**UNIVERSIDADE FEDERAL DE PELOTAS**
**Centro de Desenvolvimento Tecnológico**
**Programa de Pós-Graduação em Computação**


Tese

# Um *framework* baseado em contêineres para sistemas multiagente abertos

**Gustavo Lameirão de Lima**

Pelotas, 2024

**Gustavo Lameirão de Lima**

# Um *framework* baseado em contêineres para sistemas multiagente abertos

Tese apresentada ao Programa de Pós-Graduação em Computação do Centro de Desenvolvimento Tecnológico da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Doutor em Ciência da Computação.

Orientador: Prof. Dr. Marilton Sanchotene de Aguiar

Pelotas, 2024

Universidade Federal de Pelotas / Sistema de Bibliotecas
Catalogação da Publicação

L732f Lima, Gustavo Lameirão de

Um *framework* baseado em contêineres para sistemas multiagente abertos [recurso eletrônico] / Gustavo Lameirão de Lima ; Marilton Sanchotene de Aguiar, orientador. — Pelotas, 2024.
100 f. : il.

Tese (Doutorado) — Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, 2024.

1. Sistemas multiagente. 2. Sistemas multiagente abertos. 3. Simulação baseada em agentes. 4. Contêineres. 5. Docker. I. Aguiar, Marilton Sanchotene de, orient. II. Título.

CDD 005

Elaborada por Simone Godinho Maisonave CRB: 10/1733

**Gustavo Lameirão de Lima**

**Um *framework* baseado em contêineres para sistemas multiagente abertos**

Tese aprovada, como requisito parcial, para obtenção do grau de Doutor em Ciência da Computação, Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas.

**Data da Defesa:** 12 de abril de 2024

**Banca Examinadora:**

Prof. Dr. Marilton Sanchotene de Aguiar (orientador)
Doutor em Computação pela Universidade Federal do Rio Grande do Sul.

Profa. Dra. Diana Francisca Adamatti
Doutora em Engenharia Elétrica pela Universidade de São Paulo.

Prof. Dr. Paulo Roberto Ferreira Júnior
Doutor em Computação pela Universidade Federal do Rio Grande do Sul.

Profa. Dra. Rejane Frozza
Doutora em Computação pela Universidade Federal do Rio Grande do Sul.

Prof. Dr. Ulisses Brisolara Corrêa
Doutor em Computação pela Universidade Federal de Pelotas.

Dedico este trabalho à minha família, em especial aos meus pais, Luiz Fernando Moura de Lima e Mara Rosani Lameirão de Lima, à minha namorada e ao meu orientador.

# AGRADECIMENTOS

Aproveito este momento para agradecer a todos que, de alguma forma, participaram dessa trajetória.

Aos amigos, por todas as palavras de carinho e afeto.

Aos colegas do grupo de Sistemas Inteligentes da UFPEL, tanto pelos momentos de descontração, quanto pelos aprendizados.

À UFPEL e aos servidores, por prover toda a estrutura necessária para os estudos.

Aos professores do Programa de Pós-Graduação em Computação da UFPEL, por todos os ensinamentos que me auxiliaram a aperfeiçoar tanto o trabalho quanto meu conhecimento acadêmico.

À banca examinadora de qualificação e defesa, por aceitarem o convite e pelas contribuições para aprimorar o trabalho.

Por fim, agradeço à CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pelo auxílio financeiro durante parte da realização do estudo.

# RESUMO

LIMA, Gustavo Lameirão de. **Um *framework* baseado em contêineres para sistemas multiagente abertos**. Orientador: Marilton Sanchotene de Aguiar. 2024. 100 f. Tese (Doutorado em Ciência da Computação) – Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2024.

Os Sistemas Multiagente (SMA) são formados por agentes que interagem entre si e desenvolvem comportamento autônomo e cooperativo. Estes tipos de sistemas são utilizados para resolver problemas nos quais as entidades são descentralizadas. Existe um tipo específico de SMA que permite a interação entre agentes que participam de diferentes modelos, os Sistemas Multiagente Abertos (SMAA). Nos SMAA, agentes heterogêneos que estão inseridos em diferentes ambientes ou modelos, podem migrar de um sistema para outro levando consigo seus atributos e conhecimentos. Porém, surgem problemas complexos ao se desenvolver aplicações em SMAA, quando comparado aos SMA. Por exemplo, os problemas de implementação, cujos agentes e modelos podem ser desenvolvidos por diferentes equipes, em diferentes linguagens de programação, ou até mesmo em diferentes plataformas/arquiteturas de agentes. Além disso, a abertura de SMA não é uma tarefa fácil, por conta das incertezas e por todo o comportamento dinâmico que a troca de agentes acarreta. Dessa forma, é necessário formular técnicas para analisar essa complexidade e facilitar a compreensão do comportamento global do sistema. Dado o contexto apresentado, faz-se necessário o esforço de pesquisa na redução da complexidade apresentada pelos SMAA. O objetivo desta pesquisa é propor um *framework* para auxiliar o desenvolvimento de Sistemas Multiagente Abertos, baseado em contêineres, visando facilitar a migração de agentes entre distintos modelos que podem executar em cenários heterogêneos de *hardware* e *software*. Foram desenvolvidos cenários de simulação a partir dos modelos *Sugarscape 2 Constant Growback* (NetLogo) e *Gold Miners* (JaCaMo) para fins de análise de viabilidade de implementação do *framework*. Os testes abrangeram execuções locais, remotas (nuvem) e híbridas. Os testes realizados mostraram que o *framework* possibilita a troca de agentes entre diferentes modelos que podem ser desenvolvidos em diferentes plataformas.

Palavras-chave: sistemas multiagente; sistemas multiagente abertos; simulação baseada em agentes; contêineres; docker.

# ABSTRACT

LIMA, Gustavo Lameirão de. **A container-based *framework* for open multi-agent systems**. Advisor: Marilton Sanchotene de Aguiar. 2024. 100 f. Thesis (Doctorate in Computer Science) – Technology Development Center, Federal University of Pelotas, Pelotas, 2024.

Multi-agent Systems (MAS) are formed by agents that interact with each other and can develop autonomous and cooperative behavior. These types of systems are used to solve problems where entities are decentralized. A specific type of MAS allows interaction between agents that participate in different models, such as the Open Multi-agent Systems (OMAS). In OMAS, heterogeneous agents inserted in different environments or models can migrate from one system to another, taking their attributes and knowledge. However, complex problems arise when developing OMAS applications compared to MAS. For example, there may be implementation issues, where agents and models may be developed by different teams, in different programming languages, or even on different agent platforms/architectures. Furthermore, opening MAS is challenging due to the uncertainties and all the dynamic behavior that the exchange of agents entails. Thus, it is necessary to formulate techniques to analyze this complexity and facilitate the understanding of the system's global behavior. Given the context presented, it is necessary to make a research effort to reduce the complexity of the OMAS. This research proposes a framework to assist the development of Open Multi-agent Systems based on containers to facilitate the migration of agents between different models that can execute in heterogeneous scenarios of *hardware* and *software*. Simulation scenarios were developed based on the *Sugarscape 2 Constant Growback* (NetLogo) and *Gold Miners* (JaCaMo) models to analyze the feasibility of implementing the framework. The tests cover local, remote (cloud), and hybrid executions. The tests showed that the framework allows the exchange of agents between different models that can be developed on different platforms.

Keywords: multi-agent systems; open multi-agent systems; multi-agent-based simulation; containers; docker.

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| ACM | *Association for Computing Machinery* |
| AGR | *Agent/Group/Role* |
| AOP | *Aspect-Oriented Programming* |
| API | *Application Programming Interface* |
| ASL | *AgentSpeak Language* |
| AWS | *Amazon Web Services* |
| BDI | *Belief–desire–intention* |
| CAPES | Coordenação de Aperfeiçoamento de Pessoal de Nível Superior |
| CLI | *Command Line Input* |
| CSS | *Cascading Style Sheets* |
| CTCM | *Collaborative Team Construction Model* |
| DB | *Database* |
| DBMS | *Database Management System* |
| FIFO | *First In, First Out* |
| GUI | *Graphical User Interface* |
| HTML | *HyperText Markup Language* |
| HTTP | *Hypertext Transfer Protocol* |
| IA | Inteligência Artificial |
| IAD | Inteligência Artificial Distribuída |
| IDE | *Integrated Development Environment* |
| IP | *Internet Protocol* |
| JADE | *Java Agent DEvelopment Framework* |
| JS | *JavaScript* |
| JSON | *JavaScript Object Notation* |
| MAS | *Multi-agent Systems* |
| OMAS | *Open Multi-agent Systems* |

| | |
|---|---|
| OS | *Operating system* |
| PHP | *Hypertext Preprocessor* |
| REST | *Representational State Transfer* |
| SDK | *Software Development Kit* |
| SGBD | Sistema Gerenciador de Banco de Dados |
| SIS | *Susceptible-Infected-Susceptible* |
| SMA | Sistemas Multiagente |
| SMAA | Sistemas Multiagente Abertos |
| SPS | *Symmetric Push-Sum* |
| SQL | *Structured Query Language* |
| SSH | *Secure Shell* |
| Unix | *UNiplexed Information Computing System* |
| VM | *Virtual Machine* |

# SUMÁRIO

# 1 INTRODUÇÃO

Atualmente, diversas áreas utilizam conceitos da Inteligência Artificial para resolver problemas. Geralmente, as soluções são baseadas em Algoritmos Genéticos e Aprendizado de Máquina, utilizando métodos como *Gradient Boosting*, *Random Forest* e *Deep Learning*. Essas soluções apresentam bom desempenho quando o problema é bem definido e o conjunto de dados é grande e centralizado.

Porém, também existem problemas dinâmicos, com diversas entidades que interagem. Para solucionar tais problemas, é necessário a utilização de soluções com controle descentralizado e que sejam capazes de se adaptar em tempo de execução (Perles; Crasnier; Georgé, 2018). Para este contexto, outra solução encontrada na Inteligência Artificial são os Sistemas Multiagente (SMA).

Esses sistemas são compostos por múltiplos agentes, os quais são entidades físicas ou virtuais que têm comportamento autônomo, ou seja, conseguem agir por conta própria. Por meio de sensores, esses agentes conseguem perceber o ambiente em que estão inseridos e, via atuadores, eles podem atuar nesse ambiente (Reis, 2003; Wooldridge, 2001).

Ainda existe um tipo específico de SMA que permite a interação entre agentes que participam de diferentes modelos, os Sistemas Multiagente Abertos (SMAA) (Hewitt, 1991; Artikis; Sergot; Pitt; Busquets; Riveret, 2016). Nos Sistemas Multiagente Abertos, agentes heterogêneos (com diferentes características), os quais são analisados sob diferentes perspectivas (inseridos em diferentes ambientes/modelos), podem migrar de um modelo para outro, levando consigo seus atributos e conhecimentos. A heterogeneidade dos agentes pode vir por algumas diferenças entre os modelos, como a arquitetura, objetivos ou políticas.

Porém, surgem distintos problemas quando se desenvolvem aplicações em SMAA, quando comparado aos Sistemas Multiagente. Nos SMAA podem existir desafios de implementação, onde os agentes e modelos necessitam ser desenvolvidos por diferentes equipes, em diferentes linguagens de programação, ou até mesmo em diferentes plataformas/arquiteturas de agentes.

De maneira geral, não é possível garantir que os agentes irão agir de forma cooperativa e coordenada, devido à possibilidade de conflitos de objetivos, oriundos naturalmente do comportamento dinâmico e incerto que a troca de agentes acarreta (Artikis; Sergot; Pitt; Busquets; Riveret, 2016; Sergot, 2005; Uez, 2018; Hattab; Chaari, 2021).

Jamroga; Meski; Szreter (2013) apontam o conceito de diminuição de abertura, que está relacionado com a sobrecarga do modelo. O conceito se dá na forma de uma escala, que mensura a possibilidade de agentes entrarem e saírem do sistema sem alterar o desenho de muitos componentes. Sendo assim, quanto menor for a quantidade de alterações no modelo necessárias para ser possível receber e enviar agentes, maior o seu grau de abertura.

Assim, um sistema aberto perfeito não precisaria de transformações (passos) adicionais para acomodar novos agentes, enquanto, no outro lado, existem sistemas que iriam precisar de uma remodelagem completa para que novos agentes pudessem chegar. Dado o contexto apresentado, faz-se necessário o esforço de pesquisa na redução da complexidade apresentada pelos Sistemas Multiagente Abertos no que tange os problemas citados.

O uso de um *framework* baseado em contêineres, através de ferramentas como o Docker, para o desenvolvimento de SMAA, se favorece com a redução da complexidade do processo de migração de agentes entre modelos, executando em distintos contêineres, em cenários de *hardware* e *software* heterogêneos como, por exemplo, diferentes ambientes de desenvolvimento e de ferramentas para SMA, em diferentes sistemas operacionais.

O Docker é uma ferramenta de código aberto que permite com que aplicações e a infraestrutura destas sejam empacotadas na forma de contêineres (Turnbull, 2014; Docker, 2021a). Essa implementação facilita a maneira de implementar, gerenciar e executar as aplicações. Os contêineres do Docker incluem todas as dependências necessárias para executar uma aplicação, garantindo um funcionamento igual desde o desenvolvimento até a produção. Além disso, os contêineres criam um ambiente de execução leve e rápido. Essa simplificação permite que o processo de desenvolver, distribuir e escalonar aplicações se torne mais simples.

A dinâmica de alocação de recursos e orquestração de contêineres do Docker permite o dimensionamento das aplicações de forma rápida e eficiente, respondendo às mudanças de projeto e buscando evitar a necessidade de um provisionamento excessivo e dispendioso de recursos. Ao automatizar tarefas de implantação, dimensionamento e gerenciamento, pode-se reduzir a necessidade de intervenção manual, melhorando, assim, sua eficiência geral (Acharya; Suthar, 2022).

Outro aspecto crucial é a separação da responsabilidade operacional do *framework* em contêineres distintos dos modelos que compõem os SMAA. Por exemplo, os modelos precisam apenas definir como irão enviar e receber agentes, diminuindo a complexidade da transação no modelo, mantendo quase toda sua originalidade. Além disso, agentes que transitam entre modelos poderão transferir conhecimento, carregando consigo seus atributos.

Além disso, o uso de contêineres permite uma maior modularização do sistema, sendo que a implementação de cada módulo em um contêiner possibilita a execução de diferentes ferramentas, em diferentes sistemas operacionais, além de permitir de modo simples a substituição do código através da mudança na execução do contêiner.

Essa modularização permite que, por exemplo, uma estrutura inicial de roteamentos de agentes entre modelos seja estendida por um modelo no qual os agentes podem ser retreinados antes da nova utilização nos modelos de simulação. Além disso, através da modularização em contêineres, cada parte do *framework* pode ou não conhecer outras, conforme essa restrição for necessária. Por fim, a modularização também permite adicionar novos comportamentos ao sistema que não foram previstos em sua fase de concepção.

De maneira geral, esta Tese visa avançar na generalização de SMAA, reduzindo a complexidade do processo de abertura.

## 1.1 Objetivos

O objetivo geral desta Tese é propor um *framework* baseado em contêineres para o desenvolvimento de Sistemas Multiagente Abertos com foco no suporte ao transporte de agentes e suas informações entre modelos distintos.

Tem-se como objetivos específicos:

- especificação do *framework* e os módulos de: modelos e gatilhos, roteamento de agentes, registro de agentes, armazenamento de dados (conhecimento dos agentes), interfaces de comunicação do *framework* e de apresentação de resultados;
- desenvolvimento de um produto mínimo viável do *framework* e seus módulos para viabilizar as análises;
- especificação e desenvolvimento dos módulos dos modelos de acordo com cenários de simulação (local, nuvem e híbrido) de modo a verificar o potencial do *framework*.

## 1.2 Contribuições desta Tese

Entre as principais contribuições desta Tese pode-se elencar as seguintes:

**Proposta do *framework*:** um *framework* que apresenta o uso de contêineres, desenvolvidos através do Docker, como ferramenta para o desenvolvimento de SMAA.

**Desenvolvimento baseado em contêiner:** o *framework* permite que o código seja executado dentro de contêineres com a mesma estrutura operacional na qual foi desenvolvido, minimizando problemas que possam surgir durante a transição do desenvolvimento para a produção.

**Modularidade e adaptabilidade:** os contêineres do *framework* são responsáveis pela execução dos blocos da estrutura, tornando-o altamente modular e adaptável a diversos cenários.

**Compatibilidade com a nuvem:** a abordagem do *framework* baseado em contêineres, através de ferramentas como o Docker, facilita a implantação dos SMAA na nuvem, melhorando a portabilidade de aplicativos e expandindo sua aplicabilidade.

**Suporte para diferentes plataformas de agentes:** o suporte do *framework* para duas plataformas de agentes amplamente utilizadas, NetLogo e JaCaMo, ao mesmo tempo que pode ser estendido para suportar outras plataformas, como JADE e Mesa.

**Cenários de implantação flexíveis:** testes do *framework* em cenários de implantação locais, remotos e híbridos.

**Migração de agentes:** o *framework* permite que os agentes se movimentem livremente entre modelos, compartilhando informações do agente e reduzindo a complexidade de adaptação do código.

## 1.3 Estrutura do Texto

Este documento está dividido como segue. O Capítulo 2 apresenta o Referencial Teórico, trazendo os conceitos e definições importantes para entendimento do trabalho, além de apresentar os trabalhos relacionados. O Capítulo 3 apresenta detalhes da abordagem desenvolvida, tanto dos aspectos conceituais quanto dos tecnológicos utilizados na implementação. O Capítulo 4 mostra os resultados obtidos ao longo da implementação e validação da ferramenta, com ênfase nos cenários de simulação desenvolvidos, sendo de grande importância para constatar a viabilidade da implementação. Por fim, o Capítulo 5 apresenta as considerações finais do estudo.

# 2 REFERENCIAL TEÓRICO

Segundo Norvig; Russell (2014), existem diversas definições aceitas e utilizadas por diferentes pesquisadores e com métodos diferentes para o que é Inteligência Artificial (IA). Estas definições estão geralmente divididas em duas categorias: i) pensamento ou raciocínio e ii) comportamento.

As definições de pensamento ou raciocínio estão relacionadas com a capacidade de um sistema pensar e raciocinar como um ser humano. Esses sistemas são avaliados conforme o seu sucesso em ser fiel ao desempenho humano nas atividades de pensamento e raciocínio. Já as definições de comportamento estão relacionadas aos sistemas cujo objetivo é atingir o sucesso em comparação a uma inteligência ideal, também chamada de racionalidade, quando se tomam decisões corretas com base nos dados conhecidos.

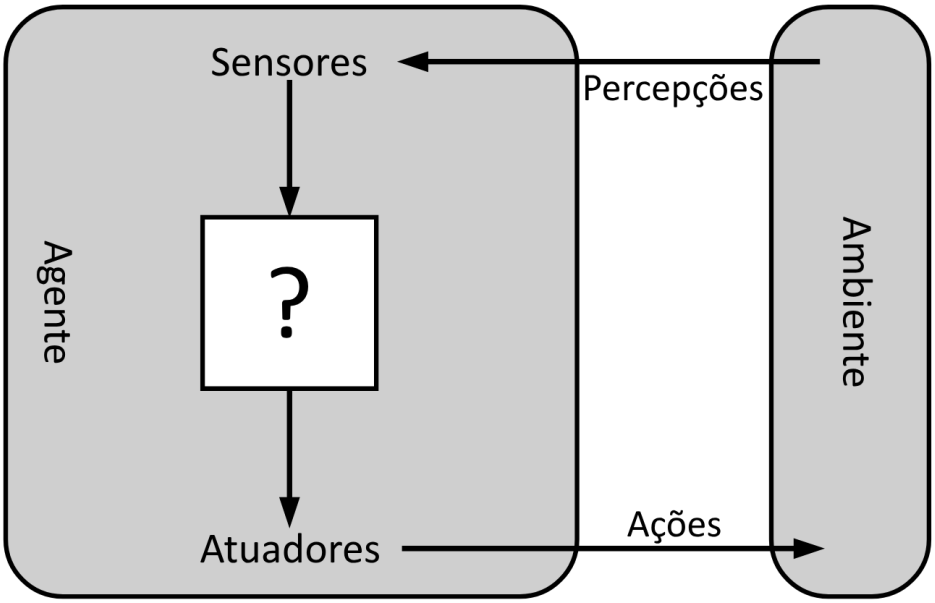
Com o avanço na computação, surgiu um novo conceito dentro da Inteligência Artificial, sendo a Inteligência Artificial Distribuída (IAD). Essa área utiliza os recursos de paralelismo na computação para que sistemas que resolvem problemas através do uso de IA possam dividir estes problemas em subproblemas, de forma que toda a arquitetura dos computadores possa ser aproveitada durante a execução destes programas, tornando essa execução mais rápida (Ferber, 1999).

Têm-se os Sistemas Multiagente (SMA) como parte da IAD, por serem compostos por um grupo de agentes inteligentes, inseridos em um ambiente, que trabalham em conjunto visando atingir um objetivo global. Estes agentes artificiais apresentam comportamento autônomo, ou seja, conseguem agir por conta própria. Porém, ao mesmo tempo que são autônomos, estes agentes podem interagir entre si para poderem concluir o objetivo global. A Seção 2.1 apresenta as características dos agentes que compõem um SMA e a Seção 2.2 apresenta características presentes nos SMA. A Seção 2.3 expõe uma introdução acerca dos Sistemas Multiagente Abertos e a Seção 2.4 apresenta os trabalhos encontrados na literatura relacionados com a abordagem desenvolvida.

## 2.1 Características de Agentes

Agentes são definidos como entidades, reais ou virtuais, com comportamento autônomo, ou seja, capazes de agir sozinhos. Os agentes conseguem perceber alterações no ambiente em que estão inseridos por meio de sensores. Além disso, os agentes também podem agir no ambiente, através da utilização de atuadores. A Figura 1 apresenta uma ilustração da interação entre um agente e o ambiente.

Figura 1 – Agentes interagem com ambientes por meio de sensores e atuadores.




Fonte: adaptado de Norvig; Russell (2014).

Um agente humano consegue perceber alterações no ambiente através de seus olhos, ouvidos e outros órgãos, além de conseguir atuar neste ambiente através das suas mãos, pernas, boca e outras partes do corpo. De maneira similar a um humano, um agente robótico pode ter sensores, como câmeras e detectores de faixa infravermelho, e podem atuar no ambiente através de seus motores (Norvig; Russell, 2014).

Segundo Norvig; Russell (2014), existem diversas características usadas para classificar agentes, dentre elas, são destacadas e definidas quatro categorias, que englobam os princípios subjacentes a quase todos os sistemas inteligentes:

**Agentes reativos simples:** realizam ações utilizando como base apenas a percepção (leitura) atual, sem considerar o histórico de percepções. Estes agentes, do tipo mais simples, leem o estado atual do ambiente e decidem com base nessa leitura. Os comportamentos reativos, embora simples, também ocorrem em parte das ações de agentes mais complexos.

**Agentes reativos baseados em modelo:** guardam informações de percepções anteriores, diferente dos agentes reativos simples, permitindo que estes agentes possam tomar decisões melhores.

**Agentes baseados em objetivos:** conseguem mudar seus objetivos parciais para atingir o objetivo global e, por apresentarem este comportamento dinâmico, são considerados mais flexíveis do que os agentes reativos.

**Agentes baseados na utilidade:** utilizam algum critério para tomarem decisões melhores. Existem diversas sequências de ações que levam ao objetivo global, porém existem soluções que são melhores que as outras, seja por serem mais rápidas, mais seguras, ou seja, por outro critério. Decisões que aumentam a função de utilidade do agente tendem a ser soluções melhores e, portanto, decisões que terão maior prioridade para o agente.

## 2.2 Características de Sistemas Multiagente

Os Sistemas Multiagente são compostos por diversos agentes que interagem ou trabalham conjuntamente. Estes agentes podem ser homogêneos ou heterogêneos. A classificação acontece através da análise das características dos agentes, como sensores, atuadores, forma de comunicação, dentre outros. Dessa forma, se os agentes têm capacidades iguais, são classificados como agentes homogêneos. Já os agentes com capacidades diferentes são classificados como heterogêneos (Reis, 2003).

Agentes são elementos que conseguem resolver problemas de forma autônoma e podem operar de maneira assíncrona. Para um agente funcionar como parte do sistema, é necessário existir uma infraestrutura que permita a comunicação e/ou interação entre esses agentes (Reis, 2003).

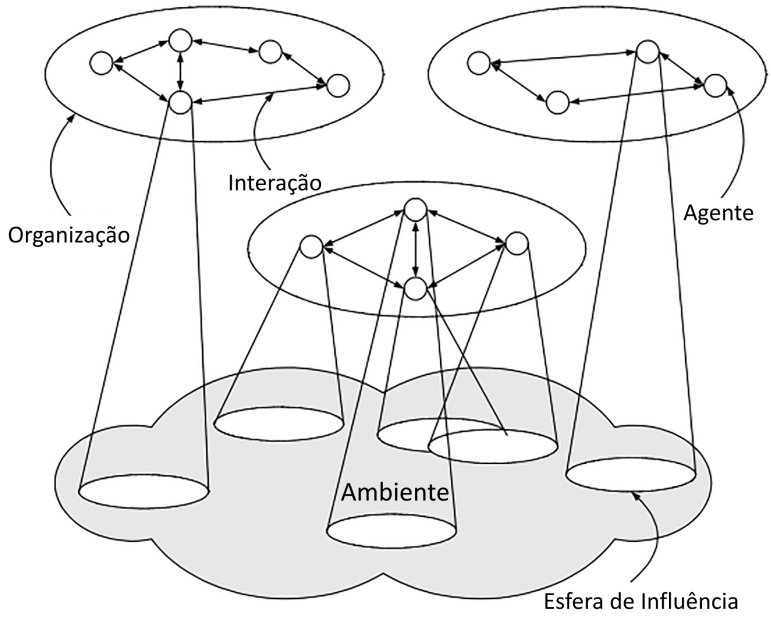
Um ponto essencial na construção de sociedades de agentes é gerenciar interações e dependências das atividades dos diferentes tipos de agentes no contexto de Sistemas Multiagente. Um exemplo desse gerenciamento é a coordenação, considerada um papel essencial nos SMA porque esses sistemas são geralmente distribuídos (Reis, 2003).

Existem diversas metodologias de coordenação, divididas geralmente em dois grupos: i) metodologias aplicáveis em domínios que contêm agentes competitivos (que agem por interesse próprio); e, ii) metodologias aplicáveis a domínios que contêm agentes cooperativos.

No primeiro caso, são sistemas nos quais os agentes se preocupam com o seu bem próprio. Nesse tipo de sistema, a coordenação através da negociação é a metodologia mais aceita. Já no segundo caso, são estudadas metodologias que permitem a construção de equipes de agentes. As metodologias mais relevantes nesse contexto

são as que permitem definir uma organização estrutural da sociedade de agentes, através da definição e troca de papéis, definição e alocação de tarefas aos agentes e o planejamento do grupo de agentes (Reis, 2003). A Figura 2 ilustra a estrutura de um Sistema Multiagente. Nela, cada agente tem uma percepção parcial do ambiente, são apresentadas a esfera de influência dos agentes, a forma com que cada agente interage com os demais e sua organização.

Figura 2 – Estrutura de um SMA.




Fonte: adaptado de Norvig; Russell (2014).

Segundo Reis (2003) e Wooldridge (2001), a principal motivação do uso de SMA se deve ao fato de que a maioria dos problemas tem características distribuídas. Além disso, também são motivações: i) problemas podem ser muito grandes para serem implementados em apenas um agente; ii) deve-se permitir que seja possível integrar novos sistemas com sistemas legados, ou seja, sistemas antigos cuja manutenção do código não é viável; iii) deve-se prover uma solução natural para problemas geograficamente e/ou funcionalmente distribuídos; iv) permitir com que a interface homem-máquina seja mais natural, para ambos serem tratados como agentes no sistema; e, v) tornar mais clara e simples a concepção de projetos.

Além disso, Reis (2003) aponta que a utilização de SMA na resolução de problemas nos quais o conhecimento ou atividade são distribuídos apresentam benefícios como: i) resolução mais rápida de problemas por conta do processamento concorrente; ii) diminuição no volume de comunicação, devido ao processamento ser realizado junto da fonte de informação e a comunicação entre as entidades ser feita em alto nível; iii) sistemas mais flexíveis e escaláveis, por conta da possibilidade de conectar múltiplos sistemas com arquiteturas diferentes; iv) maior confiabilidade, devido à inexistência

de pontos únicos de falha; v) maior capacidade de resposta, por conta de sensores, atuadores e sistemas de processamento estarem diretamente no interior dos agentes; e, vi) facilidade no desenvolvimento de sistemas complexos, devido à modularidade resultante da separação de problemas e na separação dos sistemas em agentes semiautônomos.

Em Stone; Veloso (2000) são apontadas mais razões para a utilização de SMA, tais como: i) a natureza do problema exige a utilização desses sistemas, por exemplo, devido à distribuição espacial dos seus componentes; ii) exploração do paralelismo, distribuindo funções para cada agente, de forma com que a execução do sistema se torne mais rápida; iii) robustez, pois por se utilizar diferentes agentes, não existem pontos únicos de falha no sistema, ou seja, é possível garantir que o sistema pode continuar operando em caso de falha; iv) escalabilidade, através da inserção de novos agentes ao sistema conforme a capacidade do computador utilizado para rodar o programa; v) simplificação das tarefas individuais, através da divisão do problema global em subproblemas; e, vi) estudo da IA e de comportamentos sociais, pois com frequência os agentes que compõem os SMA apresentam comportamentos de sociedade e, portanto, modelos baseados em agentes podem ser utilizados para estudar comportamentos que emergem em sociedades.

## 2.3 Características de Sistemas Multiagente Abertos

Em Sistemas Multiagente, agentes podem trocar informações entre si por diversos propósitos, visando obter conhecimento coletivo, atualizar crenças, otimizar estratégias, dentre outros. Ainda, existe um tipo específico de SMA que permite a interação entre agentes que participam de diferentes modelos, os Sistemas Multiagente Abertos.

Nos Sistemas Multiagente Abertos, agentes heterogêneos (com diferentes características) os quais são analisados sob diferentes perspectivas (inseridos em diferentes ambientes/modelos) podem migrar de um sistema para outro, levando consigo seus atributos e conhecimentos (Hewitt, 1991; Artikis; Sergot; Pitt; Busquets; Riveret, 2016). A heterogeneidade dos agentes pode surgir por algumas diferenças entre os modelos, como a arquitetura, objetivos ou políticas. Além disso, os agentes e modelos podem ser desenvolvidos por diferentes equipes, em diferentes linguagens de programação, ou até mesmo em diferentes plataformas de agentes. Dessa forma, de maneira geral, não é possível garantir que os agentes irão agir de forma cooperativa e coordenada pela possibilidade de conflitos de objetivos (Artikis; Sergot; Pitt; Busquets; Riveret, 2016; Sergot, 2005; Uez, 2018).

Os Sistemas Multiagente Abertos precisam lidar com problemas que não estão presentes nos sistemas fechados (SMA comum). A migração dos agentes entre modelos pode ocorrer em tempo de execução, e a motivação para essa migração de um agente de um sistema para outro pode ser diversa, ficando à escolha do desenvolvedor, como falhas na execução, vontade própria ou algum tipo de gatilho. Além disso, é possível ocorrerem conflitos de interesses entre os novos agentes, já que não foram projetados para trabalhar naquele conjunto (Artikis; Sergot; Pitt; Busquets; Riveret, 2016; Sergot, 2005; Uez, 2018; Hattab; Chaari, 2021).

Embora o assunto já seja conhecido há bastante tempo (Hewitt, 1991; Eijk; Boer; Van der hoek; Meyer, 1999), existem pesquisas recentes na área, mostrando que o problema é complexo e relevante, como os trabalhos relacionados que serão apresentados a seguir.

## 2.4 Trabalhos Relacionados

Nesta Seção estão apresentados os trabalhos do estado da arte onde esta Tese está inserida. Para isto, utilizaram-se três bases de dados para o levantamento, sendo elas: (i) Google Scholar; (ii) IEEE Xplore; e, (iii) ACM Digital Library. Os termos da busca utilizada foram:

- *Open Multi-agent Systems*
- *Openness in Multi-agent Systems*

Para cada termo, também foram buscadas suas permutações (como variações de hifens, espaços, dentre outros), além das siglas, como, por exemplo, OMAS (*Open Multi-agent Systems*). Já para o processo de filtragem dos artigos, foram utilizados critérios de inclusão como:

- Escritos em inglês, pela abrangência;
- Preferencialmente com data de publicação maior ou igual a 2017, para tratar de artigos mais recentes; e,
- Contivessem as palavras-chave de pesquisa no título, *abstract* ou texto.

Dentre os resultados encontrados, além das referências citadas por estes, alguns artigos foram selecionados conforme sua relevância e serão detalhados a seguir.

A primeira perspectiva da maioria dos trabalhos relacionados considera Sistemas Multiagente Abertos do ponto de vista da arquitetura/organização, como os trabalhos de Demazeau; Costa (1996), Paurobally; Cunningham; Jennings (2003), Gonzalez-Palacios; Luck (2006), Singh; Chopra (2009), Dalpiaz; Chopra; Giorgini; Mylopoulos (2010), Artikis (2012), Jamroga; Meski; Szreter (2013), Hendrickx; Martin (2017), Houhamdi; Athamena (2020), Hattab; Chaari (2021), Jiang; Zhang (2023), Eck; Soh; Doshi (2023) e de Kondylidis; Tiddi; Teije (2023), dentre outros.

Estes estudos diferem da abordagem deste trabalho, principalmente porque, dentre todos os problemas dos SMAA, este trabalho dedica-se à questão da abertura, permitindo que os agentes se movimentem entre os modelos. Os efeitos dos conflitos de interesse que podem advir da movimentação dos agentes entre os modelos devem ser resolvidos pelos próprios modelos. Este trabalho visa fornecer mecanismos para simplificar a movimentação dos agentes entre os modelos reduzindo alterações no código original do modelo.

Por outro lado, os trabalhos de Ramirez; Fasli (2017), Uez (2018), Perles; Crasnier; Georgé (2018), Dähling; Razik; Monti (2021) e de Pfeifer; Passini; Dorante; Guilherme; Affonso (2021), dentre outros, apresentam aspectos semelhantes com a abordagem desenvolvida nesta Tese.

Na Seção 2.4.1, são apresentados os trabalhos relacionados pelo ponto de vista da arquitetura/organização. Na Seção 2.4.2, são apresentados os trabalhos relacionados pelo ponto de vista da engenharia de *software*, destacando as principais diferenças com a abordagem desenvolvida neste trabalho. Por fim, a Seção 2.4.3 apresenta o fechamento da Seção de trabalhos relacionados.

### 2.4.1 Ponto de Vista da Infraestrutura ou Organização

Demazeau; Costa (1996) tratam de um modelo formal de Sistema Multiagente com organizações dinâmicas. Os autores utilizam uma abordagem de *toolbox* para programação orientada a agentes como uma diretriz computacional para sua definição. Os autores tratam de conceitos como organização dinâmica em SMA, o que poderia levar a um SMAA.

Paurobally; Cunningham; Jennings (2003) apresentam um estudo sobre os desafios em especificar e implementar protocolos de interação de agentes com a mesma interpretação por todas as partes de um Sistema Multiagente Aberto. Os autores observam esses problemas tanto na perspectiva do projetista do sistema quanto do desenvolvedor. Os autores destacam a dificuldade em implementar um protocolo que seja completo, a falta de bibliotecas, a verificabilidade dos protocolos, as propriedades e diferenças importantes na hora de escolher um protocolo.

Gonzalez-Palacios; Luck (2006) apresentam uma proposta de estrutura de como construir Sistemas Multiagente Abertos, com ênfase na observação das especificações dos agentes que irão entrar no sistema, além de garantir que não serão violadas as restrições do sistema. Para os autores, os sistemas devem contar com identificadores de papel, protocolo, serviço e identificação de conceitos gerais.

Singh; Chopra (2009) propõem desenvolver Sistemas Multiagente sobre a perspectiva de arquitetura. O trabalho apresenta como especificar os agentes, sua autonomia e interconexões em protocolos de alto nível. Através dessa modelagem, é possível modelar uma arquitetura aberta, o que tornaria o Sistema Multiagente Aberto.

Em Dalpiaz; Chopra; Giorgini; Mylopoulos (2010) destacam-se os desafios da adaptação em Sistemas Multiagente Abertos. A modelagem das interações é feita por compromissos sociais, sendo compromissos feitos entre diferentes tipos de agentes. O trabalho propõe a formalização da noção de estratégias para objetivos não apenas em termos de objetivos e planos, mas também dos compromissos necessários. Além disso, é feito um modelo conceitual e um *framework* para a adaptação dessa noção de estratégia.

O trabalho de Artikis (2012), derivado de Pitt; Mamdani; Charlton (2001); Artikis; Sergot; Pitt (2007, 2009); Artikis; Sergot (2010), apresenta uma infraestrutura para especificação dinâmica de SMA abertos, ou seja, especificação que pode ser modificada em tempo de execução pelos agentes. Os autores avaliam propostas de modificação de regras da modelagem mediante especificação dinâmica como uma métrica de espaço, além de considerar os efeitos de aceitar uma proposta de sistema de utilidade. A linguagem de programação utilizada foi o C+ (linguagem do tipo *action language*) para formalizar as especificações.

Posteriormente, este trabalho serviu como base para outros estudos, como em Artikis; Sergot; Pitt; Busquets; Riveret (2016). Neste novo estudo, os autores apresentam um modelo baseado em lógica para especificar e executar SMAA, para ser possível lidar com a imprevisibilidade dos SMAA, visto que o comportamento dos agentes pode ser previsível e/ou incontrolável. Por fim, este novo estudo também aponta aplicações da técnica em sistemas de votação (Pitt; Kamara; Sergot; Artikis, 2006), compartilhamento de recursos (Pitt; Schaumeier; Artikis, 2012), negociação (Artikis; Sergot; Pitt, 2009; Artikis; Sergot, 2010) e argumentação (Artikis; Sergot; Pitt, 2007).

Jamroga; Meski; Szreter (2013) apresentam mais detalhes do conceito de abertura em não só, mas também, Sistemas Multiagente. O conceito de abertura está relacionado com a possibilidade de agentes entrarem e saírem do sistema sem alterar o desenho de muitos componentes. A abertura pode estar relacionada a outros tipos de sistemas, como redes que recebem novos dispositivos. Os autores utilizam como base para a medição de abertura uma definição simples, tratando a abertura do sistema como a possibilidade de adicionar e remover agentes para e de um modelo. A

abertura se dá na forma de uma escala, que vai de zero até um. O zero significa um SMAA perfeito, no qual são necessários zero passos adicionais para receber novos agentes. Já o um significa um SMAA no qual seriam necessários muitos passos para incluir novos agentes (SMA fechado).

Em Hendrickx; Martin (2017) é abordada uma estratégia para lidar com SMAA, com foco em sistemas cujas interações entre os agentes são feitas por meio de fofocas (do inglês *gossip*, trata-se de comunicações feitas apenas entre nodos vizinhos). Nestes sistemas, os agentes podem chegar ou serem substituídos em momentos aleatórios. Estes eventos impedem a convergência do sistema, por serem dinâmicos. Para tal, os autores desenvolveram uma técnica que descreve o comportamento esperado do sistema, mostrando que a evolução dos momentos escalados pode ser caracterizada por um sistema dinâmico linear de duas dimensões. Os autores aplicam a técnica em dois casos, sendo eles: (i) sistemas com tamanho fixo, nos quais os agentes que saem são substituídos imediatamente, e (ii) sistemas nos quais novos agentes continuam chegando sem nunca sair, o que faz com que o tamanho cresça sem limites.

Em Houhamdi; Athamena (2020), os autores definem as características e requisitos para modelos de construção de equipes colaborativas (do inglês *Collaborative Team Construction Model* – CTCM) que permitem com que agentes possam formar grupos que cooperam em SMAA, através do compartilhamento de recursos restritos, para realizar conjuntos de tarefas. A proposta utiliza um modelo descentralizado, tanto na execução, no raciocínio, quanto na cooperação.

Hattab; Chaari (2021) propõem um modelo genérico para representação de abertura em Sistemas Multiagente. Os autores classificam os modelos já existentes na literatura em três categorias: modelos estruturais, funcionais e interacionais. É destacado que todos os estudos se encaixam em uma das categorias, mas não as três ao mesmo tempo, sendo este o diferencial da proposta. Dentre as diversas definições de abertura apontadas na literatura, os autores destacam três aspectos: i) *Adição e remoção de agentes* – os agentes são considerados entidades físicas ou caixas-pretas; ii) *Adição, remoção e modificação de tarefas e papéis de agentes* – analisa a evolução interna dos agentes, considerando seus conteúdos, objetivos, atributos e funcionalidades; e, iii) *Interoperabilidade e comportamento comunicativo entre os agentes* – considera as operações abertas entre agentes, com ênfase nas regras, normas e protocolos. Mais ainda, os autores enfatizam que o estudo de abertura de SMA não é uma tarefa fácil, por conta das incertezas e por todo o comportamento dinâmico que a troca de agentes acarreta. Dessa forma, é necessário formular técnicas para analisar essa complexidade e facilitar a compreensão do comportamento global do sistema. Por fim, para validação, os autores implementam e testam a proposta em um simulador de resgate multiagente, desenvolvido na ferramenta JADE (*Java Agent DEvelopment Framework*).

Colla; Galland; Hendrickx (2021) apresentam um algoritmo descentralizado de estimativa para Sistemas Multiagente Abertos, com entrada e saída de agentes. O algoritmo proposto calcula o valor médio dos dados de todos os agentes, lidando com o desafio de incorporar novos agentes sem esquecer as informações dos que saíram. O trabalho estabelece limites de desempenho empíricos e apresenta o algoritmo *Symmetric Push-Sum* (SPS), que se aproxima desses limites em até 1% geralmente.

Hu; Liu; Lan; Zhang (2021) propõem um modelo de sistema comunitário sobreposto. Este modelo vem como uma alternativa para outras abordagens que não são apropriadas quando os agentes estão em comunidades múltiplas e sobrepostas, cujos recursos são compartilhados e existem objetivos individuais e comuns a serem alcançados. Os agentes utilizam raciocínio social para aprimorar a compreensão dos objetivos dos outros agentes e suas dependências, facilitando a transferência de requisitos de recursos entre comunidades. O modelo é avaliado em um SMAA com 100 agentes compartilhando recursos restritos.

Galland; Hendrickx (2022) abordam limitações de desempenho para problemas intrínsecos de consenso médio em SMAA com chegadas e partidas frequentes. O objetivo é estimar colaborativamente os valores médios dos agentes no sistema, mas algoritmos podem não convergir devido à variação na composição e objetivos do sistema.

Haro (2022) aborda problemas em um SMAA, concentrando-se em cenários no qual o conjunto de agentes pode mudar independentemente da dinâmica do sistema. Duas abordagens são usadas: uma envolvendo ativação/desativação de agentes e outra com substituição de agentes. Três problemas são explorados: consenso aleatório com ruído aditivo, alocação de recursos com substituição de agentes e uma epidemia SIS (do inglês, *Susceptible-Infected-Susceptible*) contínua. A análise utiliza uma abordagem estocástica, fornecendo limites para quantidades escalares para avaliar os impactos das mudanças nos agentes. O modelo SIS é um modelo epidemiológico simples, também conhecido como modelo de processo de contato. Num modelo SIS, uma população com N indivíduos é categorizada em dois compartimentos: suscetível (S) e infectado (I). A doença é transmitida apenas quando um indivíduo suscetível entra em contato com um indivíduo infectado.

Xue; Tang; Ren; Qian (2022) abordam um sistema comutado multidimensional ou sistema multidimensional de múltiplos modos, que estende o sistema comutado clássico permitindo diferentes dimensões de subsistemas. O foco é o problema de estabilidade do sistema, considerando transições de estado descontínuas devido a variações de dimensão. Para subsistemas lineares, são propostas funções de *Lyapunov* paramétricas múltiplas para verificar condições de estabilidade, revelando uma conexão entre estabilidade e efeitos impulsivos. Os resultados de estabilidade obtidos são aplicados ao problema de consenso em um SMAA.

Galland; Vizuete; Hendrickx; Panteley; Frasca (2022) trata de um estudo que lida com o desempenho para problemas intrínsecos de consenso médio em SMAA, sujeitos a entradas e saídas frequentes de agentes. Segundo os autores, o objetivo é estimar colaborativamente a média dos agentes presentes no sistema. Algoritmos que resolvem esse tipo de problema podem nunca convergir, por conta das variações. Os autores citam que conseguem alcançar um desempenho ótimo para determinados modelos de substituições.

Dashti; Oliva; Seatzu; Gasparri; Franceschelli (2022) propõem um *framework* para realizar cálculo de sistemas de votação no contexto de SMAA, nos quais agentes podem entrar ou sair, tornando o cálculo mais desafiador. No *framework*, os agentes selecionam valores de um conjunto finito, e o cálculo é realizado utilizando um procedimento de consenso distribuído que preserva a média em paralelo dos valores. Os resultados da simulação demonstram a eficácia da abordagem. Os autores apontam que o estudo preenche um espaço de pesquisa sobre algoritmos de computação em modo distribuído no contexto de SMAA.

Restrepo; Loría; Sarras; Marzat (2022) abordam o problema de consenso, evitando colisões, em SMA com alcance sensorial limitado e a capacidade de adição de novas conexões e agentes ao longo do tempo. A topologia do grafo é representada por um grafo dinâmico não direcionado, assumido como conectado apenas em uma vez inicial, e o SMAA é modelado por uma representação comutada impulsiva multidimensional. Os resultados obtidos são aplicáveis também a SMA comuns (fechados) com adição de arestas. Uma simulação numérica demonstra a eficácia da abordagem proposta.

Chebout; Mokhati; Badri (2022) trata de um estudo intitulado *NC4OMAS*. A abordagem proposta pelos autores tem como foco lidar com o problema da controlabilidade em SMAA baseada em AGR (agente/grupo/papel, do inglês *Agent/Group/Role*). Para tal, os autores utilizam programação orientada a aspectos (AOP, acrônimo do inglês *Aspect-Oriented Programming*) para implementar o processo de monitoramento das normas. Os autores implementaram a proposta utilizando o *AspectJ*, por ser uma implementação comum para AOP utilizando Java.

Nakamura; Hayashi; Inuiguchi (2023) apresentam um método de tomada de decisão cooperativa para problema do tipo *adversarial bandit* em SMAA. Nesse tipo de problema, os agentes, iterativamente, aprendem a escolher a melhor opção dentre uma lista de candidatos. Os autores utilizam uma política distribuída (exp3) na qual grupos de agentes buscam maximizar o valor de recompensa.

Jiang; Zhang (2023) tratam do estudo de um problema de consenso em um SMAA com integrador duplo, no qual os agentes podem partir e chegar a qualquer momento. Um algoritmo distribuído é proposto para generalizar os esforços comuns na teoria multiagente. Como uma extensão, os autores discutem o problema de seguimento de líder em um ambiente de SMAA, sugerindo um esforço não trivial quando comparado

com o caso sem líder.

Eck; Soh; Doshi (2023) é um estudo no qual os autores discutem o problema da tomada de decisão em SMAA. Além da noção da abertura de agentes, os autores também discutem outras duas: abertura de tarefas e abertura de tipos. A abertura de tarefas acontece quando uma lista de tarefas que o agente pode fazer muda ao longo do tempo, pois novas tarefas podem ser incluídas e tarefas antigas podem ser removidas. Esse tipo de abertura acontece, por exemplo, em situações nas quais os agentes podem assumir novos papéis, com novas tarefas para cumprir. Já a abertura de tipos ocorre quando as capacidades e processos de tomada de decisão dos agentes podem mudar ao longo do tempo. Os autores destacam que esta terceira forma de abertura em SMA é menos compreendida na literatura, ainda que esteja presente em muitas aplicações de IA que interagem com o mundo real (físico).

Kondylidis; Tiddi; Teije (2023) apresentam um método para desenvolver uma compreensão compartilhada entre dois agentes em uma tarefa cooperativa, focando na comunicação eficiente em um SMAA. O método é direcionado a pesquisadores de interação humano-máquina, abordando desafios e limitações. Um caso de uso de resposta cooperativa a consultas é demonstrado, enfatizando um *framework* para experimentos de comunicação. O artigo discute a necessidade de adaptabilidade em sistemas autônomos, propondo um método que utiliza exemplos para aproximar a compreensão. O *framework* envolve dois agentes cooperando em um ciclo, testado em um cenário de resposta coletiva a consultas. Os resultados mostram uma comunicação bem-sucedida em um número limitado de interações, destacando a aplicabilidade do *framework*.

### 2.4.2 Ponto de Vista da Engenharia de *Software*

Em Ramirez; Fasli (2017), os autores propõem um modelo que usa os agentes desenvolvidos em NetLogo e Jason em um modelo especialmente complexo para agentes cognitivos, uma simulação de Desastre-Resgate. A abordagem para trazer essas duas arquiteturas diferentes de agentes, conectando o Jason ao NetLogo, é incluir parte das classes internas do NetLogo no código Jason ou, ao contrário, incluindo as classes internas do Jason no código NetLogo, já que ambas as aplicações usam código Java.

Embora este estudo considere um modelo multiagente fechado (não aberto), ele tem algumas semelhanças com a abordagem desta Tese no sentido de haver comunicação entre agentes com arquiteturas diferentes. A principal diferença é que, neste caso, ambas as aplicações utilizam código Java. Considera-se esta uma limitação do trabalho, caso o programador de agente queira utilizar esta abordagem para se comunicar, por exemplo, NetLogo, com outra plataforma de agente que utilize outra linguagem. A abordagem desta Tese cria uma plataforma que executa código iso-

lado (dentro de contêineres) permitindo ambientes de programação completamente diferentes, executando qualquer código (Java, Python) através do serviço de API.

Perles; Crasnier; Georgé (2018) apresentam o desenvolvimento de um *framework* para o desenvolvimento de Sistemas Multiagente Abertos adaptativos. O *framework* foi desenvolvido na linguagem Java. Baseado nos princípios de orientação a objetos, o *framework* AMAK utiliza três classes básicas: *AMAS*, *Agent* e *Environment*. Cada uma dessas classes abstratas deve ser implementada por quem for usar o *framework*, adaptando para o modelo. Os autores utilizam como estudo de caso o desenvolvimento de um sistema sociotécnico ambiental para fazer emergir o bem-estar ambiental.

A arquitetura apresentada por Perles; Crasnier; Georgé (2018) difere principalmente da abordagem desenvolvida nesta Tese por ser quase uma nova linguagem de programação, já que os autores comparam seus resultados com outras linguagens de programação de agentes (como Jade, NetLogo e GAMA). No entanto, a abordagem desta Tese não exige que o programador transforme seu modelo em algo novo. Em vez disso, o programador deve inserir o código para permitir que os agentes se comuniquem com o *framework*, levando a um menor esforço para permitir que os agentes se movam entre os modelos.

GAMA (Taillandier; Gaudou; Grignard; Huynh; Marilleau; Caillou; Philippon; Drogoul, 2018) é um ambiente de modelagem e simulação de código aberto fácil de usar para criar simulações espacialmente explícitas baseadas em agentes. Foi desenvolvido para ser utilizado em qualquer domínio de aplicação: mobilidade urbana, adaptação às alterações climáticas, epidemiologia, desenho de estratégias de evacuação de desastres, planejamento urbano, são alguns dos domínios de aplicação no qual os utilizadores do GAMA estão envolvidos e para os quais criam modelos.

Existem duas transformações de modelos, a expansão (adicionar agentes) e a redução (remover agentes). Dessa forma, a transformação é medida por sua complexidade, ou seja, o número de etapas para completar essa transformação. Um sistema perfeito não precisaria de transformações (0 passos) para acomodar novos componentes, enquanto no outro lado, existem sistemas que precisariam de um redesenho completo quando novos agentes chegam.

Além disso, a abertura de um agente depende do tipo de agente que pode ser enviado ou recebido. Por exemplo, um sistema com trens e controles pode adicionar facilmente novos trens, mas não necessariamente novos controles. Por fim, o contexto também é relevante. O foco não está em sistemas que arbitrariamente se expandem ou reduzem, o que é trivial, mas sim em adicionar e remover agentes enquanto mantém o comportamento do sistema intacto.

Uez (2018) propõe uma metodologia para especificação de Sistemas Multiagente Abertos. Para tal, a estrutura utiliza como base dois pilares: a modelagem independente de cada uma das dimensões do sistema (agente, ambiente e organização) e a especificação dos conceitos de borda, que visam prover informações em tempo de projeto que auxiliem os elementos da dimensão aberta a serem incluídos em tempo de execução. Na implementação, os elementos são projetados visando código para o *framework* JaCaMo. O trabalho conta com dois estudos de caso, que permitem visualizar os resultados ao longo das fases de desenvolvimento.

A principal diferença entre o trabalho de Uez (2018) e a abordagem desta Tese é que, no trabalho, o programador deve adaptar o modelo e a ferramenta à solução. Em contraste, o *framework* presente nesta Tese pode ser adaptado para a implementação atual em execução que o programador possui, usando os mecanismos de I/O que os agentes devem ter para se comunicarem. Em comparação com os parâmetros analisados no estudo, a abordagem desta Tese trabalha na dimensão agente e na fase de implementação (o estudo trata das dimensões agente/ambiente/organização e fases de análise/projeto/implementação). Lida-se com a dimensão do agente porque o foco principal desta Tese é transportar agentes entre modelos, embora seja possível estender a plataforma para transportar partes do ambiente, como artefatos. Por outro lado, esta Tese foca a abordagem na fase de implementação porque se quer que o programador adapte a menor parte possível do modelo original para a plataforma, incluindo os mecanismos para se comunicar com a plataforma. Dessa forma, o uso primário da plataforma está em um modelo já construído (fase de implementação). Por fim, também é importante destacar que o estudo está preparado apenas para lidar com a plataforma de agentes JaCaMo (agentes Jason), diferindo da abordagem desta Tese, cujo objetivo é comportar múltiplas plataformas de agentes.

No trabalho de Dähling; Razik; Monti (2021), os autores usam os SMA na área de IoT (do inglês, *Internet of Things*), pois ambos compartilham semelhanças (dispositivos distribuídos, cooperação). São utilizados SMA em IoT para construir sistemas de grande escala e tolerantes a falhas. É proposto um SMA nativo da nuvem, denominado cloneMAP, que utiliza técnicas de computação em nuvem para permitir tolerância a falhas e escalabilidade. Esta abordagem está relacionada com os SMA, mas não com os SMAA. A semelhança deste trabalho com a Tese deve-se ao uso de ferramentas DevOps relacionadas aos SMA, como o Docker, mas o objetivo principal do estudo não está relacionado à abertura e permitir que os agentes se movam entre os modelos. Além disso, assim como em Perles; Crasnier; Georgé (2018), este estudo compara os resultados diretamente com plataformas de programação de agentes, como JADE, o que é diferente de nossa abordagem. Nessa abordagem, mantém-se o modelo semelhante ao que era antes da abertura, fazendo alterações para inserir os mecanismos que permitem que o modelo se comunique com a plataforma.

Em Pfeifer; Passini; Dorante; Guilherme; Affonso (2021), os autores propõem um Sistema Multiagente para monitorar e gerenciar sistemas distribuídos baseados em contêineres. Seu sistema permite que os projetistas observem e verifiquem a qualidade e o progresso do aplicativo ao longo do tempo, melhorando parâmetros como QoS. Este estudo não está relacionado aos SMAA, mas possui conceitos semelhantes aos que a abordagem desta Tese utiliza. Além disso, incentivam a conexão entre SMA e DevOps, utilizando ferramentas DevOps como Docker. A principal diferença é que esta Tese não utiliza o Docker como ferramenta para monitorar sistemas SMA. Em vez disso, executam-se os modelos sobre um sistema baseado em contêineres no Docker.

### 2.4.3 Considerações Finais

A Tabela 1 sintetiza os estudos similares encontrados. Para cada estudo, é indicado: (i) o foco do trabalho (baseado no critério de divisão dos trabalhos relacionados); (ii) o ano de publicação; (iii) se o estudo utiliza alguma ferramenta DevOps (especialmente se utiliza alguma plataforma para desenvolvimento de soluções baseadas em contêineres); e, (iv) se lida com mais de uma plataforma SMA. O aspecto do DevOps foi considerado importante porque é uma estratégia de implementação das partes dos sistemas que podem executar diferentes linguagens e sistemas operacionais, possibilitando o uso de todos os tipos de ferramentas SMA. O último aspecto está relacionado com a quantidade de plataformas/ferramentas SMA que estão sendo usadas. Este aspecto é essencial para tornar a ferramenta o mais abrangente possível.

Dos 29 estudos analisados, o foco principal de 23 deles é lidar com os problemas organizacionais advindos da abertura dos SMAA. Dos outros 6, 3 são dependentes da linguagem, não permitindo a utilização de ferramentas SMA/SMAA que rodam em um ambiente de desenvolvimento/utilização diferente, como outra ferramenta que utiliza outra linguagem. Por fim, além desta Tese, apenas um estudo trata de mais de uma plataforma SMA. Este estudo é apenas para SMA, não relacionado a SMAA, portanto não lida com o problema de transporte de agentes entre modelos. Além disso, este estudo conecta duas plataformas SMA que rodam na mesma linguagem (Java), não considerando que outras ferramentas possam utilizar outras linguagens.

Este levantamento permite observar que há pesquisa atual no contexto desta Tese e o foco está voltado para a organização e a infraestrutura dos SMAA, dedicando-se às implicações e consequências das trocas de agentes entre os modelos. Entretanto, há pouco esforço em outro aspecto dos SMAA, da Engenharia de *Software* de Agentes, que se preocupa em como operacionalizar esta troca de agentes entre os modelos, por ser um campo mais complexo, ao tratar-se de forma genérica que aproveita os modelos já desenvolvidos como pouca alteração da sua originalidade.

Tabela 1 – Sumarização dos trabalhos relacionados.

| Estudo | Ano | Foco | Ferramenta DevOps | Plataforma SMA |
|---|---|---|---|---|
| (Demazeau; Costa, 1996) | 1996 | Organização | – | Única |
| (Paurobally; Cunningham; Jennings, 2003) | 2003 | Organização | – | Única |
| (Gonzalez-Palacios; Luck, 2006) | 2006 | Organização | – | Única |
| (Singh; Chopra, 2009) | 2009 | Organização | – | Única |
| (Dalpiaz; Chopra; Giorgini; Mylopoulos, 2010) | 2010 | Organização | – | Única |
| (Artikis, 2012) | 2012 | Organização | – | Única |
| (Jamroga; Meski; Szreter, 2013) | 2013 | Organização | – | Única |
| (Hendrickx; Martin, 2017) | 2017 | Organização | – | Única |
| (Ramirez; Fasli, 2017) | 2017 | Engenharia de *Software* | – | Múltiplas |
| (Perles; Crasnier; Georgé, 2018) | 2018 | Engenharia de *Software* | – | Única |
| (Uez, 2018) | 2018 | Engenharia de *Software* | – | Única |
| (Houhamdi; Athamena, 2020) | 2020 | Organização | – | Única |
| (Dähling; Razik; Monti, 2021) | 2021 | Engenharia de *Software* | Docker e Kubernetes | Única |
| (Hattab; Chaari, 2021) | 2021 | Organização | – | Única |
| (Pfeifer; Passini; Dorante; Guilherme; Affonso, 2021) | 2021 | Engenharia de *Software* | Docker | Única |
| (Colla; Galland; Hendrickx, 2021) | 2021 | Organização | – | Única |
| (Hu; Liu; Lan; Zhang, 2021) | 2021 | Organização | – | Única |
| (Galland; Hendrickx, 2022) | 2022 | Organização | – | Única |
| (Haro, 2022) | 2022 | Organização | – | Única |
| (Xue; Tang; Ren; Qian, 2022) | 2022 | Organização | – | Única |
| (Galland; Vizuete; Hendrickx; Panteley; Frasca, 2022) | 2022 | Organização | – | Única |
| (Dashti; Oliva; Seatzu; Gasparri; Franceschelli, 2022) | 2022 | Organização | – | Única |
| (Restrepo; Loría; Sarras; Marzat, 2022) | 2022 | Organização | – | Única |
| (Chebout; Mokhati; Badri, 2022) | 2022 | Organização | – | Única |
| (Nakamura; Hayashi; Inuiguchi, 2023) | 2023 | Organização | – | Única |
| (Jiang; Zhang, 2023) | 2023 | Organização | – | Única |
| (Eck; Soh; Doshi, 2023) | 2023 | Organização | – | Única |
| (Kondylidis; Tiddi; Teije, 2023) | 2023 | Organização | – | Única |
| Nossa Abordagem | 2024 | Engenharia de *Software* | Docker | Múltiplas |

Fonte: autoria própria.

Em conclusão, a abordagem desta Tese contribui para o estado da arte em como o programador usará as plataformas. Por exemplo, alguns estudos propõem novas abordagens que exigem que o programador reconstrua seus modelos conforme as propostas. No *framework* apresentado nesta Tese, os programadores não teriam que mudar substancialmente o seu modelo. Em vez disso, eles adicionarão as estruturas necessárias aos seus modelos para se comunicar com o *framework*, permitindo uma transição mais fácil de um SMA fechado para um SMA aberto. Outro trabalho relacionado já utilizou abordagens baseadas em contêiner/nuvem, relacionadas aos SMA, mostrando que essa abordagem é promissora. Além disso, alguns estudos testaram a comunicação entre agentes NetLogo e Jason.

# 3 ABORDAGEM PROPOSTA

## 3.1 Introdução

Segundo Govoni (1999), um *framework* pode ser definido como uma coleção abstrata de classes, interfaces e padrões dedicados a resolver uma categoria de problemas por meio de uma arquitetura flexível e extensível.

Os autores em Gamma; Helm; Johnson; Vlissides (1995) definem o conceito de *framework* como um conjunto de classes que compõem um projeto reutilizável para uma classe específica de *software*. Nesse sentido, diferentes *frameworks* podem ser desenvolvidos para atender problemas de diferentes domínios. Além disso, os autores também indicam que um *framework* dita a arquitetura da aplicação. Dessa forma, o *framework* define a estrutura, as divisões em classes e objetos (se utilizar este paradigma de programação) e o controle. O *framework* predefine estes parâmetros para o programador poder se concentrar nas especificidades da sua aplicação.

Solms (2012) ressalta que não existe consenso no que é exatamente arquitetura de *software* e onde estão os limites do *design* da arquitetura de *software* e o *design* de aplicação. Segundo o autor, uma das definições trata arquitetura de *software* como a infraestrutura de *software* onde os componentes de aplicação que proveem funcionalidades ao usuário podem ser especificados, implementados e executados. Além disso, o autor também aponta que a implementação de uma arquitetura de referência é chamada de *framework*, fornecendo uma implementação concreta dos elementos arquiteturais e estratégias especificadas pela arquitetura de referência.

Neste contexto, esta Tese apresenta detalhes tanto dos aspectos conceituais quanto tecnológicos da abordagem proposta. Dessa forma, o estudo engloba tanto elementos de arquitetura quanto de *framework*. Portanto, o estudo pode ser classificado tanto como arquitetura quanto por *framework*, dependendo do referencial teórico utilizado. Além disso, existem sinônimos utilizados para estas classificações, como plataforma e abordagem. Visando unificar e simplificar a terminologia, a abordagem proposta nesta Tese será majoritariamente referenciada como arquitetura deste ponto em diante.

A arquitetura proposta nesta Tese visa desenvolver um ambiente que facilite o desenvolvimento de Sistemas Multiagente Abertos. Os Sistemas Multiagente Abertos são compostos de agentes que podem migrar entre diferentes modelos. Cada modelo contém características e perspectivas de análises distintas. Por exemplo, as diferenças entre os modelos podem estar caracterizadas: i) nos grupos de programadores; ii) nas ferramentas para SMA (exemplo: NetLogo, Jason, JADE); iii) nas versões de ferramentas para SMA (exemplo: modelo X desenvolvido no NetLogo 6.1 e modelo Y no NetLogo 2.1); iv) nas ferramentas que rodam em diferentes Sistemas Operacionais (exemplo: Windows e Linux); e, v) nas ferramentas que utilizam diferentes linguagens de programação (exemplo: código Python que utiliza o NetLogo e código Java que utiliza o Jason).

Como exemplos de ferramentas de SMA distintas, é possível citar duas ferramentas amplamente citadas na literatura, o NetLogo e o JaCaMo. O NetLogo (Wilensky, 1999) é uma ferramenta de desenvolvimento de Sistemas Multiagente que conta com vários modelos de exemplos de aplicação na documentação, em áreas como ciências naturais, sociais, economia, dentre outros. O JaCaMo (Boissier; Bordini; Hübner; Ricci; Santi, 2013) é um ambiente de programação multiagente que combina três plataformas: (i) Jason, para o desenvolvimento dos agentes (Bordini; Hübner; Wooldridge, 2007); (ii) CArtAgO, para o desenvolvimento dos ambientes e artefatos (Ricci; Piunti; Viroli, 2011); e, (iii) Moise+, para o desenvolvimento das estruturas organizacionais (Hubner; Sichman; Boissier, 2007).

Para suportar os cenários heterogêneos de programação, a arquitetura proposta utiliza como base o uso de contêineres. Contêineres permitem empacotar a aplicação, suas bibliotecas e demais dependências, oferecendo ambientes isolados para executar os serviços de *software* (Google, 2024). Dentre as ferramentas encontradas para o desenvolvimento de contêineres, o Docker (Turnbull, 2014) foi escolhido pela sua relevância, embora outras ferramentas, como Kubernetes, pudessem ser utilizadas. Dessa forma, a partir deste ponto, o Docker será mencionado por ser a alternativa tecnológica encontrada para validar o uso de contêineres na arquitetura.

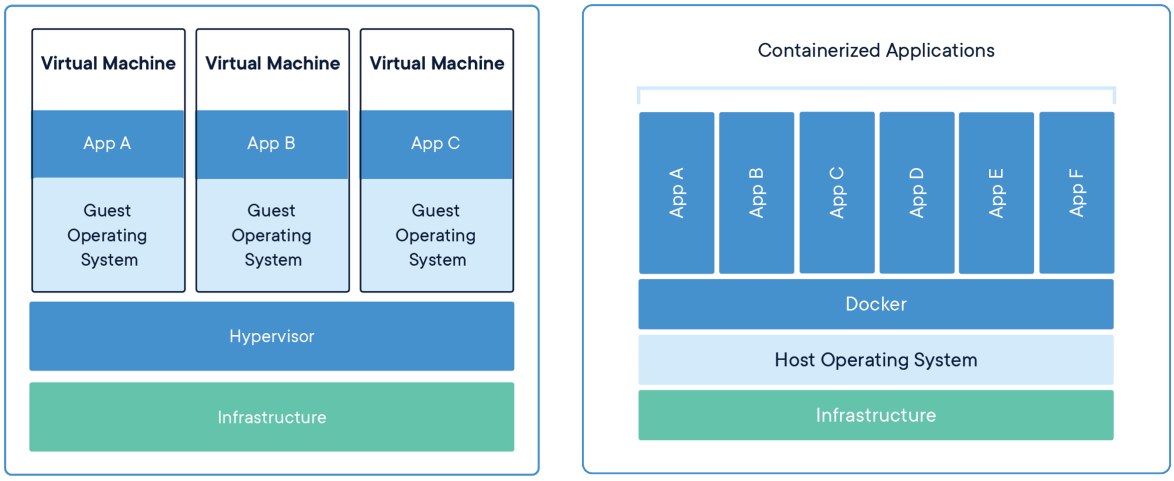
Docker é uma ferramenta de código aberto que permite a automatização do desenvolvimento de aplicações em contêineres (do inglês, *container*). O Docker adiciona um mecanismo de implementação de aplicações e dependências sobre um ambiente de execução de contêineres. Esse desenho cria um ambiente leve e rápido onde é possível executar códigos, infraestruturas e testes em um fluxo único, do desenvolvimento à produção.

Embora o conceito seja similar, existem diferenças entre as máquinas virtuais (VM, do inglês *Virtual Machine*) e os contêineres. Máquinas virtuais são uma abstração do *hardware* físico que transforma um servidor em vários servidores. O hipervisor (do inglês, *hypervisor*), *software* que cria e executa a máquina virtual, permite que vá-

rias máquinas sejam executadas em um único *host*. Cada máquina inclui uma cópia completa de um OS (sistema operacional, do inglês *Operating system*), o aplicativo, binários e bibliotecas necessários. As máquinas também podem ser lentas para inicializar (Docker, 2021a).

Em contraste, os contêineres são uma abstração na camada de aplicação que une o código e suas dependências. Múltiplos contêineres podem ser executados na mesma máquina compartilhando o *kernel* do sistema operacional, sendo cada um executando como processos isolados no espaço do usuário. As imagens dos contêineres são mais otimizadas em termos de espaço do que as máquinas virtuais (Docker, 2021a). As Figuras 3(a) e 3(b) apresentam, respectivamente, a estrutura de uma aplicação sendo executada em uma máquina virtual e um contêiner do Docker.

Figura 3 – Estruturas de (a) uma máquina virtual e de (b) um contêiner.




(a) (b)

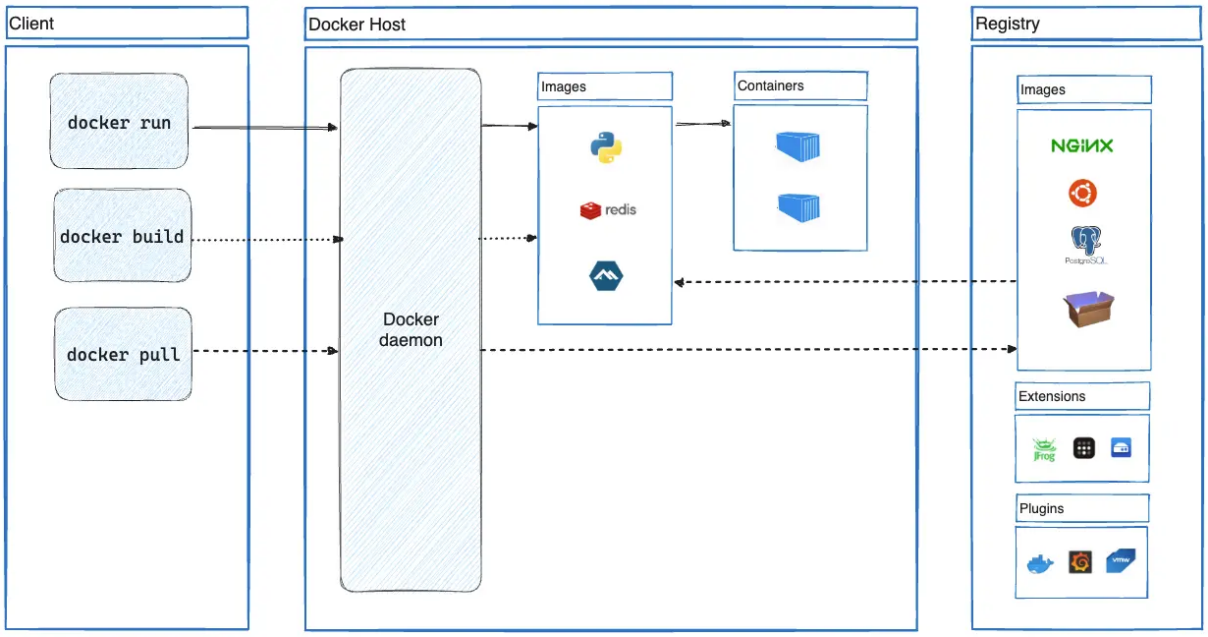
Fonte: adaptado de Docker (2021a).

A estrutura permite que os códigos sejam desenvolvidos e executados dentro de contêineres que contêm a mesma estrutura de operação, permitindo que seja possível evitar problemas como programas que rodam no desenvolvimento, mas em produção apresentam problemas.

A Figura 4 apresenta uma visão geral da arquitetura do Docker. O Docker utiliza arquitetura cliente-servidor. O *Docker Client* se comunica com o *Docker Daemon*, cujo objetivo é construir, executar e distribuir os contêineres Docker. O *Docker Host* e o *Docker Client* podem ser executados na mesma máquina, mas também é possível conectar a um *Docker Daemon* remoto. O *Client* e o *Daemon* se comunicam usando uma API REST (REST, do inglês *Representational State Transfer*), por meio de soquetes UNIX ou alguma interface de rede. O *Docker Registry* armazena as imagens do Docker. Dentre os possíveis registros, o *Docker Hub* é o registro público (qual-

quer pessoa pode utilizar) onde o Docker busca imagens por padrão, também sendo possível utilizar um registro particular. Quando são utilizados comandos para montar imagens e contêineres, o *Docker Daemon* verifica se os dados necessários estão presentes no *host* atual. Se sim, as imagens e contêineres são montados diretamente com os arquivos locais. Se não, os dados são buscados no registro apontado pelo cliente (*Docker Hub* por padrão) (Docker, 2021b).

Figura 4 – Visão geral da arquitetura do Docker.




Fonte: adaptado de Docker (2021b).

No Docker, cada imagem é gerada com base em um arquivo de descrição, chamado `Dockerfile`. Esse arquivo contém todas as informações das estruturas necessárias no contêiner, como o sistema operacional, linguagens de programação, versão de pacotes, código a ser executado, dentre outras. Além disso, o Docker conta com um banco de imagens, o *Docker Hub Container Image Library*, que contém as imagens utilizadas com frequência pelos desenvolvedores, como Apache, PHP (do inglês *Hypertext Preprocessor*), Python e outros.

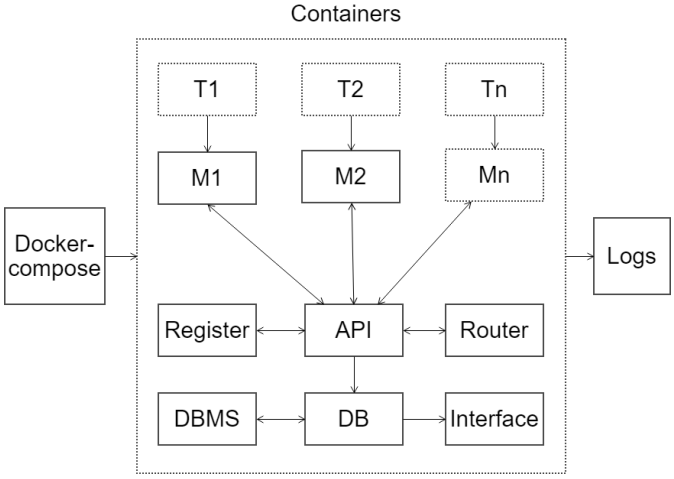
A utilização do Docker como base do sistema permite com que os contêineres responsáveis pela execução de cada bloco essencial da estrutura da arquitetura sejam facilmente substituídos, tornando a arquitetura mais modularizada e adaptável a novos cenários não previstos na concepção. Além disso, a conteinerização também permite que seja simplificada a adição de novos módulos ao sistema, tornando-o mais robusto (Turnbull, 2014; Docker, 2021a).

O uso do Docker também auxilia em questões de segurança, ao permitir a criação de redes virtuais separadas para que cada contêiner tenha uma visão parcial do sistema, limitando acesso a código sensível (Turnbull, 2014; Perrone; Romano, 2017; Docker, 2021a). Por fim, várias plataformas online como a da Amazon (AWS Docker) permitem com que toda a estrutura que está sendo desenvolvida com base no Docker seja executada na nuvem, possibilitando uma alta portabilidade da aplicação, expandindo o universo de aplicações da arquitetura.

## 3.2 Aspectos Conceituais

A Figura 5 apresenta uma descrição geral da arquitetura proposta em formato de diagrama de blocos.

Figura 5 – Organização da arquitetura proposta.




Fonte: autoria própria.

No bloco **Docker-compose**, tem-se o módulo para a geração de imagens e serviços, definidos a partir do arquivo *docker-compose.yaml* e no *Dockerfile* de cada serviço. Cada *Dockerfile* contém a sequência de instruções necessárias para montar a imagem com todos os requisitos que cada serviço necessita. Então, com base nas imagens, o **Docker-compose** usa os parâmetros de cada serviço (nome do contêiner, portas expostas, rede, `volumes`, comando/arquivo a ser executado) e os constrói. Finalmente, quando todos os serviços estão definidos no **Docker-compose**, estes podem ser executados usando apenas um comando.

Os blocos $\mathbf{M}_1$, $\mathbf{M}_2$, ..., $\mathbf{M}_n$ representam os contêineres responsáveis por executar os modelos, o modelo $i$ no bloco $\mathbf{M}_i$. No arquivo de configuração principal, pode-se definir um parâmetro chamado `auto-run`. Quando este parâmetro está definido como *True*, os modelos são executados automaticamente quando o contêiner é montado. Se for *False*, deve-se iniciar a execução dos modelos a partir de um *trigger* implementado

por um contêiner ($T_1$, $T_2$, ... , $T_n$), que irá iniciar a execução do modelo por meio de um *socket*. É necessário implementar as conexões no modelo para se comunicar com a arquitetura, como gatilhos e funções de como os agentes entram/saem do modelo. Em função dos cenários de simulação desenvolvidos nesta Tese (Capítulo 4), existem contêineres com as funções que oferecem suporte aos modelos NetLogo (contêiner Java, por meio de código intermediário Python) e JaCaMo (contêiner Java via Gradle).

$\mathbf{T}_1$, $\mathbf{T}_2$, ..., $\mathbf{T}_n$ são contêineres de gatilhos de execução, responsáveis por enviar uma mensagem aos contêineres dos modelos para iniciarem a sua execução, via *socket*. Esses contêineres são utilizados para implementar uma lógica mais robusta de disparo da execução dos modelos. Por exemplo, eles podem executar códigos que esperam por um determinado valor provindo da leitura de um sensor, executar um modelo de retreinamento de agentes, dentre outros.

O bloco **Application Programming Interface** (API) é o contêiner responsável por gerenciar o acesso ao banco de dados. Este contêiner atua como um intermediário quando qualquer parte do sistema precisa escrever, ler, atualizar ou excluir informações do banco de dados. Atualmente, a API é implementada em Python, usando o *framework* Flask (Grinberg, 2018). Todos os métodos (um método também pode ser chamado como de rota ou *endpoint*) são acessíveis via HTTP (do inglês *Hypertext Transfer Protocol*), usando o formato JSON (do inglês *JavaScript Object Notation*) em todas as mensagens, como padrão para APIs.

O bloco **Database** (DB) representa o contêiner responsável pelo banco de dados no qual, para cada modelo, todas as informações de entrada e saída de agentes são armazenadas para serem acessadas por outros contêineres que possam necessitar dessas informações. As operações realizadas no DB podem ser feitas através da API, ou seja, os contêineres de modelo, e alguns outros, não precisam ter acesso direto ao banco de dados.

O bloco **Sistema Gerenciador de Banco de Dados** (*Database Management System* – DBMS) representa o contêiner que facilita o acesso direto ao BD, fornecendo uma interface *web* (por padrão exposta à máquina *host*) que pode importar/exportar conteúdo/arquivos SQL (do inglês *Structured Query Language*) e visualizar as informações em tempo real ou criar *logs* (registros) e é escolhido conforme o DB, devido à necessidade destes serem compatíveis.

O bloco **Register** é um contêiner responsável por gerenciar a criação de todos os agentes na plataforma. Todo agente deve possuir uma identificação para ser utilizada pela arquitetura, portanto as primeiras requisições de inserção de novos agentes vindas de qualquer contêiner de modelo necessitam de uma identificação única. Assim, esse bloco é responsável por gerar uma identificação única para cada agente e, então, encaminhá-los de volta aos contêineres de modelo.

O bloco **Router** é o contêiner responsável por receber todos os agentes que saíram de um determinado contêiner de modelo, analisando e julgando para qual contêiner de modelo enviar o agente. Esta etapa especifica os protocolos de entrada e saída do agente de diferentes contêineres/modelos. Este bloco é parte essencial da arquitetura, pois os modelos delegam ao **Router** a tarefa de distribuição dos agentes entre os modelos. Quando o ambiente de simulação compartilha parcialmente informações sobre o mundo, o **Router** pode lidar com diversos problemas relacionados à ausência dessas informações. O julgamento pode ocorrer de diferentes maneiras, como: analisar os agentes mais promissores, ou executar códigos de aprendizado de máquina e executar um novo modelo que retreine os agentes. No cenário de simulação, existem duas opções de julgamento que o **Router** pode fazer da lista de agentes a serem processados: i) escolher aleatoriamente o agente e o modelo alvo (modo aleatório); e, ii) processar os agentes em uma única fila, considerando todos os modelos, e definir aleatoriamente o modelo de destino (modo sequencial geral).

O bloco **Interface** é um contêiner que recebe as informações de movimentação dos agentes através do sistema e gera relatórios expostos e formatados para visualização das métricas de execução da arquitetura. Na implementação atual, este contêiner possui um Apache Webserver, que executa código PHP que acessa todas as informações dos agentes que já passaram pelo **Router** e gera um relatório com todos os agentes, atributos e o caminho percorrido por eles. A forma geral do relatório é por meio de um código *front-end* que utiliza HTML (do inglês *HyperText Markup Language*), CSS (do inglês, *Cascading Style Sheets*) e JS (do inglês, *JavaScript*), o qual a máquina *host* pode acessar expondo a porta que o apache executa na porta 80 (por padrão). Cada tupla mostrada na interface contém todos os atributos essenciais para cada interação do agente até o momento. É possível observar informações sobre a simulação como, por exemplo, qual agente apresentou o caminho mais longo percorrido entre os modelos até o momento. Existe uma opção de pesquisa para filtrar os resultados por qualquer uma das colunas. Além disso, é possível filtrar os resultados por agente, por modelo e se a tupla foi processada pelo modelo/roteador ou não. Por fim, ainda é possível observar informações específicas de algumas ferramentas de SMA. Por exemplo, no JaCaMo, como cada agente possui um arquivo ASL (*AgentSpeak Language*), é possível ter acesso a esse arquivo e ver as informações relevantes sobre aquele agente.

Por fim, o bloco **Logs** não é um contêiner e sim um módulo responsável por gerar *logs* das saídas de cada contêiner do sistema. Pode-se obter vários tipos de *logs* da simulação como, por exemplo, *logs* regulares do Docker (via comando docker log), tuplas de banco de dados via exportação SQL e gerar arquivos *.txt* (Texto Plano) de *logs* da execução do modelo/contêiner (usando operadores de terminal ao executar o contêiner no *docker-compose*). Além disso, algumas estruturas têm tipos de *logs*

específicos, como o gerador de *log* padrão do JaCaMo (baseado na API de *log* do Java). Esses *logs* podem auxiliar na depuração de tarefas ou na geração de dados para análise de execução. O Apêndice C mostra exemplos dos *logs* que podem ser obtidos.

## 3.3 Aspectos Tecnológicos

### 3.3.1 Parametrização dos *Contêineres*, *Dockerfiles* e *Docker-compose*

Com base na descrição dos conceitos da arquitetura, os principais módulos foram desenvolvidos para a arquitetura atender os cenários de simulação da forma mais abrangente possível. Cada módulo possui parâmetros que podem ser ajustados para se adaptar ao modelo a ser testado.

O primeiro e principal arquivo de configuração é o *docker-compose*. O Docker Compose é uma ferramenta para definir, montar e executar aplicações Docker que utilizam múltiplos contêineres de forma facilitada. Através da criação de um arquivo *docker-compose*, com extensão *.yaml*, os múltiplos contêineres que compõem uma aplicação podem ser montados, por meio de um único comando. No arquivo, pode-se definir diversas informações, como: i) a ordem na qual os contêineres são montados; ii) qual imagem deve ser utilizada; iii) os nomes dos contêineres; iv) políticas de falhas; v) parâmetros de rede; e, vi) parâmetros específicos da aplicação.

Para montar as imagens e depois os contêineres, o *docker-compose* pode utilizar as descrições tanto diretamente do banco online de imagens (Docker Hub), quanto customizar imagens por arquivos do tipo *Dockerfile*. Vale destacar que os arquivos *Dockerfile* e *docker-compose* são diferentes e complementares, um não exclui o outro, por terem funções diferentes. Os arquivos *Dockerfile* descrevem a base da imagem do contêiner, contendo todas as definições do que será executado. Já os arquivos *docker-compose* funcionam como um orquestrador, sendo possível definir os serviços, a ordem de montagem dos serviços, parâmetro de tempo de execução, dentre outros.

Na implementação atual da arquitetura, são utilizados arquivos *Dockerfile* e *docker-compose* diferentes para *hosts* com arquiteturas x86/x64 e arm64. Os arquivos para arquiteturas arm64 tem a indicação no nome do arquivo, como, por exemplo, em *docker-compose-local-arm64.yaml*. Essa separação tem por objetivo ampliar a abrangência de utilização da ferramenta, pois os serviços de hospedagem na nuvem de plataformas do Docker contam com máquinas de diferentes arquiteturas.

Um exemplo de arquivo *Dockerfile* utilizado para contêineres do NetLogo é mostrado na Figura 6. Nesta Figura, é possível observar como é feita a montagem da imagem. Inicialmente, faz-se o download diretamente do site do desenvolvedor da versão do NetLogo definida na variável *NETLOGO_VERSION*. Após baixar, o arquivo é extraído e montado. Além da base da imagem do NetLogo, que utiliza o Java, também

é instalado o Python e suas dependências, para ser possível utilizar código Python (extensão `py`) no NetLogo. Os arquivos *Dockerfile*, embora simples, possuem um alto poder de isolamento de etapas. É possível montar várias partes da imagem e levar para a etapa posterior somente o que é necessário. Este processo é conhecido como *Multi-stage build*. Os demais arquivos *Dockerfile* utilizados são apresentados no Apêndice A.

Figura 6 – Exemplo de arquivo *Dockerfile* para contêineres do NetLogo.

Dockerfile_NETLOGO

```
#--------------Netlogo Instalation--------------
FROM openjdk:8-jdk
LABEL maintainer="Allen Lee <allen.lee@asu.edu>"

ARG NETLOGO_HOME=/opt/netlogo
ARG NETLOGO_VERSION=6.0.4

ENV LC_ALL=C.UTF-8 \
    LANG=C.UTF-8 \
    NETLOGO_TARBALL=NetLogo-$NETLOGO_VERSION-64.tgz

ENV NETLOGO_URL=https://ccl.northwestern.edu/netlogo/$NETLOGO_VERSION/$NETLOGO_TARBALL

WORKDIR /opt
RUN wget $NETLOGO_URL && tar xzf $NETLOGO_TARBALL && ln -sf "NetLogo $NETLOGO_VERSION" netlogo \
    && rm -f $NETLOGO_TARBALL

RUN apt-get update
RUN apt-get install -y --no-install-recommends python3
RUN apt-get update
RUN apt-get install -y python3-pip
RUN pip3 install mysql-connector-python
RUN pip3 install requests
```

Fonte: autoria própria.

No arquivo *docker-compose*, são indicados todos os parâmetros que os contêineres devem receber para garantir o seu funcionamento. Dentre os mais comuns, destacam-se:

`Version`**:** Trata da versão da ferramenta *docker-compose*. Este comando aparece antes da descrição dos serviços, pois diferentes versões podem conter diferentes sintaxes. Exemplo: `version: ‘‘3.8’’`;

`Services`**:** Início do bloco que define os serviços, pois dentro dessa organização, cada contêiner é representado como um serviço;

`Container_Name`**:** Define o nome do contêiner. Esse parâmetro é opcional, se não for definido, terá um nome adaptado com base na *stack* do *docker-compose*. Exemplo: `container_name: nome_do_contêiner`;

`Ports`: Através deste parâmetro, é possível expor portas do contêiner em questão para a máquina *host*. É possível definir múltiplas portas para serem expostas, através do formato `porta-host:porta-contêiner`. Exemplo: ‘‘`8080:80`’’;

`Volumes`: `Volumes` são mecanismos de persistência de dados, através do compartilhamento de um ou mais diretórios entre a máquina *host* e o contêiner. Diferente das VMs, quando a execução do contêiner é encerrada, os dados não são persistidos. Para manter dados, é necessário usar `volumes`. Na definição, segue o formato `diretório_host:diretório_contêiner`. Exemplo: `./directory:/var/www/html`;

`Depends On`: O parâmetro opcional `depends_on` define que, para esse contêiner ser montado, deve-se aguardar a montagem de outro contêiner. Através dessas definições, o *docker-compose* consegue definir em qual ordem os contêineres devem ser executados. Se esse parâmetro não for indicado em algum serviço, os serviços são montados na mesma ordem descrita no arquivo *.yaml*. Exemplo: ‘‘`myContainer`’’;

`Environment`: É possível definir variáveis de ambiente ao sistema. Exemplo: `PYTHONUNBUFFERED=1`.

Parte destes parâmetros utilizados nos arquivos *docker-compose* podem ser observados na Figura 7. Esta Figura aproveita o *Dockerfile* utilizado para montar contêineres do NetLogo, mostrando como é montado um serviço em um arquivo *docker-compose*.

Figura 7 – Exemplo de arquivo *Docker-compose* que monta um contêiner do NetLogo.

docker-compose-local.yaml

```yaml
model_1:
  container_name: model_1
  build:
    context: .
    dockerfile: Dockerfile_NETLOGO
  tty: true           # Container alive
  command: "/bin/sh -c 'python3 /shared_volume/python_execute_netlogo.py >> /shared_volume/netlogo_output/m1_output.txt'"
  environment:
  - PYTHONUNBUFFERED=1
  - my_hostname=model_1
  - model_file=/shared_volume/sugarscape_m1.nlogo
  - experiment=experiment1
  - auto_run=True
  - host=api
  volumes:
    - ./shared_volume:/shared_volume
  networks:
    - my_network
  depends_on:
    - "router"
    - "api"
    - "clear_files"
```

Fonte: autoria própria.

Na Figura 7, é mostrado parte do arquivo que contém o serviço do NetLogo, montando um contêiner de nome `model_1`, que utiliza como base da imagem o arquivo `Dockerfile_NETLOGO`, além de apresentar o comando que o contêiner executa (chama uma função Python que dispara o início do modelo do NetLogo e salva a saída em um arquivo *.txt*), além de mostrar as variáveis de ambiente, `volumes` compartilhados entre o contêiner e o *host*, a rede privada, e quais outros serviços devem ser montados antes deste.

Além dos parâmetros padrões utilizados pelo *docker-compose*, na implementação da arquitetura, foram adicionados parâmetros específicos para cada módulo, para auxiliar o seu funcionamento. Dentre os parâmetros e detalhes específicos (excluindo os citados anteriormente) de cada contêiner, destacam-se:

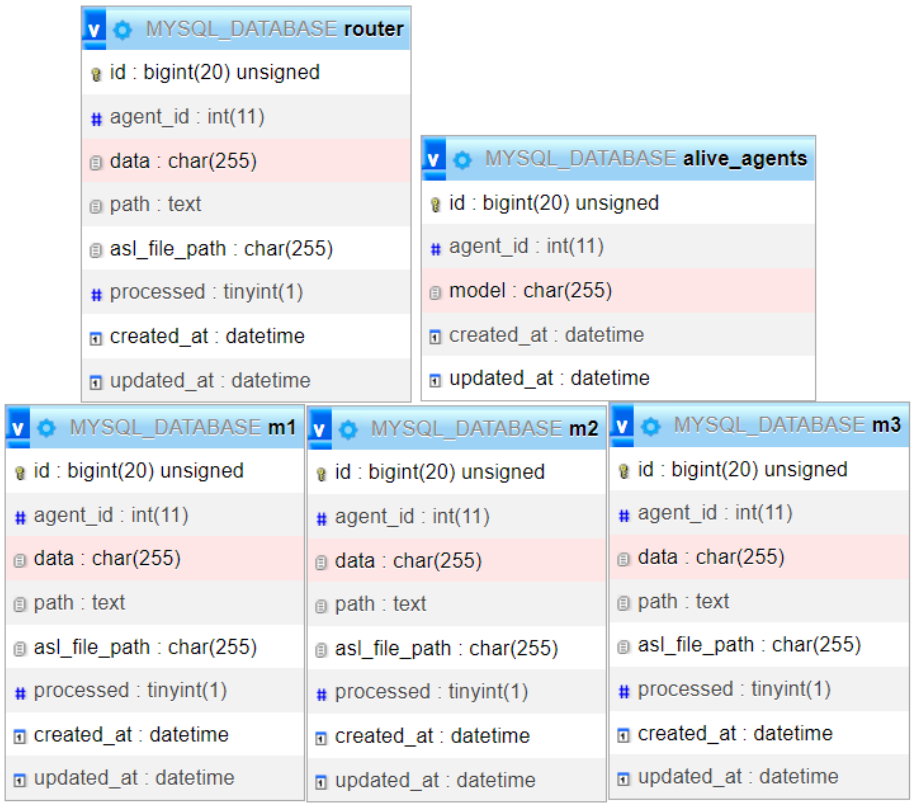
**API:** caso a API precise ser acessada pelo *host*, ou no caso de cenários de uso na nuvem (apresentados no próximo Capítulo), é necessário expor a porta que a API está escutando. Na configuração padrão, a API escuta a porta 5000. Além disso, utiliza-se um `Volume` para enviar o código da API para o contêiner. O código executado pela API segue os padrões de uso de APIs do tipo REST, através da comunicação de informações no formato JSON, utilizando HTTP *status codes*, por exemplo;

**Router:** no contêiner do **Router**, utiliza-se um `volume` para enviar o código do **Router**, vindo da máquina *host*, para o contêiner.

**DB:** no contêiner do DB é possível definir o login/senha para acessar o banco através das variáveis de ambiente. No caso de o projetista alterar o login/senha padrão, é necessário refletir essa alteração em outras partes da arquitetura que precisam dessa informação como, por exemplo, na API e no Sistema Gerenciador de Banco de Dados (SGBD). Além disso, pode-se utilizar `volumes` para cenários nos quais é necessário já inicializar o banco com algum código SQL. Neste caso, o arquivo SQL presente no `volume` pode ser enviado para a inicialização do contêiner. Para acessar o banco de fora da rede privada do Docker, pode-se expor a porta para o *host*. Por padrão, a porta 3306 do contêiner é redirecionada para a 9906 do *host*. Por fim, como outros contêineres dependem que o DB esteja não só preparado para receber conexões (garantido pelo `depends_on` do Docker), como também que o banco de dados e suas tabelas já estejam montadas, é possível criar um *healthcheck*, para garantir que o serviço esteja pronto para uso.

Na implementação atual, o banco de dados utilizado é o MySQL. A Figura 8 apresenta informações sobre a estrutura do banco utilizado no cenário de teste. A Figura também permite observar os parâmetros das tabelas de modelo, do **Router** e da tabela `alive_agents`, que guarda ao fim da execução dos modelos, quais agentes permaneceram vivos (ativos) ao fim da execução dos modelos NetLogo;

Figura 8 – Informações sobre as tabelas e seus respectivos campos utilizados nos cenários de teste da arquitetura.

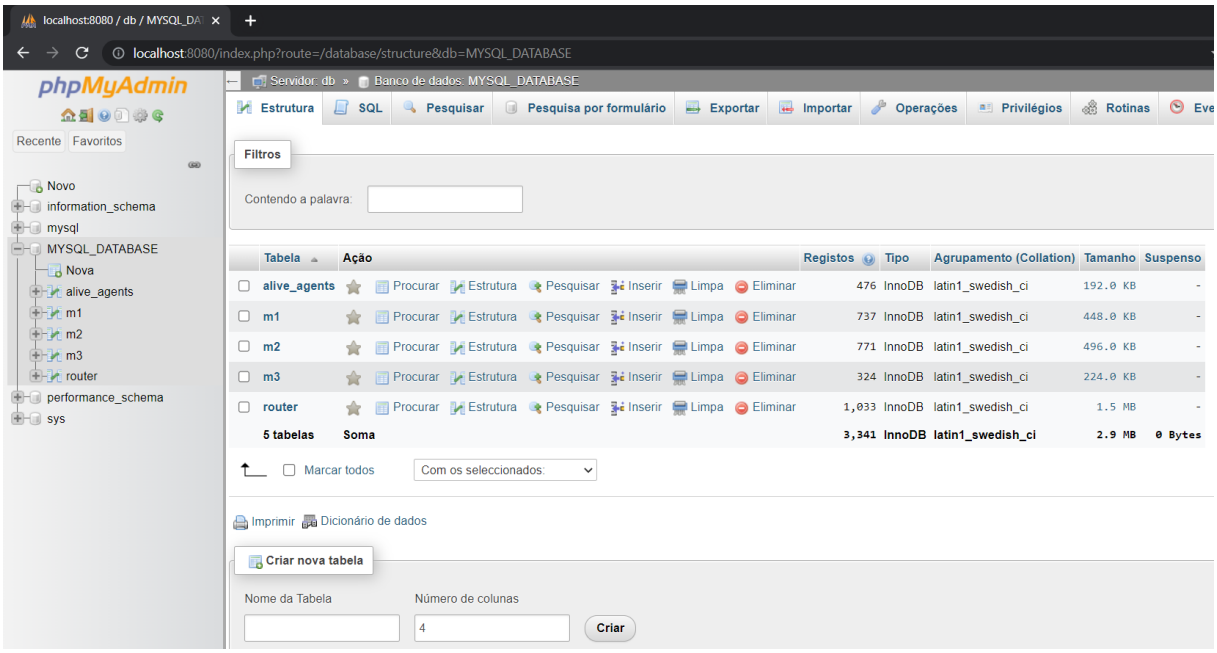



Fonte: autoria própria.

**DBMS**: Esse contêiner contém o SGBD e, para que este possa ser acessado pelo *host*, é necessário expor a porta, que no cenário padrão é da porta 80 do contêiner para a 8080 do *host*. Na implementação atual, o SGBD utilizado para o MySQL é o PHPMyAdmin.

A Figura 9 ilustra a interface do SGBD, contendo as tabelas utilizadas na simulação e a quantidade de registros. Por meio do SGBD, é possível ler, inserir, editar ou excluir qualquer passagem de agente pela simulação;

Figura 9 – Exemplo da interface do SGBD PHPMyAdmin.

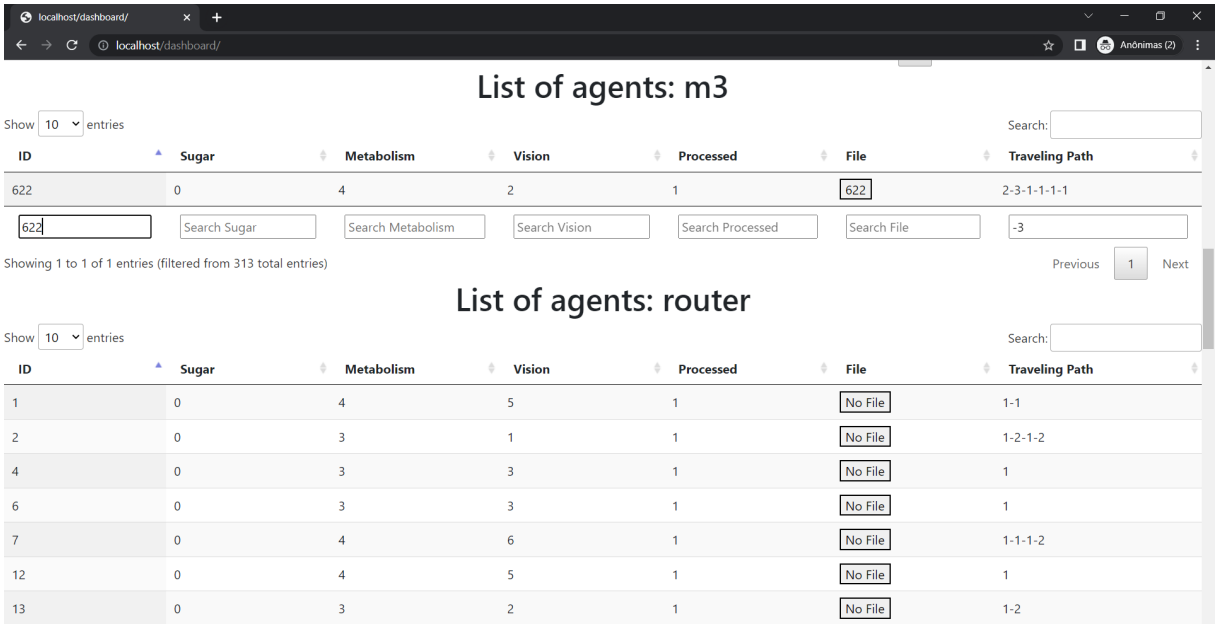



Fonte: autoria própria.

`Clear Files`**:** Este contêiner trata de uma funcionalidade para *debug* (depuração), definido especificamente para os casos de simulação, não caracterizando sua participação na arquitetura proposta. Este é um contêiner com a função de executar um código Python que limpa certos arquivos de *logs*, para que informações não sejam acumuladas de uma simulação para outra, em especial os *logs* do NetLogo. Estes *logs* ficam no `Volume` compartilhado entre os contêineres e o *host*. Além disso, também são removidos os arquivos ASL dos agentes gerados durante a simulação anterior, para não interferir em simulações posteriores;

**Interface:** No contêiner da interface, é necessário indicar um `volume`, para que o código da página *web* que está no *host* possa ser processado para dentro do contêiner. Para a página poder ser acessada depois do processamento no *host*, é necessário também expor a porta. Por padrão, a exposição é da porta 80 para a 80 (`host:contêiner`). Por fim, este contêiner também utiliza a variável `host` para identificar o caminho de comunicação com a API, conforme descrito anteriormente. A Figura 10 apresenta parte da visão geral da interface *web* atual.

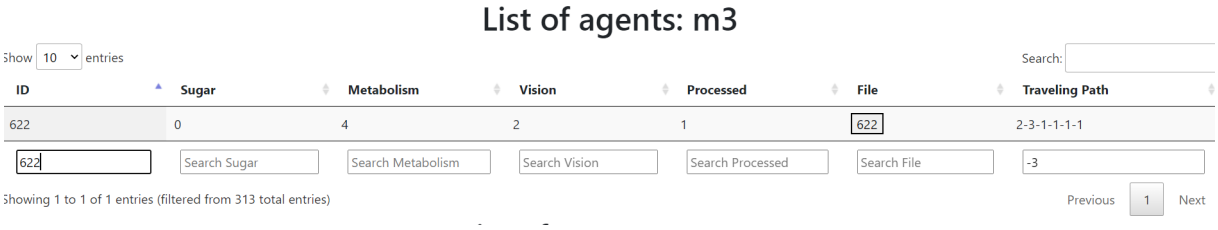
Figura 10 – Exemplo da interface *web* geral, mostrando a passagem dos agentes pela arquitetura.




Fonte: autoria própria.

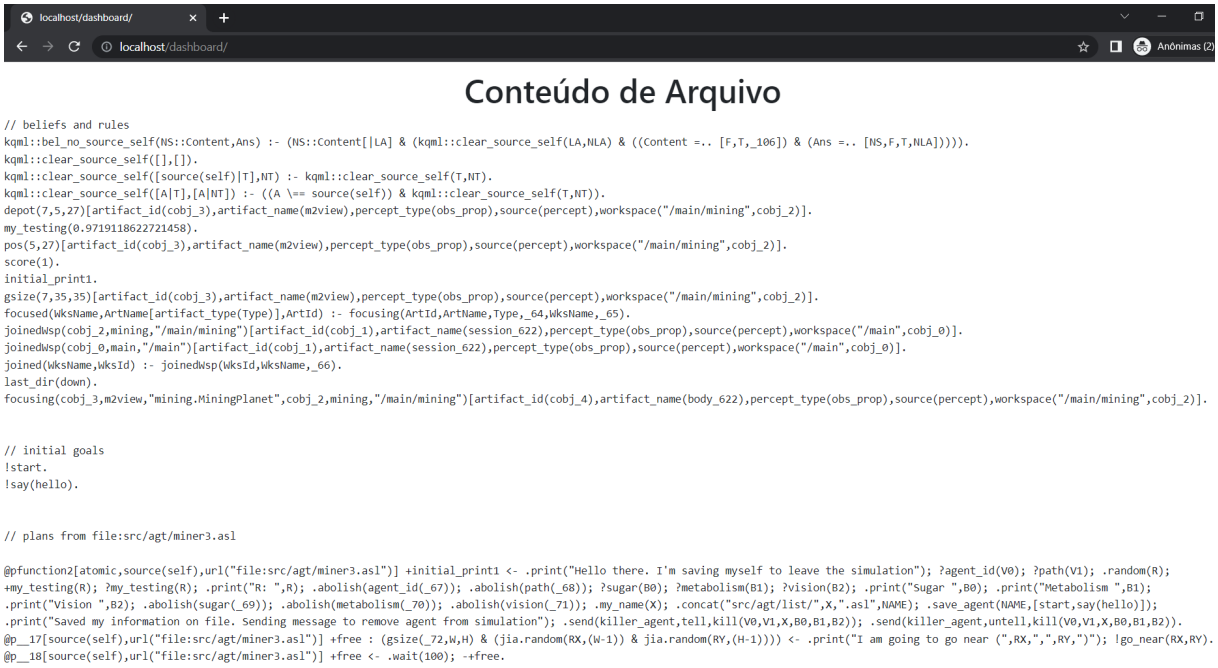
A interface *web* conta com os atributos de todos os agentes, as informações separadas por cada modelo, os agentes no **Router**, informações do arquivo ASL de cada agente, o maior caminho entre modelos percorrido pelos agentes, dentre outros. Nessa visão, é possível observar as informações de cada passagem de agente pelo sistema, observando os atributos referentes ao NetLogo nas colunas, além da indicação do arquivo do Jason para o caso de o agente já ter entrado alguma vez na simulação do JaCaMo e, por fim, qual caminho percorreu entre os modelos. As representações dos agentes são separadas entre os modelos (m1, m2 e m3) e o **Router**. A visão de uma tupla de agente específico e de parte da representação das informações referentes ao JaCaMo (obtidas ao clicar no botão referente ao id do agente) são mostrados nas Figuras 11 e 12, respectivamente;

Figura 11 – Exemplo da representação de uma tupla de agente na interface.




Fonte: autoria própria.

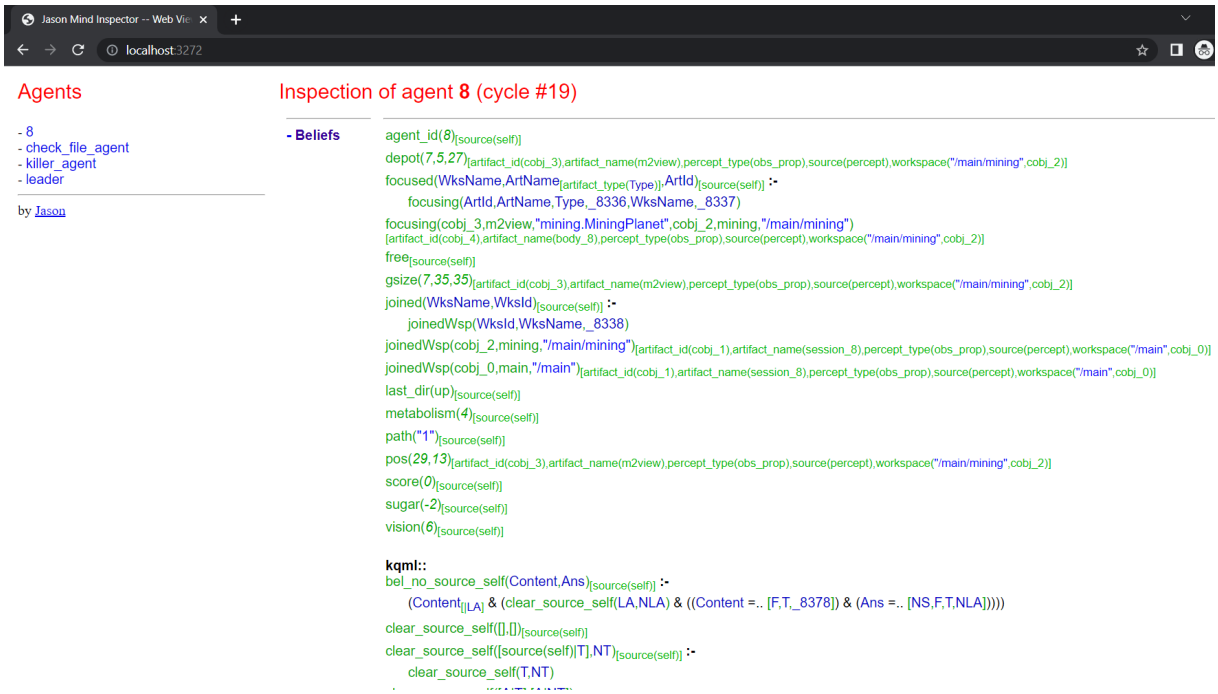
Figura 12 – Exemplo da representação dos dados que constam no arquivo ASL de um agente específico.




Fonte: autoria própria.

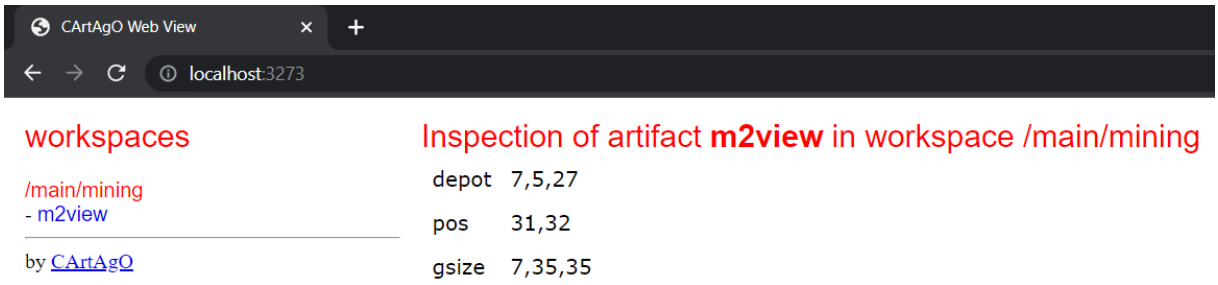
**Contêiner JaCaMo:** no contêiner responsável por executar código JaCaMo, para ser possível acessar de fora do contêiner a representação *web* da execução dos agentes no Jason e dos artefatos do CArtAgO, as portas 3271, 3272 e 3273 são expostas, por meio de um projeto montado via Gradle. Ilustrações da interface mostrada pelo Jason e pelo CArtAgO são mostradas nas Figuras 13 e 14, respectivamente.

Figura 13 – Exemplo da interface *web* do Jason mostrando informações de um agente.




Fonte: autoria própria.

Figura 14 – Exemplo da interface *web* do CArtAgO.




Fonte: autoria própria.

Além disso, também é utilizada uma variável de ambiente `host`, utilizada para indicar o caminho de comunicação com a API. Caso a execução seja local, `host` recebe o nome do contêiner da API (por padrão “api”). No caso de a API ser executada fora da rede padrão, deve-se indicar o IP (do inglês *Internet Protocol*) da máquina neste parâmetro. Por fim, utiliza-se `Volumes` para expor o projeto, que está na máquina local, para que o contêiner possa compilar e executar;

**Contêineres NetLogo:** Para os contêineres de modelo que executam código NetLogo, existem alguns parâmetros para serem definidos. Inicialmente, o parâmetro `auto-run` deve ser informado, indicando se o contêiner deve executar automaticamente o modelo ao ser montado ou não. Por padrão, esse parâmetro tem valor *True*.

Além disso, o parâmetro `command` deve ser definido para apontar o arquivo Python que executa o NetLogo. Ainda neste parâmetro, é possível indicar que a saída dessa execução vá para um arquivo *.txt*, permitindo que seja possível guardar um *log* extra de toda a execução do modelo. Um `volume` deve ser apontado entre *host* e contêiner para enviar o modelo a ser executado.

Variáveis de ambiente também são utilizadas. A variável `model_file` indica qual o arquivo *.nlogo* deve ser executado. Como o NetLogo é utilizado no modo *headless* (execução da simulação sem GUI (do inglês *Graphical User Interface*), por meio de experimentos definidos no *BehaviorSpace* do NetLogo), é necessário indicar qual experimento deve ser executado, através do parâmetro `experiment`. Por fim, este contêiner também utiliza a variável **host** para identificar o caminho de comunicação com a API, conforme descrito anteriormente.

Conforme apresentado, `Volumes` auxiliam na persistência de dados. No caso da arquitetura, parte dos arquivos compartilhados são disponibilizados por meio de um `Volume` compartilhado entre o *host* e os contêineres. A Tabela 2 apresenta a descrição dos principais diretórios e arquivos do `Volume`.

Tabela 2 – Principais diretórios e arquivos presentes no `Volume` compartilhado entre o *host* e os contêineres.

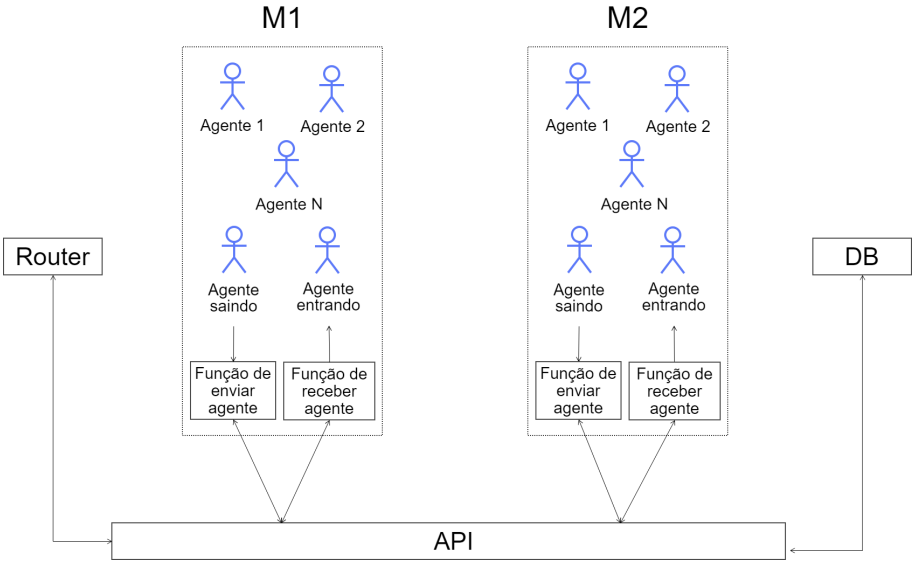
| Arquivo ou Diretório | Descrição |
|---|---|
| `api.py` | Código utilizado pelo contêiner da API. Contém todos os métodos da API que é o ponto de comunicação central da arquitetura |
| `clean_simulation_files.py` | Arquivo utilizado pelo contêiner clear_files. Utilizado para limpar *logs* das simulações anteriores. Os arquivos que serão excluídos podem ser escolhidos dentro deste código |
| `dashboard` | Contém os arquivos da interface *web* apresentada pelo contêiner **Interface** |
| `db_file` | Contém o arquivo SQL utilizado para montar o banco para o cenário de teste |
| `init.py` | Arquivo do Python utilizado para sinalizar o diretório como um diretório de pacotes do Python |
| `initialize_db.py` | No caso de não utilizar uma estrutura pronta para o DB, através do arquivo SQL, a estrutura do banco pode ser montada através do exemplo apresentado neste arquivo. Deve ser executado durante a criação do DB |
| `jacamo` | Contém um diretório onde deve ser inserido a estrutura do JaCaMo, via Gradle, para o caso de utilizar o contêiner de modelo que roda JaCaMo |
| `netlogo_agent_handler.py` | Arquivo intermédio que contém as funções que o NetLogo utiliza para acessar a API, através da extensão `py`. Através do arquivo, o NetLogo consegue executar o código Python necessário para a entrada e saída de agentes no modelo, através da comunicação com a API |
| `netlogo_output` | Contém os arquivos de saída com *logs* do NetLogo que são utilizados para testar a saída do(s) modelo(s). Além disso, também cria arquivos ao final da simulação indicando quais agentes permaneceram vivos ao fim da simulação, podendo ser utilizado para *debug* |
| `pycache` | Arquivos de cache utilizado pelos códigos Python executados no `volume` |
| `python_execute_netlogo.py` | Arquivo utilizado pelos contêineres opcionais de gatilho, que irão condicionar o início da execução dos contêineres de modelo através de alguma política. Contém código Python para executar *sockets* que aguardam o comando para iniciar a execução dos modelos |
| `py` | Arquivos da extensão `py`, utilizada pelo NetLogo, para utilizar código Python dentro do NetLogo. Na arquitetura, é utilizado para rodar o código Python que se comunica com a arquitetura |
| `router.py` | Código executado pelo contêiner **Router**. Contém uma função em *loop* que chama a API no método para processar agentes designados ao **Router**. No método da API estão contidas as políticas de roteamento. Neste arquivo é possível determinar um tempo entre o processamento de cada agente, para permitir que a arquitetura consuma mais recursos e seja capaz de processar mais agentes por intervalo de tempo, ou aumentar o tempo entre o processamento, diminuindo a sobrecarga da máquina |
| `sugar-map.txt` | Arquivo utilizado pelos outros dois arquivos do NetLogo, contendo informação da simulação em relação às posições do mapa que contêm açúcar e sua quantidade (contido na versão original do modelo) |
| `sugarscape_m1.nlogo` | Arquivo do NetLogo, que será utilizado pelo contêiner de modelo |
| `sugarscape_m2.nlogo` | Arquivo do NetLogo, que será utilizado pelo contêiner de modelo |

Fonte: autoria própria.

### 3.3.2 Entrada e Saída de Agentes dos Modelos

A Figura 15 ilustra parte do fluxo de transição dos agentes entre os modelos e a API. Os agentes estão inseridos em uma simulação e, para os agentes saírem ou entrarem na simulação, são implementados os gatilhos e as funções para entrada e saída. O modelo envia o agente para a API, que por sua vez prepara os dados e envia ao DB. Também através da API, o **Router** consegue perceber que existem agentes para serem processados. Após ler e processar os agentes, através das políticas de roteamento escolhidas, os agentes podem ser enviados aos modelos de destino.

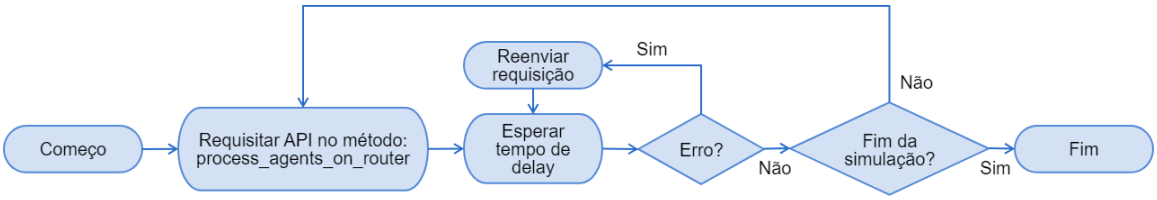
Figura 15 – Fluxo geral da entrada e saída de agentes nos modelos, mediados pela API.




Fonte: autoria própria.

Um fluxograma de funcionamento do **Router** é mostrado na Figura 16. É possível observar que a função principal se trata de um *loop* que faz uma requisição para a API processar os agentes que estão na fila, conforme o tipo de roteamento escolhido. Após aguardar o *delay*, caso aconteça algum erro, a requisição é reenviada. Se não houver erro, o *loop* de processar os agentes na fila do **Router** recomeça, e continua sendo executado até o fim da simulação.

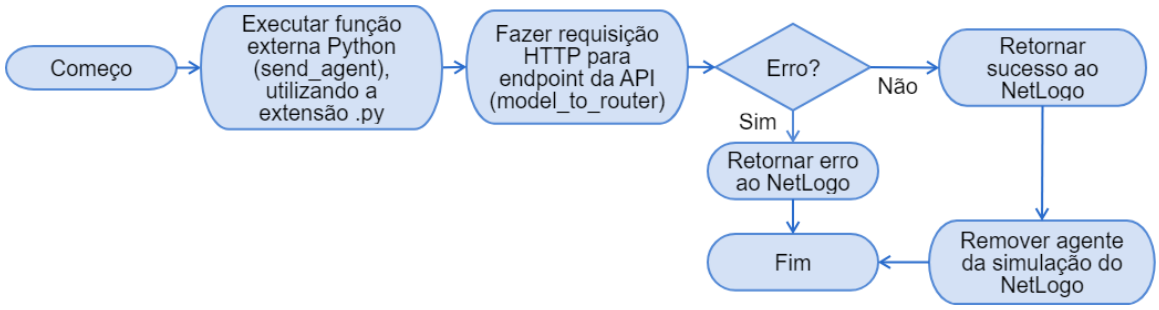
Figura 16 – Fluxo simplificado da função principal do **Router**.




Fonte: autoria própria.

As funções de envio/recebimento de agentes podem variar conforme a ferramenta de SMA escolhida. No caso do NetLogo, a comunicação com a API é feita via código Python. Quando um agente deve sair da simulação, é chamada a função `send_agent`, a qual é executada externamente ao NetLogo. Ao executar essa função, é feita uma requisição HTTP para o método (também chamado de rota ou *endpoint*) `model_to_router` da API. Se algum erro for detectado, é retornado erro ao NetLogo, para poder ser tratado. Se não tiver erro, o NetLogo é notificado de que o envio do agente foi feito com sucesso, permitindo que o agente seja removido da simulação. A Figura 17 ilustra este fluxo.

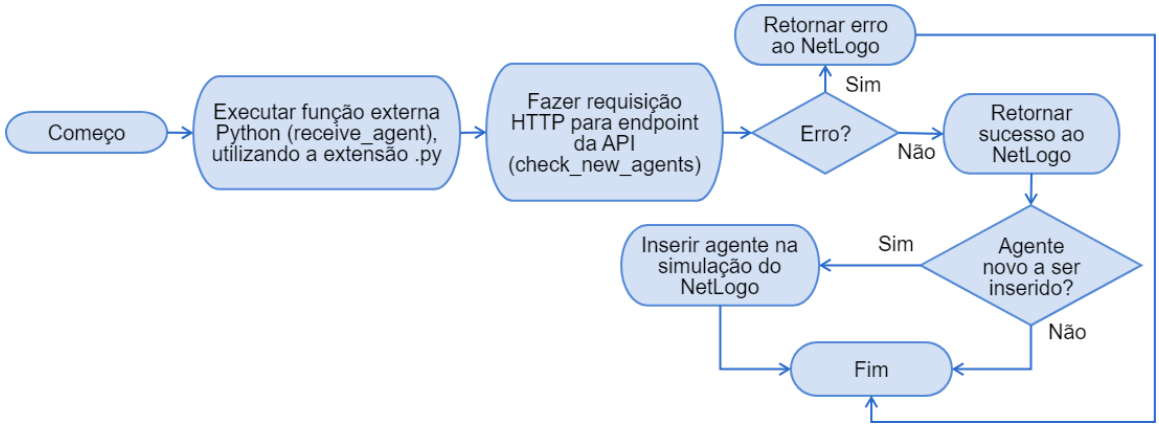
Figura 17 – Fluxo simplificado da função de envio de agentes do NetLogo.




Fonte: autoria própria.

A Figura 18 apresenta o fluxo executado para o recebimento de agentes no NetLogo. O NetLogo chama uma função externa, chamada `receive_agent`, a qual é executada externamente no Python. Essa função faz uma requisição HTTP para a API, no método `check_new_agents`. Se algum erro for detectado, é retornado erro ao NetLogo, para poder ser tratado. Se não tiver erro, o NetLogo é notificado que a execução foi feita com sucesso. Caso exista algum agente esperando para ser inserido no modelo, esse agente é inserido pelo NetLogo.

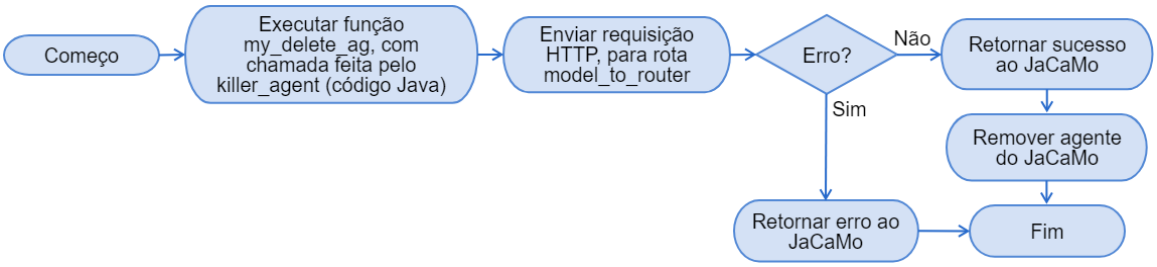
Figura 18 – Fluxo simplificado da função de recebimento de agentes do NetLogo.




Fonte: autoria própria.

O envio e recebimento de agentes no JaCaMo funciona de maneira similar ao NetLogo. A remoção do agente começa quando o `killer_agent`, através do recebimento de uma mensagem ASL enviada por um agente que será removido da simulação, começa o processo de remoção. Este agente faz a chamada de uma função Java (`my_delete_ag`), que faz uma requisição HTTP para a API, através do método `model_to_router`. Caso não haja erro na execução, o JaCaMo é notificado que tudo deu certo e o agente pode ser removido da simulação. A Figura 19 ilustra o fluxo.

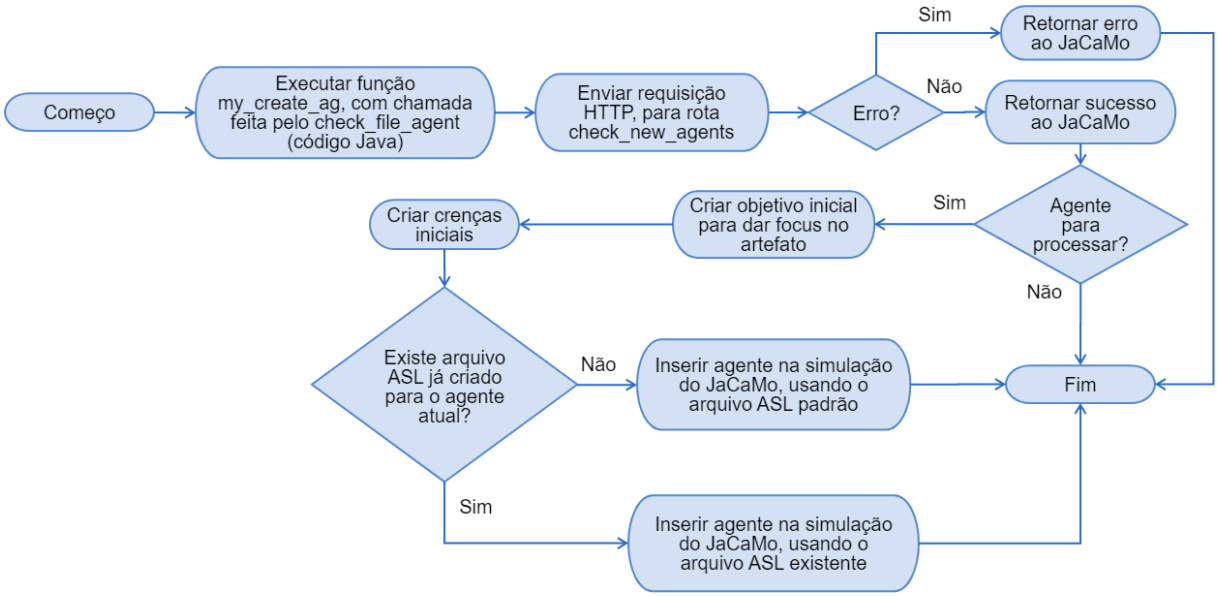
Figura 19 – Fluxo simplificado da função de envio de agentes do JaCaMo.




Fonte: autoria própria.

A Figura 20 mostra o fluxo de recebimento de agentes no JaCaMo. O recebimento começa com o agente `check_file_agent` verificando se há agentes que devem ser inseridos na simulação, através da execução da função Java `my_create_ag`. Essa função faz uma requisição HTTP para a API, através do método `check_new_agents`. Este método é ligeiramente diferente do utilizado pelo NetLogo, pois ao invés de receber todos os agentes que estão na fila de uma vez só, o JaCaMo utiliza uma rota que retorna apenas o primeiro agente da fila. Caso não haja erro na execução, é verificado se existem agentes para entrar na simulação. Se sim, o agente é criado com um objetivo inicial para dar foco no artefato do CArtAgO, para o agente conseguir ver o ambiente. Além disso, ele é criado com crenças iniciais, que podem ser ajustadas conforme necessidade.

Figura 20 – Fluxo simplificado da função de recebimento de agentes do JaCaMo.

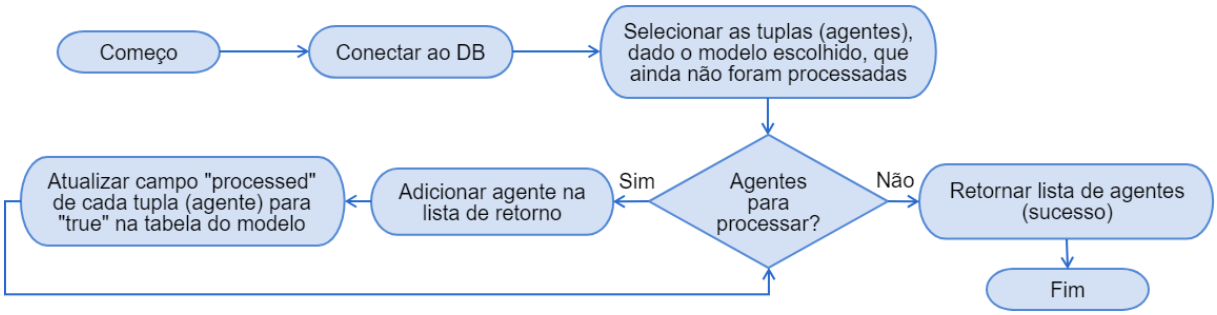



Fonte: autoria própria.

Por fim, é verificado se já existe um arquivo ASL com o ID do agente. Se sim, significa que o agente já passou pela simulação e, nesse caso, o agente é inserido na simulação utilizando o arquivo ASL existente, para trazer as informações referentes a sua última passagem pelo modelo. Se não existir o arquivo, o agente é inserido na simulação utilizando um arquivo ASL padrão, definido pelo desenvolvedor.

A Figura 21 ilustra o fluxo do método `check_new_agents` da API. Este método é executado pelos contêineres de modelo quando é necessário verificar se existem agentes para entrar no modelo. Após conectar ao DB, verifica-se na tabela do modelo todos os agentes que ainda não foram processados. Enquanto tiver agentes para serem processados, o agente é inserido em uma lista e sinalizado como processado. Quando não houver mais agentes para processar, a lista é retornada ao modelo. Além da versão do método que retorna todos os agentes que estão na fila da tabela, também existe uma versão que retorna apenas um por vez. Esse método pode ser utilizado em modelos que recebem apenas um agente por vez. Ressalta-se que existe uma variação desse método, chamado `check_new_agents_1`, que funciona de maneira muito similar ao comportamento descrito, sendo que a única diferença está no fato de que o método processa e retorna apenas o primeiro agente da fila, ao invés de todos (utilizado pelo modelo do JaCaMo do cenário de teste).

Figura 21 – Método da API: `check_new_agents`.



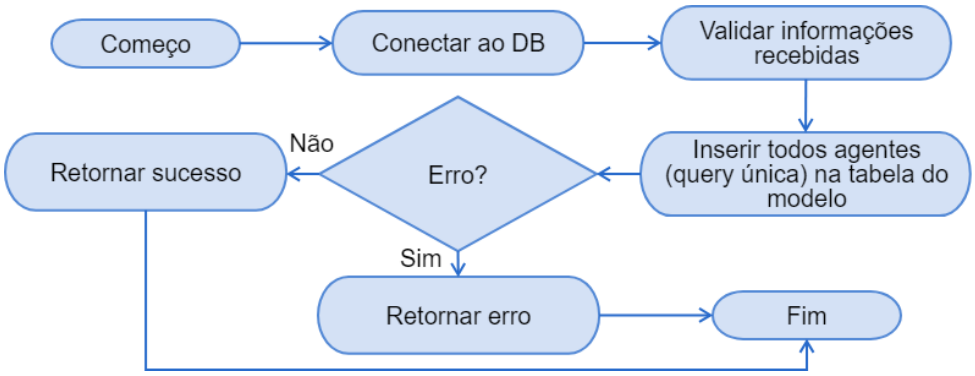

Fonte: autoria própria.

### 3.3.3 Register e Roteamento de Agentes para os Modelos

A Figura 22 ilustra o funcionamento do método `register_agents_on_platform`. Este método é executado para inserir novos agentes na arquitetura, atribuindo a eles um `id` para identificação. Após conectar com o DB, as informações do agente são validadas. Depois da validação, os agentes são inseridos no DB, em uma inserção única, na tabela do modelo escolhido (sem passar pelo **Router**, pois os agentes não passam pela política de roteamento quando recém-criados). A inserção de todos os agentes em uma única *query* é mais eficiente do que inserir um por um, diminuindo a sobrecarga do DB. Se houver algum problema durante a execução, é retornada

uma mensagem com o erro correspondente. Caso contrário, é retornada uma mensagem indicando sucesso na operação. Ressalta-se que, por conta dos cenários de simulação implementados (mais detalhes na Seção 4.3) inicializarem criando agentes apenas nos modelos do NetLogo, o fluxo de criação de agentes durante a fase de inicialização dos modelos, passando pelo **Register**, está sendo executado, inicialmente, pela solicitação do modelo, intermediado pelo código Python e, finalmente, sendo processado pela API, que retorna as agentes com a devida identificação da arquitetura ao modelo. Dessa forma, para esta implementação, não foi necessário adicionar um contêiner extra para o **Register** neste fluxo.

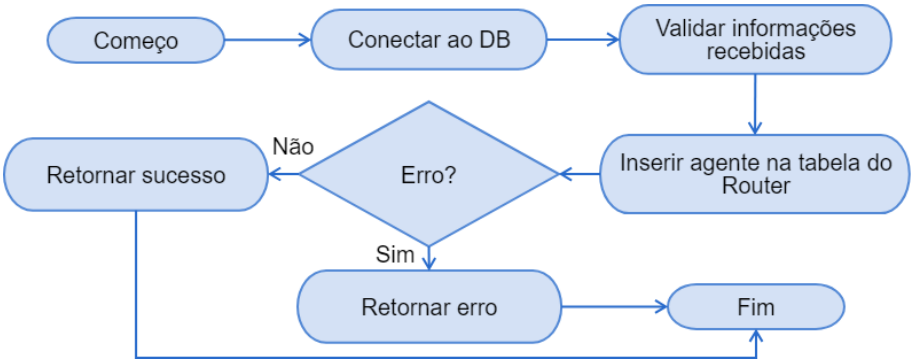
Figura 22 – Método da API: `register_agents_on_platform`.




Fonte: autoria própria.

A Figura 23 apresenta o fluxograma de funcionamento do método `model_to_router`. Este método trata do envio de agentes que saíram dos contêineres de modelo e devem ser processados pelo **Router**. Após conectar ao DB, as informações do agente são validadas. Após a validação, o agente é inserido na tabela do **Router**. Caso haja algum problema durante a execução, é retornada uma mensagem indicando o erro. Se tudo estiver correto, é retornada uma mensagem que indica sucesso.

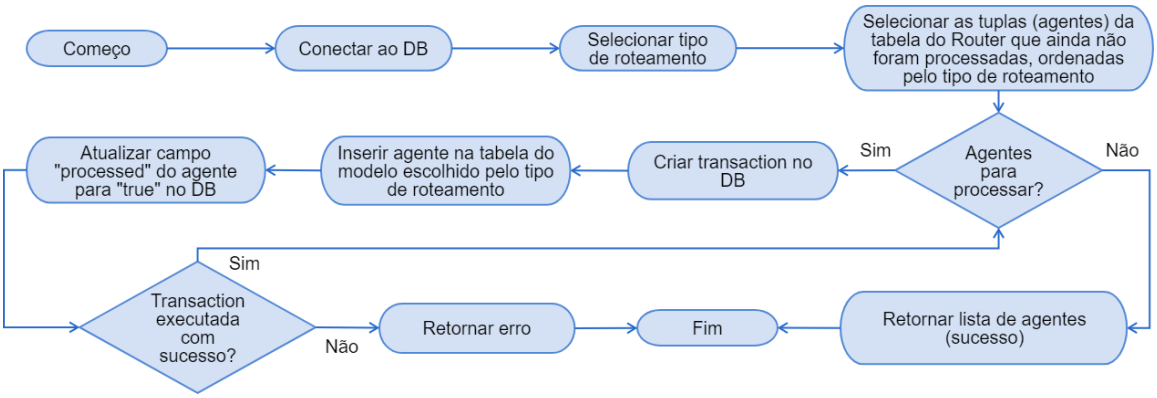
Figura 23 – Método da API: `model_to_router`.




Fonte: autoria própria.

A Figura 24 apresenta o fluxo do método `process_agents_on_router` da API. Este método é chamado pelo **Router**, tendo como responsabilidade processar os agentes que saíram dos modelos e encaminhá-los para outros modelos. Após conectar com o banco, é escolhido o tipo de roteamento. Com base no tipo escolhido, são processados os agentes que ainda não foram processados, ordenados pelo tipo de roteamento. Enquanto existirem agentes para processar, cria-se uma transação (do inglês, *transaction*) no DB, para garantir que no caso de qualquer falha no processo entre processar o agente, remover da fila do **Router** e enviar ao modelo, o processo possa ser revertido, evitando inconsistências.

Figura 24 – Método da API: `process_agents_on_router`.




Fonte: autoria própria.

Dentro de cada *transaction*, o agente é inserido na tabela do modelo escolhido. Depois disso, atualiza-se o agente na tabela do **Router**, indicando que foi processado. Se tudo ocorreu certo, o agente foi processado. No caso de acontecer algum erro no processamento de qualquer agente, é enviada uma mensagem indicando erro. Se todos os agentes forem processados corretamente, é retornada uma mensagem para o **Router** indicando sucesso.

Em relação aos tipos de roteamento, o estágio atual da abordagem conta com duas possibilidades, o modo aleatório e o modo sequencial geral. O modo aleatório ordena a fila de agentes do **Router** aleatoriamente. Já o modo sequencial geral ordena os agentes da fila pela ordem que eles chegaram, no formato FIFO (do inglês, *First In, First Out*. Baseado no tipo de roteamento escolhido, o **Router** processa os agentes da fila individualmente, atribuindo-os a um modelo (o modelo é escolhido aleatoriamente para cada passagem do agente pela fila). Ressalta-se que, embora a implementação conte com estes dois tipos de roteamento, a arquitetura é flexível para implementar novos comportamentos não previstos na concepção. Por exemplo, como o usuário tem acesso aos dados do agente, poderia ser desenvolvido um tipo de roteamento mais complexo, que considere os agentes mais promissores com base nos dados das simulações anteriores.

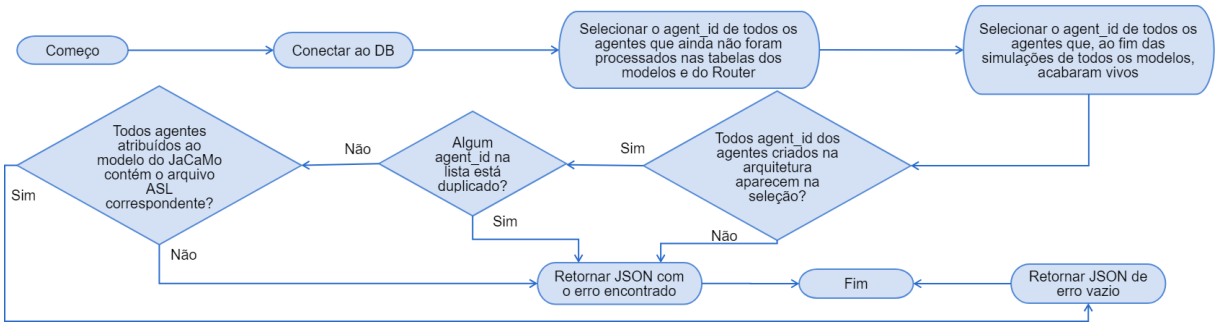
### 3.3.4 Testes de Sanidade

Testes de sanidade são testes criados para garantir que o funcionamento principal de partes ou da totalidade do sistema estejam conforme o esperado, visando atender os requisitos básicos (Fecko; Lott, 2002; Sammi; Masood; Jabeen, 2011). Esses testes são importantes para garantir que o funcionamento da arquitetura esteja correto, tanto durante as alterações feitas ao longo do desenvolvimento quanto durante a fase de execução. Dentre os testes de sanidade desenvolvidos para a arquitetura, é possível destacar: o (i) teste de sanidade geral e o (ii) teste de sanidade de rechecagem de caminho.

Para o teste de sanidade geral, primeiro busca-se no DB todos os agentes das tabelas dos modelos e do **Router** que ainda não foram processados, pois a execução foi parada para teste. Também se buscam os agentes que continuam vivos nos modelos quando a simulação parou. No cenário de simulação, os agentes vivos dos modelos *m1* e *m2* do NetLogo são armazenados na tabela de *alive_agents*. Já para o modelo *m3* (JaCaMo), como somente um agente fica vivo por vez, para obter o agente que acabou vivo ao final da simulação, é realizada uma consulta na tabela do modelo *m3*, buscando ordenadamente pela entrada do agente na tabela, o último agente processado. A união destas buscas forma um agrupamento que, se o funcionamento da simulação estiver correto, deve conter todos os agentes criados no início da simulação.

O teste de sanidade geral conta com 3 fases, que verificam: (i) se todos os agentes criados no início estão presentes na seleção, pois todo agente deve estar ou em um modelo (vivo), ou na fila para entrar em um modelo, ou na fila de processamento do **Router**; (ii) se nenhum agente da seleção está duplicado, pois nenhum agente deve existir em mais de um local simultaneamente; e, (iii) se todos os agentes que passaram pelo JaCaMo possuem um arquivo ASL (exceto o último agente vivo, que pode estar na primeira passagem). Se nenhum dos testes der problema, a execução ocorreu conforme esperado e o retorno do JSON de erros é feito em branco. Se alguma etapa teve erro, o erro é indicado no JSON de retorno do método. A Figura 25 representa esquematicamente o teste de sanidade geral criado para testar parte do funcionamento da arquitetura durante uma simulação.

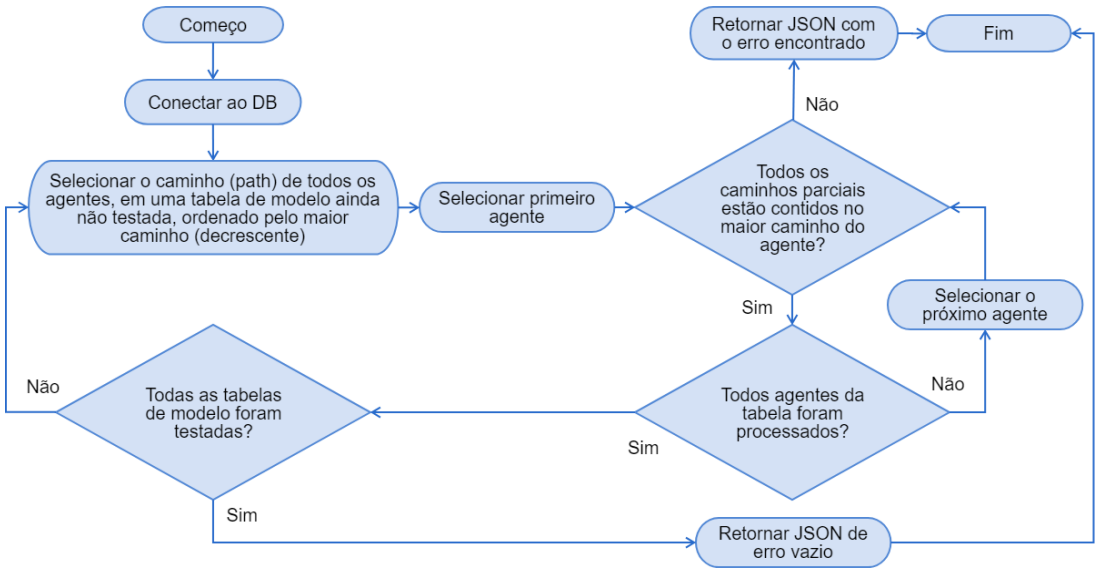
Figura 25 – Método da API: `sanity_test`.




Fonte: autoria própria.

Já o teste de sanidade de rechecagem de caminho, conforme ilustrado na Figura 26, tem por objetivo garantir que não houve equívoco durante a transição dos agentes entre os modelos. Para isso, é feita a leitura de todas as tabelas de modelo do banco, buscando o caminho (*path*) de cada agente, em ordem decrescente. Depois, para cada agente da tabela, verifica-se se todos os caminhos parciais estão contidos no caminho maior. Por exemplo, se o caminho final foi **1-2-3**, o que significa que o agente começou no modelo **m1**, depois foi para o **m2** e por fim para o **m3**, e, portanto, os caminhos parciais como **1** e **1-2** devem estar contidos no caminho final. Este teste é feito, separadamente, para todos os agentes de todos os modelos. Se acontecer qualquer diferença entre os caminhos, o teste retorna um JSON indicando o erro e qual foi o agente com o caminho errado. Se não, significa haver consistência nas trajetórias dos agentes entre os modelos.

Figura 26 – Método da API: `sanity_test_recheck_agent_path`.




Fonte: autoria própria.

# 4 RESULTADOS

No cenário de simulação proposto, foram construídos contêineres para permitir que a arquitetura tenha suporte a duas diferentes ferramentas de desenvolvimento para Sistemas Multiagente: NetLogo (Wilensky, 1999) e JaCaMo (Boissier; Bordini; Hübner; Ricci; Santi, 2013). Com base nestas ferramentas, foram escolhidos dois modelos clássicos, apresentados nas próprias documentações das ferramentas: Sugarscape 2 Constant Growback (Epstein; Axtell, 1996; Li; Wilensky, 2009) do NetLogo e Gold Miners (Bordini; Hübner; Tralamazza, 2007) do JaCaMo, com adaptações.

Foram utilizados três contêineres de modelo, executando duas cópias isoladas do modelo Sugarscape (em dois contêineres) e uma cópia do Gold Miners (em um contêiner). O principal objetivo deste cenário de simulação é validar a viabilidade da abordagem, permitindo que os agentes se movam livremente entre os modelos, mesmo que as duas ferramentas utilizem diferentes arquiteturas de agentes.

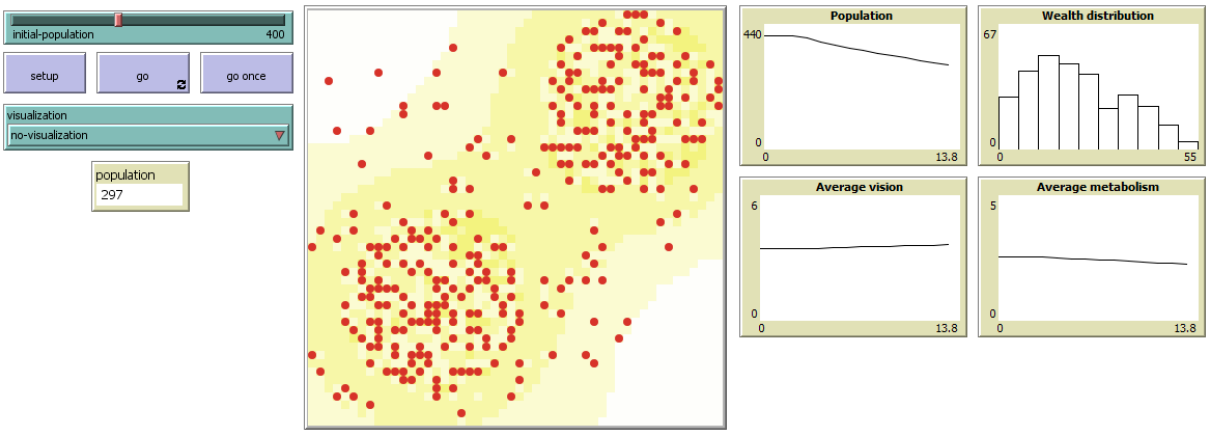
## 4.1 O Modelo Sugarscape

O Sugarscape (Epstein; Axtell, 1996; Li; Wilensky, 2009) é um modelo utilizado na simulação de sociedades artificiais, simulando uma população com recursos limitados, na qual cada agente é representado como uma formiga que, para sobreviver, deve se deslocar pelo ambiente em busca de açúcar (alimentação). Cada formiga nasce com uma quantidade de açúcar (de 5 a 25), metabolismo (de 1 a 4) e visão (de 1 a 6). O metabolismo define quanta energia a formiga perde ao se mover. Se a energia da formiga chega a zero, ela morre e sai da simulação. A visão determina até quantas posições de distância a formiga consegue ver o açúcar, baseado na posição atual. A Figura 27 ilustra a interface da IDE (do inglês *Integrated Development Environment*) do NetLogo simulando o modelo original.

Na Figura 27, pode-se visualizar o ambiente (centro), os parâmetros de entrada (esquerda) e algumas métricas (direita). É possível configurar o número de agentes na inicialização e, através dos botões, controlar a execução, indefinidamente ou passo a passo. No centro, vê-se o ambiente. Os pontos vermelhos representam os agentes,

enquanto os demais representam o açúcar inserido no ambiente, variando do amarelo ao branco, respectivamente, de maior para menor intensidade. Por fim, os gráficos apresentados na direita representam as métricas de saída da simulação, como tamanho da população, médias de metabolismo e de visão.

Figura 27 – Exemplo de execução do modelo Sugarscape na IDE do NetLogo.




Fonte: adaptado de Li; Wilensky (2009).

Um exemplo de como o agente é representado no DB é apresentado na Figura 28. É possível observar as informações do agente com `agent_id` 762 (o `id` 1033 é apenas para identificação da transação da tupla no DB). Ainda, observam-se os dados do agente no NetLogo na coluna `data`, cujos valores 0, 4 e 2 representam o açúcar, o metabolismo e a visão, respectivamente. A coluna `path` mostra por quais modelos o agente passou (neste caso, ele começou no modelo 2 e foi para o 3). A coluna `processed` informa que este dado já foi processado pelo modelo, ou seja, o agente já entrou na simulação para qual foi designado pelo **Router**. Por fim, os campos `created_at` e `updated_at` são utilizados apenas para gerenciamento interno das tabelas.

Figura 28 – Informações da tupla de um agente no DB.

| | id | agent_id | data | path | asl_file_path | processed | created_at | updated_at |
|---|---|---|---|---|---|---|---|---|
| ☐ Editar Copiar Apagar | 1033 | 762 | [0 4 2] | 2-3 | | 1 | 2023-02-08 18:32:49 | 2023-02-08 18:44:23 |

Fonte: autoria própria.

Figura 29 – Funções NetLogo responsáveis por (a) enviar e (b) receber agentes entre a arquitetura e o modelo NetLogo.

```
to send_agent_to_api
  py:setup py:python
  py:run "from netlogo_agent_handler import send_agent"
  ;//Example: "send_agent('12', '[1 2 3]', '1-1')"

  let updated_historic ""
  ifelse (historic = "")
  [
    set updated_historic 1
  ]
  [
    set updated_historic (word "" (historic) "-1")
  ]
  let tuple []
  set tuple lput sugar tuple
  set tuple lput metabolism tuple
  set tuple lput vision tuple

  print("NL: sending this agent to router:")
  print(agent_id)
  let result py:runresult (word "send_agent('" (agent_id) "', '" tuple "', '" (updated_historic) "')")

  print("NL: Agent send successfully?")
  print(result)
end
```

(a)

```
to check_new_agents_on_api
  py:setup py:python
  py:run "from netlogo_agent_handler import receive_agent"
  let result py:runresult (word "receive_agent('" ("m1") "')")
  ifelse(length result > 0)
  [
    foreach result
    [
      x ->
      let tuple_intern read-from-string item 1 x

      create-turtles 1
      [
        set agent_id item 0 x
        set sugar random-in-range 5 25
        set metabolism item 1 tuple_intern
        set vision item 2 tuple_intern
        set historic item 2 x

        set shape "circle"
        move-to one-of patches with [not any? other turtles-here]
        set vision-points []
        foreach (range 1 (vision + 1)) [ n ->
          set vision-points sentence vision-points (list (list 0 n) (list n 0) (list 0 (- n)) (list (- n) 0))
        ]
        run visualization
        print "Agent created"
      ]
    ]
  ]
  [
    print("There is no new agent on DB")
  ]
end
```

(b)

Fonte: autoria própria.

A Figura 29 mostra o código adicionado ao modelo original que contém as funções que permitem ao NetLogo enviar (Figura 29(a)) e receber (Figura 29(b)) dados da arquitetura. A arquitetura fornece as funções `send_agent_to_api` e `check_new_agents_on_api` ao usuário para poder enviar (e quais informações devem ser enviadas) e receber um agente. Para este modelo específico, definiu-se que o gatilho de saída será baseado na morte do agente, ou seja, quando seu alimento chega a zero (conforme o modelo original), o modelo envia este agente para a arquitetura e remove este agente da simulação. A adição do gatilho que dispara a função de envio de agente ao código original é apresentada na Figura 30.

Figura 30 – Exemplo de código do NetLogo com adição do gatilho `send_agent_to_api`.

```
ask turtles [
  turtle-move
  turtle-eat
  if sugar <= 0
  [
    send_agent_to_api
    die
  ]
  run visualization
]
```

Fonte: autoria própria.

Para este modelo especificamente é necessário fazer outro ajuste, pois o agente volta para a simulação com a sua informação sobre comida sobrescrita com um valor definido aleatoriamente (nos limites estabelecidos originalmente no modelo), enquanto os outros parâmetros permanecem os mesmos da última simulação. Isto ocorre porque se o agente sai da simulação quando sua comida for zero e, ao retornar, o agente retornaria com o mesmo valor da saída, causaria um *loop*, pois o agente chegaria à simulação com zero comida e morreria novamente.

Foram adicionados dois novos atributos aos atributos originais dos agentes NetLogo, os quais são `agent_id` e `historic`. Utiliza-se o `agent_id` para verificar o `id` único do agente para a arquitetura, que difere do `id` regular do NetLogo, o qual é usado apenas para a própria simulação do NetLogo. O atributo `historic` armazena o caminho de quais contêineres de modelos os agentes passaram.

Para adaptar qualquer modelo NetLogo à plataforma, a única dependência necessária é a instalação da extensão `py` (Wilensky, 2022), que pode ser encontrada na documentação do NetLogo. Utilizou-se esta extensão para enviar/receber informações da API através do uso de código Python. Desenvolveram-se funções simples para esta tarefa, bastando o programador incluir essas funções e utilizá-las sempre que o modelo precisar enviar/receber informações sobre os agentes. Exemplos de como adaptar e como usar estas funções estão no GitHub da ferramenta (Lima; Aguiar, 2022).

Por fim, a Figura 31 mostra as informações obtidas em parte de um *log* gerado pela saída obtida do contêiner durante a execução de um modelo do NetLogo. O modelo atual implementado apresenta informações para *debug*, como o horário em que o modelo começou a ser executado, além do nome, tempo de duração e retorno obtido durante a execução de algumas funções do Python. Ressalta-se que essas informações são obtidas através dos comandos de *print* utilizados no NetLogo. Outros tipos de *logs* podem ser gerados conforme a necessidade do programador, bastando imprimir as informações que forem relevantes.

Figura 31 – Exemplo de *log* da execução do NetLogo.

```
Starting model on:  14:48:14
/opt/netlogo/netlogo-headless.sh --model /shared_volume/sugarscape_m1.nlogo --experiment experiment1
Starting model...
NL: Python function - new_request_to_register
NL: On NetLogo, Start time of request_to_register
02:48:25.551 PM 06-Feb-2023
NL: Starting register:
Start time request_to_register: 1707230905.8719466
Host: 129.159.61.46
Response from API: true
Function worked well
Left loop
End time request_to_register: 0.9180641174316406

NL: return from register:
true
NL: On NetLogo, final time from request_to_register
02:48:26.833 PM 06-Feb-2023
NL: ------------BEGIN TICK --------------
NL: Tick number:
0
NL: Python Function - new_check_new_agent
NL: NetLogo, Start time - new_check_new_agent
02:48:26.973 PM 06-Feb-2024
Start time receiving_agents: 1707230907.1461995
Function worked well
Left loop
End time receiving_agents: 0.4478728771209717

NL: receiving_agents: -----------
NL: All agents to be processed:
[[1 [23 3 2] ] [2 [15 4 3] ] [3 [18 1 2] ] [4 [20 4 1] ] [5 [18 1 1] ] [6 [16 1 4] ] [7 [18 3 2] ] [8 [15 4 4] ] [9 [5 2 6] ]]
```

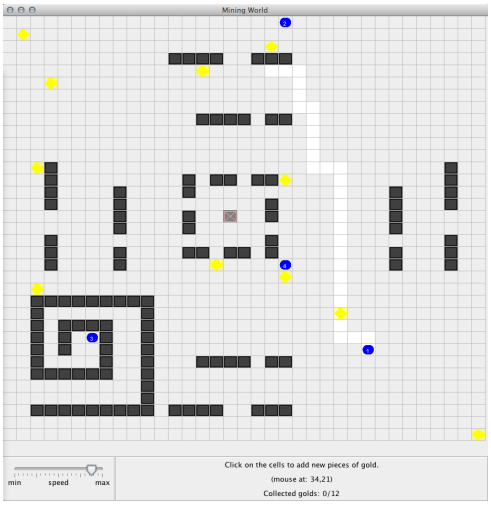
Fonte: autoria própria.

## 4.2 O Modelo Gold Miners

O modelo Gold Miners (Bordini; Hübner; Tralamazza, 2007) simula um conjunto de agentes representando mineradores cujo papel é navegar por um ambiente. Por exemplo, quando um agente encontra um nó de ouro, ele interrompe seu objetivo atual, pega o ouro e o traz de volta ao depósito central. O modelo também conta com um agente do tipo líder. O papel desse agente é receber mensagens dos agentes quando estes recuperam um nó de ouro, armazenando um histórico de quantos nós cada agente conseguiu recuperar. Este agente, embora simples, representa o papel de liderança, podendo auxiliar na coordenação dos agentes, se for necessário.

A Figura 32 ilustra a interface do JaCaMo simulando o modelo Gold Miners. Na Figura, é possível ver a representação dos agentes (pontos azuis), o depósito (caixa com X no centro), as barreiras ambientais – posições não acessíveis aos agentes (caixas pretas), os nós de ouro (caixas amarelas), as posições já visitadas pelos agentes (caixas brancas) e as não acessadas (caixas cinza).

Figura 32 – Exemplo de execução do modelo Gold Miners na GUI do JaCaMo.




Fonte: adaptado de Bordini; Hübner; Tralamazza (2007).

Foram feitas algumas adaptações neste modelo para criar o cenário de simulação em conjunto com o Sugarscape. Criou-se um novo mundo (o modelo original conta com alguns mundos de exemplo), alterando a posição das barreiras, dos nós de ouro, dentre outros. Para o cenário no qual o modelo do JaCaMo é executado dentro do Docker, a opção de utilizar a GUI do JaCaMo foi desativada, porém, caso o programador queira utilizar o JaCaMo fora do Docker e comunicar com a plataforma, foi adicionado um parâmetro `using_docker`, para que isso possa ser ajustado.

Além disso, ao invés de simular múltiplos agentes do tipo minerador, foi definido que apenas um agente minerador funciona por vez, para evitar sobrecarga no modelo (embora seja possível múltiplos mineradores coexistirem). No ciclo padrão, o agente que vem da arquitetura para este modelo é inserido na simulação, faz a execução regular até encontrar um nó de ouro, faz a entrega e, após a entrega ser computada, o agente envia uma mensagem para o `killer_agent`, sinalizando para ser removido da simulação.

Após a remoção, outro agente que esteja na fila é inserido na simulação, criando um ciclo no qual cada agente, durante sua vida útil, entra na simulação, coleta um nó de ouro, entrega, é removido da simulação e dá lugar para um novo agente. Como cada agente traz um nó de ouro para o depósito central, removendo esse ouro do ambiente, a quantidade de nós de ouro no ambiente vira o limitante de quantos agentes

irão participar da simulação. Como cada agente entra na simulação na intenção de encontrar um nó de ouro, quando esses nós se esgotarem, o último agente inserido irá vagar pelo ambiente indefinidamente. Ressalta-se que isso é uma decisão do projeto do cenário de simulação proposto, e não uma limitação da ferramenta.

As Figuras 33(a), 33(b), 34(a) e 34(b) apresentam, respectivamente, os trechos do código-fonte necessário para que o JaCaMo possa receber e enviar dados da arquitetura, via funções Java e ASL.

Figura 33 – Funções Java responsáveis por (a) inserir e (b) remover agentes no modelo JaCaMo.

```
System.out.println("Sugar: " + sugar);
System.out.println("Metabolism: " + metabolism);
System.out.println("Vision: " + vision);
char ch='"';
String bels = "agent_id("+agent_id+")";
bels = bels + ",path("+ ch + agent_path + ch + ")";
bels = bels + ", sugar("+sugar+"), metabolism("+metabolism+"), vision("+vision+")";
System.out.println("Creating agent with this beliefs: "+bels);
Settings s = new Settings();
s.addOption(Settings.INIT_BELS, bels);
s.addOption(Settings.INIT_GOALS, "jcm::focus_env_art([art_env(mining,m2view,default)],5)");
try {
    String asl_file_name = "list/"+agent_id+".asl";

    if (!Files.exists(Paths.get("src/agt/"+asl_file_name))){
        asl_file_name = "default_agent.asl";
    }

    System.out.println("asl_file_name: "+asl_file_name);

    rs.createAgent(agent_id, asl_file_name, null, null, null, s, ts.getAg());
    rs.startAgent(agent_id);
    System.out.println("Agent created by custom file");
} catch (Exception e) {
    e.printStackTrace();
}
```

(a)

```
String tuple_agent_id = String.valueOf(args[0]);
String tuple_data = "[" + String.valueOf(args[3]) +
" " + String.valueOf(args[4]) + " " + String.valueOf(args[5]) + "]";
String tuple_path = String.valueOf(args[1]) == "" ? "3" :
String.valueOf(args[1]).replace("\"", "")+"-3";

System.out.println("Tuple - agent_id: "+tuple_agent_id);
System.out.println("Tuple - data: "+tuple_data);
System.out.println("Tuple - path: "+tuple_path);

if (rs.killAgent(tuple_agent_id, null, 0)){
    System.out.println("Agent "+tuple_agent_id+" removed from the simulation...");

    String postUrl = "http://"+host+":5000/api/v1/resources/model_to_router";
    JSONObject json_obj = new JSONObject();
    json_obj.put("agent_id",tuple_agent_id);
    json_obj.put("data",tuple_data);
    json_obj.put("path",tuple_path);

    HttpClient httpClient = HttpClientBuilder.create().build();
    HttpPost post = new HttpPost(postUrl);
    StringEntity postingString = new StringEntity(json_obj.toString());
    post.setEntity(postingString);
    post.setHeader("Content-type", "application/json");
    HttpResponse  response = httpClient.execute(post);
} else {
    System.out.println("Error while removing agent from simulation...");
}
```

(b)

Fonte: autoria própria.

Figura 34 – Funções ASL responsáveis por (a) verificar se existem agentes aguardando para serem inseridos e (b) remover agentes no modelo JaCaMo.

```
+!check_new_agent : true
<-
    .print("Checking and creating agent if it exists on API");
    mylib.check_new_agent;
    .wait(1000);
    !check_new_agent.
```

(a)

```
+kill(AGENTID, PATH, MYNAME, SUGAR, METABOLISM, VISION) : true
  <- .print("I've received a message to kill agent ",AGENTID);
     .print("Killing agent ",AGENTID);
     /* Calling Java code to remove agent from simulation */
     /* and send it to the Router (via API) */
     mylib.my_delete_ag(AGENTID, PATH, MYNAME, SUGAR, METABOLISM, VISION);
     .wait(2000);
     /* Belief to keep all removed agents */
     +killed_agent(AGENTID);
     .print("This agent has being removed from the simulation: ",AGENTID).
```

(b)

Fonte: autoria própria.

Mais especificamente, a arquitetura apresenta dois agentes inseridos ao modelo para lidar com o envio/recebimento de informações: `killer_agent` e `check_file_agent`. O agente `killer_agent` espera por mensagens para remover agentes da simulação, vindas dos próprios agentes. Quando isso acontece, este agente salva todas as informações do agente em um arquivo ASL (que será utilizado quando este agente reentrar na simulação) e então retira o agente da simulação. Já o agente `check_file_agent` tem a função de executar o código Java que se comunica com a API para verificar se existe algum agente na fila para entrar no modelo do JaCaMo. Se sim, o agente é montado com base no arquivo ASL (arquivo padrão se o agente nunca passou pelo modelo ou arquivo específico com dados anteriores se o agente já passou pelo modelo).

A Figura 35 mostra o código ASL que é adicionado ao agente padrão da simulação no JaCaMo. Neste código, são adicionadas as crenças referentes às variáveis utilizadas no NetLogo (neste caso, `sugar`, `metabolism` e `vision`), além do `agent_id`, utilizado pela plataforma. Além disso, também é adicionada a crença `my_testing`, que mostra algo que pode ser carregado de uma simulação para a outra dentro do JaCaMo. Na implementação atual, esse código é executado assim que o agente minerador cumpre o seu papel (deposita um nó de ouro), mas o momento da chamada pode ser alterado conforme a necessidade. A chamada poderia ser feita antes, caso o agente precisasse das informações do NetLogo para realizar alguma ação, bastando separar a parte final, na qual o agente salva seu arquivo e solicita a sua remoção da simulação ao `killer_agent`. Como um complemento, a Figura 36 mostra parte do código ASL original do agente minerador, com a adição do gatilho (última linha) que dispara a chamada da função apresentada na Figura 35.

Figura 35 – Função ASL adicionada no agente padrão de simulação do JaCaMo.

```
+main : true
<-
  .print("Hello there. I'm a regular agent");
  /* Getting Agent's ID and Path */
  ?agent_id(AGENTID);
  ?path(PATH);

  /* Something that is carried through the simulations */
  .random(R);
  +my_testing(R);
  ?my_testing(R);
  .print("R: ", R);

  .abolish(agent_id(_));
  .abolish(path(_));

  /* Getting Agent's Sugar, Metabolism and Vision (From NetLogo) */
  ?sugar(SUGAR);
  ?metabolism(METABOLISM);
  ?vision(VISION);

  .print("Sugar ", SUGAR);
  .print("Metabolism ", METABOLISM);
  .print("Vision ", VISION);

  /* Removing previous values, so the attributes keep updated */
  .abolish(sugar(_));
  .abolish(metabolism(_));
  .abolish(vision(_));

  .my_name(MYNAME);
  .concat("src/agt/list/",MYNAME,".asl",NAME)
  /* It is possible to save the agent, adding the beliefs, for examply: say(hello) */
  .save_agent(NAME,[start,say(hello)]);
  .print("Saved my information on file. Sending message to remove agent from simulation");

  .send(killer_agent, tell, kill(AGENTID, PATH, MYNAME, SUGAR, METABOLISM, VISION));
  /* Do "untell" because the same agent can pass through the simulation more than once,so, the belief must be removed */
  .send(killer_agent, untell, kill(AGENTID, PATH, MYNAME, SUGAR, METABOLISM, VISION)).
```

Fonte: autoria própria.

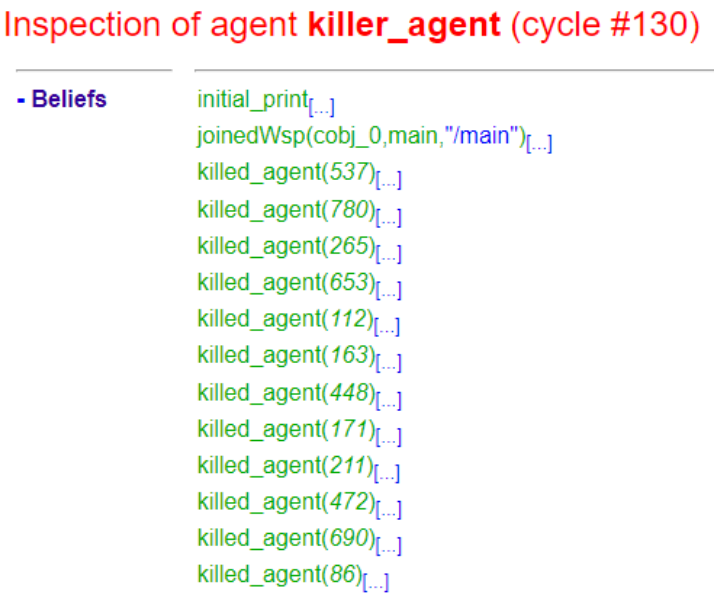
Figura 36 – Alteração no código ASL original do agente para inserção do gatilho que dispara a função responsável por se comunicar com a arquitetura para enviar e receber dados.

```
/* new plan for event +!handle(_) */
@pfunction[atomic]
+!handle(gold(X,Y))
  :  not free
  <- .print("Handling ",gold(X,Y)," now.");
     !pos(X,Y);
     !ensure(pick,gold(X,Y));
     ?depot(_,DX,DY);
     !pos(DX,DY);
     !ensure(drop, 0);
     .print("Finish handling ",gold(X,Y));
     ?score(S);
     -+score(S+1);
     .send(leader,tell,dropped);
     +main.
```

Fonte: autoria própria.

Após criar as crenças dos atributos, cria-se o caminho para o novo arquivo (baseado no nome do agente, o qual é o próprio *agent_id*), salva-se o agente (através da função *save_agent* do Jason) e envia-se o comando para o agente `killer_agent` remover o agente da simulação. O `killer_agent` também guarda, para cada agente removido, uma crença chamada `killed_agent`, que armazena a identificação do agente removido da simulação do JaCaMo por este agente. A Figura 37 mostra, por meio da interface *web* do JaCaMo, a lista das crenças guardadas dos agentes removidos pelo `killer_agent` em uma simulação.

Figura 37 – Informações sobre agentes removidos da simulação, através das crenças do *killer_agent* na interface *web* do JaCaMo.




Fonte: autoria própria.

A função do agente `check_file_agent` é verificar se algum agente está esperando para entrar no modelo atual. Se sim, o agente é incluído no modelo. Este agente usa código Java e ASL para se comunicar com a API e inclui o agente na simulação. Este código também pode ser adaptado para inserir informações iniciais extras ao agente, como crenças, objetivos ou foco nos artefatos do CArtAgO.

Quando o modelo inclui um agente na simulação, o agente `check_file_agent` verifica se existe um arquivo ASL com o mesmo `agent_id` do agente, ou seja, verifica se o novo agente já passou pelo modelo anteriormente. Se o arquivo existir, o modelo cria o agente com suas informações anteriores (usando o arquivo ASL existente). Caso contrário, o agente é criado utilizando como base um arquivo ASL padrão. A Figura 38 apresenta um exemplo com parte do conteúdo de um arquivo ASL, especificamente do agente **472**. É possível observar que o agente passou pelo modelo mais de uma vez, ou seja, foram transferidas informações de uma simulação para outra.

Figura 38 – Parte do arquivo ASL de um agente inserido no JaCaMo mais de uma vez.

```
// beliefs and rules
kqml::bel_no_source_self(NS::Content,Ans) :- (NS::Content[|LA] & (kqml::clear_source_self(LA,NLA) & ((Content =.. [F,T,_71]) & (Ans =.. [
NS,F,T,NLA])))).
kqml::bel_no_source_self(NS::Content,Ans) :- (NS::Content[|LA] & (kqml::clear_source_self(LA,NLA) & ((Content =.. [F,T,_554]) & (Ans =.. [
NS,F,T,NLA])))).
kqml::clear_source_self([source(self)|T],NT) :- kqml::clear_source_self(T,NT).
kqml::clear_source_self([A|T],[A|NT]) :- ((A \== source(self)) & kqml::clear_source_self(T,NT)).
kqml::clear_source_self([],[]).
kqml::clear_source_self([source(self)|T],NT) :- kqml::clear_source_self(T,NT).
kqml::clear_source_self([A|T],[A|NT]) :- ((A \== source(self)) & kqml::clear_source_self(T,NT)).
depot(7,5,27)[artifact_id(cobj_3),artifact_name(m2view),percept_type(obs_prop),source(percept),workspace("/main/mining",cobj_2)].
my_testing(0.3619893330067251).
my_testing(0.020466827705073354).
pos(5,27)[artifact_id(cobj_3),artifact_name(m2view),percept_type(obs_prop),source(percept),workspace("/main/mining",cobj_2)].
score(1).
main.
gsize(7,35,35)[artifact_id(cobj_3),artifact_name(m2view),percept_type(obs_prop),source(percept),workspace("/main/mining",cobj_2)].
focused(WksName,ArtName[artifact_type(Type)],ArtId) :- focusing(ArtId,ArtName,Type,_512,WksName,_513).
joinedWsp(cobj_0,main,"/main")[artifact_id(cobj_1),artifact_name(session_472),percept_type(obs_prop),source(
percept),workspace("/main",cobj_0)].
joinedWsp(cobj_2,mining,"/main/mining")[artifact_id(cobj_1),artifact_name(session_472),percept_type(obs_prop),source(
percept),workspace("/main",cobj_0)].
joinedWsp(cobj_0,main,"/main")[artifact_id(cobj_1),artifact_name(session_472),percept_type(obs_prop),source(
percept),workspace("/main",cobj_0)].
joined(WksName,WksId) :- joinedWsp(WksId,WksName,_514).
last_dir(down).
focusing(cobj_3,m2view,"mining.MiningPlanet",cobj_2,mining,"/main/mining")[artifact_id(cobj_4),artifact_name(body_472),percept_type(
obs_prop),source(percept),workspace("/main/mining",cobj_2)].

// initial goals
!start.
!say(hello).

// plans from file:src/agt/list/472.asl

@p__73[source(self),url("file:src/agt/list/472.asl"),url("jar:file:/home/gradle/.gradle/caches/modules-2/files-2.1/org.jacamo/jacamo/1.0/
bda076187adb93bc05a91ab0cdcb2fb44b039ed6/jacamo-1.0.jar!/templates/common-cartago.asl")] +!jcm::focus_env_art([],_509).
```

Fonte: autoria própria.

Na Figura, é possível observar em destaque a crença `my_testing` com os valores das duas simulações, além dos objetivos iniciais inseridos pelo código Java e dos planos que constavam da última execução (carregados através do código ASL do agente gerado anteriormente, conforme destacado). Já quando o agente é inserido pela primeira vez no modelo, o agente é inserido através da criação de um novo arquivo ASL usado como um arquivo modelo (na implementação atual é o `default_agent.asl`), cujo nome deste arquivo é o `agent_id` do próprio agente.

Por fim, a Figura 39 complementa as informações sobre a inserção do agente. Trata-se de um *log* obtido através do Docker, que contém algumas informações de capturas de tela tanto de informações ASL quanto outros feitos pelo código Java. Estão contidas nas informações tanto dados do agente recebido no momento, quanto a confirmação de que o agente foi criado utilizando dados da passagem anterior e, por fim, o agente é removido da simulação pelo *killer_agent*.

Figura 39 – Parte de um *log* gerado pelo Docker, sobre a saída do contêiner do JaCaMo.

```
New agents
ID: 472
Stats: [0 4 1]
Path: 2-3-1
Sugar: 0
Metabolism: 4
Vision: 1
Creating agent with this beliefs: agent_id(472),path("2-3-1"), sugar(0), metabolism(4), vision(1)
asl_file_name: list/472.asl
Agent created by custom file
[472] Hello there. I'm a regular agent
[472] R: 0.3619893330067251
[472] Sugar 0
[472] Metabolism 4
[472] Vision 1
[472] Saved my information on file. Sending message to remove agent from simulation
[killer_agent] I've received a message to kill agent 472
[killer_agent] Killing agent 472
Executing JAVA custom code - delete
[472] focusing on artifact m2view (at workspace /main/mining) using namespace default
Java Args0 - agent_id: 472
Java Args1 - path: "2-3-1"
Java Args2 - id: 472
Java Args3 - sugar: 0
Java Args4 - metabolism: 4
Java Args5 - vision: 1
Tupla - agent_id: 472
Tupla - data: [0 4 1]
Tupla - path: 2-3-1-3
Agent 472 removed from the simulation...
```

Fonte: autoria própria.

Na adaptação do modelo original, desenvolvida para esta Tese, utilizou-se apenas um agente minerador, juntamente com os agentes da arquitetura (`check_file_agent` e `killer_agent`) e o agente líder. Assim, quando o agente entra na simulação, ele assume o controle do agente minerador. Este agente traz suas informações consigo (sobre simulações anteriores), navega pelo mapa, pega ouro, traz de volta ao depósito e, então, sai da simulação enviando uma mensagem para o agente `killer_agent` para ser removido da simulação. Quando o agente sai da simulação, o modelo verifica o próximo agente que irá entrar na simulação e assumir o controle do agente minerador para a simulação poder continuar sua execução.

Para adaptar um modelo JaCaMo à plataforma, este deve estar rodando via Gradle. A instalação via Gradle é uma das principais opções de instalação apontadas na documentação do JaCaMo (Boissier; Bordini; Hübner; Ricci; Santi, 2022a). Se o projeto ainda não estiver rodando via Gradle, precisa-se adaptar o modelo para isso. Os passos necessários para alcançar este objetivo são apresentados na documentação do JaCaMo (Boissier; Bordini; Hübner; Ricci; Santi, 2022b), na parte referente à instalação do JaCaMo via Gradle. No caso do Gold Miners, foram utilizadas implementações propostas nos tutoriais da ferramenta (Leite; Boissier; Bordini; Hübner; Ricci; Santi, 2022).

Depois da adaptação, é necessário incluir duas bibliotecas no arquivo Gradle de compilação: `JSON-simple` – para lidar com informações no formato JSON – e cliente HTTP – para lidar com solicitações HTTP.

A Figura 40 apresenta um exemplo das dependências de um arquivo Gradle, incluindo as originais do JaCaMo e as adicionadas para o funcionamento da arquitetura.

Figura 40 – Dependências adicionadas ao arquivo *build.gradle* do projeto JaCaMo.

build.gradle

```
dependencies {
    // Default from JaCaMo
    implementation group: 'org.twitter4j', name: 'twitter4j-async', version: '4.0.7'
    implementation group: 'org.twitter4j', name: 'twitter4j-stream', version: '4.0.7'
    implementation group: 'org.jacamo', name: 'jacamo', version: '1.0'
    implementation group: 'search', name: 'search'

    // New dependencies, from the architecture
    implementation group: 'com.googlecode.json-simple', name: 'json-simple', version: '1.1.1'
    implementation group: 'org.apache.httpcomponents', name: 'httpclient', version: '4.5.13'
}
```

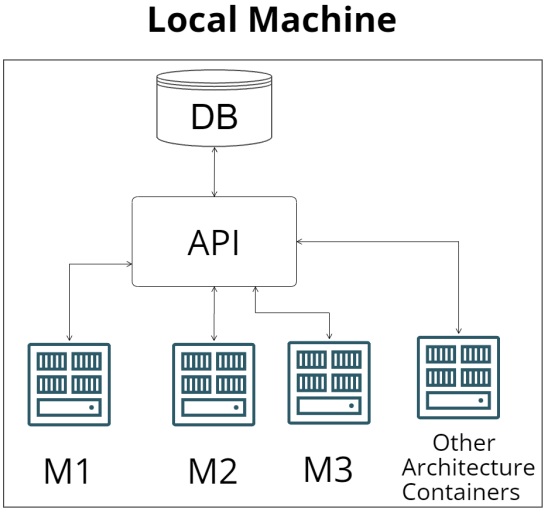
Fonte: autoria própria.

O último passo compreende incluir os agentes `killer_agent` e `check_file_agent` (com seus arquivos ASL) que lidam com a API. Com essas etapas concluídas, o modelo está pronto para se comunicar com a API e o programador pode usar as funções do sistema para enviar e receber agentes entre a API e o modelo. Exemplos de como adaptar e usar as funções estão disponíveis no GitHub (Lima; Aguiar, 2022).

## 4.3 Cenário de Simulação – Modo Local

Neste cenário de simulação, foi utilizada apenas uma máquina como *host* para todos os serviços (modo local), ou seja, todos os serviços do Docker rodam localmente e compartilham os recursos que são necessários através da rede isolada do Docker. A Figura 41 apresenta a configuração da arquitetura para este cenário, destacando que todos os contêineres estão sendo executados na máquina local.

Figura 41 – Cenário de simulação local, executando todos os contêineres na máquina local.




Fonte: autoria própria.

Após baixar a plataforma da documentação (disponível no GitHub (Lima; Aguiar, 2022)), o usuário inicia a simulação com o comando `docker-compose -f docker-compose-local.yaml up -d`. Este comando diz ao Docker para baixar e construir todas as imagens e contêineres necessários, conforme o arquivo `docker-compose-local.yaml`. Depois que tudo estiver definido e em execução, todos os *logs* do contêiner poderão ser acessados por meio do Docker Desktop ou do Docker CLI (do inglês *Command Line Input*). Enquanto a CLI é uma interface de linha de comando, o Docker Desktop é uma interface gráfica (é possível observar essa interface nos vídeos disponíveis em (Lima; Aguiar, 2024a), que serão explicados a seguir) que permite ao usuário verificar o que está acontecendo dentro dos contêineres, executar comandos nos contêineres e assim por diante.

Na simulação, os contêineres $\mathbf{M}_1$ e $\mathbf{M}_2$ (veja Figura 5, página 40) estão rodando simultaneamente instâncias do modelo Sugarscape do NetLogo (isoladas, uma em cada contêiner), e o contêiner $\mathbf{M}_3$ está executando o modelo Gold Miners, do JaCaMo. O modelo do JaCaMo roda indefinidamente, por ser um cenário controlado, com poucos nós de ouro no ambiente. Já nos modelos do NetLogo, foram criadas limitações artificiais nos modelos do NetLogo, para as simulações serem executadas até completarem 1000 ciclos, pois o modelo Sugarscape foi criado para que, trabalhando em uma faixa de equilíbrio, a simulação nunca acabe. Essa limitação se trata apenas para que o cenário de simulação tenha um fim, ou seja, trata-se de um limite do cenário de simulação, e não da ferramenta. Os três modelos usam a opção `auto-run`, ou seja, nenhum gatilho é necessário para começar a simulação, além da montagem do contêiner em ordem adequada.

No início da execução, os modelos $\mathbf{M}_1$ e $\mathbf{M}_2$ (NetLogo) solicitam ao **Register** para criar os agentes (cada modelo solicita 400 agentes inicialmente) e para receberem uma identificação que será utilizada pela arquitetura. No cenário de simulação projetado, apenas os modelos NetLogo solicitam novos agentes. O modelo $\mathbf{M}_3$ (JaCaMo), neste cenário, está recebendo e enviando agentes, mas não os criando na inicialização, embora seja possível.

Após a etapa inicial de criação e cadastro de todos os agentes, os contêineres de modelos $\mathbf{M}_1$, $\mathbf{M}_2$ e $\mathbf{M}_3$ continuam a execução dos seus modelos, juntamente com o processamento do **Router**. Sempre que um agente sai (morre) de algum dos modelos, ele é enviado para a **API** que fará a inserção no **DB**. Uma vez que uma nova informação está no **DB**, o **Router** consegue perceber que existem novos agentes a serem processados, então ele lê os agentes e os envia para um modelo conforme o tipo de roteamento (por padrão, aleatoriamente). Tanto os **Logs** quanto o contêiner **Interface** podem acessar o fluxo desta informação.

Além disso, cada contêiner mostra *logs* e informações sobre si. Por exemplo, no contêiner da API é possível verificar todas as requisições que a API recebe e processa, enquanto no contêiner JaCaMo o usuário pode verificar todos os agentes vivos, mentes, artefatos, entre outras características. Além dos *logs* do Docker, os blocos **DBMS** e **Interface** são acessíveis via navegador (endereço no arquivo `docker-compose`), permitindo ao usuário acessar a navegação de todos os agentes através dos contêineres de modelo e seus valores de parâmetros. O Apêndice C mostra exemplos dos *logs* que podem ser obtidos. Ressalta-se que novos tipos de *logs* podem ser gerados pelo usuário, como, por exemplo, de capturas de tela da execução, conforme a necessidade do projetista.

É importante ressaltar que ambos os modelos (NetLogo e JaCaMo) rodando nos contêineres têm acesso a todas as informações do agente. Por exemplo, o modelo do JaCaMo poderia acessar qualquer atributo do agente do NetLogo (açúcar, metabolismo e visão) para utilizá-lo na simulação. Por outro lado, o NetLogo também tem acesso ao arquivo ASL de cada agente e poderia utilizar estas informações, como uma crença, para algo útil na simulação.

Os cenários de simulação executados com e sem GUI (vídeos mostrando a execução disponíveis em (Lima; Aguiar, 2024a)) foram essenciais para verificar a factibilidade da arquitetura. Com estes testes, a plataforma permitiu dois modelos de SMA, provenientes da documentação original de cada plataforma (NetLogo e JaCaMo), passando de uma execução fechada de SMA para uma aberta com poucos passos extras.

É fundamental ressaltar que a necessidade de adaptações nos modelos não são substanciais, que implique descaracterizar os modelos originais para serem utilizados na arquitetura; ao contrário, essas adaptações são devidas porque esses modelos não estão originalmente preparados para a abertura e receberem (enviarem) agentes de

(para) fora. São fornecidos exemplos das funções dos gatilhos para enviar e receber informações para a arquitetura. Os usuários precisam adicioná-lo ao seu modelo, adaptar as informações que desejam enviar ou receber e usar os gatilhos quando quiserem que o modelo envie ou receba agentes. Ressaltamos que esse recurso é importante porque se objetiva reduzir a complexidade da adaptação do código.

Além disso, essas adaptações mostram que a arquitetura permite que plataformas de agentes completamente diferentes se comuniquem entre si. Por exemplo, a programação NetLogo usa a linguagem Logo, enquanto JaCaMo usa ASL, baseada em conceitos BDI (crenças, desejos e intenções, do inglês *Belief–desire–intention*). Embora sejam modelos heterogêneos, o agente carrega todas as suas informações enquanto se movimenta entre os contêineres. Assim, é possível utilizar as informações de um modelo em outro, mesmo em ferramentas distintas.

Este cenário executa os modelos dentro do Docker, como contêineres. Entretanto, é possível executar os modelos fora do Docker. Esta forma de execução da arquitetura permite com que seja possível, por exemplo, executar modelos com interfaces gráficas, ainda se utilizando da arquitetura para o conceito de abertura. Uma vez que a estrutura esteja montada e executada no Docker, é possível expor portas dos contêineres para a máquina *host*. Dessa forma, é possível, por exemplo, executar o modelo Gold Miners (ou qualquer outro) na máquina local usando GUI e ainda assim fazer com que o modelo participe de uma estrutura montada na arquitetura.

Além de expor a porta do contêiner da API, indica-se este novo *host* para os arquivos que se comunicam com a arquitetura (neste caso, são os arquivos `my_create_ag` e o `my_delete_ag`). Para facilitar esta integração, estes arquivos já contam com um parâmetro `using_docker`, onde o valor `true` define que o uso dos arquivos deve seguir a execução padrão, onde toda a arquitetura está dentro do Docker, enquanto o valor *false* define que a identificação do *host* difere da nomenclatura padrão. Além disso, também é necessário alterar um arquivo do modelo original que trata do ambiente, chamado `MiningPlanet.java`, pois este também trata da possibilidade de utilizar ou não a interface gráfica, através da variável `hasGUI`.

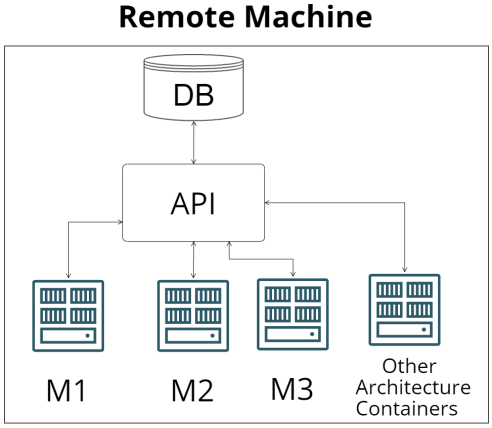
## 4.4 Cenário de Simulação – Modos Remoto e Híbrido

Para explorar uma nova capacidade da arquitetura, além do modo local (*host* único, máquina local), a arquitetura foi testada nas modalidades remoto e híbrido, utilizando uma máquina na nuvem (do inglês, *cloud*). Ressalta-se que, para estes testes na nuvem (modo remoto e híbrido), foi utilizada como base a mesma dinâmica do cenário apresentado no teste local, com três contêineres de modelo, executando duas cópias do modelo Sugarscape (independentes, iniciando com 400 agentes cada, limitando a simulação em 1000 ciclos) e uma cópia do Gold Miners. A diferença está na distribuição dos contêineres (incluindo os de modelo) da arquitetura.

Para o serviço de *cloud*, foi utilizada a Oracle Cloud (Jakóbczyk, 2020). A Oracle Cloud disponibiliza ferramentas de *cloud* em modalidades pagas. Além da modalidade paga, o usuário pode testar a plataforma completa por 30 dias gratuitamente. Entretanto, parte dos serviços contêm a modalidade *always free*, cujos recursos podem ser utilizados por tempo indeterminado gratuitamente.

De maneira similar ao cenário local, todos os serviços são montados/executados em uma mesma máquina *host*. Porém, ao invés do usuário executar isto na própria máquina, ele pode executar em uma máquina na *cloud*. Nos testes feitos utilizando o serviço da Oracle Cloud, foram utilizadas as máquinas disponibilizadas pela plataforma. Para tal execução, basta instalar o Docker na máquina, clonar o projeto do GitHub, e utilizar o arquivo correspondente à arquitetura da máquina remota (*docker-compose-local* ou *docker-compose-local-arm64*). Mais informações sobre a instalação do Docker na máquina utilizada para o cenário de simulação são apresentadas no Apêndice B. Por fim, a Figura 42 ilustra como os contêineres foram distribuídos, conforme descrito.

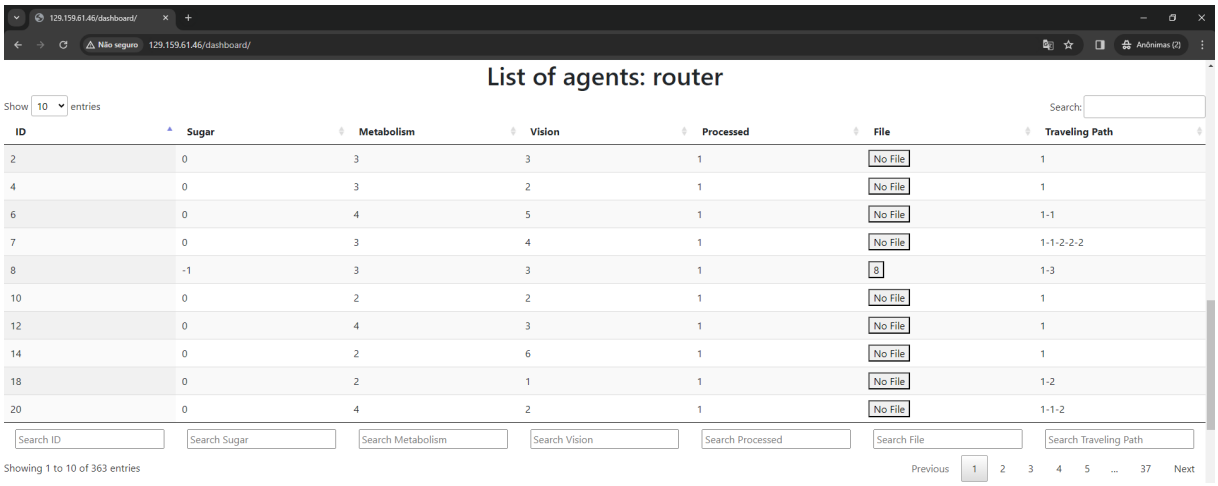
Figura 42 – Cenário de simulação remoto, executando todos os contêineres em uma máquina na nuvem.



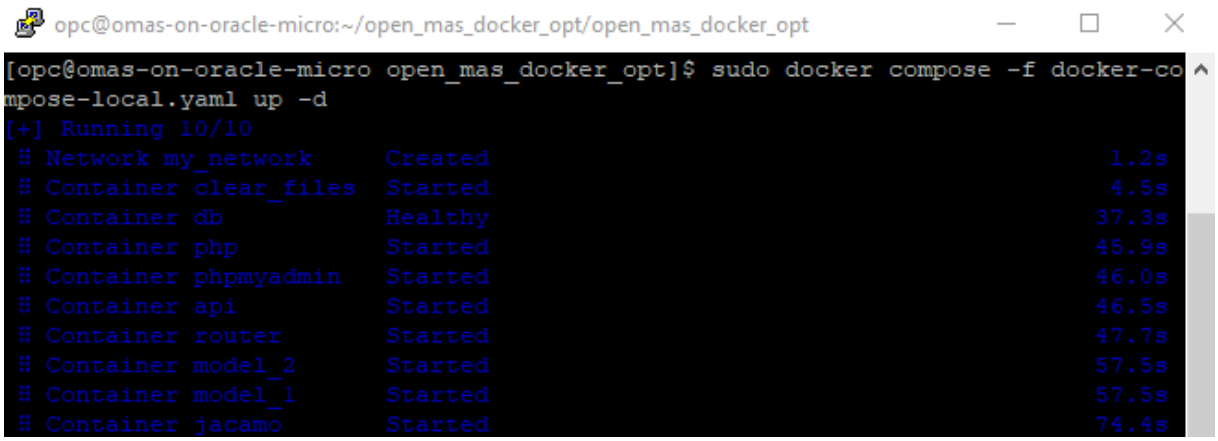

Fonte: autoria própria.

Esse arquivo conta com uma pequena adaptação nas imagens de alguns contêineres, pois as imagens originais foram desenvolvidas para utilizar arquiteturas x86/x64, enquanto algumas das máquinas disponíveis na versão gratuita da Oracle Cloud podem utilizar máquinas arm64, com instruções diferentes. A alteração é bem simples, bastando indicar o arquivo correspondente com a arquitetura da máquina que se está utilizando (x86/x64 ou arm64). A plataforma já conta com arquivos preparados para rodar em ambas arquiteturas. A Figura 43 ilustra um teste realizado na Oracle Cloud, onde é possível observar (a) parte da interface (utilizando o IP remoto) e (b) a conexão com a máquina para executar o cenário de simulação remoto, via SSH (do inglês *Secure Shell*).

Figura 43 – Representação de (a) parte da interface e (b) da conexão remota com máquina da Oracle Cloud, para um cenário de simulação remoto.




(a)

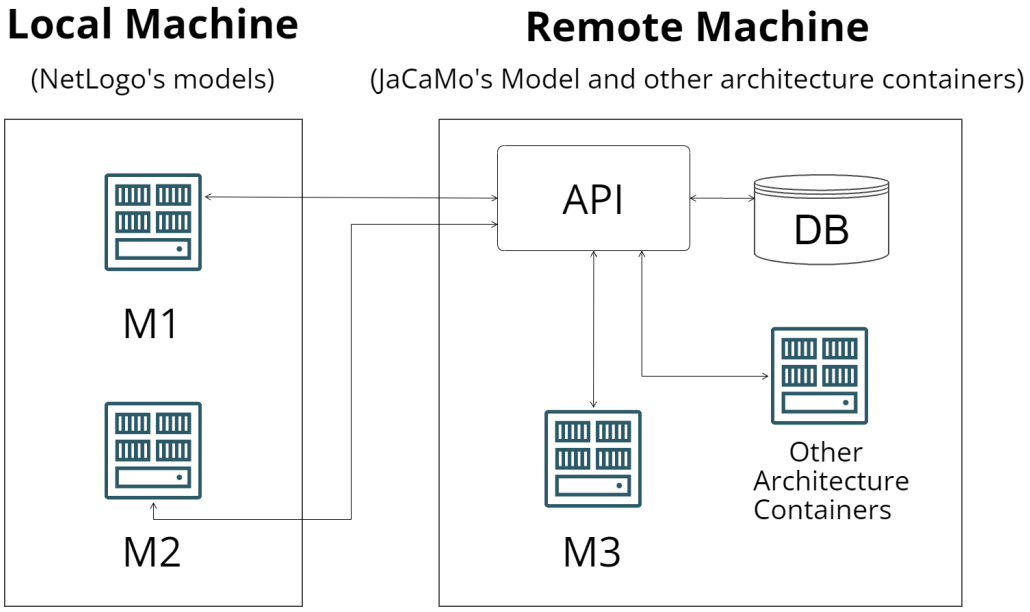



(b)

Fonte: autoria própria.

Ainda, foi criado um cenário de teste híbrido, no qual alguns serviços da arquitetura rodam na máquina local e outros na máquina na *cloud*. Mais especificamente, os contêineres dos dois modelos NetLogo executam localmente e as demais partes da arquitetura executam na *cloud*. Essa divisão das partes se dá por dois motivos: (i) algumas estruturas necessitam estar na mesma rede, como o modelo JaCaMo e a interface *web*, para ser possível acessar os arquivos ASL; e, (ii) os modelos NetLogo, no cenário de simulação citado, tendem a gastar mais recursos computacionais. A Figura 44 apresenta como a arquitetura ficou disposta para esse cenário. Na Figura é possível observar que os contêineres dos modelos do NetLogo ($\mathbf{M}_1$ e $\mathbf{M}_2$) são executados na máquina local, enquanto o modelo do JaCaMo ($\mathbf{M}_3$) e o restante da arquitetura é executado na máquina na nuvem. A comunicação entre as partes acontece através da **API**, sendo necessário expor a porta na máquina da nuvem e indicar o IP no **docker-compose** na máquina local.

Figura 44 – Cenário de simulação híbrido. Neste cenário, parte dos contêineres são executados na máquina local e outra parte em uma máquina na nuvem.




Fonte: autoria própria.

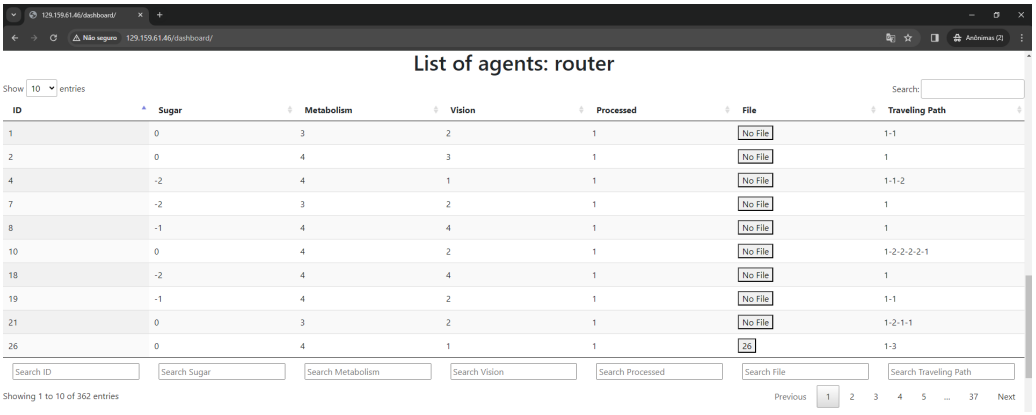
Da mesma forma que o modelo remoto, para executar este cenário, basta instalar o Docker na máquina, clonar o projeto do GitHub, e utilizar os arquivos respectivos da máquina *host* (`docker-compose-hybrid-local`) e do servidor remoto (`docker-compose-hybrid-server`). As mesmas adaptações das imagens são necessárias, pois a máquina remota da Oracle pode ter um processador com instruções arm64. Mais informações sobre a instalação do Docker na máquina utilizada para o cenário de simulação são apresentadas no Apêndice B.

Para que as duas partes da arquitetura se comuniquem, é necessário indicar que a máquina remota precisa expor a porta de comunicação com a API, por ser o meio onde os modelos irão enviar/receber os agentes. A única exposição da máquina é a porta de comunicação com a API (por padrão é a porta 5000). Além disso, é necessário indicar o endereço IP da máquina remota, para os modelos saberem como se comunicar com a API. Essa indicação é feita no arquivo `docker-compose-hybrid-local`, que roda na máquina local, através da variável de ambiente *host*.

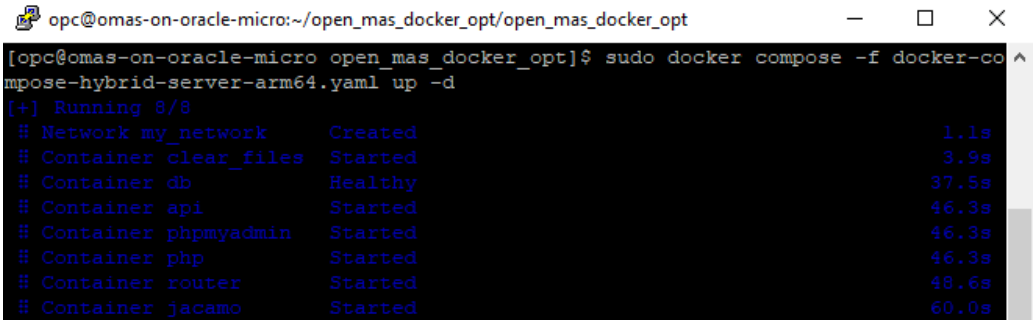
No momento de executar as duas partes separadamente, primeiro precisa-se montar a estrutura remota, para depois executar a parte local. Esse ordenamento é necessário pelo fato de que a estrutura local depende de o restante da arquitetura estar pronta para uso. A Figura 45 mostra o cenário híbrido, executando na máquina local os dois modelos NetLogo e o restante da arquitetura em uma máquina da Oracle Cloud. Na Figura 45 é apresentada parte da interface (a), a conexão com a máquina remota através do SSH (b) e o terminal local para execução da parte local (c).

Estes cenários (remoto/híbrido) mostram como a arquitetura pode ser estendida de diversas formas, como a própria migração de sistemas locais para Sistemas Multiagente que rodam em *cloud*. Além disso, a utilização da arquitetura no Docker permite com que esses serviços sejam distribuídos, podendo rodar em diversas máquinas diferentes, e também escaláveis, no caso de os modelos necessitarem sistemas mais robustos. É importante destacar que ao levar a plataforma para a *cloud*, os recursos utilizados para executar a arquitetura são os da máquina remota, e não da local. Isso permite que seja possível executar a arquitetura em computadores com menos poder computacional, caso tenham acesso à internet.

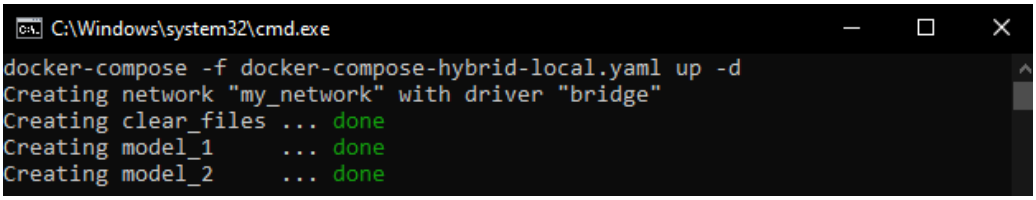
Figura 45 – Representação de (a) parte da interface, (b) da conexão remota com máquina da Oracle Cloud e (c) da execução dos modelos do NetLogo localmente, para um cenário híbrido de simulação.




(a)




(b)




(c)

Fonte: autoria própria.

# 5 CONSIDERAÇÕES FINAIS

Este trabalho apresentou as etapas para o desenvolvimento de uma proposta de arquitetura para auxiliar o desenvolvimento de Sistemas Multiagente Abertos. Primeiro, o Capítulo 2 apresentou a área de Sistemas Multiagente, passando por todos os conceitos e definições necessárias para entendimento do texto, além de apresentar trabalhos relacionados com este e a discussão das diferenças entre eles. Depois, o Capítulo 3 apresentou em detalhes a abordagem proposta, mostrando a arquitetura, suas etapas e funções. Por fim, o Capítulo 4 mostrou os resultados obtidos ao realizar as simulações dos cenários propostos, enfatizando o estudo de caso desenvolvido no protótipo da arquitetura, sendo de grande importância para verificar a viabilidade da implementação.

A abordagem proposta apresenta um ambiente que facilita o desenvolvimento de Sistemas Multiagente Abertos utilizando contêineres do Docker. Essa arquitetura permite a migração de agentes entre modelos que podem rodar em cenários heterogêneos. Além disso, a estrutura permite que o código seja executado dentro de contêineres que contenham a mesma estrutura de operação na qual foram desenvolvidos, contornando desafios comuns a programas que rodam na fase de desenvolvimento mas que, entretanto, na fase de produção apresentam problemas.

A arquitetura tem o potencial de facilitar a etapa de união entre os modelos. O Docker permite que os modelos sejam executados em diferentes cenários, com diferentes parâmetros, em diferentes plataformas, usando diferentes linguagens de desenvolvimento, que podem ser executados em diferentes sistemas operacionais.

A implementação apresentada confirma a viabilidade de testes com plataformas de agentes diferentes, como foi o caso do NetLogo (agentes logo) e do JaCaMo (agentes BDI). Embora o cenário de simulação utilize modelos clássicos e simplificados, é possível observar como os atributos dos agentes podem ser compartilhados entre modelos de diferentes plataformas, através da disponibilização de todos os dados em tempo real durante a simulação.

Também foi possível verificar que, através da ferramenta proposta, a troca de agentes entre modelos pode acontecer com a adição de poucas linhas ao código original dos modelos, reduzindo assim parte da complexidade dos SMAA, segundo a escala do grau de abertura. Para realizar a integração de modelos com a arquitetura, limita-se a adicionar as funções de comunicação e utilizá-las (disparar os gatilhos) conforme a necessidade do modelo de enviar e/ou receber agentes.

Além disso, a arquitetura permite que a migração de agentes entre modelos ocorra em tempo de execução. A motivação para essa migração de um agente de um sistema para outro pode ser diferente, sendo de escolha do desenvolvedor, como falhas de execução, vontade própria ou algum gatilho.

Em relação às limitações da arquitetura, duas coisas podem ser apontadas: *overhead* e GUI. Embora o Docker tenha menos sobrecarga que as VMs, não é zero. Portanto, a *engine* do Docker pode utilizar mais recursos em comparação aos modelos nativos. No entanto, em nossos testes em uma máquina em nuvem com poucos recursos, o impacto foi mínimo. Já para a limitação relacionada a GUI, projetamos contêineres do Docker para uso em linha de comando, embora existam opções de utilizar modelos com interface gráfica. Simulações que requerem GUI devem ser executadas localmente, com comunicação com a arquitetura via exposição de porta API, conforme demonstrado em um de nossos cenários de simulação.

Em trabalhos futuros, é possível explorar novos gatilhos que fazem os agentes mudarem de modelo, como limites geográficos entre modelos e agentes que vivem em modelos paralelos. Limites geográficos podem ser entendidos nas situações quando o agente chega na borda (limite geográfico) do ambiente, ele passa para o outro modelo. Modelos paralelos são modelos nos quais o mesmo agente participa de mais de um modelo simultaneamente, mas cada modelo trata de certos atributos do agente ou evolui o agente em contextos diferentes.

Além disso, também é possível explorar novos tipos de roteamento. A implementação atual da abordagem conta com tipos de roteamento simples (aleatório e sequencial), mas a arquitetura é flexível e está preparada para receber novas implementações. Novos tipos de roteamento mais complexos podem rotear os agentes de forma mais sofisticada, utilizando informações como os dados das passagens dos agentes, informações dos próprios modelos, retreinar agentes mais promissores, dentre outros. As possíveis novas implementações de roteamento devem ser inseridas no código que o contêiner do *Router* executa (inserção ou substituição do código).

Também tem-se a intenção de tornar o código da plataforma *open-source*. A abertura do código irá permitir com que outros programadores possam contribuir com códigos que irão aprimorar a arquitetura, tornando a plataforma cada vez mais robusta.

Além disso, os testes com a Oracle Cloud (porém não limitado apenas a esta, por existirem outras plataformas de nuvem como a AWS Docker) permitiram verificar a viabilidade da arquitetura ser executada na nuvem, por conta de o desenvolvimento ser feito com base no Docker. Isso possibilita a portabilidade de aplicações e amplia o universo de aplicações na arquitetura.

Finalmente, a implementação da arquitetura atual suporta duas plataformas de agentes relevantes, NetLogo e JaCaMo. No entanto, é de interesse em evoluir a arquitetura para oferecer suporte a outras plataformas de agentes, como JADE (baseado em Java) ou Mesa (baseado em Python). Para novas integrações, é necessário adicionar cada nova plataforma em um contêiner separado e implementar a integração com a API, para permitir a transferência de agentes. Nos exemplos citados, JADE e Mesa, é possível aproveitar o código já desenvolvido (por utilizarem como base o Java e o Python, respectivamente), mas a adição de plataformas que utilizam outras linguagens de programação também é possível, bastando implementar o código de comunicação intermediário.

Por fim, além deste documento, o estudo desenvolvido pela tese gerou três publicações, sendo elas:

| | |
|---|---|
| **Título:** | Towards a Docker-based architecture for open multi-agent systems |
| **Autores:** | Gustavo Lameirão de Lima, Marilton Sanchotene de Aguiar |
| **Evento:** | IAES International Journal of Artificial Intelligence (IJ-AI) |
| **Tipo de publicação:** | Revista |
| **Ano:** | 2024 |
| **Qualis:** | A4 (baseado no índice Scopus) |
| **Link:** | `https://ijai.iaescore.com/index.php/IJAI/article/view/22652` |
| **Referência completa:** | (Lima; Aguiar, 2024b) |

| | |
|---|---|
| **Título:** | Exploring Heterogeneous Open Multi-Agent Systems on Cloud Using a Docker-Based Architecture |
| **Autores:** | Gustavo Lameirão de Lima, Marilton Sanchotene de Aguiar |
| **Evento:** | 2023 IEEE Symposium Series on Computational Intelligence (SSCI) |
| **Tipo de publicação:** | Conferência |
| **Ano:** | 2023 |
| **Qualis:** | A2 |
| **Link:** | `https://ieeexplore.ieee.org/abstract/document/10371883` |
| **Referência completa:** | (Lima; Aguiar, 2023a) |

| | |
|---|---|
| **Título:** | Uma arquitetura baseada em Docker<br>para Sistemas Multiagentes Abertos |
| **Autores:** | Gustavo Lameirão de Lima, Marilton Sanchotene de Aguiar |
| **Evento:** | XVII Workshop-Escola de Sistemas de Agentes,<br>seus Ambientes e Aplicações (WESAAC 2023) |
| **Tipo de publicação:** | Conferência |
| **Ano:** | 2023 |
| **Qualis:** | - |
| **Link:** | `https://doi.org/10.5281/zenodo.8329081` |
| **Referência completa:** | (Lima; Aguiar, 2023b) |

# REFERÊNCIAS


ACHARYA, J. N.; SUTHAR, A. C. Docker Container Orchestration Management: A Review. In: INTERNATIONAL CONFERENCE ON INTELLIGENT VISION AND COMPUTING (ICIVC 2021), 2022, Oman. **Proceedings...** Springer International Publishing, 2022. p.140–153.

ARTIKIS, A. Dynamic specification of open agent systems. **Journal of Logic and Computation**, UK, v.22, n.6, p.1301–1334, 07 2012.

ARTIKIS, A.; SERGOT, M. Executable specification of open multi-agent systems. **Logic Journal of the IGPL**, UK, v.18, n.1, p.31–65, 2010.

ARTIKIS, A.; SERGOT, M.; PITT, J. An executable specification of a formal argumentation protocol. **Artificial Intelligence**, Netherlands, v.171, n.10, p.776–804, 2007. Argumentation in Artificial Intelligence.

ARTIKIS, A.; SERGOT, M.; PITT, J. Specifying norm-governed computational societies. **ACM Trans. Comput. Logic**, New York, NY, USA, v.10, n.1, jan 2009.

ARTIKIS, A.; SERGOT, M.; PITT, J.; BUSQUETS, D.; RIVERET, R. **Specifying and Executing Open Multi-agent Systems**. Germany: Springer International Publishing, 2016. v.30, p.197–212.

BOISSIER, O.; BORDINI, R. H.; HÜBNER, J. F.; RICCI, A.; SANTI, A. Multi-agent oriented programming with JaCaMo. **Science of Computer Programming**, Netherlands, v.78, n.6, p.747–761, 2013.

BOISSIER, O.; BORDINI, R. H.; HÜBNER, J. F.; RICCI, A.; SANTI, A. **JaCaMo's GitHub Page**. 2022. Disponível em: `https://github.com/jacamo-lang/jacamo`. Acesso em: 28 fev. 2024.

BOISSIER, O.; BORDINI, R. H.; HÜBNER, J. F.; RICCI, A.; SANTI, A. **JaCaMo's GitHub Page - JaCaMo Installation**. 2022. Disponível em: `https://github.com/jacamo-lang/jacamo/blob/master/doc/install.adoc`. Acesso em: 28 fev. 2024.

BORDINI, R. H.; HÜBNER, J. F.; TRALAMAZZA, D. M. Using Jason to implement a team of gold miners. In: INTERNATIONAL WORKSHOP ON COMPUTATIONAL LOGIC IN MULTI-AGENT SYSTEMS, 2007, Japan. **Proceedings...** Springer, 2007. p.304–313.

BORDINI, R. H.; HÜBNER, J. F.; WOOLDRIDGE, M. **Programming multi-agent systems in AgentSpeak using Jason**. USA: John Wiley & Sons, 2007.

CHEBOUT, M. S.; MOKHATI, F.; BADRI, M. NC4OMAS: A Norms-based Approach for Open Multi-Agent Systems Controllability. In: INTERNATIONAL CONFERENCE ON AGENTS AND ARTIFICIAL INTELLIGENCE (ICAART 2022), 14., 2022, Online. **Proceedings...** SCITEPRESS – Science and Technology Publications: Lda, 2022. p.164–171.

COLLA, S.; GALLAND, C. M. de; HENDRICKX, J. M. Decentralized Estimation in Open Multi-Agent Systems. In: EUROPEAN CONTROL CONFERENCE, 2021, Virtual Conference. **Proceedings...** IEEE, 2021.

DÄHLING, S.; RAZIK, L.; MONTI, A. Enabling scalable and fault-tolerant multi-agent systems by utilizing cloud-native computing. **Autonomous Agents and Multi-Agent Systems**, Germany, v.35, n.1, p.1–27, 2021.

DALPIAZ, F.; CHOPRA, A. K.; GIORGINI, P.; MYLOPOULOS, J. Adaptation in Open Systems: Giving Interaction Its Rightful Place. In: CONCEPTUAL MODELING – ER 2010, 2010, Berlin, Heidelberg. **Proceedings...** Springer Berlin Heidelberg, 2010. p.31–45.

DASHTI, Z. A. Z. S.; OLIVA, G.; SEATZU, C.; GASPARRI, A.; FRANCESCHELLI, M. Distributed mode computation in open multi-agent systems. **IEEE Control Systems Letters**, Italy, v.6, p.3481–3486, 2022.

DEMAZEAU, Y.; COSTA, A. R. Populations and organizations in open multi-agent systems. In: NATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED AI (PDAI'96), 1., 1996. **Proceedings...** University of Hyderabad, 1996. p.1–13.

DOCKER, I. **What is a Container? | Docker**. 2021. Disponível em: `https://www.docker.com/resources/what-container`. Acesso em: 28 fev. 2024.

DOCKER, I. **Docker overview | Docker Docs**. 2021. Disponível em: `https://docs.docker.com/get-started/overview/`. Acesso em: 28 fev. 2024.

ECK, A.; SOH, L.-K.; DOSHI, P. Decision making in open agent systems. **AI Magazine**, USA, v.44, n.4, p.508–523, 2023.

EIJK, R. M. van; BOER, F. S. de; VAN DER HOEK, W.; MEYER, J.-J. C. Open multi-agent systems: Agent communication and integration. In: INTERNATIONAL WORKSHOP ON AGENT THEORIES, ARCHITECTURES, AND LANGUAGES, 1999. **Proceedings...** Springer, 1999. p.218–232.

EPSTEIN, J. M.; AXTELL, R. **Growing artificial societies**: social science from the bottom up. USA: The MIT Press, 1996.

FECKO, M. A.; LOTT, C. M. Lessons learned from automating tests for an operations support system. **Software: Practice and Experience**, USA, v.32, n.15, p.1485–1506, 2002.

FERBER, J. **Multi-Agent Systems**: An Introduction to Distributed Artificial Intelligence. 1st.ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

GALLAND, C. M. de; HENDRICKX, J. M. Fundamental performance limitations for average consensus in open multi-agent systems. **IEEE Transactions on Automatic Control**, USA, v.68, n.2, p.646–659, 2022.

GALLAND, C. M. de; VIZUETE, R.; HENDRICKX, J. M.; PANTELEY, E.; FRASCA, P. Random coordinate descent for resource allocation in open multi-agent systems. **arXiv preprint arXiv:2205.10259**, USA, 2022.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns**: elements of reusable object-oriented software. Germany: Pearson Deutschland GmbH, 1995.

GONZALEZ-PALACIOS, J.; LUCK, M. Towards compliance of agents in open multi-agent systems. In: INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING FOR LARGE-SCALE MULTI-AGENT SYSTEMS, 2006. **Proceedings...** Springer, 2006. p.132–147.

GOOGLE. **O que são contêineres? | Google Cloud**. 2024. Disponível em: `https://cloud.google.com/learn/what-are-containers?hl=pt-br#section-1`. Acesso em: 28 fev. 2024.

GOVONI, D. **Java application frameworks**. USA: John Wiley & Sons, Inc., 1999.

GRINBERG, M. **Flask web development**: developing web applications with python. USA: O'Reilly Media, Inc., 2018.

HARO, R. S. V. **Contributions to open multi-agent systems**: consensus, optimization and epidemics. 2022. Thesis (Doctorate in Computer Science) — Université Paris-Saclay.

HATTAB, S.; CHAARI, W. L. A generic model for representing openness in multi-agent systems. **The Knowledge Engineering Review**, Cambridge, United Kingdom, v.36, 2021.

HENDRICKX, J. M.; MARTIN, S. Open multi-agent systems: Gossiping with random arrivals and departures. In: IEEE ANNUAL CONFERENCE ON DECISION AND CONTROL (CDC), 56., 2017, Australia. **Proceedings...** IEEE, 2017. p.763–768.

HEWITT, C. Open information systems semantics for distributed artificial intelligence. **Artificial intelligence**, Netherlands, v.47, n.1-3, p.79–106, 1991.

HOUHAMDI, Z.; ATHAMENA, B. Collaborative Team Construction in Open Multi-Agents System. In: INTERNATIONAL ARAB CONFERENCE ON INFORMATION TECHNOLOGY (ACIT), 21., 2020, Egypt. **Proceedings...** IEEE, 2020. p.1–7.

HU, Y.; LIU, W.; LAN, J.; ZHANG, J. Forming Multi-agents Collaborative Communities in Overlapping Communities. In: INTERNATIONAL CONFERENCE ON ROBOTICS, CONTROL AND AUTOMATION ENGINEERING (RCAE), 4., 2021, China. **Proceedings...** IEEE, 2021. p.114–120.

HUBNER, J. F.; SICHMAN, J. S.; BOISSIER, O. Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels. **International Journal of Agent-Oriented Software Engineering**, Switzerland, v.1, n.3-4, p.370–395, 2007.

JAKÓBCZYK, M. T. **Practical Oracle Cloud Infrastructure**. USA: Apress, 2020.

JAMROGA, W.; MESKI, A.; SZRETER, M. Modularity and Openness in Modeling Multi-Agent Systems. **Electronic Proceedings in Theoretical Computer Science**, Waterloo, Australia, v.119, p.224–239, Jul 2013.

JETHVA, H. **How to Install Docker and Docker Compose on Oracle Linux**. 2023. Disponível em: `https://www.atlantic.net/dedicated-server-hosting/how-to-install-docker-and-docker-compose-on-oracle-linux/`. Acesso em: 28 fev. 2024.

JIANG, S.; ZHANG, Y. Coordination on Open Double-Integrator Multi-Agent Systems. In: CHINESE CONTROL CONFERENCE (CCC), 42., 2023, China. **Proceedings...** IEEE, 2023. p.5687–5692.

KONDYLIDIS, N.; TIDDI, I.; TEIJE, A. t. Establishing Shared Query Understanding in an Open Multi-Agent System. **arXiv preprint arXiv:2305.09349**, USA, 2023.

LEITE, J.; BOISSIER, O.; BORDINI, R. H.; HÜBNER, J. F.; RICCI, A.; SANTI, A. **JaCaMo - Tutorials - Gold Miners**. 2022. Disponível em: https://github.com/jacamo-lang/jacamo/tree/main/doc/tutorials/gold-miners. Acesso em: 28 fev. 2024.

LI, J.; WILENSKY, U. **NetLogo Sugarscape 2 Constant Growback model**. 2009. Disponível em: https://ccl.northwestern.edu/netlogo/models/Sugarscape2ConstantGrowback. Acesso em: 28 fev. 2024.

LIMA, G. L. de; AGUIAR, M. S. de. **Architecture's GitHub Page**. 2022. Disponível em: https://github.com/GustavoLLima/open_mas_docker_opt. Acesso em: 28 fev. 2024.

LIMA, G. L. de; AGUIAR, M. S. de. Exploring Heterogeneous Open Multi-Agent Systems on Cloud Using a Docker-Based Architecture. In: IEEE SYMPOSIUM SERIES ON COMPUTATIONAL INTELLIGENCE (SSCI), 2023, Mexico. **Proceedings...** IEEE, 2023. p.842–849.

LIMA, G. L. de; AGUIAR, M. S. de. Uma arquitetura baseada em Docker para Sistemas Multiagentes Abertos. In: XVII WORKSHOP-ESCOLA DE SISTEMAS DE AGENTES, SEUS AMBIENTES E APLICAçõES - WESAAC 2023, 2023, Brazil. **Anais...** UFPEL, 2023. v.XVII, p.20–31.

LIMA, G. L. de; AGUIAR, M. S. de. **Videos from the simulation scenario, with GUI ON and OFF**. 2024. Disponível em: https://github.com/GustavoLLima/open_mas_docker_opt/tree/main/videos. Acesso em: 28 fev. 2024.

LIMA, G. L. de; AGUIAR, M. S. de. Towards a Docker-based architecture for open multi-agent systems. **IAES International Journal of Artificial Intelligence**, Indonesia, v.13, n.1, p.45–56, 2024.

NAKAMURA, T.; HAYASHI, N.; INUIGUCHI, M. Cooperative Learning for Adversarial Multi-Armed Bandit on Open Multi-Agent Systems. **IEEE Control Systems Letters**, Italy, v.7, 2023.

NORVIG, P.; RUSSELL, S. **Inteligência Artificial**: Tradução da 3a Edição. Brazil: Elsevier, 2014. v.1.

PAUROBALLY, S.; CUNNINGHAM, J.; JENNINGS, N. R. Ensuring consistency in the joint beliefs of interacting agents. In: INTERNATIONAL JOINT CONFERENCE ON AUTONOMOUS AGENTS & MULTIAGENT SYSTEMS, AAMAS 2003, JULY 14-18, 2003, MELBOURNE, VICTORIA, AUSTRALIA, PROCEEDINGS, 2., 2003, Melbourne, Australia. **Proceedings...** ACM, 2003. p.662–669.

PERLES, A.; CRASNIER, F.; GEORGÉ, J.-P. AMAK - A Framework for Developing Robust and Open Adaptive Multi-agent Systems. In: INTERNATIONAL CONFERENCE ON PRACTICAL APPLICATIONS OF AGENTS AND MULTI-AGENT SYSTEMS – HIGHLIGHTS OF PRACTICAL APPLICATIONS OF AGENTS, MULTI-AGENT SYSTEMS, AND COMPLEXITY: THE PAAMS COLLECTION, 2018, Spain. **Proceedings...** Springer International Publishing, 2018. p.468–479.

PERRONE, G.; ROMANO, S. P. The docker security playground: a hands-on approach to the study of network security. In: PRINCIPLES, SYSTEMS AND APPLICATIONS OF IP TELECOMMUNICATIONS (IPTCOMM), 2017, USA. **Proceedings...** IEEE, 2017. p.1–8.

PFEIFER, V.; PASSINI, W. F.; DORANTE, W. F.; GUILHERME, I. R.; AFFONSO, F. J. A multi-agent approach to monitor and manage container-based distributed systems. **IEEE Latin America Transactions**, USA, v.20, n.1, p.82–91, 2021.

PITT, J.; KAMARA, L.; SERGOT, M.; ARTIKIS, A. Voting in multi-agent systems. **The Computer Journal**, UK, v.49, n.2, p.156–170, 2006.

PITT, J.; MAMDANI, A.; CHARLTON, P. The open agent society and its enemies: a position statement and research programme. **Telematics and Informatics**, Netherlands, v.18, n.1, p.67–87, 2001.

PITT, J.; SCHAUMEIER, J.; ARTIKIS, A. Axiomatization of socio-economic principles for self-organizing institutions: Concepts, experiments and challenges. **ACM Transactions on Autonomous and Adaptive Systems (TAAS)**, USA, v.7, n.4, p.1–39, 2012.

RAMIREZ, W. A. L.; FASLI, M. Integrating NetLogo and Jason: a disaster-rescue simulation. In: COMPUTER SCIENCE AND ELECTRONIC ENGINEERING (CEEC), 9., 2017, UK. **Proceedings...** IEEE, 2017. p.213–218.

REIS, L. P. **Coordenação em Sistemas Multi-Agente**: Aplicações na Gestão Universitária e Futebol Robótico. 2003. 451p. Thesis (PhD in Electrical and Computer Engineering) — University of Porto, Faculty of Engineering, Porto, Portugal.

RESTREPO, E.; LORÍA, A.; SARRAS, I.; MARZAT, J. Consensus of Open Multi-agent Systems over Dynamic Undirected Graphs with Preserved Connectivity and Collision Avoidance. In: CONFERENCE ON DECISION AND CONTROL (CDC), 61., 2022, Mexico. **Proceedings...** IEEE, 2022. p.4609–4614.

RICCI, A.; PIUNTI, M.; VIROLI, M. Environment programming in multi-agent systems: an artifact-based perspective. **Autonomous Agents and Multi-Agent Systems**, Germany, v.23, p.158–192, 2011.

SAMMI, R.; MASOOD, I.; JABEEN, S. A framework to assure the quality of sanity check process. In: SOFTWARE ENGINEERING AND COMPUTER SYSTEMS: SECOND INTERNATIONAL CONFERENCE, ICSECS 2011, KUANTAN, PAHANG, MALAYSIA, JUNE 27-29, 2011, PROCEEDINGS, PART III 2, 2011, Malaysia. **Anais...** Springer, 2011. p.143–150.

SANKA, S. **Docker Part 1 – Install Docker CE and Docker Compose in Oracle Linux Server 8.6 – OCI-IAAS**. 2023. Disponível em: `https://bit.ly/4bX1ap7`. Acesso em: 28 fev. 2024.

SERGOT, M. Modelling Unreliable and Untrustworthy Agent Behaviour. In: MONITORING, SECURITY, AND RESCUE TECHNIQUES IN MULTIAGENT SYSTEMS, 2005, Germany. **Anais...** Springer Berlin Heidelberg, 2005. p.161–177.

SINGH, M. P.; CHOPRA, A. K. Programming multiagent systems without programming agents. In: INTERNATIONAL WORKSHOP ON PROGRAMMING MULTI-AGENT SYSTEMS, 2009, Budapest, Hungary. **Proceedings...** Springer, 2009. p.1–14.

SOLMS, F. What is software architecture? In: SAICSIT '12 PROCEEDINGS OF THE SOUTH AFRICAN INSTITUTE FOR COMPUTER SCIENTISTS AND INFORMATION TECHNOLOGISTS CONFERENC, 2012, South Africa. **Anais...** Association for Computing Machinery, 2012. p.363–373.

STONE, P.; VELOSO, M. Multiagent systems: A survey from a machine learning perspective. **Autonomous Robots**, Germany, v.8, n.3, p.345–383, 2000.

TAILLANDIER, P.; GAUDOU, B.; GRIGNARD, A.; HUYNH, Q.-N.; MARILLEAU, N.; CAILLOU, P.; PHILIPPON, D.; DROGOUL, A. Building, composing and experimenting complex spatial models with the GAMA platform. **GeoInformatica**, Germany, v.23, n.2, p.299–322, Dec. 2018.

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

UEZ, D. M. **Open AEOlus**: um método para especificação de sistemas multiagentes abertos. 2018. Thesis (PhD in Automation and Systems Engineering) — Federal University of Santa Catarina, Technological Center, Florianópolis, Brazil.

WILENSKY, U. **NetLogo. Evanston, IL**: Center for connected learning and computer-based modeling, Northwestern University. 1999.

WILENSKY, U. **NetLogo Py Extension**. 2022. Disponível em: `https://ccl.northwestern.edu/netlogo/docs/py.html`. Acesso em: 28 fev. 2024.

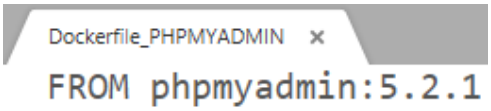
WOOLDRIDGE, M. **Introduction to multiagent systems**. EUA: John Wiley & Sons, Inc., 2001.

XUE, M.; TANG, Y.; REN, W.; QIAN, F. Stability of multi-dimensional switched systems with an application to open multi-agent systems. **Automatica**, Netherlands, v.146, p.110644, 2022.

# Apêndices

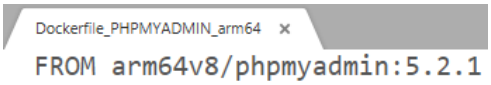
# APÊNDICE A – Dockerfile Utilizados

Figura 46 – Exemplo de arquivo *Dockerfile* para contêineres do PHPMyAdmin.




Fonte: autoria própria.

Figura 47 – Exemplo de arquivo *Dockerfile* para contêineres do PHPMyAdmin para a arquitetura arm64.

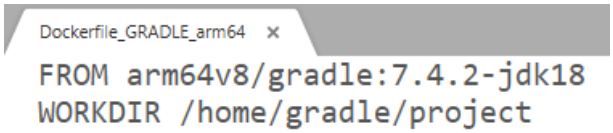



Fonte: autoria própria.

Figura 48 – Exemplo de arquivo *Dockerfile* para contêineres do JaCaMo (utilizando o Gradle).

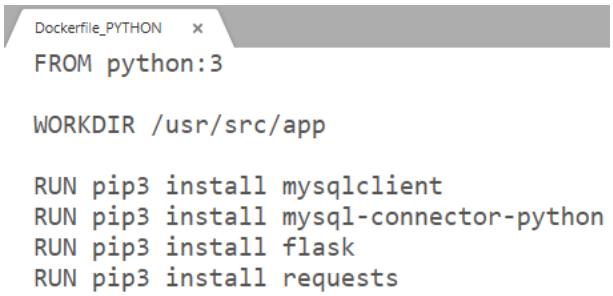

Fonte: autoria própria.

Figura 49 – Exemplo de arquivo *Dockerfile* para contêineres do JaCaMo (utilizando o Gradle) na arquitetura arm64.



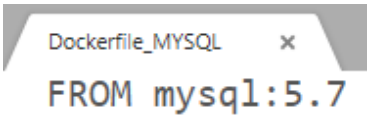

Fonte: autoria própria.

Figura 50 – Exemplo de arquivo *Dockerfile* para contêineres que utilizam o Python.



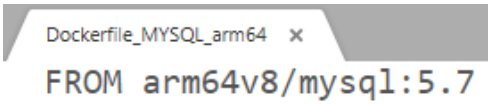

Fonte: autoria própria.

Figura 51 – Exemplo de arquivo *Dockerfile* para contêineres que utilizam o MySQL.



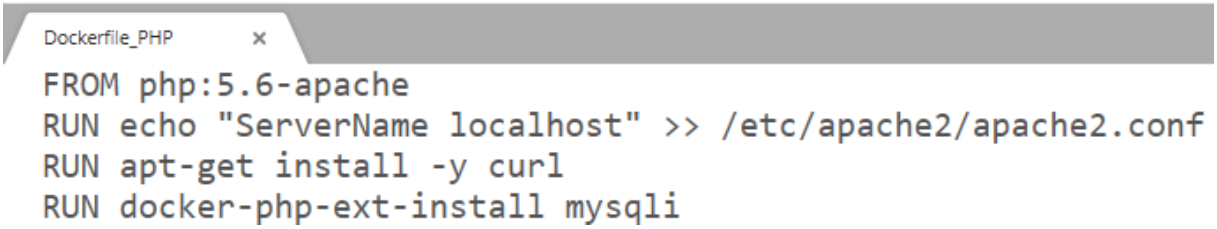

Fonte: autoria própria.

Figura 52 – Exemplo de arquivo *Dockerfile* para contêineres que utilizam o MySQL na arquitetura arm64.




Fonte: autoria própria.

Figura 53 – Exemplo de arquivo *Dockerfile* para contêineres que utilizam o PHP (através do Apache).




Fonte: autoria própria.

# APÊNDICE B – Instalação do Docker na VM da Oracle

A instalação utiliza como base nos tutoriais apresentados em Jethva (2023) e Sanka (2023), que trata da instalação do Docker em máquina da Oracle, que utilizam Oracle OS. A forma da instalação pode variar conforme o sistema operacional e a máquina utilizada. Para este caso, os passos são:

- Instalar e executar o Docker
  - `sudo dnf update -y`
  - `sudo dnf config-manager --add-repo=https://download.docker.com/linux/centos/docker-ce.repo`
  - `sudo dnf install docker-ce docker-ce-cli containerd.io docker-compose-plugin`
  - `sudo systemctl enable docker`
  - `sudo systemctl start docker`
- Instalar o git e clonar o projeto
  - `sudo dnf install git`
  - `git clone https://github.com/GustavoLLima/open_mas_docker_opt`
- Abrir a pasta do projeto e usar o comando `docker-compose -f arquivo up -d`, onde o arquivo é o nome do cenário a ser testado, seja ele o remoto ou o servidor do híbrido.

# APÊNDICE C – Exemplos de Logs obtidos durante a simulação

*Logs* podem ser gerados de diversas formas, desde os que apresentam informações de algum contêiner que podem ser capturados pelo *docker logs*, salvando a saída da execução em um arquivo *txt*, quanto os gerados nativamente, como no arquivo *mas.log* gerado pelo JaCaMo. Como um complemento aos *logs* já apresentados anteriormente, as Figuras a seguir apresentam outros exemplos de *logs* que podem ser obtidos em simulações.

Figura 54 – Parte do *log* obtido no arquivo de texto salvo pela arquitetura sobre o contêiner do JaCaMo, a partir do `docker logs`.

```
Starting a Gradle Daemon, 1 busy Daemon could not be reused, use --status for details
<-------------> 0% INITIALIZING [397ms]
> IDLE
2024-02-05T17:59:58.179237000Z Starting a Gradle Daemon, 2 busy Daemons could not be reused, use --status for details

> Task :run
CArtAgO Http Server running on http://172.19.0.2:3273
Jason Http Server running on http://172.19.0.2:3272
[Cartago] Workspace mining created.
[Cartago] artifact m2view: mining.MiningPlanet(7,1) at mining created.
All agents at the moment:
[killer_agent] I'm a killer agent
[leader, killer_agent, check_file_agent]
3 agents and no df, checking if there is a new agent on API to be created...
New agents
ID: 472
Stats: [0 4 1]
Path: 2-3-1
Sugar: 0
Metabolism: 4
Vision: 1
Creating agent with this beliefs: agent_id(472),path("2-3-1"), sugar(0), metabolism(4), vision(1)
asl_file_name: list/472.asl
Agent created by custom file
[472] Hello there. I'm a regular agent
[472] R: 0.3619893330067251
[472] Sugar 0
[472] Metabolism 4
[472] Vision 1
[472] Saved my information on file. Sending message to remove agent from simulation
[killer_agent] I've received a message to kill agent 472
[killer_agent] Killing agent 472
Executing JAVA custom code - delete
```

Fonte: autoria própria.

Figura 55 – Parte do *log* gerado nativamente pelo JaCaMo, no arquivo *mas.log*.

```
[Cartago] Workspace mining created.
[Cartago] artifact m2view: mining.MiningPlanet(7,1) at mining created.
[killer_agent] I'm a killer agent
[472] Hello there. I'm a regular agent
[472] R: 0.3619893330067251
[472] Sugar 0
[472] Metabolism 4
[472] Vision 1
[472] Saved my information on file. Sending message to remove agent from simulation
[killer_agent] I've received a message to kill agent 472
[killer_agent] Killing agent 472
[472] focusing on artifact m2view (at workspace /main/mining) using namespace default
[killer_agent] This agent has being removed from the simulation: 472
```

Fonte: autoria própria.