UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Instituto de Ciência e Tecnologia
Câmpus de Sorocaba

JOÃO PHELYPE DA SILVA CAVALCANTE

Conteinerização e gerenciamento de aplicações industriais de uma arquitetura orientada a microsserviços para a Indústria 4.0

Sorocaba
2023

JOÃO PHELYPE DA SILVA CAVALCANTE

Conteinerização e gerenciamento de aplicações industriais de uma arquitetura orientada a microsserviços para a Indústria 4.0

Trabalho de Conclusão de Curso apresentado ao Instituto de Ciência e Tecnologia de Sorocaba, Universidade Estadual Paulista (UNESP), como parte dos requisitos para obtenção do grau de Bacharel em Engenharia de Controle e Automação.

Orientador: Me. Ricardo Pasquati Pontarolli.
Coorientador: Prof. Dr. Eduardo Paciência Godoy.

Sorocaba
2023

C376c

Cavalcante, João Phelype da Silva
Conteinerização e gerenciamento de aplicações industriais de uma arquitetura orientada a microsserviços para a indústria 4.0 / João Phelype da Silva Cavalcante. -- Sorocaba, 2023
55 p. : il., tabs., fotos

Trabalho de conclusão de curso (Bacharelado - Engenharia de Controle e Automação) - Universidade Estadual Paulista (Unesp), Instituto de Ciência e Tecnologia, Sorocaba
Orientador: Ricardo Pasquati Pontarolli
Coorientador: Eduardo Paciência Godoy

1. Portainer. 2. Docker. 3. Raspberry Pi. 4. Microsserviços.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca do Instituto de Ciência e Tecnologia, Sorocaba. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada

UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Instituto de Ciência e Tecnologia de Sorocaba

Conteinerização e gerenciamento de aplicações industriais de uma arquitetura orientada a microsserviços para a Indústria 4.0

João Phelype da Silva
Cavalcante

ESTE TRABALHO DE GRADUAÇÃO FOI JULGADO ADEQUADO COMO PARTE DO REQUISITO PARA A OBTENÇÃO DO GRAU DE
BACHAREL EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Prof. Dr. Everson Martins
Coordenador

BANCA EXAMINADORA:

Me. Ricardo Pasquati Pontarolli
Orientador / UNESP – Campus Sorocaba

Prof. Dr. Eduardo Paciência Godoy
Coorientador / UNESP – Campus Sorocaba

Prof. Dr. Márcio Alexandre Marques
Avaliador / UNESP – Campus Sorocaba

Prof. Dr. Leopoldo André Dutra Lusquino Filho
Avaliador / UNESP – Campus Sorocaba

Sorocaba
2023

Este trabalho é dedicado à minha mãe Audenice e irmã Priscila, que me apoiaram e proporcionaram a realização e conclusão de mais uma etapa de grande importância em minha vida.

AGRADECIMENTOS

Expresso meus sinceros agradecimentos a Universidade Estadual Paulista Júlio de Mesquita Filho - Instituto de Ciência e Tecnologia de Sorocaba (UNESP - ICTS), em especial ao Laboratório do Grupo de Automação e Sistemas Integráveis – GASI e aos meus orientadores Me. Ricardo Pasquati Pontarolli e Prof. Dr. Eduardo Paciência Godoy por me guiar em todo o desenvolvimento do estudo, realizando correções, esclarecendo dúvidas e tornando meu processo de aprendizado enriquecedor.

Agradeço a Fundação de Amparo à Pesquisa do Estado de São Paulo – FAPESP pelo apoio a realização deste estudo, por meio do processo Nº 2018/19984-4.

A todos meus professores e servidores públicos que sempre estavam de prontidão para orientar os discentes e tornaram minha passagem pela universidade ainda mais especial, me proporcionando um ensino público e de qualidade.

A todos os meus colegas pelo apoio e parceria durante todos os anos dentro da universidade, por fim, a todas as pessoas que de alguma forma contribuíram para minha formação pessoal e acadêmica.

CAVALCANTE, João Phelype Silva. **Conteinerização e gerenciamento de aplicações industriais de uma Arquitetura Orientada a Microsserviços para a Indústria 4.0**. 2023. 55 f. PFC (Graduação) - Curso de Engenharia de Controle e Automação, Instituto de Ciência e Tecnologia, Universidade Estadual Paulista "Júlio de Mesquita Filho", Sorocaba, 2023.

RESUMO

Tendo em vista o crescimento no segmento da Indústria 4.0, principalmente voltado para automatização de processos industriais, surge a necessidade de desenvolvimento e aperfeiçoamento da *Industrial Internet of Things* (IIoT). Inserida neste contexto tem-se a planta industrial controlada por microsserviços estudada neste trabalho. Muito se fala das vantagens e benefícios de utilizar uma arquitetura orientada a microsserviços, no entanto, o objetivo deste trabalho é mitigar suas desvantagens e malefícios. Com o crescimento do número de serviços no sistema, uma das maiores desvantagens neste tipo de arquitetura torna-se óbvia: a governança. Gerenciar um sistema complexo de múltiplos serviços não é tarefa fácil, para isso foi decidido realizar a conteinerização desses serviços para gerenciá-los por meio de um gerenciador de *containers*. Para isso, é necessário o entendimento básico de conceitos, dentre eles, *Docker* que é uma ferramenta de virtualização a nível de sistema operacional com maior performance quando comparado às máquinas virtuais, já o *Portainer* define-se como uma interface gráfica simplificada, facilitando a interação entre o usuário final e a aplicação construída com *MoleculerJS*, *framework* de microsserviços que auxilia na elaboração de serviços. O estudo tem como base o protótipo de uma planta piloto desenvolvida no Laboratório do Grupo de Automação e Sistemas Integráveis (GASI) do Instituto de Ciência e Tecnologia de Sorocaba (ICTS), a metodologia utilizada é fundamentada em *Docker* e *Portainer*, a mesma, será testada no intuito de entender qual a funcionalidade da solução desenvolvida. Ao final deste trabalho foi realizada a conteinerização dos serviços de uma planta industrial orientada a microsserviços, também foi implementado o Gerenciador de Containers *Portainer* para monitorar e orquestrar os serviços disponibilizados na planta. Um conjunto de dados foi obtido através de testes de carga para validar o tempo médio de resposta de requisições realizadas aos serviços. Conclui-se que o tempo médio de resposta não foi afetado significativamente pela conteinerização e a gestão e monitoria dos serviços foi melhorada drasticamente de forma qualitativa.

Palavras-chave: *Portainer*. *Docker*. *Raspberry Pi*. Microsserviços.

CAVALCANTE, João Phelype Silva. **Containerization and management of industrial applications of a Microservices Oriented Architecture for Industry 4.0**. 2023. 55 p. Undergraduate Thesis - Control and Automation Engineering Course, Institute of Science and Technology, São Paulo State University "Júlio de Mesquita Filho", Sorocaba, 2023.

ABSTRACT

With the rapid growth of the Industry 4.0 sectores and its focus on automating industrial processes, there is an urgent need to develop and enhance the *Industrial Internet of Things* (IIoT). In this context, this study examines the use of a microservices-based architecture to control an industrial plant. While much is said about the benefits of using microservices, this work aims to address the risks and drawbacks associated with this approach. One of the key challenges of managing a complex system of multiple services is governance, which becomes increasingly evident as the number of services in the system grows. To tackle this issue, the study proposes containerizing the services and orchestrating them using a container manager. To achieve this, a basic understanding of concepts is necessary, such as Docker, which is a high-performance virtualization tool at the operating system level, and Portainer, which is a simplified graphical interface that facilitates interaction between end-users and the application built with *MoleculerJS*, a microservices framework that aids in the creation of services. The study is based on a pilot plant prototype developed at the Laboratory of the Group of Automation and Integrated Systems (GASI) of the Institute of Science and Technology of Sorocaba (ICTS). The methodology used is based on Docker and Portainer, and the goal is to understand the functionality of the developed solution. The study concludes that containerizing the services of an industrial plant oriented to microservices and implementing the Container Manager Portainer significantly improves the management and monitoring of services qualitatively, without affecting the average response time of requests made to services. Load tests were conducted to validate this claim, and the results confirm that the average response time was not affected by containment.

Keywords: Portainer. Docker. Raspberry Pi. Microservices.

Lista de Figuras

Sumário

1 Introdução e justificativa

Com a chegada da Indústria 4.0, máquinas inteligentes estão sendo empregadas para tornar os processos mais produtivos, eficientes e seguros. Uma das tecnologias-chave da Indústria 4.0 é a Internet das Coisas Industrial que permite a interconexão digital de múltiplos dispositivos. Com essa rede de dispositivos conectados é possível inferir informações do meio físico através de sensores. Esses dados podem ser compartilhados rapidamente entre múltiplos dispositivos, o acesso facilitado à informação permite separar os serviços em módulos, cada qual com sua responsabilidade.

De acordo com Santos *et al.* (2018), tomando como base o ponto de vista histórico é possível observar diversos marcos na revolução industrial, na última década nota-se uma evolução da tecnologia da informação e introdução de processos produtivos no intuito de trazer uma transformação da indústria tradicional, levando a outro grau de desenvolvimento organizacional. No intuito de validar os benefícios dessas ferramentas tecnológicas e para fortalecer a competição no mercado, há uma mudança de paradigma na manufatura sendo questionada a nível global. Essa transformação é conhecida como Indústria 4.0, em que se visa descrever a implementação de ferramentas inteligentes que podem se comunicar entre si a fim de trazer maior automação ao longo da cadeia produtiva.

De modo geral, Bertolucci (2016) defende que o fundamento básico da Indústria 4.0 está na adoção da integração de máquinas, sistemas e ativos, possibilitando a criação de redes inteligentes por toda cadeia de valor, que por sua vez permitirá o controle autônomo dos módulos de produção.

Chilanti (2022) em sua dissertação expõe que a Indústria 4.0 possui cinco características mais marcantes: digitalização; flexibilidade; conectividade; adaptabilidade e inteligência. O emprego da digitalização se dá para a análise computacional do objeto de estudo, sejam eles dados oriundos de processos ou máquinas. Já a flexibilidade proporciona um ajuste fino dos setores para obter serviços ainda mais customizados. O papel da conectividade é a integração da comunicação dos diversos sistemas em tempo real, possibilitando uma maior segurança para as tomadas de decisões. A adaptação está atrelada a autonomia de reação dos sistemas em situações adversas e inesperadas. A inteligência aplicada à análise dos dados permite uma produção mais sustentável e eficiente, trazendo um melhoramento contínuo.

O autor acredita que a implementação da Indústria 4.0 nas empresas, devem acontecer de forma gradativa e contínua, pois tais mudanças, em sua maioria, levam tempo, possuem alto valor e risco. Em contrapartida possibilita a mudança cultural no âmbito operacional da empresa, onde traz a tecnologia a favor para otimizar os processos de produção e distribuição destes serviços e recursos.

O termo Internet das Coisas é proveniente da expressão inglesa *Industrial Internet of Things*, usualmente empregada pela sua abreviação IIoT. Sacomano (2018) entende que esta ferramenta se fundamenta no acesso remoto, a informações oriundas de objetos conectados à internet.

Neste trabalho foi utilizado o protótipo de planta industrial do Laboratório do Grupo de Automação e Sistemas Integráveis (GASI) localizado no quinto andar do Instituto de Ciência e Tecnologia de Sorocaba. O protótipo é responsável por fazer o controle de nível de dois

tanques. O controle é realizado por múltiplos *Raspberry Pi* conectados à rede local, sendo que diversos serviços estão distribuídos entre os microcontroladores. Com o aumento no número de serviços surge a necessidade de escalar horizontalmente, ou seja, aumentar o número de microcontroladores. O crescimento da planta aumenta a complexidade e o volume de serviços instalados em cada placa, e para sistemas dessa magnitude uma organização impecável é essencial.

Atualmente o gerenciamento de versões e dependências das aplicações utilizadas no sistema são feitas de forma manual. Esse gerenciamento manual pode trazer comportamentos inesperados, como problemas de compatibilidade entre versões. Além disso, configurar o ambiente em novas placas consome tempo e é necessário conhecer diversas ferramentas e também existe certa dificuldade no controle de versões dos serviços, trazendo receio na implantação de novas soluções.

Visto isso, a fim de propor uma solução serão criadas imagens dos serviços utilizados no sistema. Essas imagens possuem diversas versões e todas as dependências para a execução da aplicação, a instalação das imagens são facilitadas utilizando ferramentas apropriadas. Além disso, será utilizado um aplicativo capaz de gerenciar os serviços, imagens e contêineres de cada *Raspberry Pi* da planta.

2 Revisão de Literatura

2.1 Microsserviços e arquiteturas

Para Oliveira *et al.* (2017), esta arquitetura de software traduz em pequenos serviços a interação dos próprios processos e seus mecanismos de comunicação. O principal incentivo para o uso de Microsserviços está associado à capacidade de serem introduzidos em ambientes de desenvolvimento simplificados, tendo inúmeros benefícios como a fácil manutenção e rápida migração.

O autor ainda define que máquinas virtuais são caracterizadas pela execução de aplicações em um sistema operacional específico do hóspede, já no caso dos microsserviços é segmentado por meio de conceitos de *Linux Containers* (LXC), utilizando assim o mesmo sistema operacional. Tendo o isolamento das aplicações em contestes individualizados, resultando em uma eliminação de dependências do código do programa. Esta aplicação traz alguns benefícios, como a escalabilidade, o potencial do empreendimento agregar valor ao seu negócio e atender a demandas, e manutenção simplificada, seja ela corretiva, remediativa ou preventiva.

A arquitetura de Microsserviços consiste na criação de módulos apartados, cada um responsável por um serviço. Desta forma, cada módulo executa em instâncias diferentes, isso permite uma maior escalabilidade, disponibilidade e facilita sua manutenção (MOLECULERJS, 2022).

Já Domingos *et al.* (2020) entende que Microsserviços representa um modelo estrutural da aplicação de um conjunto de serviços, suas principais características também são facilidade de manutenção, praticidade de testes e implementação independente, podendo ser chamada de "*deploy*". Esta arquitetura é caracterizada por ser formada por unidades independentes de funcionalidades de outros serviços, assim seu projeto, desenvolvimento, teste e aplicações são autônomos. Além da organização dos recursos, que trazem a versatilidade da tecnologia empregada.

A principal responsabilidade de acordo com Domingos *et al.* (2020) é metaforicamente fornecer pontes para que os sistemas sejam conectados, isto só é possível por meio de um recurso de aplicação interligado a uma *Application Programum Interface* (API) e tendo seu banco de dados próprio. Outras características fundamentais dos microsserviços são a capacidade de adaptação em diversas configurações e recebimento de dados, a automação do início ao fim de seu ciclo de vida e capacidade de dimensionar corretamente soluções aplicáveis em função do cenário a ser utilizado.

No entanto em seu estudo o autor traz alguns pontos desfavoráveis no uso de microsserviços, levando a contraindicação da arquitetura em alguns casos, pois também possui complexidade e desvantagens na implementação. É fundamental uma equipe experiente de operações, tendo em vista que são manipulados diversos serviços com inúmeras formas de comunicação, trazendo grande complexidade na organização de suporte para os microsserviços. Ainda, está arquitetura possui uma fase de teste mais robusta, sendo separada em duas etapas. Na primeira etapa são realizados os testes unitários chamados de *Test-driven Development* (TDD), que são realizados antes do desenvolvimento do código. Já na segunda etapa é testado a implementação dos serviços, a comunicação entre a base de dados e os testes

de protocolo, em que as APIs recebem requisições, por exemplo, *Hypertext Transfer Protocol* (HTTP), ferramenta recorrente usada pelos profissionais para comunicação entre sistemas de informação na internet.

Domingos *et al.* (2020) entende que as arquiteturas de microsserviços se diferenciam das demais pois impactam diretamente no desempenho e desenvolvimento do perfil do time responsável pela construção e pela implementação dos sistemas. Outro benefício é que o ponto de falha não é unificado, ou seja, quando houver falha em alguns dos serviços, o restante da aplicação não sofrerá danos, deixando apenas o produto afetado indisponível.

Em conjunto, Oliveira *et al.* (2017) expõe que a principal motivação para o uso de microsserviços é o isolamento das falhas inteirando o funcionamento da aplicação ainda que com danos em serviços isolados, ainda quando comparada a estruturas que utilizam arquitetura monolítico, em que o sistema compartilha integralmente os recursos, resultando em um ciclo de vida dependente que reage a falha de modo global, deixando toda a aplicação fora de funcionamento.

O conceito de arquitetura monolítica é entendido por Domingos *et al.* (2020) como o empacotamento integral dos componentes do servidor em apenas uma unidade. Possui três fases básicas, a primeira é a *client-side* ou interface de usuário, representada pelas páginas em HTML e Javascript. Em seguida têm-se a fase de banco de dados, onde é realizado o gerenciamento dos dados por tabelas ou outras ferramentas. Por fim, o *server-side* responsável pelo suporte das requisições HTTP, além da execução, acesso aos bancos de dados e demais outras funcionalidades.

Uma desvantagem da arquitetura monolítica de acordo como autor, é que possui um ponto de falha único, isto é, caso uma funcionalidade tenha qualquer tipo de problema toda a aplicação será afetada, deixando o sistema indisponível e não permitindo o acesso a nenhuma funcionalidade. Quando as aplicações começam a expandir e ampliar as funcionalidades, passam a ser um empecilho para o desenvolvimento de novas funcionalidades, além da dificuldade no entendimento e na manutenção do código para os profissionais. Tais operações se apresentam de forma complexa e arriscada, exigindo o uso de recursos de diversas formas, como, tempo, testes e desenvolvedores.

2.2 Docker

Para Anderson (2015) a ferramenta *Docker* é uma tecnologia de virtualização de contêineres, similar a uma máquina virtual, porém com consumo de recursos reduzidos. Através do *Docke*r é possível automatizar aplicações implantadas em *containers*. A aplicação é virtualizada dentro deste contêiner fornecendo um ambiente rápido e leve onde o código será executado de forma eficiente e garante que o ambiente de desenvolvimento será muito parecido com o ambiente de produção.

Já para Oliveira *et al.* (2017) *Docker* é estruturada através de uma plataforma de microsserviços, em que é utilizado conceitos de *containers*, imagens e fórmulas, chamados *Dockerfiles*. De maneira geral é criada uma comunicação entre *containers* e o sistema operacional por meio de uma API, que por sua vez faz a função de segregar a interação da aplicação do restante do sistema.

O documento *Docker overview* (2021), disponibilizado pela empresa *Docker Inc,* afirma que a tecnologia *Docker* é composta por quatro componentes principais, sendo eles *Docker Client* e *Server, Docker Images*, *Docker Registries* e *Docker Containers*. Para isso é utilizado uma arquitetura cliente-servidor onde o cliente comunica-se com o *Docker Daemon* que executa e distribui os *containers*.

O *Docker Client* é a principal forma de interagir com o *Docker* através de comandos, que utilizam a *Docker Application Interface (API)*, assim o usuário é capaz de se comunicar com múltiplos *daemons*. Um dos três componentes do servidor é o *Docker Daemon que* recebe requisições da *Docker* API e orquestra os objetos *Dockers* (*images, containers, networks* e *volumes*). Além disso, também é capaz de comunicar-se com outros *daemon* com o objetivo de gerenciar serviços do tipo *Docker*. Outra peça fundamental do servidor são as *Docker Images,* componente este que pode ser definido como um *template read-only* contendo instruções de como realizar a criação de um *Docker Container*. Por fim, o último componente do servidor é o *Docker Container*. Os *containers* são instâncias executáveis de uma *Docker Image.* É possível criar, iniciar, parar, mover ou deletar um *container* através da API ou *Command Line Interface* (CLI) do *Docker*.

Outra forma de interagir com o *Docke*r é através do *Docker Compose*, uma ferramenta desenvolvida para auxiliar na definição e compartilhamento de sistemas com múltiplos *containers*. A maior vantagem do *Docker Compose* é definir todos os containers utilizados pelo sistema em um arquivo de configuração e através de um comando disponibilizar toda a arquitetura da aplicação em instantes.

Oliveira *et al.* (2017) diz que a criação de uma *Docker image* é feita por meio de *Dockerfiles* disponibilizadas pelas comunidades. A ferramenta possui um serviço de repositório em *cloud,* onde pode ser executado remotamente e está disponível nos principais sistemas operacionais. Ainda, o *Docker* é adaptável de modo que são aplicáveis para outras arquiteturas de computadores, permitindo o bom funcionamento em máquinas com dispositivos simples e de pouco recurso.

No caso de Chilanti (2022), *Docke*r é entendido como uma ferramenta com o intuito de automatizar o desenvolvimento de *containers,* resultando em uma implementação e execução das aplicações de forma simplificada. Ainda, a aplicação *Docker* possibilita que a execução de *containers* seja feita em um ambiente virtual, como por exemplo, *Kernel Linux*. Diferente de uma máquina virtual, onde é utilizado um sistema operacional próprio para cada *kernel*, no caso do *Docker* cada *container* compartilha o mesmo *kernel,* trazendo uma eficácia na aplicação, deixando-o mais leve, rápido e portátil. Outro benefício elencado é a otimização de espaço de armazenamento por meio da reutilização de *layers*, ou seja, caso o *Docker* entenda que duas imagens distintas compartilhem a mesma camada o sistema é capaz de aproveitá-la, trazendo maior eficiência para a aplicação.

A comunidade científica de uma forma geral percebe que *Docker e Kubernetes*, estão se tornando cada vez mais populares para a implementação de virtualização de nível de sistema operacional para aplicações de IIoT que exigem baixa latência. Okwuibe *et al.* (2020) acreditam que isto se deve à sua arquitetura, que resulta em menor utilização de recursos e tempo de inicialização com maior velocidade. Além disso, tem como maior vantagem a compatibilidade técnica com várias plataformas de implantação e ambientes em *cloud,* tornando-se mais fáceis de adotar tanto no meio acadêmico quanto na indústria. Os *containers*

são projetados principalmente para fornecerem uma plataforma padronizada de isolamento para a implantação de aplicativos, trazendo a facilidade para os desenvolvedores isolarem seus aplicativos do ambiente, resolvendo, assim, o problema de dependência da plataforma.

Os autores enxergam que a orquestração de virtualização fornecida pelo *Docker* é necessária para permitir a implantação de sistemas distribuídos. Visto que os desenvolvedores enfrentam um desafio significativo ao desenvolver aplicativos e arquiteturas que atendam aos requisitos de todos os protocolos subjacentes, pois a IIoT depende de dispositivos heterogêneos que trabalham juntos e compartilham grandes quantidades de dados.

Em seu estudo, Okwuibe *et al.* (2020) observou que existe um atraso de aproximadamente 5 segundos para a inicialização da sessão de *streaming* no nó móvel quando o servidor *FFmpeg* está sendo executado em um *Docker* em comparação com a implementação nativa. A partir disso os autores entendem que uma solução viável para aplicativos IIoT sem restrições graves, é a combinação, *edge* e orquestração de *containers* por meio do uso da ferramenta *Docker*.

O objetivo do estudo realizado por Sollfrank *et al*. (2021) foi examinar uma arquitetura baseada em tecnologias da IIoT que utiliza processamento e comunicação virtualizados em nós para atender aos requisitos de tempo da automação. Este estudo foi focado em sistemas com baixo consumo de *hardware* para monitoramento em tempo real.

Os autores definem *Docker* como uma plataforma que permite a criação, compartilhamento e execução de aplicativos em *containers*, suportando a escalabilidade dinâmica de recursos de aplicativos, permitindo que sejam flexíveis em relação às demandas. O *Docker* usa a virtualização de nível do sistema operacional para criar um ambiente onde o aplicativo é executado independentemente do *host*. As imagens podem ser carregadas em repositórios para desenvolvimento de *software*. O *Docker hub* é semelhante ao *git* em termos de funcionalidade e suporta controle de versão. Estando disponível para vários sistemas operacionais, o *Docker* traz a facilidade de portabilidade de *containers*. A orquestração de containers *Docker* pode ser gerenciada através de arquivos *Docker-compose* ou *Kubernetes*.

O *Docker Engine* detém o fluxo de trabalho para criar e realizar a conteinerização de aplicativos, enquanto o *Docker Hub* é uma biblioteca para compartilhar e gerenciar pilhas de aplicativos como imagens *Docker*. Estes dois juntos formam a plataforma *Docker* de acordo com Sollfrank *et al.* (2021). O *Docker Image* serve como base para um *container Docker*, onde pode ser descrito como a combinação de um sistema de arquivos com parâmetros estruturados organizados em diversas camadas de imagem. Todas as imagens *Docker* de repositórios públicos podem ser puxadas do *Docker hub* para qualquer máquina local. As imagens podem ser personalizadas ou autoprojetadas e certificadas. Já o *Dockerfile* é um arquivo de texto que contém os comandos necessários para montar uma imagem. Para virtualização, o *Docker* emprega uma rede pré estabelecida e isolada por padrão enquanto o container é executado sem outras especificações de rede.

Sollfrank *et al.* (2021), a partir dos seus experimentos conseguiu avaliar os impactos da conteinerização *Docker* em um aplicativo monitoria em tempo real. Os autores entendem que a virtualização baseada em *Docker* pode atender aos requisitos de respostas em tempo real em aplicativos de automação. Ainda foi observado um atraso adicional significativo de processamento no tempo médio de execução no nó. Todos estes pontos indicam que a

virtualização pode ser utilizada não apenas para aplicativos de tempo real, mas também para os aplicativos de multitarefas, que se beneficiam com o isolamento dos containers.

Logo, a ferramenta *Docker* é uma tecnologia de virtualização de containers que permite a automação de aplicativos conteinerizados, fornecendo um ambiente rápido e leve onde o código será executado de forma eficiente, garantindo que o ambiente de desenvolvimento seja semelhante ao ambiente de produção. O *Docker* é estruturado por meio de uma plataforma de microsserviços que utiliza containers, imagens e fórmulas chamadas de *Dockerfiles*. É uma arquitetura cliente-servidor onde o cliente se comunica com o *Daemon* do *Docker* que executa e distribui os containers. Os principais componentes da tecnologia *Docker* são *Docker Images, Registries* e *Containers*. O *Docker Compose* é uma ferramenta desenvolvida para definir e compartilhar sistemas com vários containers, tornando possível definir todos os containers usados pelo sistema em um arquivo de configuração e, por meio de um comando, fornece toda a arquitetura da aplicação em segundos. Os autores argumentam que a orquestração de virtualização do *Docker* é necessária para permitir a implantação de sistemas distribuídos.

A ferramenta *Docker* é adaptável, permitindo a implementação e execução de aplicativos de maneira simplificada, resultando em uma aplicação leve, rápida e portátil. A comunidade científica reconhece que o *Docker* e o *Kubernetes* estão se tornando cada vez mais populares para a implementação de virtualização de nível do sistema para aplicativos IIoT que requerem baixa latência, graças à sua arquitetura resultando em menor uso de recursos e tempo de inicialização, com maior compatibilidade com vários ambientes de implantação e nuvem.

2.3 Portainer

O manual *Portainer architecture* (2021), disponibilizado pela empresa *Portainer* Documentation define *Portainer* como uma interface gráfica de fácil utilização que abstrai a complexidade de gerenciar *containers*. Removendo a necessidade de usar CLI, YAML ou entender os manifestos.

A versão gratuita da ferramenta *Portainer* pode conectar-se com quatro diferentes tipos de ambientes, sendo eles descritos a seguir:

a) *Azure Container Instances (ACI)* é um serviço de contêiner da Microsoft Azure que permite implantar e executar contêineres facilmente, sem a necessidade de gerenciar a infraestrutura subjacente. Ele fornece uma maneira simples e rápida de implantar contêineres isolados em nuvem, sem a necessidade de provisionar máquinas virtuais ou gerenciar um cluster de orquestração de contêineres;
b) *Docker Standalone* é uma maneira de usar o *Docker* como um motor de contêiner autônomo. Neste modo, o *Docker* é executado diretamente em um único sistema *host* e é usado para construir, executar e gerenciar contêineres nesse host. O *Docker Standalone* é projetado para cenários em que você precisa executar um pequeno número de contêineres em um único host, ou quando você precisa experimentar com a conteinerização sem a complexidade de uma ferramenta de orquestração;
c) *Docker Swarm* é uma ferramenta de orquestração de contêineres do *Docker* que permite gerenciar um *cluster* de *hosts Docker* e orquestrar a execução de aplicativos em contêineres nesse cluster. Com o *Docker Swarm*, você pode criar um grupo de

hosts Docker que trabalham juntos como um único sistema virtual, permitindo que você execute aplicativos em contêineres em vários hosts;

d) *Kubernetes*, também conhecido como K8s, é uma plataforma de orquestração de contêineres de código aberto que permite gerenciar, implantar e escalar aplicativos em contêineres de maneira eficiente e consistente em um ambiente de computação em nuvem. Ele é projetado para trabalhar com contêineres *Docker*, mas também suporta outros tipos de contêineres, como *rkt*.

Para a proposta deste trabalho o ambiente mais indicado é o *Docker Standalone*, pois para este momento ainda não existe a necessidade de balanceamento de carga (*Docker Swarm* ou *Kubernetes*). E quanto ao ACI, o sistema atual da planta piloto é hospedado localmente, mas caso seja necessário o *Docker Standalone* pode ser migrado para nuvem posteriormente.

Em uma arquitetura distribuída em múltiplos *hosts*, como a apresentada neste trabalho, é necessário que o *Portainer* tenha acesso a todas as máquinas. Para o ambiente do tipo *Docker Standalone* são oferecidos 4 tipos de conexões, listadas a seguir:

a) *Docker API*: É um meio de comunicação entre o *Portainer Server* e o *Docker* API. Ele permite que o *Portainer Server* se comunique com o *Docker* API e execute tarefas como criar e excluir contêineres, serviços e volumes;

b) *Portainer Agent*: O *Portainer Agent* é um agente leve que é executado em cada host. Ele permite que o *Portainer* gerencie o host e execute tarefas como iniciar e parar contêineres, criar e excluir serviços e gerenciar volumes. Diferente do *Portainer Edge Agent*, a conexão parte do *Portainer Server* para o host;

c) *Portainer Edge Agent*: O *Portainer Edge Agent* é um agente que permite que o *Portainer* gerencie contêineres em hosts remotos e dispositivos IoT. Ele se conecta ao *Portainer Server* e fornece uma maneira de gerenciar contêineres em hosts remotos e dispositivos IoT;

d) *Socket*: O *Portainer Server* se conecta diretamente na *Docker Engine*, permitindo gerenciar contêineres, imagens, redes e volumes em um host.

Das 4 opções de conexão listadas anteriormente, os tipos c. e d. necessitam de uma porta de rede alocada no *host*, isso traz duas desvantagens: a primeira delas impactando a segurança, pois os *hosts* ficam suscetíveis a ataques através das portas abertas. Quanto a segunda desvantagem, é a necessidade de configurações: nesse tipo de conexão é necessário configurar no *Portainer Server* quais portas devem ser acessadas, assim como abrir essas portas nos *hosts*. É preciso realizar essa configuração para cada nova máquina adicionada na rede, ou seja, essa solução não é escalável, portanto, para o sistema utilizado pela planta estudada neste trabalho o modelo agente seria o mais indicado.

A arquitetura utilizando um agente é composta por dois elementos: *Portainer Server e Portainer Agent*, onde ambos são executados em *containers*. Sendo necessário realizar o *deploy* do *Portainer Agent* em cada nó do *cluster*, além disso, cada agente deve reportar ao contêiner do *Portainer Serve*r, em que é capaz de receber conexões de múltiplos *Portainer Agents* e para isso é necessário persistência de dados. Com isso, é possível gerenciar diversos *clusters* através de uma interface centralizada.

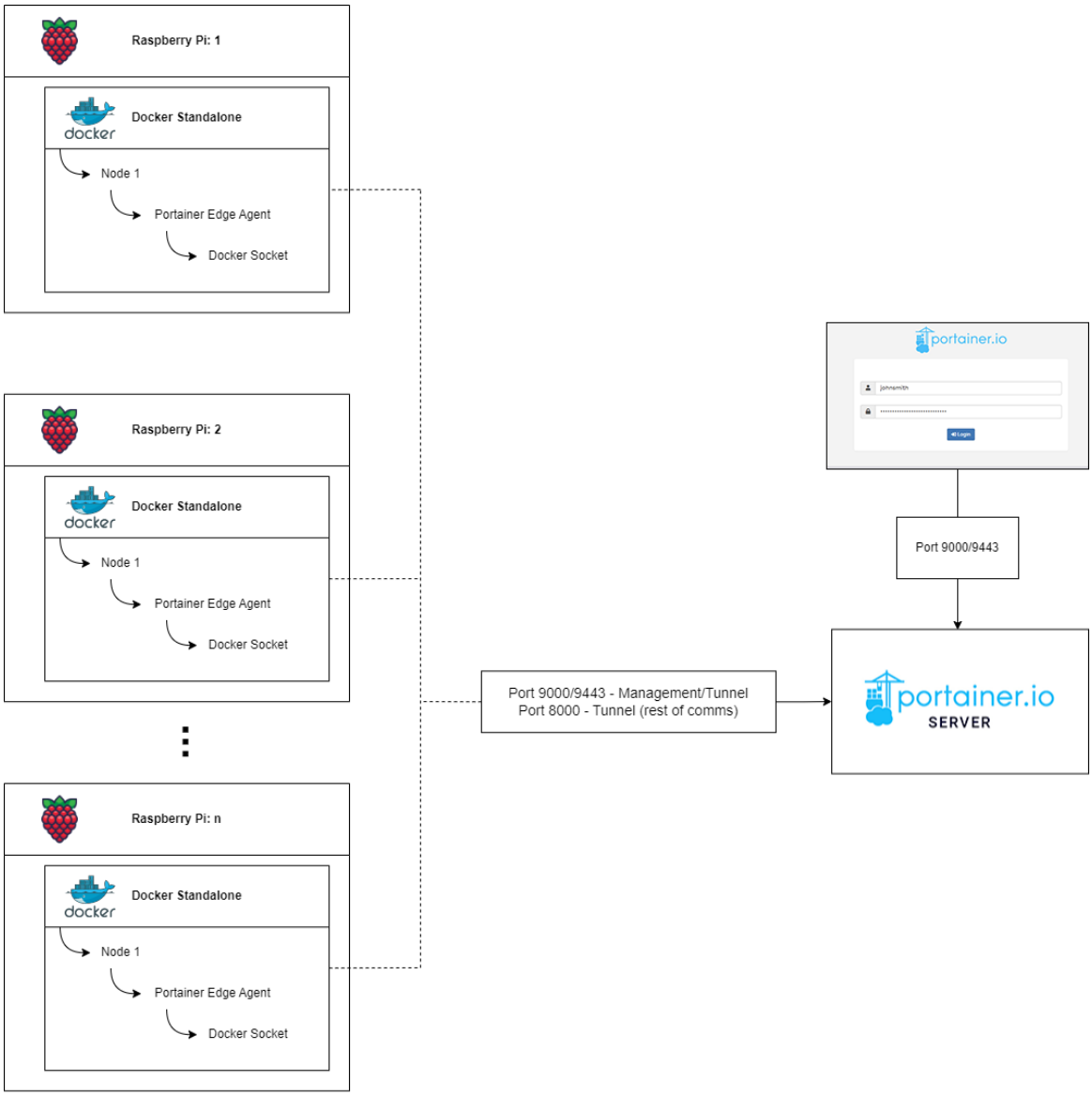
O *Portainer Edge Agent* é utilizado em configurações em ambientes remotos que estão completamente separados da rede onde o *Portainer Server* está hospedado. Trazendo diversos benefícios, como, maior segurança na maioria dos casos, visto que a única preocupação do

escopo de projeto será a instância do *Portainer*, não tendo a necessidade de projetar individualmente cada rede do *Agent* como forma de conexão do *Portainer*.

As conexões tipo a. e b. são mais indicados para redes com um maior número de *hosts* e/ou com expectativa de crescimento. Essas duas soluções são semelhantes, porém, o *Portainer Edge Agent* (b.) oferece maior segurança. Neste caso, é necessário alocar portas apenas no host do *Portainer Server*, pois neste método quem receberá as conexões é o *Portainer Server*, logo, existe um menor número de portas aberta e todas as medidas de segurança podem ser aplicadas no mesmo lugar.

Devido a maior segurança oferecida assim como melhor configuração para ambientes remotos, o modelo escolhido para este trabalho foi o *Portainer Edge Agent*. O diagrama na Figura 1 descreve como ocorre a conexão de um ambiente *Raspberry Pi* com o *Portainer Server* utilizando *Portainer Edge Agent*.

Figura 1 - Diagrama de conexão entre *Portainer Edge Agent* e *Portainer Server*



Fonte – Autoria própria.

2.4 MoleculerJS

O manual de *MoleculerJS* (2022) licenciado e disponibilizado pela empresa CC BY4.0 entende que Moleculer é um *framework* de microsserviços rápido, moderno e poderoso para *Node.js.* Auxilia na construção eficiente de serviços confiáveis e escaláveis.

O *MoleculerJS* utiliza uma Arquitetura de Microsserviços onde todos os serviços estão sendo executados em nós independentes que se comunicam através de um *Transporter*. Neste tipo de arquitetura a latência de rede não é desprezível, mas em contrapartida oferece a possibilidade de escalar serviços e mantê-los altamente disponíveis e resilientes.

O componente principal do *Moleculer* é o *ServiceBroker*. Ele é responsável por resolver serviços e eventos, invocar ações e se comunicar com nós remotos. É necessário iniciar uma nova instância do *ServiceBroke*r em cada nó, entretanto não é necessário criá-lo manualmente. É possível utilizar o *Moleculer Runner,* que é capaz de criar e executar um *broke*r e carregar serviços.

O elemento denominado *Transporter* é o responsável e se faz essencial para trabalhar com múltiplos nós. Este componente é responsável por comunicar-se com outros nós, transferindo eventos, requisições e processando respostas. Caso um serviço seja executado em múltiplas instâncias em nós diferentes, as requisições serão balanceadas entre os nós disponíveis. Os nós são processos do sistema operacional sendo executados em uma rede local ou externa, cada instância dessas é capaz de hospedar um ou vários serviços.

O API *gateway* expõe os serviços do *Moleculer* para o usuário final, ele é basicamente um servidor que recebe e mapeia requisições para invocar serviços, em seguida retornando respostas apropriadas.

2.5 Trabalhos relacionados

Oliveira *et al.* (2017) entende que a aplicação para o controle remoto no contexto da Indústria 4.0 é possível através da configuração de uma arquitetura que combina *Docker*, microsserviços, *cloud*, sensores e dispositivos *Raspberry Pi*, em que a distribuição é feita por meio da IoT. No estudo feito pelo autor, provou-se que com a geração e execução de containers a partir de *Dockerfiles* é possível realizar, sem dificuldades, a migração de uma aplicação para um novo e diferente ambiente com êxito. Ainda neste mesmo caso não foi quantificado o estresse do sistema para verificar a perda de processamento, aspecto que pode ser um fator limitante para o uso de alguns dispositivos mais simplificados.

O pesquisador entende que apesar dos resultados positivos obtidos, ainda há a necessidade de se aprofundar nos estudos de *containers*, principalmente aqueles focados em performance das máquinas e quais seus impactos. Além disso, deve haver uma preocupação adicional quanto aos aspectos de segurança dos *containers* e da rede, uma vez que o compartilhamento das mesmas bibliotecas do sistema operacional e interface física significa uma fragilidade no procedimento como um todo.

Conforme exposto por Domingos *et al.* (2020) em seu estudo, entende-se que a aplicação da arquitetura de microsserviços requer conhecimentos específicos e deve ser implementada por profissionais qualificados. A utilização de uma arquitetura monolítica pode ser uma aliada no início do desenvolvimento dos projetos, tendo em vista o déficit de conhecimento da ferramenta e sua complexidade. Ainda assim, esta ferramenta se mostra de

enorme importância para a Indústria 4.0, à medida que permite a simplificação de tarefas, como, a migração entre arquiteturas.

A preocupação com o usuário final é notória, por isso deve-se entender quais são as principais necessidades do consumidor para entregar as melhores soluções, garantindo um mercado competitivo que acompanhe o crescimento exponencial das novas tecnologias.

3 Desenvolvimento do trabalho

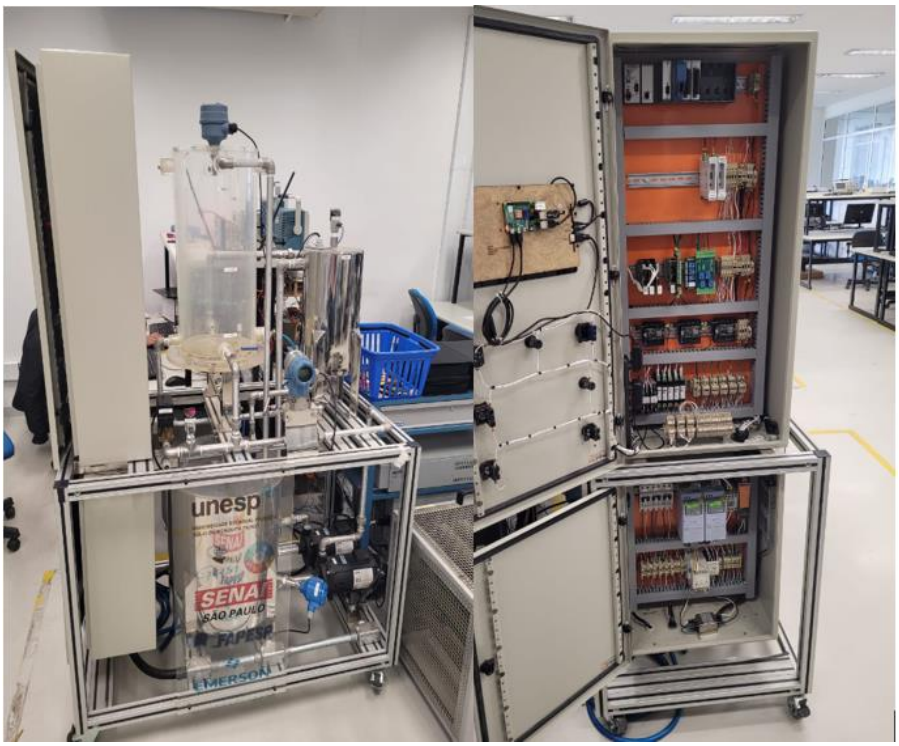
3.1 Proposta do trabalho

A proposta do trabalho é a melhora do gerenciamento de serviços de uma planta piloto ao realizar a conteinerização dos serviços escritos em *NodeJS* de uma planta-piloto industrial que será descrita posteriormente. Será implantado um sistema de gerenciamento de containers através da ferramenta *Portainer*. Essa ferramenta é capaz de gerenciar *containers* remotamente e vai permitir adicionar, remover, iniciar, parar e observar *containers* através de sua interface gráfica. A arquitetura anterior que executa *NodeJS* diretamente no host suporta apenas interface de linha de comandos sendo necessário acessar cada *Raspberry Pi* manualmente para executar ações nos serviços.

3.2 Visão geral

Para este trabalho foi utilizado a planta piloto, conforme Figura 2, desenvolvida no GASI, localizado no Instituto de Ciência e Tecnologia de Sorocaba (ICTS). A planta utilizada é descrita por Pontarolli *et al.* (2020), sendo composta por um tanque principal de 83 litros construído em material inox localizado na parte inferior do protótipo. Na parte superior estão localizados dois reservatórios, um de 15 litros de capacidade construído em inox e outro de 38 litros de capacidade construído em acrílico. O fluido armazenado no tanque principal pode ser bombeado para um desses reservatórios. O protótipo conta com duas bombas, cada uma responsável por levar líquido para um dos reservatórios localizados na parte superior. O sistema controla quatro variáveis, sendo elas: nível, vazão, pressão da linha e pressão do reservatório.

Figura 2 - Planta-piloto utilizada neste estudo



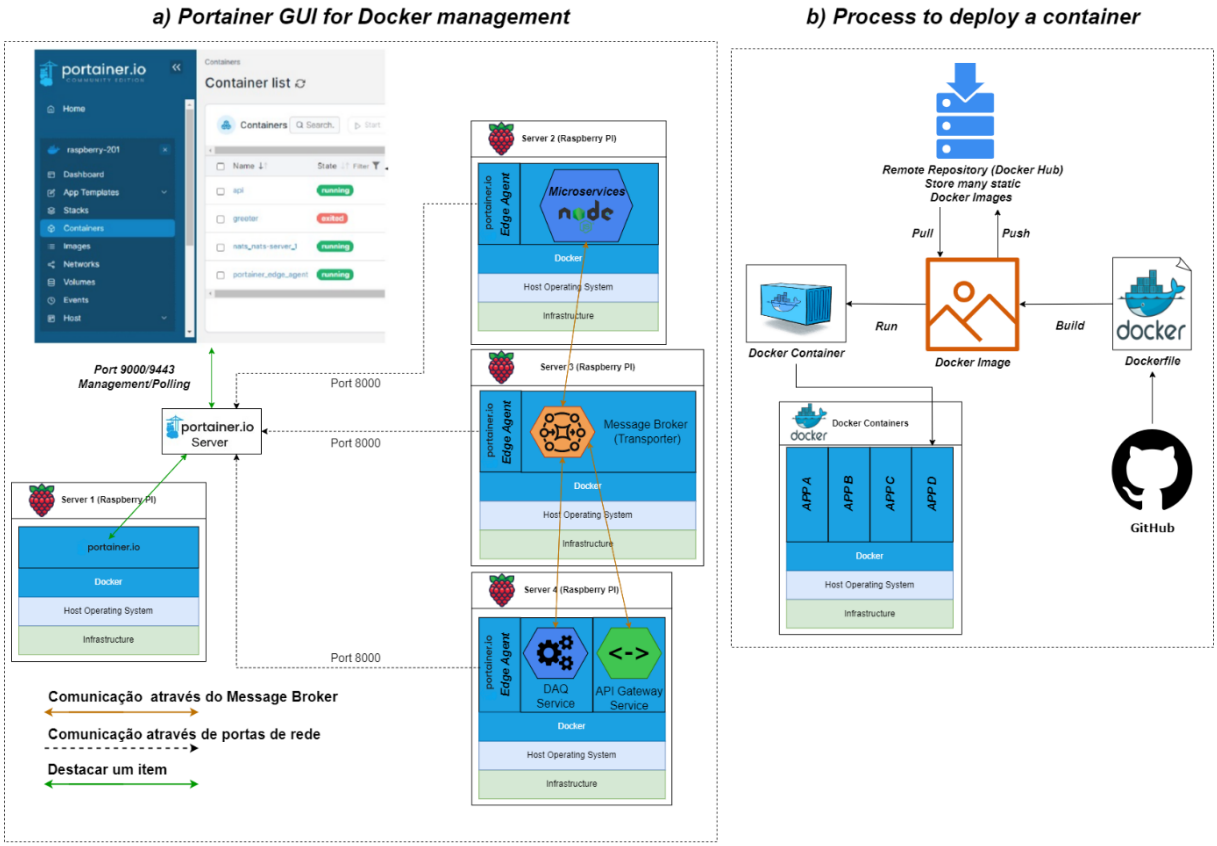
Fonte - Autoria própria.

O painel elétrico da planta-piloto (Figura 2) é composto por uma rede de microcomputadores *Raspberry Pi*. Esses microcomputadores hospedam os serviços utilizados para operar a planta.

A conteinerização foi realizada nos serviços escritos em *NodeJS* que utilizam o *framework MoleculerJS* listados a seguir:

a) *Application Programming Interface* (API): Responsável pela interface com os demais serviços;
b) control: Responsável pelas malhas de controle;
c) *megaind-RPI*: Sistema RPI;
d) PID 4.0: Responsável pelas malhas de controle, suporta redundância;
e) *Data Aquisition* (DAQ): responsável pela leitura das entradas (sensores) e atualização de saídas (atuadores).

Figura 3 - Configuração geral de comunicação entre placas *Raspberry* e processo de deploy de um container



Fonte - Autoria própria.

Na Figura 3 tem-se uma visão geral da proposta: Figura 3 (a) descreve como funciona a comunicação entre as *Raspberry Pis* com o *Potainer Server*, e o diagrama na Figura 3 (b) mostra o ciclo de vida das imagens *Docker* dos serviços, desde a criação até a disponibilização em um repositório remoto. De forma a facilitar o gerenciamento da planta foi realizada a conteinerização dos serviços e a implementação da *Graphic User Interface* (*GUI*) do *Portainer*, Figura 3 (a) para gerenciar esses *containers* e Figura 3 (b) para disponibilizar as imagens dos serviços utilizados nas conteinerizações.

Foi necessário implantar um processo de criação e disponibilização de *Docker Images* e configurar a comunicação dos *containers* com a *GUI*, que serão detalhados nas sessões posteriores. Com isso, a arquitetura proposta é uma rede de containers orquestrados por um gerenciador de containers e um repositório remoto de serviços tornando a implantação e remoção de serviços performática e amigável.

3.3 Conteinerização de serviços

Para criar um *container* utilizando a ferramenta *Docker* é necessária uma *Docker Image* que pode ser criada a partir de um arquivo de configuração chamado *Dockerfile*. Este arquivo contém instruções de como o container será criado, o Apêndice I mostra um *Dockerfile* com uma aplicação escrita em *NodeJS*.

Instruções inseridas em um *Dockerfile* seguem fluxos semelhantes a instalação manual. A Figura 4 mostra um exemplo de como criar uma imagem *Docker* para uma aplicação *NodeJS*; foi necessário indicar a *Docker Image* (L1) que será utilizada como imagem base pela nossa aplicação.

Figura 4 - Configuração geral de comunicação entre placas *Raspberry* e processo de deploy de um container

```
(1) FROM node:14
(2) ENV NODE_ENV production
(3) WORKDIR /app
(4) COPY [ "package.json", "package-lock.json", "./" ]
(5) RUN npm install --production
(6) COPY . .
(7) CMD [ "npm,", "start" ]
```

Fonte - Autoria própria.

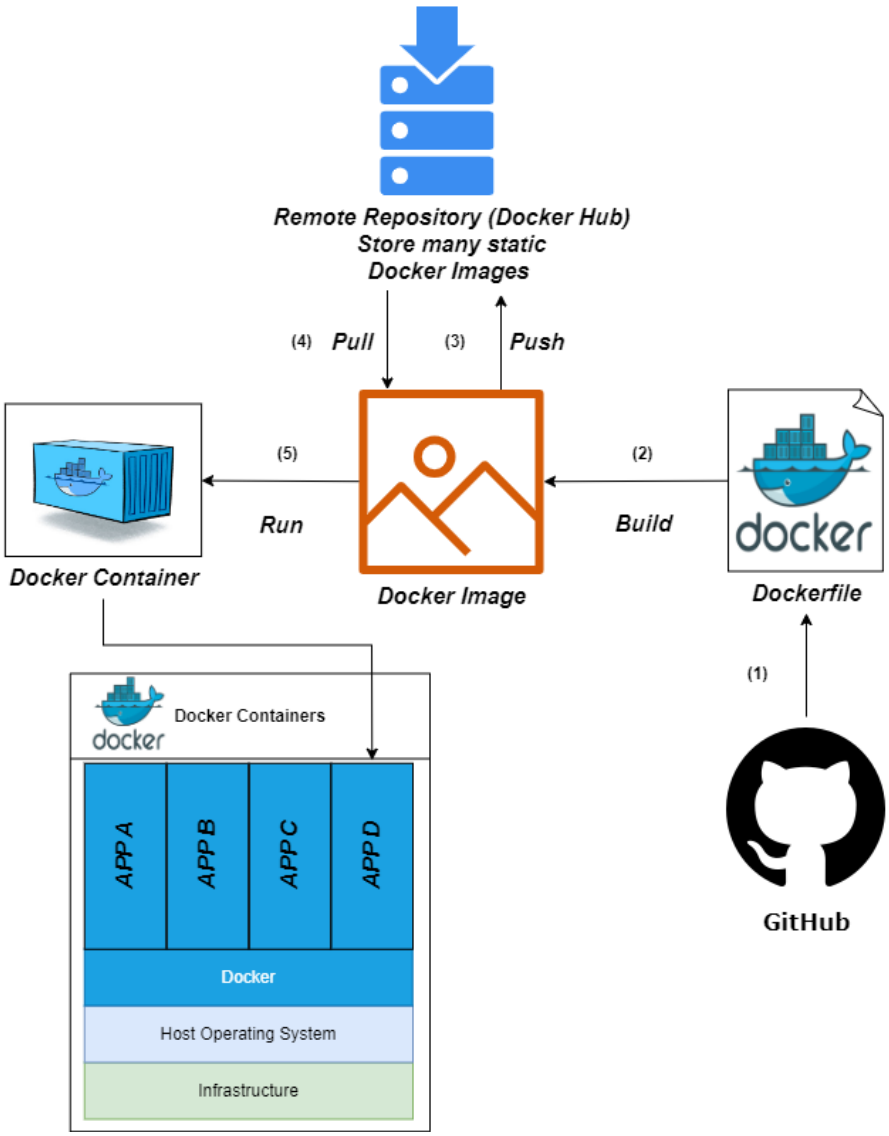
Docker Images podem ser herdadas de outras imagens. Portanto, foi utilizada a imagem oficial do *NodeJS* para as conteinerizações. Essa imagem já possui todos os pacotes e ferramentas necessárias para executar uma aplicação *NodeJS*. Aplicações escritas em NodeJS são mais performáticas em ambientes produtivos, é possível indicar o ambiente através da variável de ambiente NODE_ENV (L2).

Para facilitar a execução dos próximos comandos foi criado um diretório (L3) e instruído ao *Docker* que o utilize como padrão. Com o objetivo de executar o comando *npm install* os arquivos *package. json* e *package-lock. json* são necessários. O comando *COPY* (L4), recebe como parâmetro múltiplos arquivos e copia para o diretório indicado no último parâmetro.

Agora que os arquivos para gerenciamento de pacote estão no diretório desejado, para instalar as dependências necessárias para a aplicação basta executar o comando *npm install* (L5). Neste ponto, a *Docker Image* será baseada na versão 14 do node e todas as dependências da aplicação estão instaladas. O comando *COPY* (L6) foi utilizado novamente para copiar todos

os arquivos do diretório para a imagem. O último passo foi indicar ao *Docker* o comando (L7) que deve ser utilizado ao executar a *Docker Image* descrita. O *Dockerfile* mostrado na Figura 4 foi construído para ser genérico o suficiente para ser utilizada na maioria das aplicações da planta.

Figura 5 - Processo de *deploy* de um container.



Fonte - Autoria própria.

A Figura 5 numera o ciclo de vida de um serviço, desde o repositório de código (*GitHub*) até a implantação do container. Depois de criado o *Dockerfile* de uma aplicação, o primeiro passo é armazenar esse arquivo no *GitHub*. Recuperou-se o *Dockerfile* do *Github* (1), em seguida as *Docker Images* foram montadas através da CLI utilizando o comando *docker build -t gasiepgodoy/moleculer:api* (2), o parâmetro -t nomeia a imagem com uma *tag* de identificação.

Em seguida a imagem é carregada através do comando *docker push gasiepgodoy/moleculer:api* (3) em um repositório remoto, para indicar a imagem correta foi necessário indicar a *tag* de identificação informada no passo anterior. Depois de carregada no

repositório, qualquer máquina autorizada poderá realizar o download dessa imagem com o comando *docker pull* (4).

Qualquer máquina, que possua a ferramenta *Docker*, em posse dessa *Docker Image* será capaz de executá-la com o comando d*ocker run* (5). Neste trabalho, foram criadas *Docker Images* para todos os serviços escritos em *NodeJS* disponibilizando uma biblioteca de imagens onde qualquer *Raspberry* autorizada pode fazer o *download* e disponibilizar o serviço em instantes.

3.4 Gerenciamento de serviços

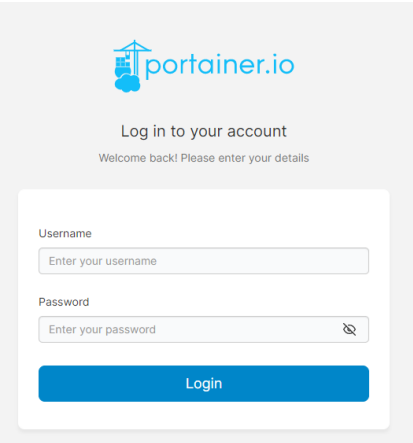
Devido a popularização dos *containers* surgiram diversas ferramentas para auxiliar no uso dessa tecnologia. Uma dessas ferramentas é o *Portainer*, um *software* utilizado para gerenciar *containers*. Com a conteinerização dos serviços tornou-se possível a implantação do *Portainer* para realizar a orquestração dos serviços da planta.

Uma instancia do *Portainer* foi implantada em uma das placas através do arquivo *Docker Compose*, apresentado no Apêndice II. *Docker Compose* é uma funcionalidade utilizada para configurar e executar um container através de um arquivo de instruções. Vale destacar algumas das configurações utilizadas neste *Docker Compose* como o atributo *ports*, utilizado para mapear portas do *container* com as portas da máquina hospedeira. As portas mapeadas são utilizadas para acessar a *Portainer GUI* (9000) e para realizar a conexão com o *Edge Agent* (8000). O atributo *volumes* foi utilizado para indicar o local onde o *Portainer Server* irá persistir seus dados.

Depois de executar o arquivo do Apêndice II, a porta 9000 da máquina hospedeira passou a disponibilizar uma interface gráfica do *Portainer*. Ao acessar essa interface (Figura 6) são solicitadas credenciais conforme mostra a Figura 5. No primeiro acesso foi necessário configurar as credenciais de administrador.

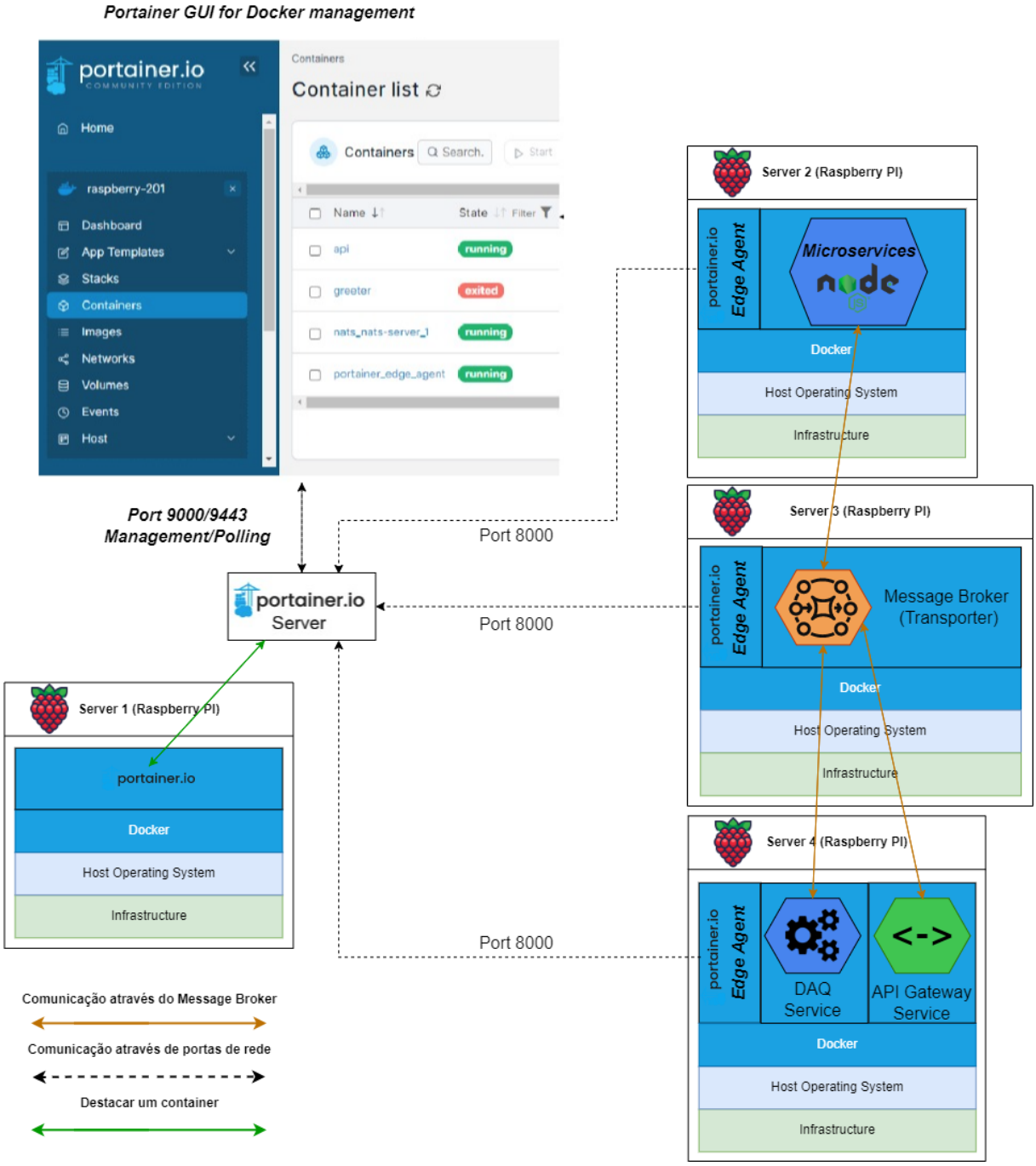
Com acesso a essa interface é possível orquestrar containers hospedados na máquina onde o *Portainer* foi instalado, que apesar de já ser um ganho significativo para a arquitetura da planta não seria eficiente instalar uma instância de *Portainer* Server para todas as *Raspberry Pis*. Portanto, foi necessário conectar o *Portainer* Server com todas as placas da planta, para isso foi escolhido o método de comunicação utilizando o *Portainer Edge Agent*.

Figura 6 - Tela de acesso a *Portainer* GUI



Fonte - Autoria própria.

Figura 7 - Diagrama de comunicação entre *Raspberry* e *Portainer Server*



Fonte - Autoria própria.

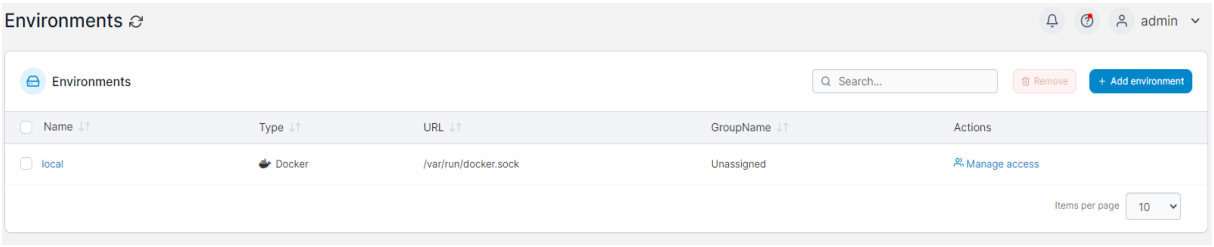
No modelo de configuração *Edge Agent* o *Portainer Server* se conecta com os containers hospedados em outras máquinas. Para realizar essa conexão é necessário configurar uma instância de *Edge Agent* em cada máquina que se deseja operar através da interface gráfica do *Portainer*. A Figura 7 mostra um diagrama de comunicação entre *Portainer Server* e *Edge Agent*, onde o *Edge Agent* conecta-se no *Portainer Server* através da porta 8000.

Quando a conexão entre *Portainer* Server e *Edge Agent* é estabelecida, é possível requisitar dados assim como instruir ações através da *Portainer GUI* permitindo monitorar, implantar e remover serviços em todas as placas Raspberry conectadas através da mesma interface.

Para manter um histórico da configuração assim como tirar maior proveito das funcionalidades oferecidas pela ferramenta *Docker*, o *Edge Agent* foi implantado através do *Docker Compose*. O Apêndice III mostra a configuração escolhida, vale comentar sobre as configurações *EDGE_ID* e *EDGE_KEY* e como adquiri-las, esses atributos funcionam como credenciais para validar a conexão do *Edge Agent* ao *Portainer Server*.

Essas credenciais são geradas na criação de um ambiente, ou seja, cada dispositivo conectado através de um *Edge Agent* possui um ambiente. A Figura 22 ilustra os ambientes utilizados pelas Raspberry Pis da planta, eles foram criados através da aba *environment* da *Portainer GUI*.

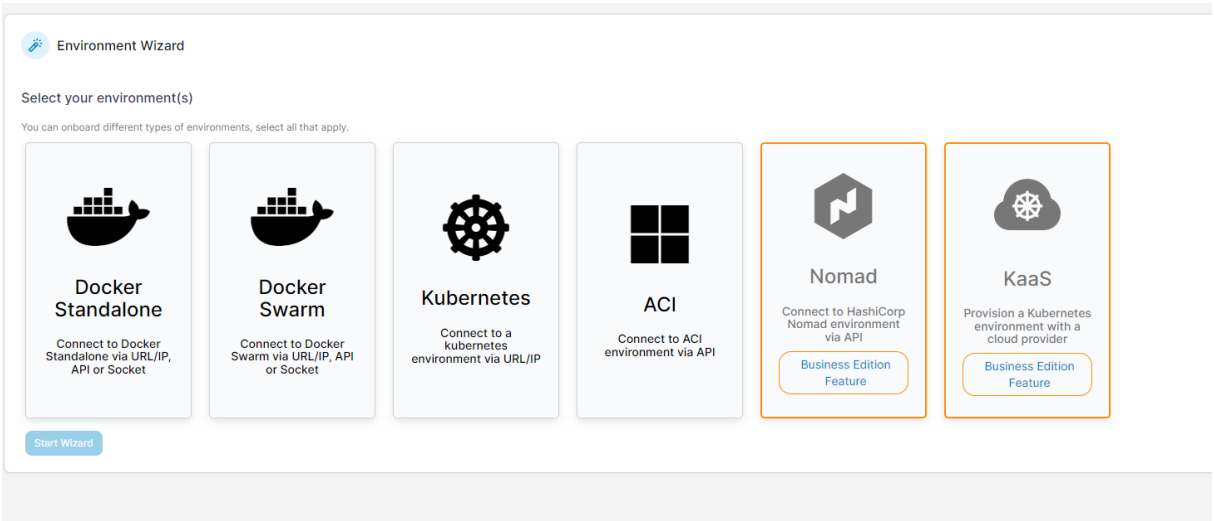
Figura 8 - Aba de ambientes da *Portainer* GUI



Fonte - Autoria própria

Para adicionar um novo ambiente foi selecionado o botão *Add enviroment*, o qual se encontra na aba *enviroment*, mostrada na Figura 8. Na página seguinte foi selecionado o tipo de ambiente mais adequado conforme o desejado, levando em consideração o objetivo e intuito do estudo, conforme mostra a Figura 9. Isto indica a funcionalidade do sistema, podendo ser adaptável de acordo com as necessidades do usuário.

Figura 9 - Aba de ambientes da *Portainer* GUI



Fonte - Autoria própria

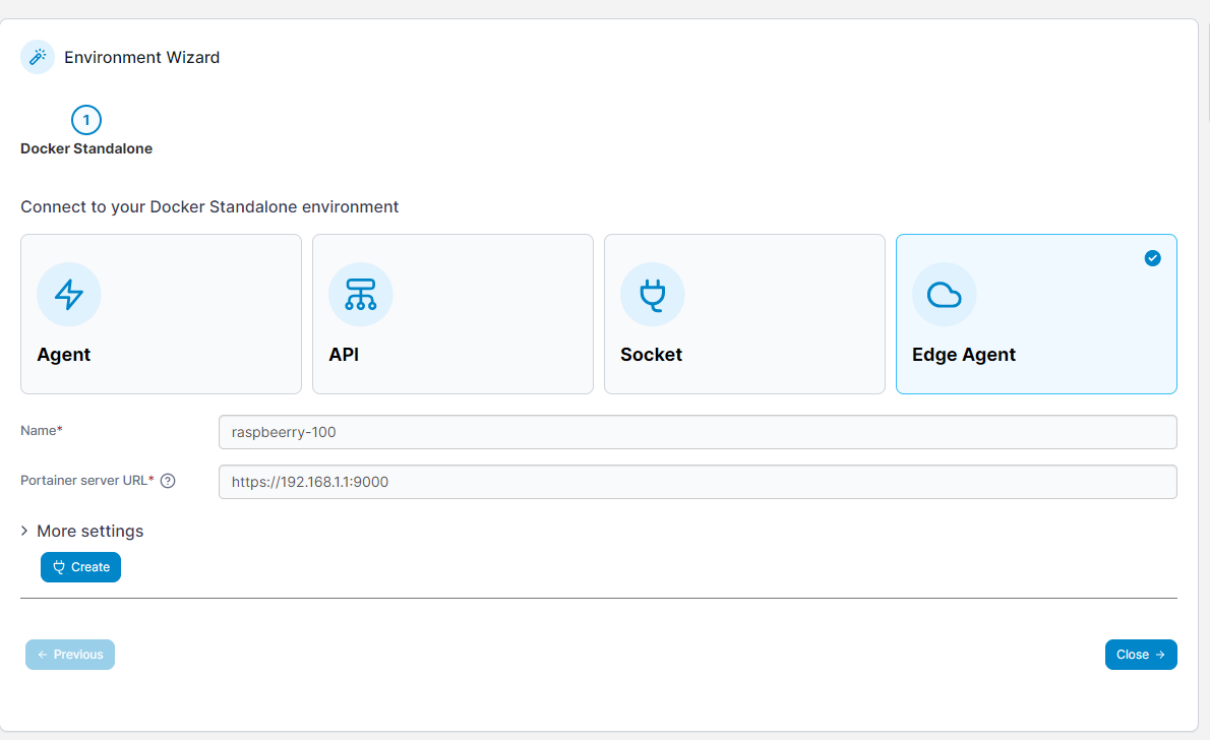
Partindo do princípio de que cada Raspberry Pi é um ambiente isolado, então podemos considerar que cada placa utiliza uma instancia *Docker* própria e com serviços destintos, assim assume-se que o ambiente mais indicado é o *Docker Standalone*.

No próximo passo, foi necessário utilizar o botão *Start Wizard*, o qual redireciona para uma nova página ilustrada na Figura 10, onde é necessário escolher a opção de conexão desejada e atribuir um nome para o ambiente. Para facilitar a identificação foi utilizado o padrão

raspberry-[três últimos dígitos] do IP da placa, por exemplo, para um IP 192.168.1.100 a nomenclatura correta é *raspberry*-100.

Também foi necessário informar o endereço IP da máquina hospedeira do *Portainer Server* para assim criar uma relação com a conexão indicada anteriormente. Depois do ambiente ter sido criado, é preciso informar os valores para as chaves EDGE_KEY e EDGE_ID, juntamente com isso é atribuído uma sugestão de instruções para instalar um container com o *Edge Agent* utilizando *Docker*, conforme ilustra a Figura 11.

Figura 10 - Aba de ambientes da *Portainer* GUI



Fonte - Autoria própria

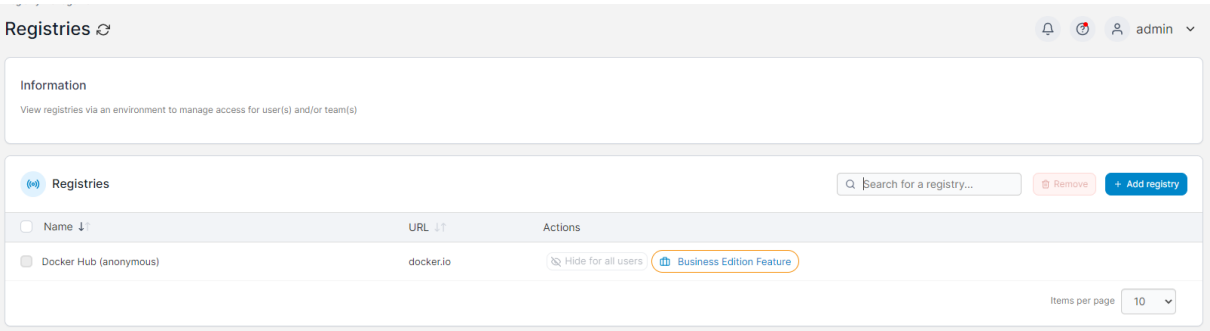
Figura 11 - Credenciais de conexão dos ambientes do *Portainer* Server

Fonte - Autoria própria

O comando sugerido pode ser executado através do *Docker Client*, mas como comentado anteriormente para este processo foi escolhido o arquivo de configuração *Docker Composer*, para que seja mantido o histórico da configuração em um arquivo armazenado no repositório de código *GitHub*.

Para acessar o acervo de imagens criadas foi necessário realizar a configuração de autorização do *Docker Hub*, pois se trata de um repositório de imagens privado e é necessário credencial para acessá-lo. Através da aba *Registries*, é possível obter esta autorização, selecionado o botão *Add registry* ilustrado na Figura 12.

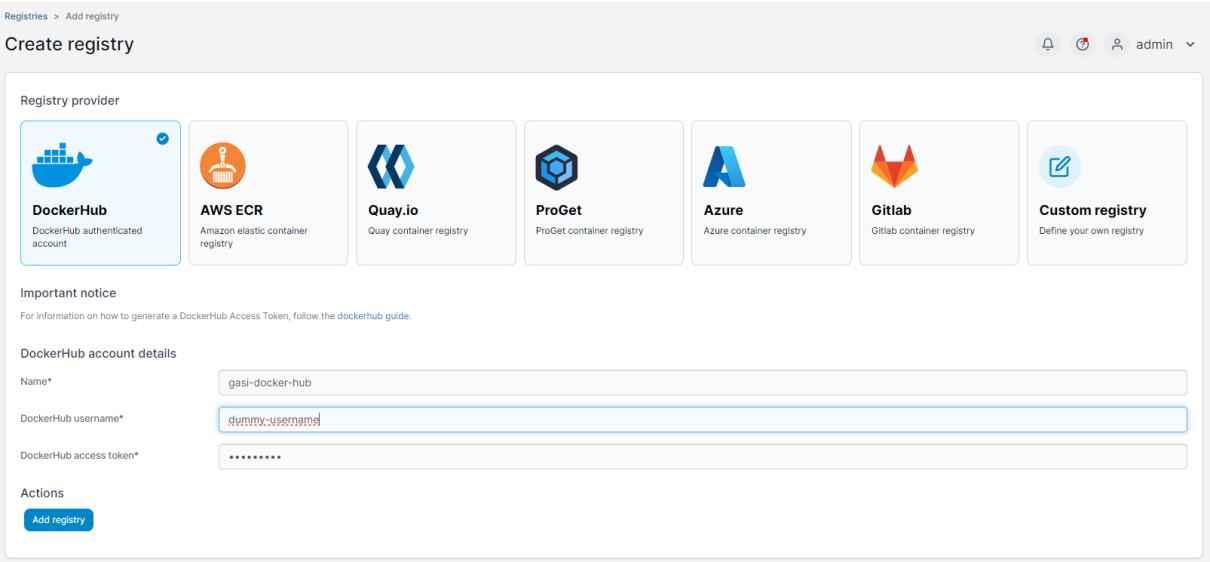
Figura 12 - Aba de gerenciamento de repositórios de *Docker* Images



Fonte - Autoria própria

Na aba de configuração foi necessário selecionar o repositório desejado, as opções disponíveis são mostradas na Figura 13, dentre elas estão os tipos de repositórios disponíveis do mercado para hospedagem de *Docker Images. E*m seguida informou-se as credenciais de acesso do GASI para acessar o repositório.

Figura 13 - Aba de configuração de um novo repositório de *Docker Images*



Fonte - Autoria própria

Com isso, ao estabelecer a comunicação entre *Edge Agent e Portainer* Server foi implantada uma ferramenta de gerenciamento remoto de containers com sucesso, finalizando assim o desenvolvimento deste trabalho. Esta comunicação impacta diretamente na melhoria contínua do gerenciamento dos serviços indicados anteriormente neste estudo.

Uma vez que a ferramenta é implementada pode-se adicionar ou remover serviços, oferece uma ferramenta de logs que auxilia na pesquisa através de palavras-chaves e retem esses logs por um período configurado. Além disso, é possível analisar o consumo de hardware, por exemplo, memória RAM, CPU, dentre outros.

3.5 Network e Segurança

Na arquitetura anterior qualquer aplicação que tivesse acesso a rede seria capaz de enviar requisições ao *broker* de mensagens assim como acessar qualquer serviço disponível na rede. Com a conteinerização, as *Raspberry Pis* possuem uma rede interna entre outros containers do *host*. Dessa forma, o acesso a tais serviços deve ser feito através do *broker* de mensagens ou pelo serviço de API. Neste modelo, os pontos possíveis de ataques são reduzidos, podendo focar as medidas de segurança nestes dois pontos de entrada. A ferramenta *Docker* oferece diversas formas de organizar os containers em redes, sendo as principais: *bridge*, *host* e *none*.

Uma rede de ponte (bridge) é uma rede virtual que permite a comunicação entre os contêineres *Docker* que estão sendo executados no mesmo *host*. Ela age como uma ponte entre os contêineres, permitindo que se comuniquem e compartilhem dados. Por padrão, o *Docker* cria uma rede de ponte chamada "bridge" quando o primeiro container é implementado, e novos contêineres se conectam a ela automaticamente. Você também pode criar suas próprias redes de ponte personalizadas para seus contêineres. A rede de ponte isola os contêineres da máquina host e de outras redes, o que ajuda a melhorar a segurança e torna mais fácil gerenciar seus contêineres.

No *Docker*, uma rede do tipo *host* permite que um contêiner use a rede da máquina hospedeira, em vez de ter sua própria rede isolada. Isso significa que o contêiner pode acessar diretamente as interfaces de rede do *host* e usar o mesmo endereço IP do *host*. O uso da rede do host pode fornecer melhor desempenho e menor latência para o contêiner, pois ele não precisa passar por camadas de rede adicionais. No entanto, isso também significa que o contêiner tem menos isolamento e segurança do *host* e de outros contêineres na mesma máquina. A rede do *host* pode ser útil para aplicativos que exigem acesso a uma interface de rede específica ou têm requisitos de rede específicos, mas deve ser usada com cautela e apenas para aplicativos confiáveis.

O modo de rede "*none*" do *Docker* é uma configuração que desabilita todas as capacidades de rede para um contêiner. Isso significa que o contêiner não pode se comunicar com o mundo externo ou com outros contêineres, e não possui um endereço IP atribuído a ele. O modo de rede "*none*" é útil quando você não deseja que um contêiner tenha nenhum acesso à rede, seja por motivos de segurança ou porque ele simplesmente não requer conectividade de rede. No entanto, tenha em mente que se um contêiner não tiver acesso à rede, ele pode não ser capaz de realizar determinadas tarefas ou acessar recursos que exigem conectividade de rede.

Para a planta utilizada neste trabalho, o modo de rede escolhido foi o de ponte (*bridge*) para que os *containers* pudessem se comunicar entre si dentro de um mesmo host enquanto ficam isolados da rede exterior com maior segurança. Boa parte dos serviços não recebe conexões, apenas se conectam a serviços externos (*Portainer Server* e *Broker* de Message), ou seja, não existe necessidade da rede externa acessar esses contêineres.

Existem duas exceções na planta: serviço de *API* e o *broker* de mensagens. Para essas exceções foram abertas portas de rede em seus respectivos *hosts*, dessa forma concedendo acesso apenas aos contêineres desses serviços mantendo os outros contêineres hospedados na mesma máquina isolados.

Figura 14 - Arquivo de instrução *Docker Compose* concedendo acesso a dispositivos externos

```
1   version: '3.5'
2   services:
3     daq:
4       image: gasiepgodoy/moleculer:4relind-rpi
5       container_name: 4relind-rpi
6       env_file:
7         - .env
8       restart: always
9       devices:
10        - /dev/i2c-1:/dev/i2c-1
11        - /dev/i2c-1:/dev/i2c-2
```

Fonte - Autoria própria.

Existem serviços que utilizam dispositivos externos, por exemplo o DAQ é dependente de um Inter-Integrated Circuit (I^2C). Nestes casos é necessário conceder acesso aos dispositivos. Essa configuração foi realizada através do atributo *devices* no arquivo de instrução *Docker Compose* mostrado na Figura 14.

4 Resultados e discussões

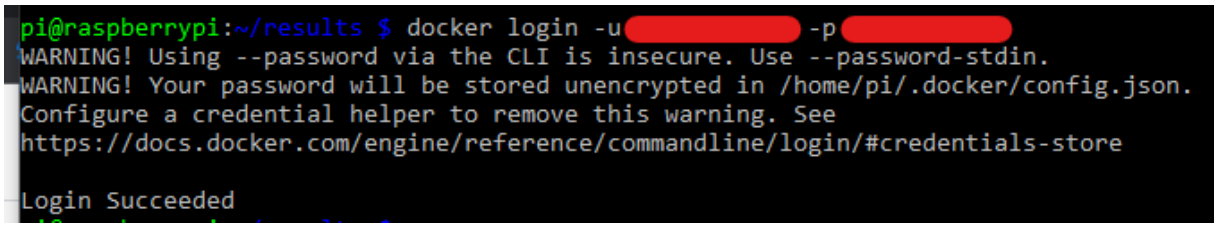
Seguindo a metodologia proposta, foi possível migrar todos os serviços para *containers* e durante este processo foram propostas algumas normas que devem ser seguidas para implantar um novo serviço. Para garantir a estabilidade dos serviços, deve-se manter a versão mais recente do código atualizada no *GitHub* e sua *Docker Image* hospedada no *Docker Hub*. Além de garantir serviços estáveis, traz liberdade para realizar alterações e adicionar novas funcionalidades com segurança, pois existem cópias armazenadas das versões funcionais.

Nessa arquitetura implantada, o *Docker Hub* age como um *Marketplace* de serviços tornando possível o *deploy* facilitado através do *Portainer Server*, utilizando sua funcionalidade de instanciar *containers* carregando imagens diretamente de um repositório remoto. A Figura 16 mostra o acervo de *Docker Images* desenvolvido durante este trabalho. Tais imagens se encontram disponíveis para serem utilizadas na implantação remota de serviços.

Durante este trabalho foi implantado três formas de disponibilizar serviços: (i) via *Docker Client* através da *Command Line Interface* (CLI), (ii) via *Docker Compose* através de arquivos de instruções e (iii) através da *Portainer GUI*, que é mais amigável com o usuário pois entrega suas funcionalidades por meio de uma interface gráfica.

O exemplo a seguir demonstra como implantar um serviço através do *Docker* Client: primeiro foi necessário começar uma sessão no *Docker Hub* através do comando *docker login*.

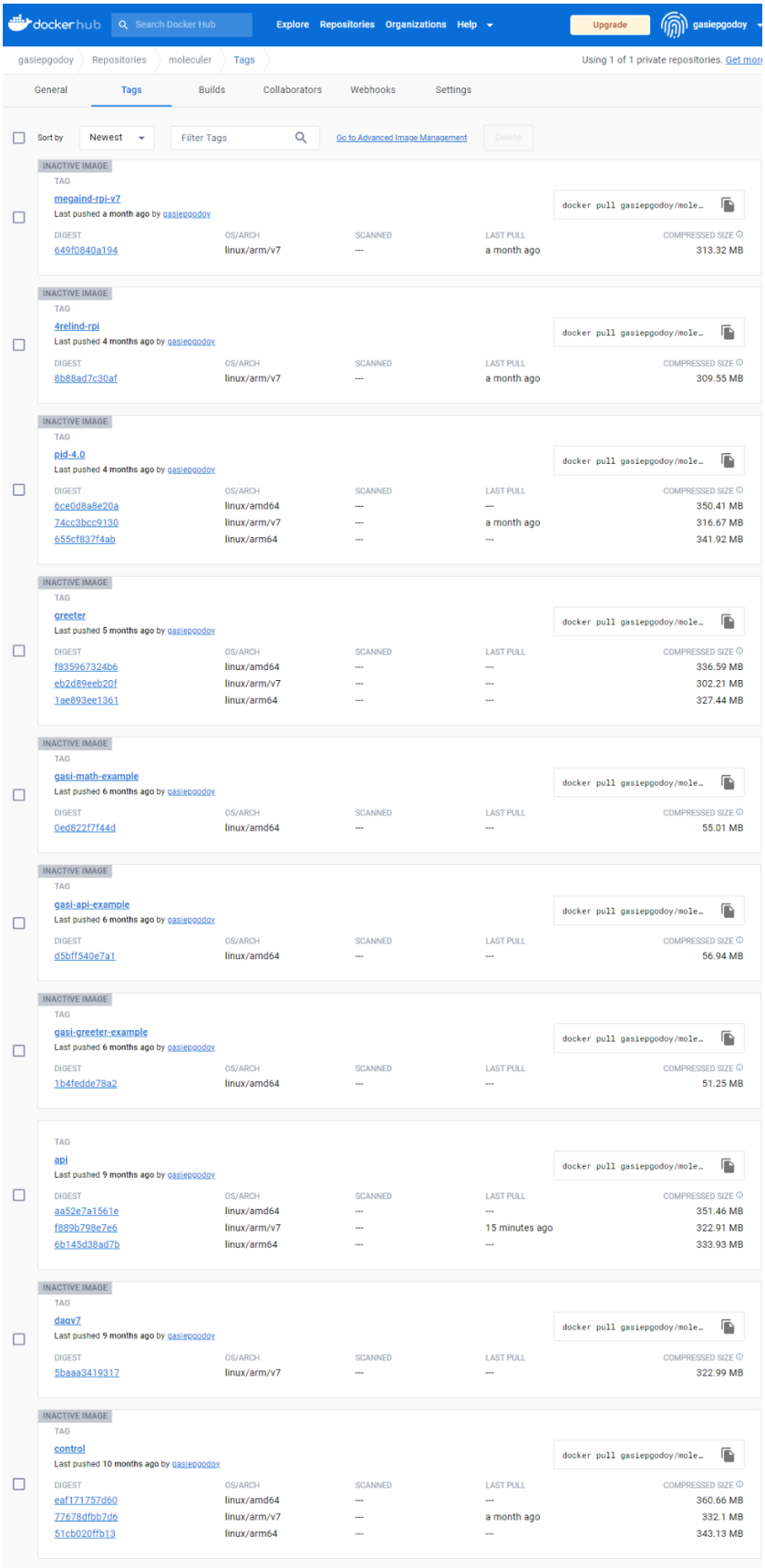
Figura 15 - Autenticação via *Docker* Cliente ao repositório de imagens



Fonte - Autoria própria

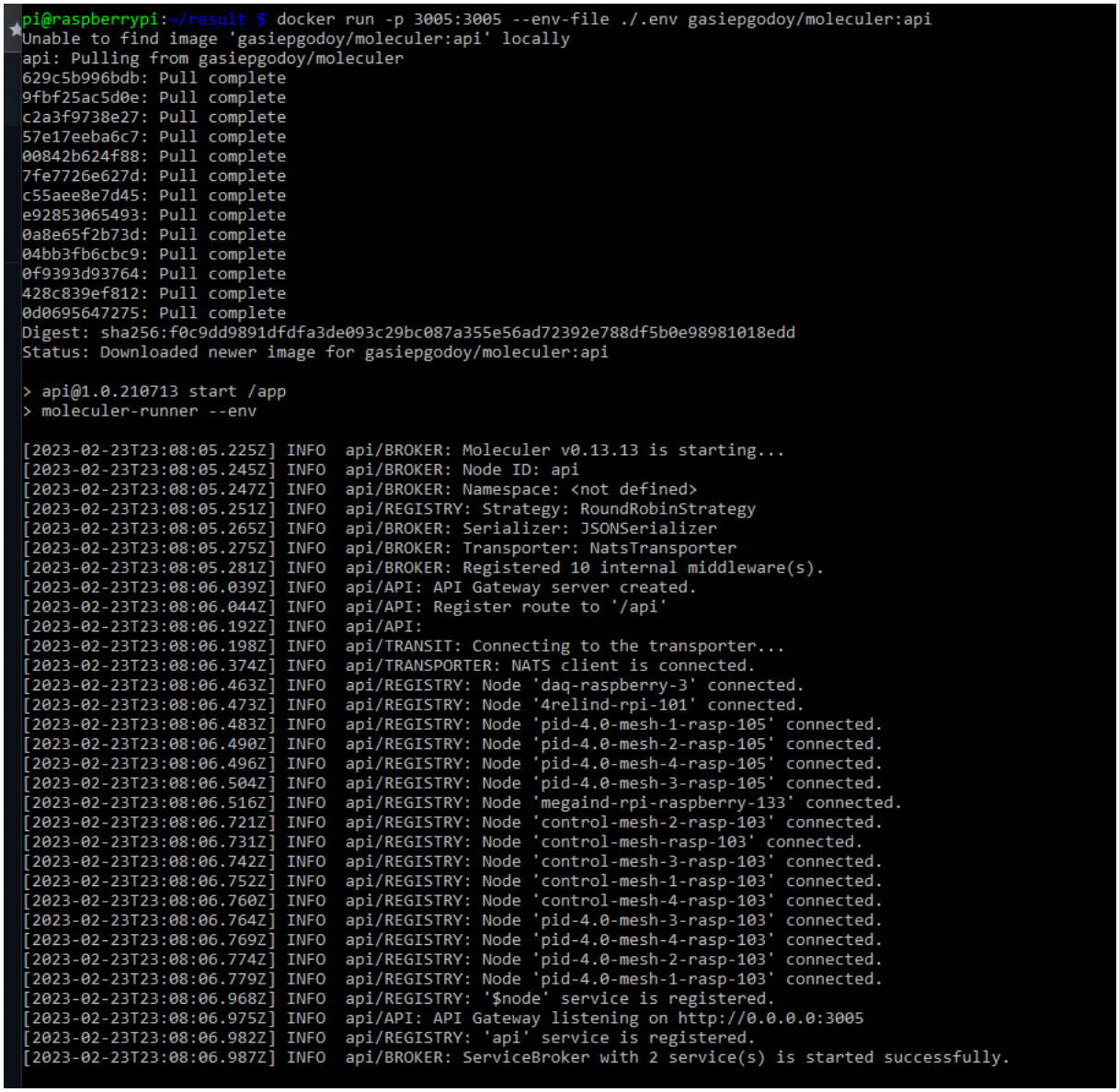
A Figura 15 mostra o processo de autenticação com o repositório de imagens através do comando *docker login -u username -p password*. A flag *-u* indica qual é o usuário, enquanto a flag *-p* indica a senha da credencial a ser utilizada. Tendo sucesso, a máquina possui acesso ao acervo de imagens hospedados no *Docker Hub* apresentado na Figura 16. Neste ponto foi possível executar o comando *docker run* para recuperar uma imagem do serviço *API* e disponibilizá-lo.

Figura 16 - *Docker* Images hospedadas no repositório remoto de imagens.



Fonte - Autoria própria

Figura 17 - Implantação de um serviço através do *Docker Client*



Fonte - Autoria própria

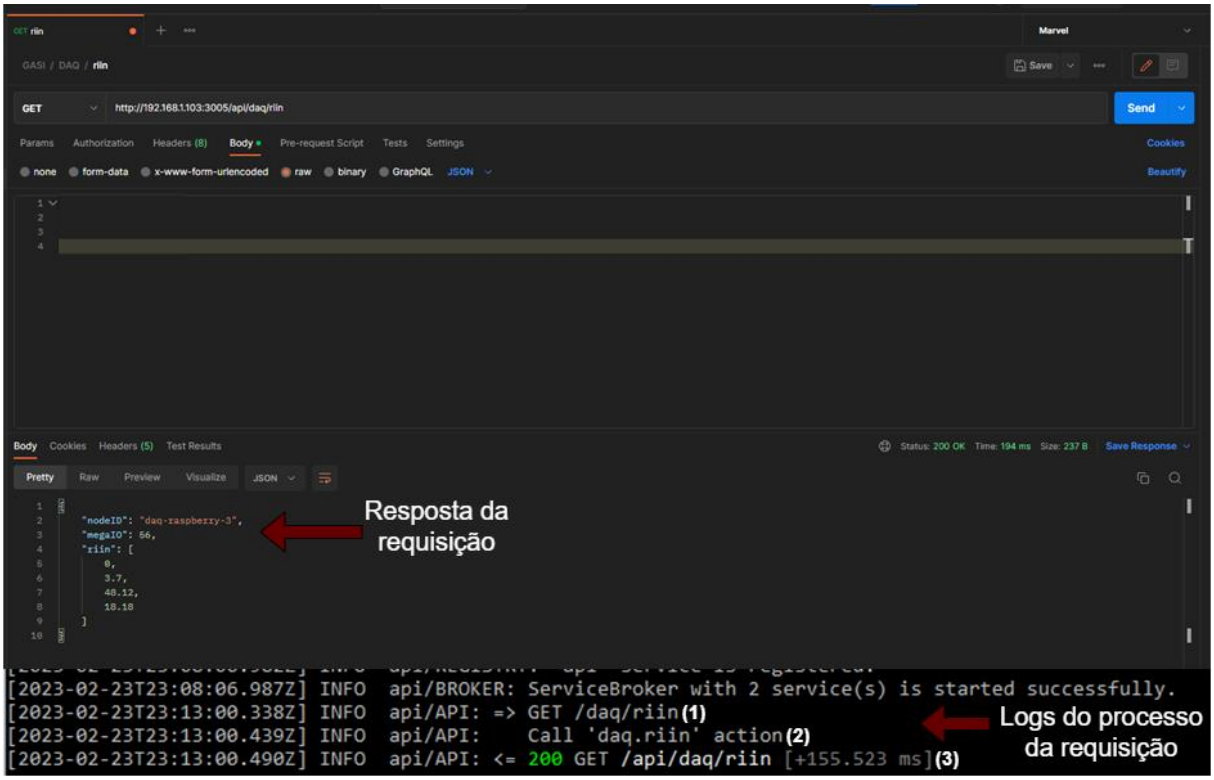
Logo após a execução do comando *docker run*, a Figura 17 mostra que não foi possível encontrar a imagem solicitada localmente, em seguida realiza seu *download* do repositório remoto. Após finalizado o download e instalação da imagem, o serviço é rapidamente disponibilizado e está pronto para ser requisitado. A Figura 18 mostra uma requisição no serviço API recém disponibilizado.

Apesar deste método ser prático, são necessárias algumas configurações manuais. Ao observar a primeira linha da Figura 17 foi indicado um mapeamento de porta através do parâmetro -p e foi carregado um arquivo de variáveis de ambientes através do comando *--env-file*. Fazer a implantação de múltiplos serviços simultaneamente torna-se inviável por esse método, mas o arquivo de instrução *Docker Compose* soluciona esse problema.

A Figura 18 mostra uma requisição HTTP no serviço DAQ através do aplicativo *Postman*, ainda na figura é indicado a resposta da requisição trazendo as informações solicitadas. Na parte inferior é apresentada os logs de processamento da requisição com cada

linha (L) de interesse numerada, nesses logs é possível identificar o método e o *endpoint* solicitado (L1), a ação (*L2*) e finalmente a resposta (L3).

Figura 18 - Requisição no serviço DAQ através da API recém implantada



Fonte - Autoria própria

O *Docker* Client oferece alguns comandos para gerenciar *containers*, a Figura 19 mostra a aplicação do comando *docker container ls* que tem a função de listar os containers ativos na máquina. A lista de contêineres mostra as seguintes informações básicas:

a) CONTAINER ID: Identificador único do container;
b) IMAGE: *Docker Image* utilizada para executar o container;
c) COMMAND: Comando executado no momento que o container foi implantado;
d) CREATED: Tempo passado desde a criação do container;
e) STATUS: Status do container;
f) PORTS: Portas do container, neste caso, indica as portas mapeadas entre container e host;
g) NAMES: Nome escolhido para o container.

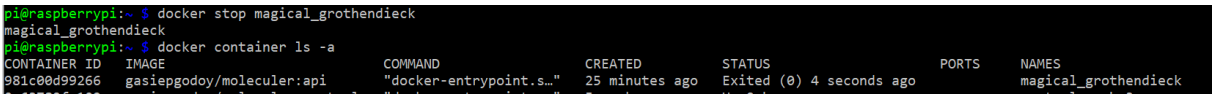
Figura 19 - Utilizando *Docker* Client para listar containers ativos

Fonte - Autoria própria

Já o comando *docker stop container_name* tem a função de parar a execução do *container* indicado, conforme mostra a Figura 20, em seguida é apresentado todos os containers parados pela execução do comando. Depois foi executado novamente o comando *docker*

container ls -a neste caso foi utilizada o parâmetro -a para que o comando *container ls* liste também *containers* que não estejam ativos. Nessa nova listagem, nota-se que o *status* mudou e não existem mais portas mapeadas.

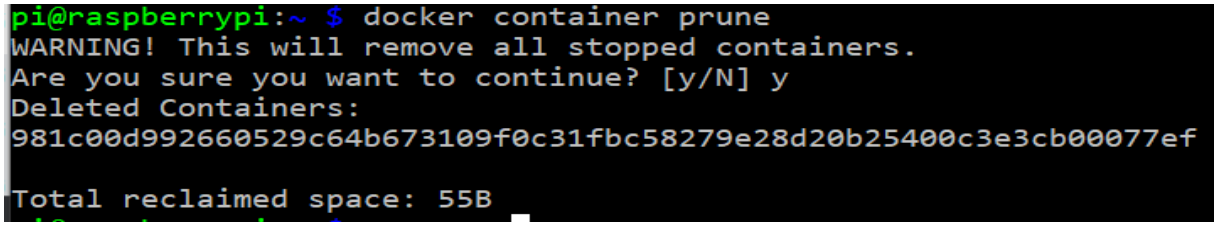
Figura 20 - Utilizando *Docker Client* para inativar um container



Fonte - Autoria própria

Se tratando de um exemplo, foi executado o comando *docker container prune* para remover containers inativos e liberar espaço em memória, a Figura 21 mostra esse processo.

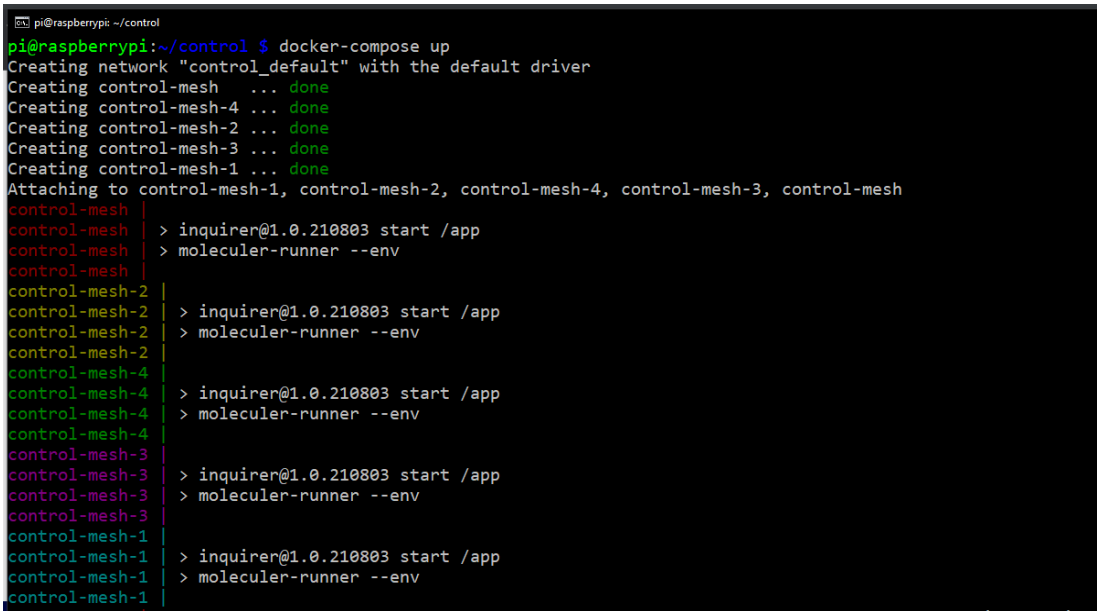
Figura 21 - Utilizando *Docker Client* para desligar um container ativo



Fonte - Autoria própria

Como comentado anteriormente, implantar múltiplos *container* através do *Docker Client* é trabalhoso e nada eficiente, por tanto, para implantar os 5 serviços de controle foi decidido pela utilização do arquivo de instruções *Docker Compose* anexado no Apêndice IV. No anexo, os blocos nomeados *control*, control_1, control_2, control_3 e control_4 são configurações dos *containers* responsáveis por cada malha de controle. Já o Apêndice V exemplifica um arquivo contendo variáveis de ambientes para serem carregadas no *container*.

Figura 22 - Implantação de múltiplos serviços simultaneamente através de *Docker Compose*



Fonte - Autoria própria

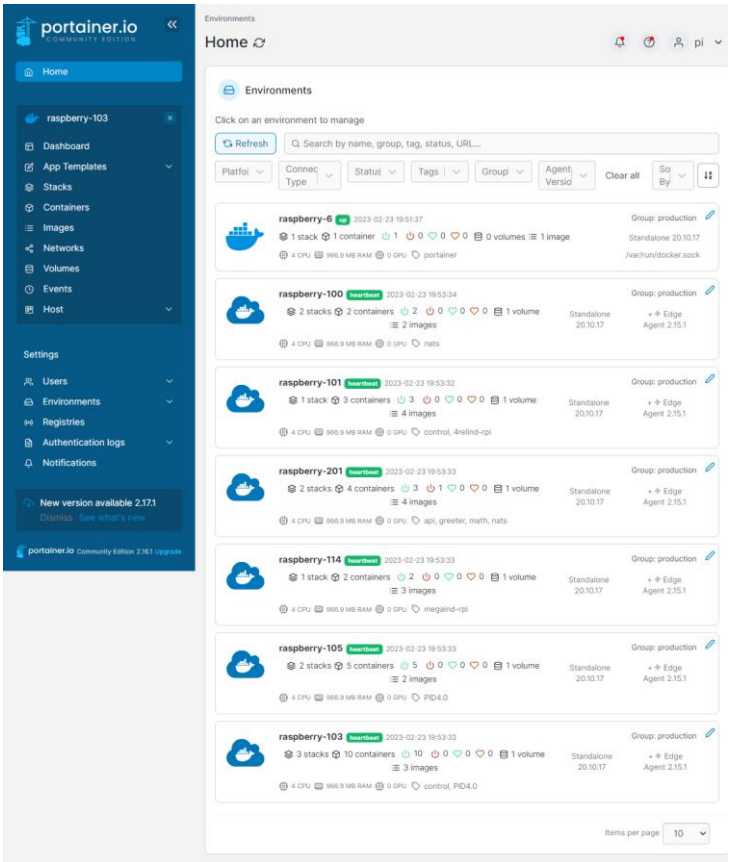
Para executar um arquivo de instrução *Docker Compose*, basta navegar até o diretório onde o arquivo está localizado e executar o comando d*ocker-compose up* em seguida a ferramenta *Docker* realiza o download das imagens caso necessário, no cenário apresentado na Figura 22 a máquina hospedeira já possuía as imagens e, portanto, pulou este passo indo diretamente para a criação da rede e em seguida dos contêineres. Ainda na Figura 22, após a criação dos containers observa-se que cada um deles executou o comando *moleculer-runner --env*, comando utilizado para disponibilizar os serviços escritos em *MoleculerJS*.

O comando para executar um arquivo *Docker Compose* no ambiente de produção é seguido pelo parâmetro -d utilizado para execução desacoplada, fazendo com que seja possível fechar o terminal sem encerrar os *containers*.

O arquivo de instruções *Docker Compose* é muito útil para disponibilizar arquiteturas completas de forma rápida, mas ainda necessita um conhecimento técnico mais profundo e não é muito amigável e ainda era necessário acessar as *Raspberry Pis* diretamente para orquestrar e monitorar *containers*. Neste ponto a implantação e remoção de serviços já havia melhorado significantemente, entretanto, ainda possuía suas limitações.

A implantação do *Portainer* traz mais facilidade para implantação de container, não sendo necessário o conhecimento técnico muito profundo devido a sua interface gráfica amigável. A Figura 23 ilustra a página inicial da ferramenta onde cada *Raspberry Pi* possui um ambiente exclusivo.

Figura 23 - Página inicial da ferramenta *Portainer*

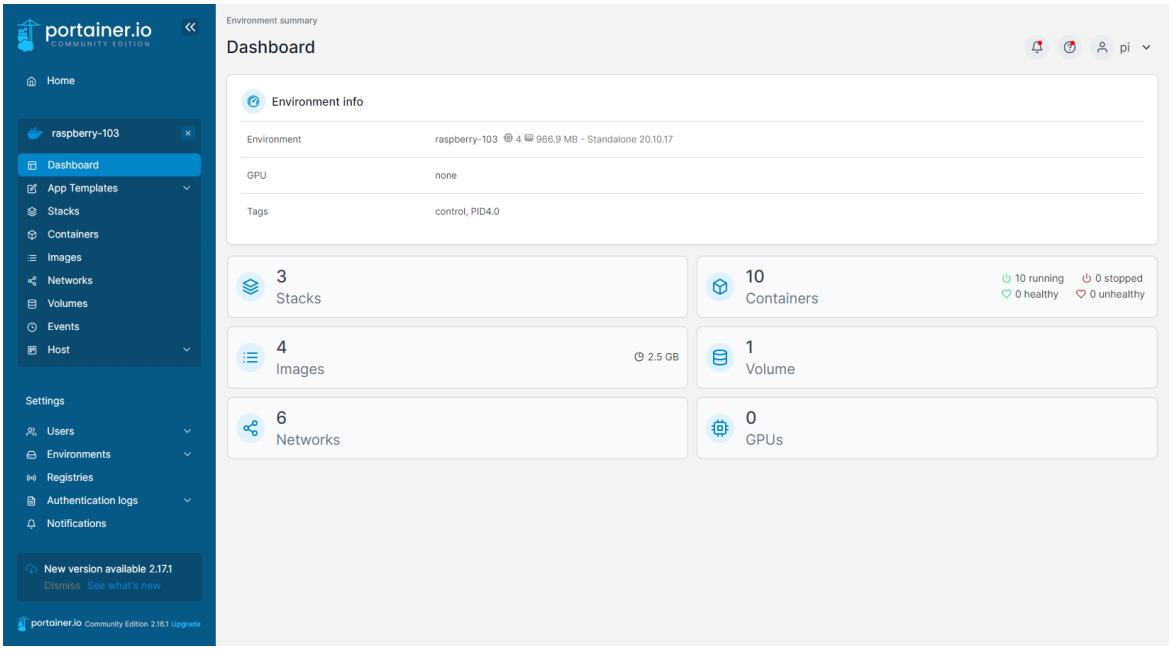


Fonte - Autoria própria

Com o *Portainer*, obteve-se acesso remoto aos *containers* instanciados nas *Raspberry* da planta. Através da página de ambientes do *Portainer Server* verifica-se, conforme mostra a Figura 23, um pequeno resumo dos *containers*, volumes e imagens instaladas em cada *Raspberry Pi* conectada. Os ambientes foram rotulados com o recurso de *tags* para indicar quais serviços estão sendo executados em cada placa.

Ao acessar um dos ambientes a ferramenta redireciona para o *Dashboard* do ambiente selecionado, essa nova página mostra informações básicas ilustradas na Figura 24. O *dashboard* de ambiente apresenta algumas informações básicas como o número de imagens, contêineres, volumes, networks etc. Ainda neste dashboard de ambiente, é possível selecionar qualquer um desses atributos de container para conseguir informações mais especializadas assim como editar, deletar ou adicionar novos itens.

Figura 24 - Dashboard de um ambiente da ferramenta *Portainer*

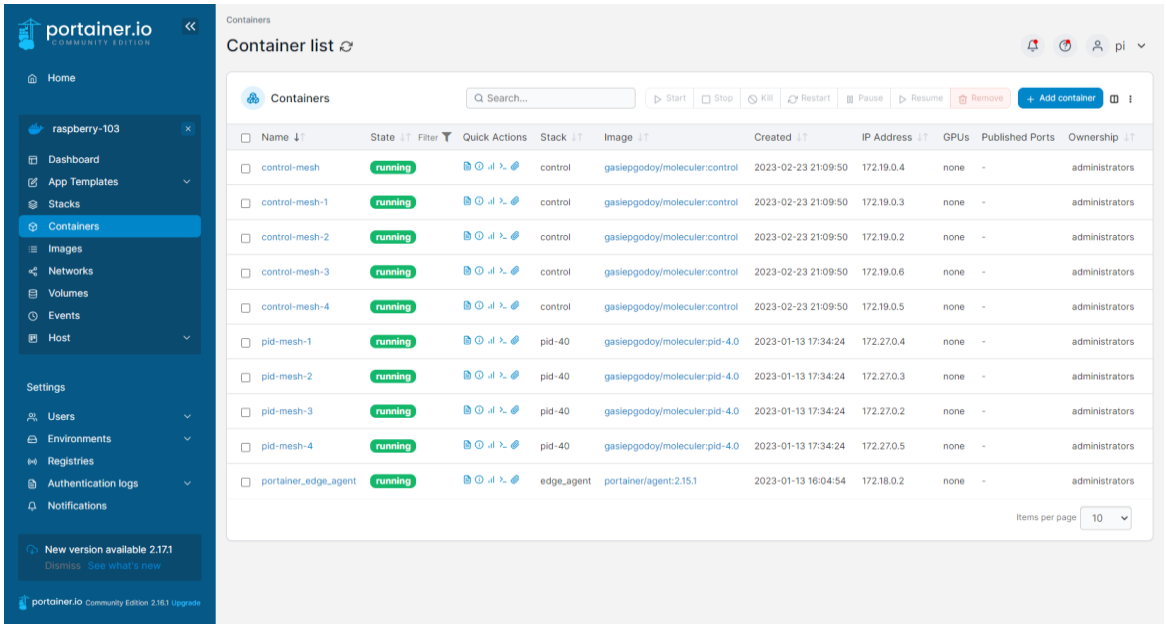


Fonte - Autoria própria

Para este trabalho, o maior ponto de interesse é a aba de *containers*, pois é nessa aba onde é será possível adicionar, remover e editar serviços. Ao acessar a página de contêineres, ilustrada na Figura 25, observa-se uma lista de contêineres ativos para o ambiente selecionado. Nesta lista é possível verificar algumas informações básicas, como nome, status e imagem utilizada. Este mesmo *dashboard* oferece algumas ações que podem ser aplicadas a contêineres selecionados, tais como pausar, parar, remover ou resumir.

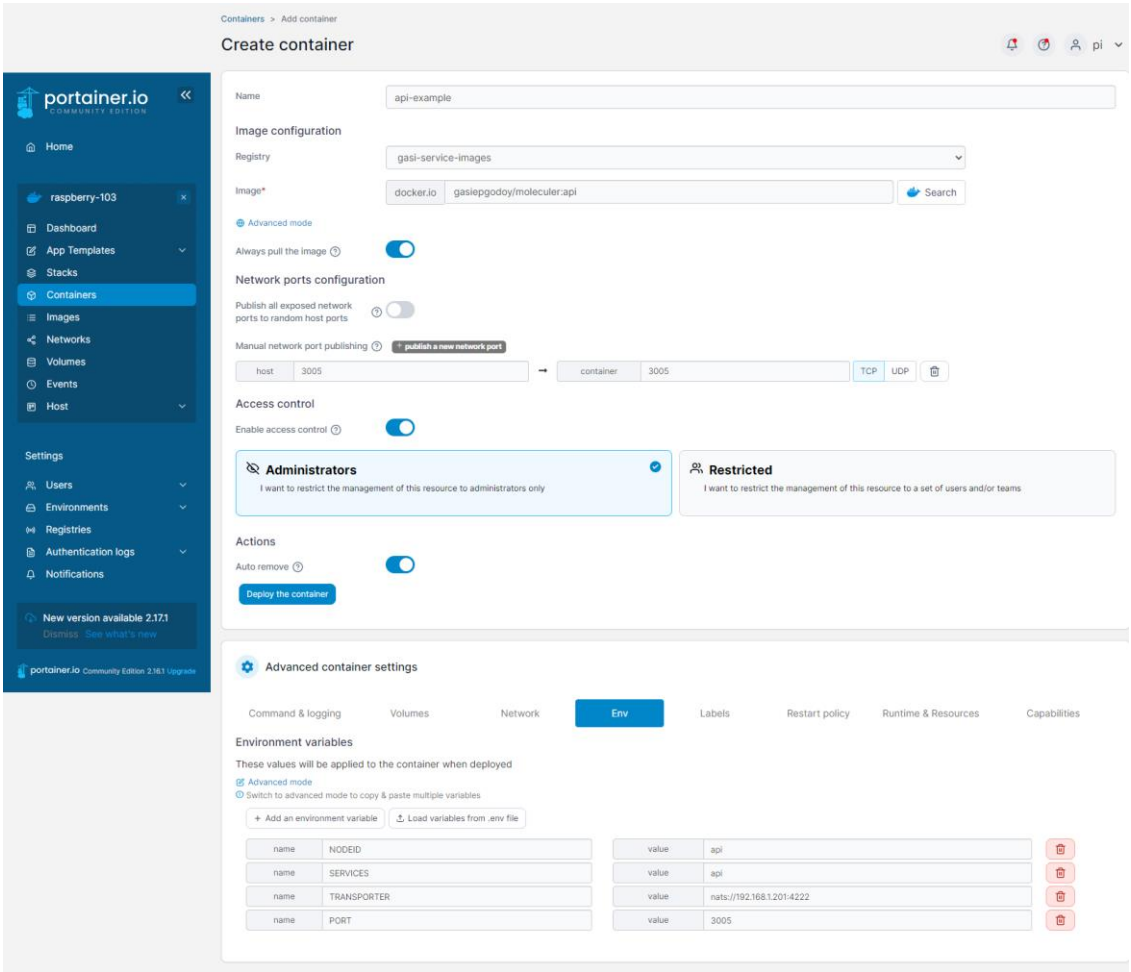
De forma a implantar o serviço API na máquina apresentada na Figura 25, foi selecionado o botão *Add container* mostrado no dashboard, após ser direcionado para a página de criação de containers, foi necessário preencher o formulário apresentado na Figura 26 com as configurações desejadas, como assim como carregar as variáveis de ambiente utilizadas pelo serviço.

Figura 25 - Dashboard de um ambiente da ferramenta *Portainer*



Fonte - Autoria própria

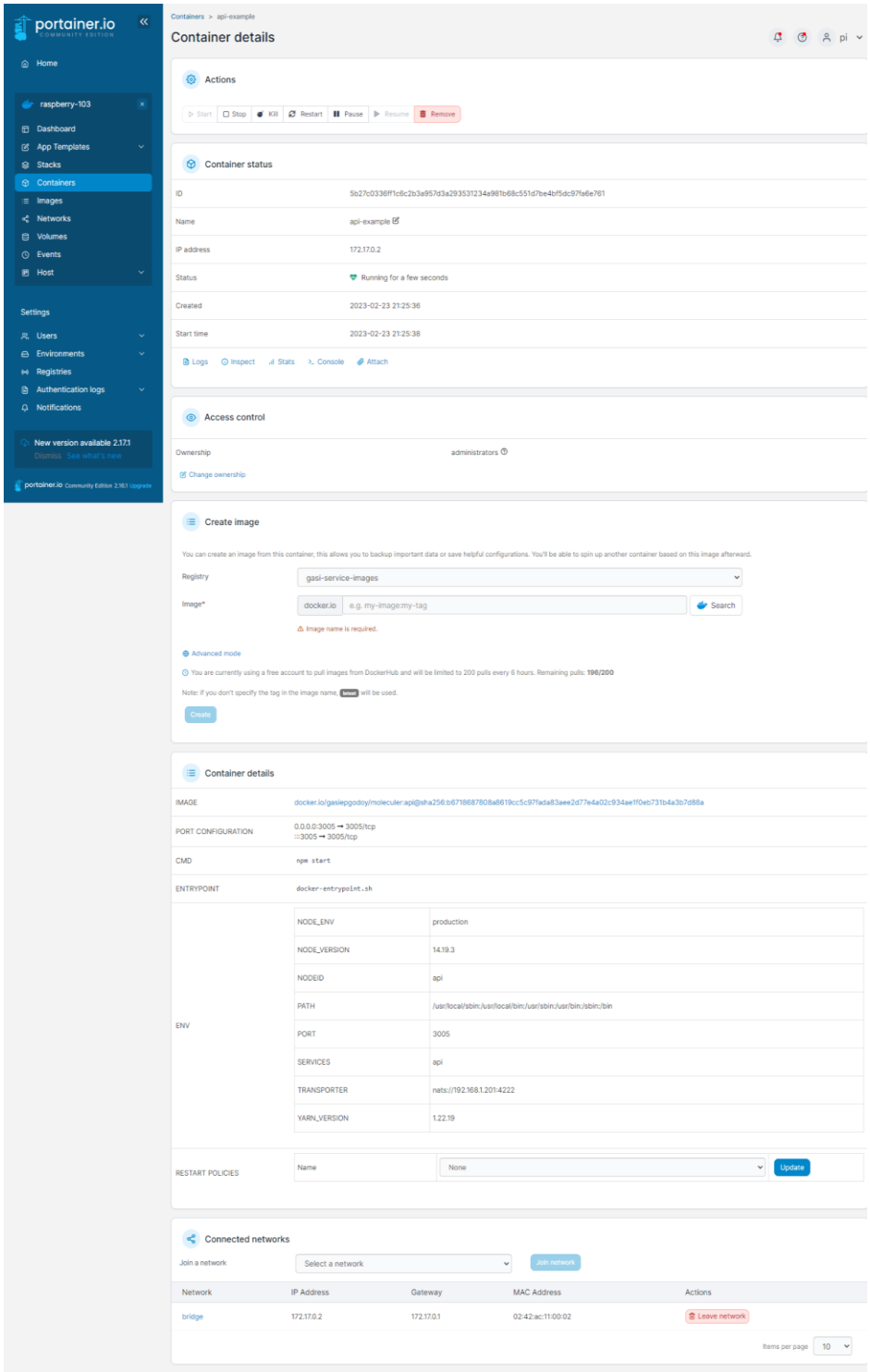
Figura 26 - Dashboard de um ambiente da ferramenta *Portainer*



Fonte - Autoria própria

Terminadas as configurações, o botão *Deploy the container* foi usado para disponibilizar o serviço. Voltando a página de *containers*, ao acessar o *container* recém implantado foi possível verificar informações das configurações escolhidas, conforme ilustra a Figura 27. Nesta mesma página também é possível editar essas configurações, caso necessário. Existem outras funcionalidades que podem ser acessadas por essa mesma página, como clonar, remover, pausar ou parar um container.

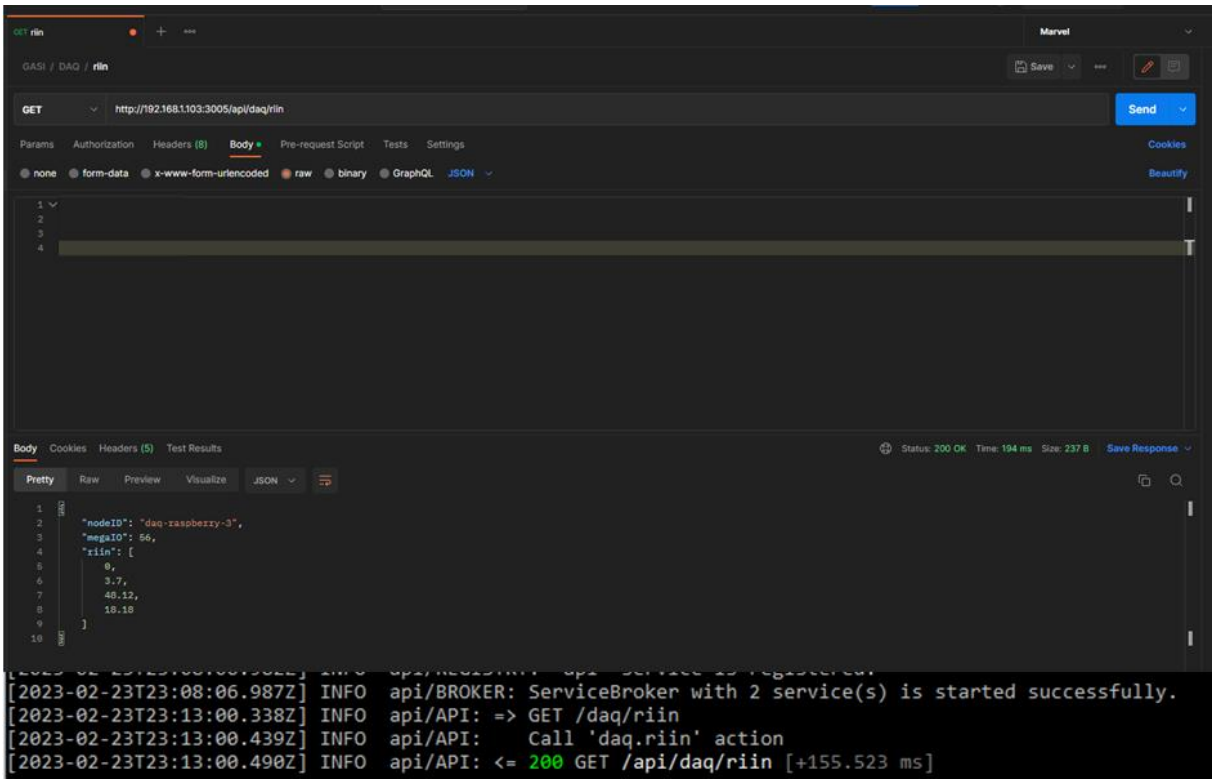
Figura 27 - Página de detalhes de um container



Fonte - Autoria própria

A Figura 28 mostra uma requisição no *container* recém implantado através do *Portainer*, essa requisição foi realizada no serviço DAQ solicitando uma ação de *riin*. Essa requisição é semelhante a apresentada na Figura 18, a única diferença foi o processo de *deploy* utilizado.

Figura 28 - Aplicativo Postman realizando uma requisição HTTP

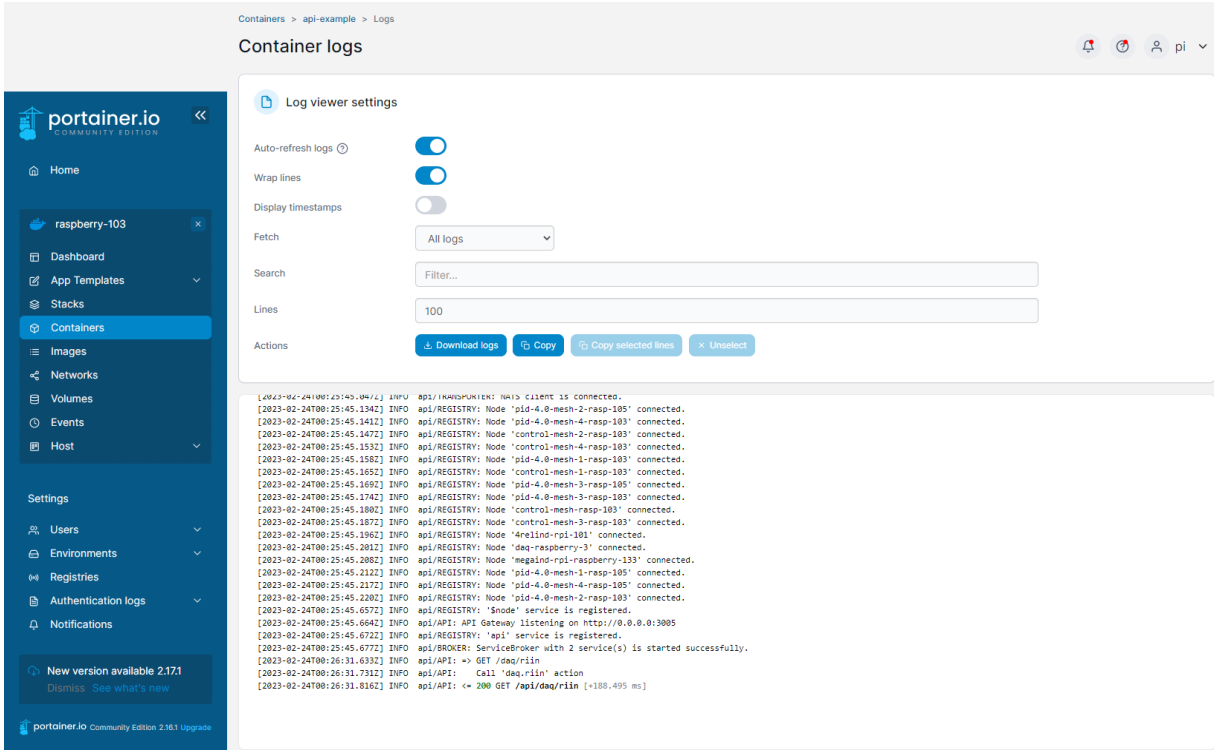


Fonte - Autoria própria

A Figura 29 mostra essa requisição sendo processada dentro do *container*, o processo é mostrado pela ferramenta de *logs* oferecida pela *Portainer GUI*. Diferente da visualização através de uma *Command Line Interface* (CLI), essa ferramenta de oferece filtros e retenção de *logs* por um período, facilitando a consulta e mantendo um histórico.

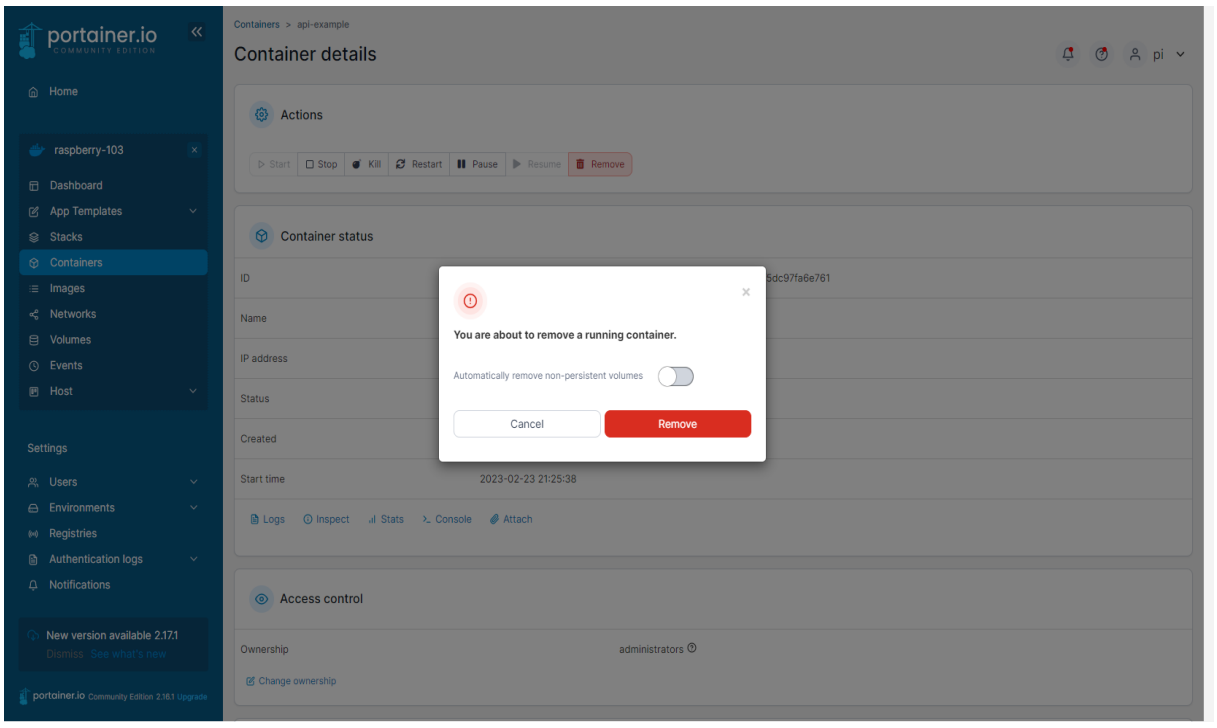
Como a placa utilizada para este exemplo não hospeda um serviço API no ambiente de produção, foi necessário removê-lo. Para isto, o container em questão foi acessado e o botão *Remove* selecionado em seguida foi necessário aprovar a remoção conforme ilustra a Figura 30. A Figura 31 mostra que após a confirmação de remoção uma mensagem indica que a operação foi um sucesso e o container não está presente na lista de containers ativos.

Figura 29 - Logs de um container, acessada através da página de detalhes de containers



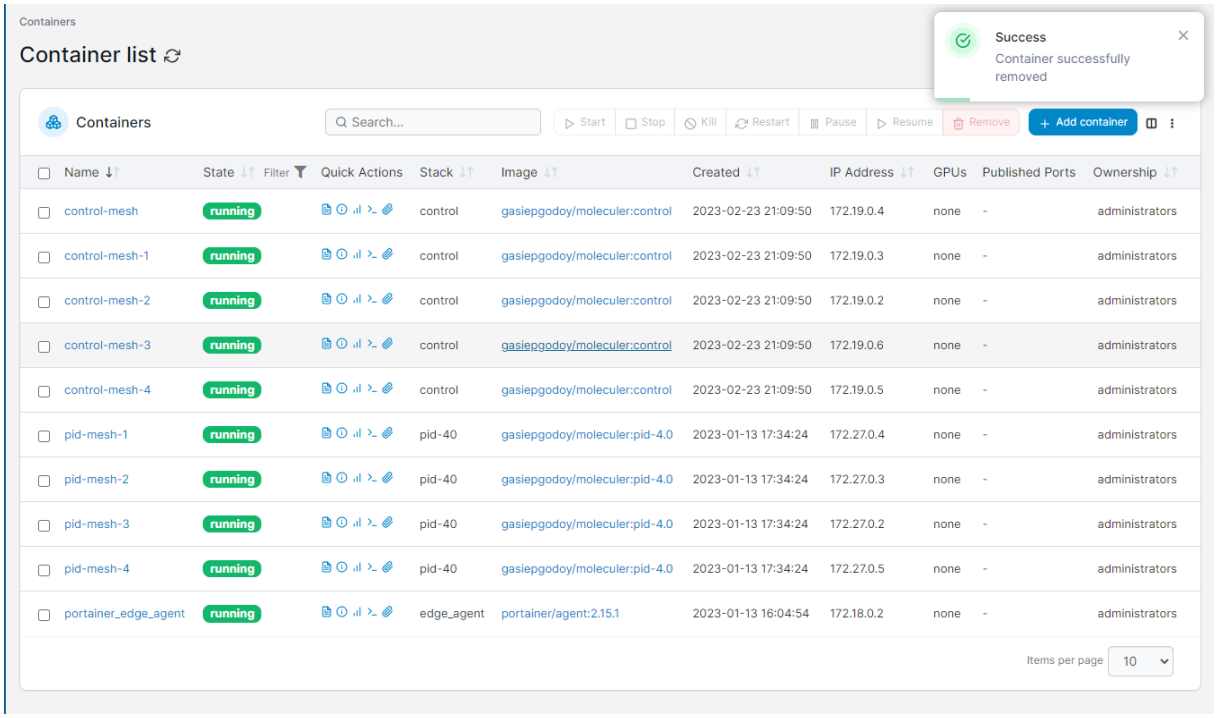
Fonte - Autoria própria

Figura 30 - Remoção de um container através do *Portainer*



Fonte - Autoria própria

Figura 31 - Remoção de um container através do *Portainer*



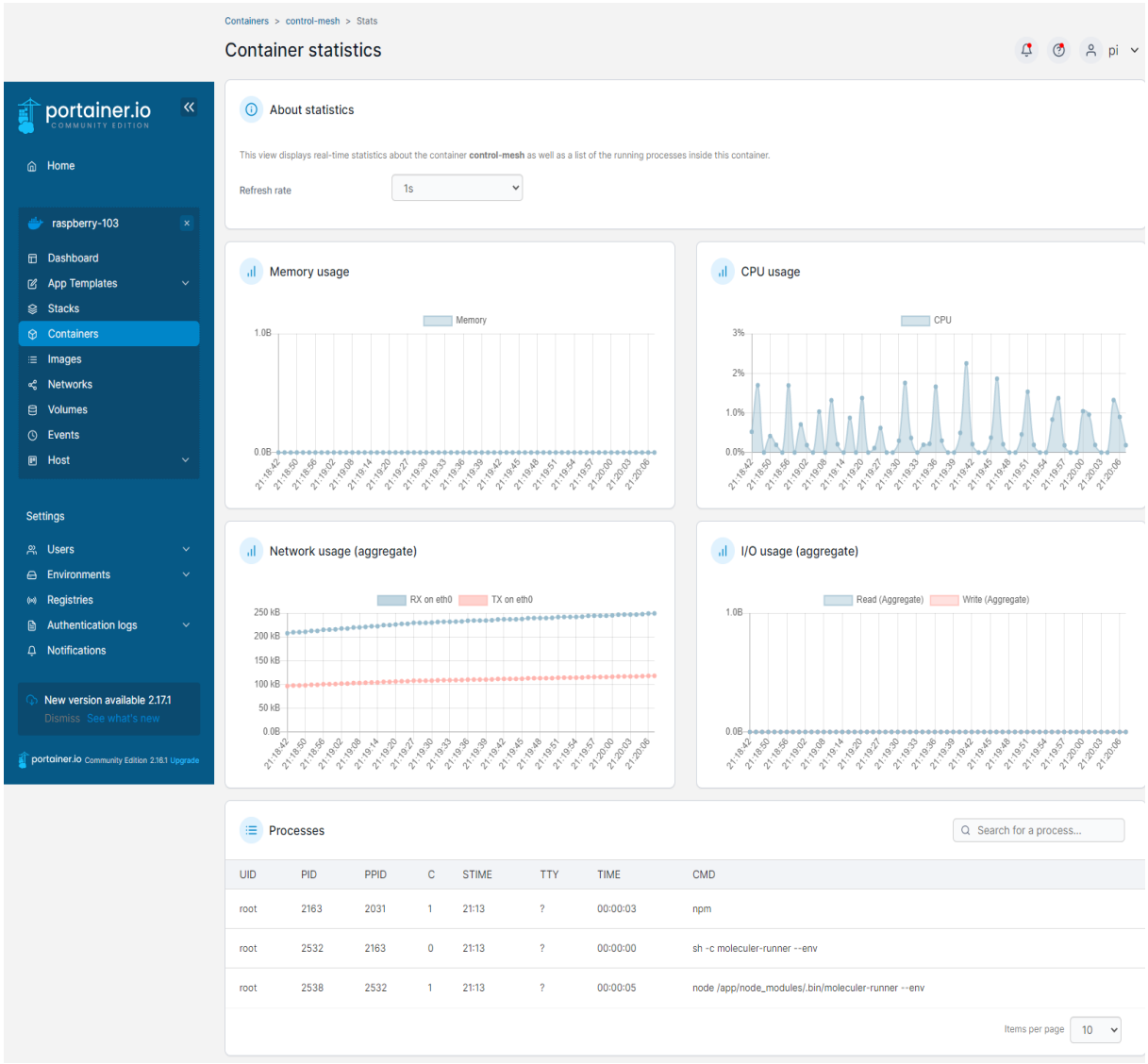
Fonte - Autoria própria

A Figura 32 mostra outra das funcionalidades do *Portainer*: a possibilidade de monitorar o consumo de hardware através da aba de estatísticas do container. Os dados apresentados na Figura 32 foram observados em uma malha de controle enquanto a planta estava ativa. Através desta funcionalidade é possível identificar gargalos de *hardware* indicando a necessidade de melhoria no processo, seja com diminuição de carga ou melhoria da máquina hospedeira.

O *dashboard* apresentado na Figura 32 permite escolher a taxa de atualizações das monitorias e conta com 5 visões diferentes, sendo elas:

a) Memory Usage: Gráfico informando a memória utilizada;
b) CPU Usage: Gráfico informando o poder computacional utilizado;
c) Network Usage: Gráfico informando a quantidade de dados transacionados através da rede;
d) I/O Usage: Gráfico informando a quantidade de dados consumidos por processos de entrada e saída;
e) Processes: Tabela mostrando os processos ativos no container.

Figura 32 - Página de estatísticas de container, acessada através do dashboard de um container



Fonte - Autoria própria

A Figura 33 mostra a disposição dos serviços em sua respectiva *Raspberry Pi* ao finalizar a configuração de todos os ambientes e conectar o *Portainer a* cada uma das *Raspberry Pis.*

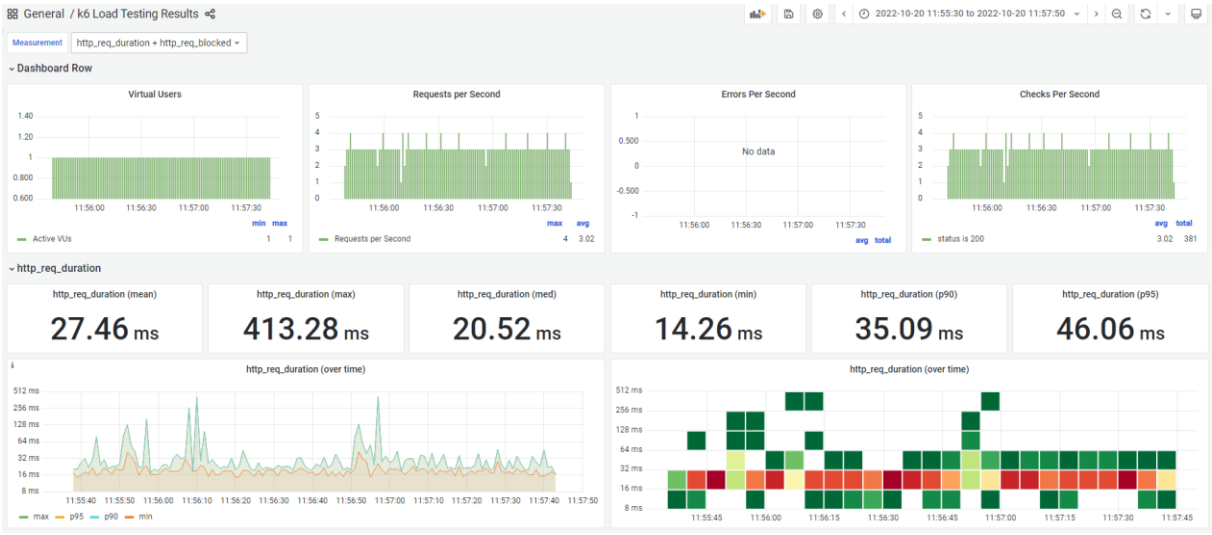
Figura 33 - Disposição final dos serviços disponíveis em cada placa *Raspberry*.

Fonte - Autoria própria

Por fim, avaliou-se o impacto na performance desta nova arquitetura. Para realizar os testes de carga foi utilizado *software* k6 e para a visualização dos dados foi construído um *dashboard* com auxílio do Grafana, ambas as ferramentas foram disponibilizadas pela Grafana Labs (2022).

As baterias de testes consistiram em disparos ao serviço DAQ através do serviço API para mensurar o tempo de resposta da planta. Cada bateria disparou requisições a partir de uma máquina virtual continuamente durante 60 segundos, o resultado do teste era apresentado no *dashboard* da Figura 34.

Figura 34 - Dashboard dos tempos de respostas ao requisitar um serviço.



Fonte - Autoria própria

A primeira configuração utilizada para os testes foi executar o serviço diretamente pelo *Node.JS*. Esse teste foi repetido cinco vez e se resultado é mostrado na Tabela 1.

Tabela 1 - Tempo de resposta utilizando NodeJS

Teste	média (ms)	p90 (ms)	p95 (ms)
1	19,28	23,88	27,65
2	19,42	25,58	28,16
3	18,79	23,79	26,56
4	19,09	22,56	24,7
5	19,75	25,26	29,77
Desvio Padrão	**0,36**	**1,22**	**1,89**
Média	**19,28 ± 0,36**	**23,88 ± 1,22**	**27,65 ± 1,89**

Fonte - Autoria própria

A segunda coluna da Tabela 1 traz o tempo de resposta médio enquanto as segunda e terceira colunas trazem o percentil. Um percentil é uma medida estatística que representa uma posição específica dentro de uma distribuição de dados. É frequentemente usada para descrever a posição relativa de um indivíduo ou grupo de indivíduos dentro de uma população maior.

Para calcular um percentil, primeiro classifica-se os dados do mais baixo para o mais alto. Então, determina-se o percentil de um valor específico comparando-o com os outros valores no conjunto de dados. Por exemplo, se um valor está no 70º percentil, isso significa que 70% dos valores no conjunto de dados são menores do que esse valor. Ou seja, as colunas dois e três mostram que 90% e 95% dos valores no conjunto são menores que o valor apresentado.

Para a segunda configuração de testes, foi utilizado o mesmo *endpoint*, desta vez o serviço foi executado em um *container* resultado é mostrado na Tabela 2.

Tabela 2 - Tempo de resposta utilizando Container

Teste	média (ms)	p90 (ms)	p95 (ms)
1	19,93	25,52	29,43
2	19,72	28,39	34
3	19,85	27,31	32,06
4	19,63	25,3	30,37
5	19,84	26,01	30,23
Desvio Padrão	**0,12**	**1,31**	**1,83**
Média	**19,84 ± 0,12**	**25,77 ± 1,31**	**30,3 ± 1,83**

Fonte - Autoria própria

Ao comparar os resultados mostrados nas Tabelas 1 e 2, nota-se que o tempo de reposta ao executar o serviço através do *NodeJS* é ligeiramente menor e, portanto, mais performático. Este resultado não surpreende, já que ao executar em um *container* é adicionado o tempo de requisição e resposta entre *host* e *container*.

Diante do que foi visto, a arquitetura implantada foi capaz de centralizar o acesso remoto aos *Raspberry Pi* na ferramenta *Portainer*. Acessar todos os ambientes em uma mesma ferramenta permite orquestrar ações em dispositivos diferentes de forma mais eficiente e amigável.

Outra problemática da arquitetura anterior, é a falta de uma ferramenta de auto execução no *NodeJS* durante a inicialização do dispositivo. Para contornar esse problema, primeiro foi adotada uma configuração através de *script* para executar os serviços desejados ao ligar a *Raspberry Pi*. Este método se tornou ineficaz por não possuir uma forma amigável de gerenciar os serviços e a necessidade de configurações específicas para cada placa. Em seguida, foi adotada uma ferramenta chamada PM2, que é um gerenciador de processos e apesar de possuir uma interface amigável para gerenciar os serviços em execução é uma dependência que consome recursos que podem ser mais bem aproveitados pela operação da planta, além disso, para gerenciar os serviços é necessário acessar individualmente cada placa. Os *containers* possuem uma ferramenta de auto inicialização nativa e com configuração escalável, obtendo como resultado uma melhor gestão de recursos de *hardware* sem renunciar à alta disponibilidade logo após a inicialização do dispositivo.

Por fim, foi possível encontrar uma alternativa para gerenciar, visionar e disponibilizar múltiplos microsserviços que podem ser combinados para construção de aplicações industriais.

5 Conclusão

A nova arquitetura trouxe segurança e estabilidade a esteira de desenvolvimento com o versionamento dos serviços, além disso, a padronização implantada no processo de desenvolvimento de um serviço desde a criação até o *deploy* em produção torna a introdução de novos desenvolvedores ao projeto mais independente, diminuindo a necessidade de acompanhamento.

O objetivo principal foi atingido com o gerenciamento remoto dos *containers* através do *Portainer*, tornando possível o *deploy* ou desligamento de serviços, monitoria da saúde dos *containers* e consumo de *hardware*, tudo agrupado em uma mesma interface de usuário.

Um ponto negativo que a nova arquitetura trouxe é a necessidade de criar uma imagem sempre que houver alguma alteração em código, diferente da arquitetura antiga onde era possível realizar mudanças no código e já o executar novamente. Esta é uma desvantagem devido ao comportamento de *snapshot* trazido pela ferramenta *Docker*. A palavra s*napshot* se traduz para fotografia, no contexto deste trabalho o termo significa tirar uma foto do código em um momento específico e transformá-lo em uma imagem que pode ser utilizada em qualquer momento, logo, a imagem representa o estado do código no momento de sua criação.

Vale lembrar que os testes de carga realizados tinham como objetivo verificar se o tempo de resposta dos serviços havia sido impactado negativamente e quantificar o tamanho desse impacto. Sendo assim, o objetivo principal do trabalho não era melhorar a performance do tempo de resposta e sim oferecer melhor gerencia e monitoria para os serviços.

Ainda, como apontamento final, é possível abrir o *Portainer* para a internet, ampliando a disponibilidade de gerenciamento remoto da planta. Indica-se também a implementação de um novo sistema de monitoria de recursos de *hardware* consumidos, pois o sistema atual permite a monitoria dos *containers* individualmente, porém seria mais performático agrupar esses dados em um só lugar.

Considerando todos os pontos levantados neste estudo, pode-se concluir que a conteinerização do sistema trouxe diversos benefícios para a *operação* da planta. É interessante para a evolução do trabalho a implementação de *Kubernetes* para escalar automaticamente os *containers* de serviços de acordo com a demanda. Apesar das melhorias aplicadas a monitoria do sistema, a planta não possui um sistema que acompanhe informações do consumo de *CPU*, *RAM* e temperatura de cada *Raspberry Pi*, tal serviço pode ser adicionado e mostrado no monitor *LCD* da planta.

REFERÊNCIAS

ANDERSON, Charles. "Docker [Software engineering]". **IEEE Software,** vol. 32, no. 3, pp. 102-c3, maio – jun. 2015, doi: 10.1109/MS.2015.62.

BERTOLUCCI, Cristiano Silveira. **Indústria 4.0: O que é, e como ela vai impactar o mundo** fev. 2016. Disponível em: https://www.citisystems.com.br/industria-4-0 Acesso em: 14 nov. 2022.

CHILANTI, Daniel Peretti. **GEN.IO: Desenvolvimento e aplicação de dispositivo IoT para coleta de dados de produção em ambiente industrial.** 2022. 87 f. TCC (Graduação) - Curso de Engenharia de Controle e Automação, Centro Tecnológico Departamento de Automação e Sistemas, Universidade Federal de Santa Catarina, Florianópolis, 2022. Disponível em: https://repositorio.ufsc.br/handle/123456789/240189. Acesso em: 14 nov. 2022.

DOCKER DOCS. **Docker overview**. 2021. Disponível em: https://docs.docker.com/get-started/overview/. Acesso em: 14 nov. 2022.

DOMINGOS, Leonardo Luis; FARINA, Renata Mirella. MICROSSERVIÇOS: um estudo de caso apontando suas potencialidades. **Revista Interface Tecnológica**, Taquaritinga - Sp, v. 17, n. 2, p. 18-30, 18 dez. 2020. Disponível em: https://revista.fatectq.edu.br/index.php/interfacetecnologica/article/view/851. Acesso em: 14 nov. 2022.

DOMINGUES, Felipe de Oliveira *et al*. **Historiador de Processos como um microsserviços: uma abordagem para a Indústria 4.0**. 2021. 89 f. TCC (Graduação) - Curso de Engenharia de Controle e Automação, Instituto de Ciência e Tecnologia de Sorocaba, Universidade Estadual Paulista “Júlio de Mesquita Filho”, Sorocaba, 2021

MOLECULERJS. **Documentation licensed under CC BY 4.0**. 2022. Disponível em: https://moleculer.services/docs/0.12/index.html. Acesso em: 14 nov. 2022.

OLIVEIRA, Rodrigo Felipe Albuquerque P. de *et al*. Uma Arquitetura de Microserviços de Internet das Coisas para Casas Inteligentes: an internet of things microservices architecture for smart homes. **Revista de Engenharia e Pesquisa Aplicada**, Recife, v. 2, n. 2, p. 15-23, 27 jul. 2017. ISSN: 2525-4251. Disponível em: http://revistas.poli.br/index.php/repa/article/view/547. Acesso em: 14 nov. 2022.

OKWUIBE, Jude *et al*. SDN Enhanced Resource Orchestration of Containerized Edge Applications for Industrial IoT. **IEEE Access**, Finland, v. 8, n. 1, p. 229117-229131, 26 nov. 2020.

PORTAINER DOCUMENTATION. **Portainer architecture**. 2021. Disponível em: https://docs.portainer.io/v/ce-2.9/start/architecture. Acesso em: 14 nov. 2022.

Grafana Labs. **The best developer experience for load testing**. 2022. Disponível em: https://k6.io/. Acesso em: 14 nov. 2022.

PONTAROLLI, Ricardo P. *et al*. Microservice Orchestration for Process Control in Industry 4.0. **IEEE Xplore**, Sorocaba, p. 245-249, 2020.

SACOMANO, José Benedito. **Indústria 4.0: conceitos e fundamentos**. São Paulo: Editora Bluncher, 2018.

SANTOS, B. P. Alberto, Lima, T. D. F. M., Charrua-Santos, F. M. B. (2018). INDUSTRY 4.0: CHALLENGES AND OPPORTUNITIES**. Revista Produção e Desenvolvimento**, 4(1), 111-124. https://doi.org/10.32358/rpd.2018.v4.316

SOLLFRANK, Michael *et al*. Evaluating Docker for Lightweight Virtualization of Distributed and Time-Sensitive Applications in Industrial Automation. **IEEE Transactions On Industrial Informatics,** Munich, v. 17, n. 5, p. 3566-3576, maio 2021.

APÊNDICE

Apêndice I – *Docker Compose* para implementação do *Portainer Server*

```
version: '3.5'
services:
  portainer:
    container_name: portainer
    image: portainer/portainer-ce:2.16.1
    ports:
      - "8000:8000"
      - "9000:9000"
    restart: always
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ./data:/data
```

Apêndice II – Configuração para container do *Portainer Edge Agent.*

```
version: "3.3"
services:
  portainer_edge_agent:
    image: portainer/agent:2.15.1
    container_name: portainer_edge_agent
    restart: always
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - /var/lib/docker/volumes:/var/lib/docker/volumes
      - /:/host
      - portainer_agent_data:/data
    environment:
      - CAP_HOST_MANAGEMENT=1
      - EDGE=1
      - EDGE_ID={environment_edge_id)
      - EDGE_KEY={environment_edge_key}
      - EDGE_INSECURE_POLL=1
volumes:
  portainer_agent_data:
```

Apêndice III – *Docker Compose* para implantação de múltiplos serviços simultaneamente.

```
version: "3.5"
services:
  control:
    image: gasiepgodoy/moleculer:control
    container_name: control-mesh
    env_file:
      - control-mesh.env
    restart: always

  control_1:
    image: gasiepgodoy/moleculer:control
    container_name: control-mesh-1
    env_file:
      - control-mesh-1.env
    restart: always

  control_2:
    image: gasiepgodoy/moleculer:control
    container_name: control-mesh-2
    env_file:
      - control-mesh-2.env
    restart: always

  control_3:
    image: gasiepgodoy/moleculer:control
    container_name: control-mesh-3
    env_file:
      - control-mesh-3.env
    restart: always

  control_4:
    image: gasiepgodoy/moleculer:control
    container_name: control-mesh-4
    env_file:
      - control-mesh-4.env
    restart: always
```

Apêndice IV – Arquivo *.env* contendo as variáveis de ambiente do serviço control.

```
NODEID=control-mesh-rasp-103
SERVICES=control
TRANSPORTER=nats://192.168.1.201:4222
CONTAINER_PORT_MAP=5902:5902
NAME=control
LOGGER=true
SERIALIZER=JSON
REQUESTTIMEOUT=100
MAXCALLLEVEL=100
TRACKING_ENABLED=true
TRACKING_SHUTDOWNTIMEOUT=5000
REGISTRY_STRATEGY=RoundRobin
REGISTRY_PREFERLOCAL=true
BULKHEAD_ENABLED=false
BULKHEAD_CONCURRENCY=10
BULKHEAD_MAXQUEUESIZE=100
VALIDATION=TRUE
VALIDATOR=NULL
METRICS=TRUE
METRICSRATE=1
```