UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROJETO DE GRADUAÇÃO

LUIZ CARLOS COSMI FILHO

UMA PROPOSTA DE EXPANSÃO DE UM ESPAÇO INTELIGENTE PROGRAMÁVEL ATRAVÉS DE DISPOSITIVOS ROBÓTICOS E COMPUTAÇÃO EM BORDA

VITÓRIA-ES

DEZEMBRO/2023

Luiz Carlos Cosmi Filho

UMA PROPOSTA DE EXPANSÃO DE UM ESPAÇO INTELIGENTE PROGRAMÁVEL ATRAVÉS DE DISPOSITIVOS ROBÓTICOS E COMPUTAÇÃO EM BORDA

Parte manuscrita do Projeto de Graduação do aluno Luiz Carlos Cosmi Filho, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Vitória-ES

Dezembro/2023

Luiz Carlos Cosmi Filho

UMA PROPOSTA DE EXPANSÃO DE UM ESPAÇO INTELIGENTE PROGRAMÁVEL ATRAVÉS DE DISPOSITIVOS ROBÓTICOS E COMPUTAÇÃO EM BORDA

Parte manuscrita do Projeto de Graduação do aluno Luiz Carlos Cosmi Filho, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Aprovado em 13 de dezembro de 2023.

COMISSÃO EXAMINADORA:

Profa. Dra. Raquel Frizera Vassallo
Universidade Federal do Espírito Santo
Orientadora

Prof. Dr. Alexandre Pereira do Carmo
Instituto Federal do Espírito Santo
Coorientador

Prof. Dr. Rodolfo da Silva Villaça
Universidade Federal do Espírito Santo
Examinador

Profa. Dr. Mariana Rampinelli Fernandes
Instituto Federal do Espírito Santo
Examinadora

Vitória-ES

Dezembro/2023

Ao meus pais, por todo apoio e carinho.

AGRADECIMENTOS

Gostaria de agradecer a todas as pessoas especiais na minha vida. A todos vocês, minha eterna gratidão. Em especial, à minha família, por todo apoio, carinho e dedicação. Sem vocês, nada disso seria possível. Obrigado por todas as marmitinhas especiais preparadas com muito amor que fizeram meus dias mais felizes, pelas conversas diárias por telefone para matar a saudade, pela companhia e compreensão, por tudo.

Agradeço imensamente à minha orientadora, Raquel Frizera Vassallo, por todos os ensinamentos, tanto acadêmicos quanto da vida. Não tenho dúvidas da sua importância para a minha formação, sou muito grato por todas as oportunidades que me deu ao longo desses anos.

Agradeço também ao meu coorientador, Alexandre Pereira do Carmo, por todo o tempo que dedicou a mim. A sua paciência, conhecimento e incentivo foram fundamentais.

Agradeço aos meus amigos de laboratório: Cabrunco, Flavinho, Fifi, Ricardinho, Wagner, Jorge, Bento, Bruna, Lucas, Witalo, Vinicius, Matheus Dutra, Matheus Ribeiro, Sobrinho, André, Zuelzer, Henrique, Artur, Aiury, Miquelly e muitos outros que por lá passaram nos últimos anos. Obrigado pelas conversas, momentos de descontração e apoio nesse trabalho. Em especial, gostaria de agradecer aos empresários Cabruno, Flavinho e Ricardinho pelas oportunidades de aprendizado durante o estágio.

À banca examinadora pela aceitação do convite e pelo tempo investido para leitura e avaliação desse trabalho.

Agradeço à Universidade Federal do Espirito Santo pela minha formação.

Por fim, a todos que, de forma direta ou indireta, contribuíram para a minha formação pessoal e acadêmica.

RESUMO

Um espaço inteligente pode ser definido como um ambiente físico equipado com sensores e dispositivos interconectados, que são capazes de coletar dados, analisá-los e responder de forma inteligente às necessidades dos usuários. Com esse objetivo, foi desenvolvido no Laboratório de Visão Computacional e Robótica na Universidade Federal do Espírito Santo (UFES) um Espaço Inteligente Programável (PIS), onde é possível que os usuários utilizem tais funcionalidades com o uso de câmeras fixas no ambiente, dispositivos robóticos e serviços de computação em nuvem. No entanto, um dos problemas recorrentes encontrados é a abrangência do espaço inteligente, que é limitada pelo campo de visão das câmeras fixas. Assim, neste trabalho propõe-se a integração de dispositivos robóticos à plataforma de computação e disponibilização das informações dos sensores presentes no PIS. Tais dispositivos robóticos podem ser utilizados para computação em borda, disponibilizando e processando informações, sempre que necessário para atender aos requisitos impostos pelas aplicações. Para que isso fosse possível, foi desenvolvido um controlador responsável por avaliar onde as aplicações devem ser executadas, ou em nuvem ou em borda, baseado no tempo de resposta especificado e visando a utilização racional dos recursos de infraestrutura. O controlador foi testado em dois estudos de caso, o primeiro envolvendo a detecção e localização de marcadores visuais de realidade aumentada e o segundo envolvendo a detecção de objetos por meio de redes neurais. Os resultados obtidos mostram que, em ambos os casos, controlador proposto agiu sobre o conjunto de serviços especificados procurando sempre minimizar, quando possível, o tempo de resposta definido.

Palavras-chave: Computação em borda; Computação em Nuvem; Espaços Inteligentes.

ABSTRACT

An intelligent space can be defined as a physical environment equipped with interconnected sensors and devices capable of collecting data, analyzing it, and intelligently responding to the users' needs. With this goal, a Programmable Intelligent Space (PIS) was developed at the Computer Vision and Robotics Laboratory at Universidade Federal do Espírito Santo (UFES), where users can access the functionalities present in the environment using fixed cameras, robotic devices, and cloud computing services. However, one of the recurring problems encountered is the limited coverage of the intelligent space, usually constrained by the field of view of the fixed cameras in the environment. Therefore, this work proposes the integration of robotic devices into the computing platform and sharing sensor information present in the PIS. Such robotic devices can then be used for edge computing and data processing, whenever necessary, to meet application requirements. To achieve this, a controller was developed to assess where the applications should run, either in the cloud or at the edge, based on the specified response time and focusing on the rational use of the infrastructure resources. The controller was tested in two case studies, the first involving the detection and localization of augmented reality visual markers and the second involving object detection through neural networks. The obtained results demonstrate that, in both cases, the proposed controller acted on the set of specified services, always aiming to minimize, whenever possible, the defined response time.

Keywords: Edge computing; Cloud computing; Intelligent Spaces.

LISTA DE FIGURAS

Figura 1 – Representação do campo de visão das câmeras fixas e de um VANT. . . 14
Figura 2 – Modelo conceitual da arquitetura do sistema proposto. 14
Figura 3 – Camadas de hardware e software em soluções de virtualização baseadas em máquinas virtuais e em contêineres. 18
Figura 4 – Representação visual de uma arquitetura monolítica e uma arquitetura de microsserviços. 21
Figura 5 – Representação visual dos componentes de um *cluster* Kubernetes composto por 3 nós. 22
Figura 6 – Modelo Conceitual da arquitetura do PIS. 25
Figura 7 – Representação da comunicação entre um produtor e dois consumidores, com um *exchange*, duas filas e três diferentes *bindings* 28
Figura 8 – Exemplo de *logs* de uma aplicação do PIS. 29
Figura 9 – Diagrama de exemplo de *trace* e sua respectiva árvore de *spans* no tempo. 30
Figura 10 – Esquemático de funcionamento do controlador proposto para o PIS. . . 33
Figura 11 – Exemplos de marcadores visuais ArUco. 34
Figura 12 – Esquemático de comunicação entre os serviços do experimento de detecção de marcadores visuais. 34
Figura 13 – Exemplos de detecção de objetos. 35
Figura 14 – Esquemático de comunicação entre os serviços do experimento de detecção de objetos. 35
Figura 15 – Divisão da camada de suporte à aplicação do PIS. 37
Figura 16 – Modificação da camada de suporte à aplicação para medição de tempo de comunicação. 37
Figura 17 – Esquemático dos parâmetros de consulta dos *traces* na API do Zipkin em uma linha temporal. 38
Figura 18 – Esquemático de implantação da ferramenta Zipkin neste trabalho. . . . 39
Figura 19 – Representação da comunicação de um serviço genérico no PIS em borda. 40
Figura 20 – Representação da comunicação de um serviço genérico orquestrado em nuvem para atender uma aplicação em borda. 41
Figura 21 – Exemplo de criação de uma *Shovel* para o compartilhamento de imagens de uma câmera através da interface web do RabbitMQ. 41
Figura 22 – Esquemático com o fluxograma de funcionamento do controlador. . . . 45
Figura 23 – Esquemático de comunicação entre os serviços do experimento de detecção de marcadores visuais. 48

Figura 24 – Exemplo de *trace* com processamento de imagens em borda. 48
Figura 25 – Exemplo de *trace* com processamento de imagens em nuvem. 49
Figura 26 – Média móvel do tempo de resposta em nuvem ao longo do experimento. 51
Figura 27 – Média móvel do tempo de resposta total ao longo do experimento. . . . 52
Figura 28 – Esquemático de comunicação entre os serviços do experimento de detecção de objetos. 53
Figura 29 – Exemplo de *trace* com processamento de imagens em borda. 53
Figura 30 – Exemplo de *trace* com processamento de imagens em nuvem. 53
Figura 31 – Média móvel do tempo de resposta em nuvem ao longo do experimento. 56
Figura 32 – Média móvel do tempo de resposta total ao longo do experimento. . . . 57

LISTA DE TABELAS

Tabela 1 – Tempo de resposta total na tarefa de detecção e localização de marcadores com o processamento de imagens em borda e em nuvem. 49
Tabela 2 – Tempo de resposta total na tarefa de detecção de objetos com o processamento de imagens em borda e em nuvem. 54

LISTA DE ABREVIATURAS E SIGLAS

AMQP	*Advanced Message Queuing Protocol*
API	*Application Programmable Interface*
CPU	*Central Processing Unit*
DNS	*Domain Name Services*
HTTP	*Hypertext Transfer Protocol*
IaaS	*Infrastructure as a Service*
IoT	*Internet of Things*
MSA	*Microservice Architecture*
NFS	*Network File System*
NTP	*Network Time Protocol*
OSI	*Open System Interconnection*
PaaS	*Platform as a Service*
PIS	*Programmable Intelligent Space*
PPGEE	Programa de Pós-Graduação em Engenharia Elétrica
SaaS	*Software as a Service*
SO	Sistema Operacional
UFES	Universidade Federal do Espírito Santo
VANT	Veículo Aéreo Não Tripulado
WAN	*Wide Area Network*

SUMÁRIO

1	**INTRODUÇÃO**	**12**
1.1	**Objetivos**	**15**
1.1.1	Objetivo Geral	15
1.1.2	Objetivos Específicos	15
1.2	**Estrutura do Texto**	**15**
2	**REFERENCIAL TEÓRICO**	**17**
2.1	**Virtualização**	**17**
2.2	**Computação em nuvem e borda**	**18**
2.3	**Arquitetura de software**	**19**
2.4	**Kubernetes**	**21**
2.5	**Espaço Inteligente Programável**	**24**
2.5.1	Camadas	25
2.5.2	Comunicação	26
2.5.3	Observabilidade	28
2.6	**Trabalhos relacionados**	**30**
3	**PROPOSTA**	**33**
3.1	**Medição do tempo de resposta total**	**36**
3.2	**Conexão entre *brokers***	**39**
3.3	**Desenvolvimento do controlador para o PIS**	**42**
4	**EXPERIMENTOS E RESULTADOS**	**46**
4.1	**Recursos Computacionais**	**46**
4.2	**Estudo de caso I - Detecção e localização de marcadores visuais**	**47**
4.3	**Estudo de caso II - Detecção de pessoas**	**52**
5	**CONCLUSÃO E TRABALHOS FUTUROS**	**58**
5.1	**Conclusão**	**58**
5.2	**Trabalhos futuros**	**59**
	REFERÊNCIAS	**61**
	APÊNDICES	**65**
.1	**Definição de recurso personalizado**	**66**

1 INTRODUÇÃO

O conceito de "computação ubíqua", também conhecido por "computação pervasiva", foi inicialmente proposto por Weiser (1991). Para ele, a tecnologia do futuro estaria imersa na vida das pessoas, imperceptível. Assim, os usuários se concentrariam na tarefa e não na ferramenta que estão utilizando. Em outras palavras, a ideia central da computação ubíqua é a de que a tecnologia deve se adaptar ao usuário e ao contexto em que ele se encontra, em vez de exigir que o usuário se adapte à tecnologia. Dessa forma, os dispositivos e sistemas computacionais estariam incorporados ao ambiente, tornando-se uma parte natural e invisível do dia a dia das pessoas. Quando proposta, sua ideia foi revolucionária e a frente do seu tempo. Isso, porque a tecnologia de hardware necessária para viabilizar o projeto não existia (SATYANARAYANAN, 2001).

Como destacado por Satyanarayanan (2001), o avanço tecnológico na área de sistemas distribuídos e computação móvel tornou possível a implementação do conceito de computação ubíqua. São exemplos de dispositivos de computação ubíqua *smartphones*, *tablets*, sensores, dispositivos inteligentes em casas, carros conectados, entre outros. Também pode ser considerado como uma aplicação de computação ubíqua um "espaço inteligente", um espaço físico equipado com uma rede de sensores, atuadores e serviços computacionais que atuam com o objetivo de atender às necessidades dos usuários presentes no ambiente (LEE; HASHIMOTO, 2002; CARMO, 2021). Enquanto a computação ubíqua se refere à integração quase invisível da tecnologia ao ambiente físico e ao cotidiano das pessoas, os espaços inteligentes são ambientes físicos que utilizam as tecnologias de computação ubíqua para se tornarem mais eficientes e personalizados, atendendo às necessidades das pessoas que os utilizam.

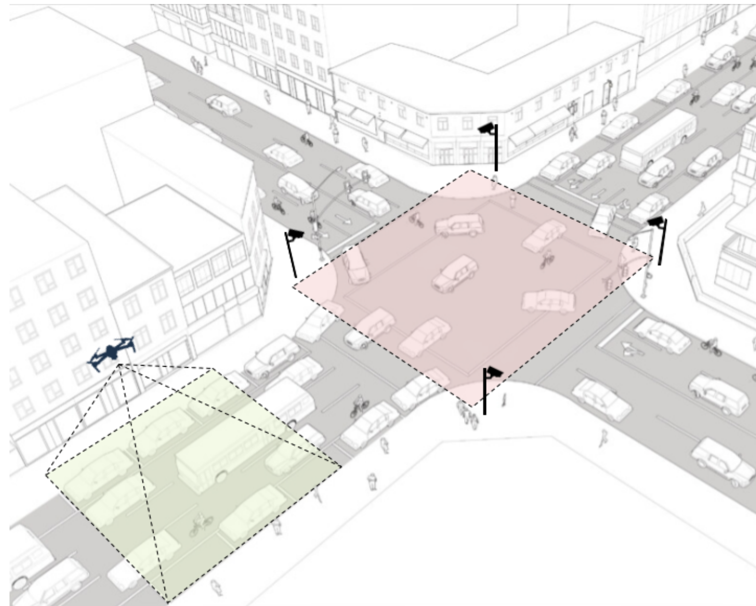
Um espaço inteligente pode ser um ambiente fechado, como uma sala de reuniões ou laboratório, ou um área aberta bem definida, como um pátio ou quadra, ou até mesmo o espaço abrangido por uma cidade, dando origem ao conceito de "cidades inteligentes" (COTTA, 2020; CARMO, 2021). Nesse contexto, sensores e dispositivos inteligentes podem ser implantados em toda a cidade para coletar dados em tempo real. Então, essas informações são utilizadas para otimizar o funcionamento da cidade, melhorar a mobilidade urbana, a segurança e a qualidade de vida dos cidadãos. Diversos sensores podem ser utilizados para adquirir informações, tais como câmeras, microfones e sensores de proximidade. Além disso, os serviços computacionais que controlam este ambiente podem atuar diretamente sobre ele, através do uso de telas e alto falantes, ou indiretamente, por meio de atuadores ou dispositivos robóticos como robôs terrestres ou mesmo Veículos Aéreos Não-Tripulados (VANTs).

No Lab VISIO - Laboratório de Visão Computacional e Robótica do Programa de Pós-Graduação em Engenharia Elétrica (PPGEE) da Universidade Federal do Espírito Santo (UFES) foi desenvolvido um Espaço Inteligente Programável (PIS, do inglês *Programmable Intelligent Space*) baseado em visão computacional, tendo câmeras como principais sensores. O PIS, além de ter as características de um espaço inteligente, também incorpora uma arquitetura baseada em microsserviços centrada na observabilidade multinível e a programabilidade granular da infraestrutura (CARMO, 2021).

Assim, diversas aplicações foram desenvolvidas e integradas ao PIS desenvolvido no Lab VISIO. Por exemplo, os autores Almonfrey et al. (2018) propuseram um detector de pessoas, além de experimentos envolvendo interação humano-robô como prova de conceito da aplicação. Já os autores Queiroz et al. (2018) propuseram um método para estimativa tridimensional de coordenadas de juntas de esqueletos. Continuando o trabalho desenvolvido por Queiroz et al. (2018), os autores Custodio et al. (2020) propuseram um Ambiente Interativo para o espaço inteligente através de esqueletos tridimensionais, bem como uma aplicação para construção de um mapa de ocupação do ambiente. No trabalho proposto pelos autores Izabel et al. (2022), um sistema que integra o processo de localização, controle e navegação de um cão-guia robô (Lysa) para ambientes internos, baseado em visão computacional, também foi desenvolvido para o PIS. É importante notar que, em todas as aplicações citadas, a abrangência do PIS é limitada pelo campo de visão das câmeras, enquanto os dispositivos robóticos realizam parte do sensoriamento e principalmente ações no ambiente através dos comandos obtidos pelos serviços em nuvem, estes responsáveis por processar as informações de sensoriamento.

Ao integrar sensores (câmeras, *lasers*, entre outros) em robôs terrestres ou VANTs e disponibilizar essas informações, a abrangência do PIS poderia ser expandida e não estar limitada pelo campo de visão das câmeras instaladas no ambiente. Considere a Figura 1, onde o espaço físico monitorado pelas câmeras fixas no cruzamento das avenidas poderia ser considerado como um espaço inteligente. Aplicações de videomonitoramento com foco em segurança pública poderiam ser executadas em tal espaço, por exemplo, visando reconhecer veículos ou pessoas, estimar velocidade, etc. No entanto, a abrangência desse espaço é limitada pelo campo de visão das câmeras de videomonitoramento (área vermelha destacada na imagem). Ao levantar voo com um VANT equipado com uma câmera, o espaço físico monitorado é expandido (área verde destacada na imagem). Isso permitiria, por exemplo, rastrear um veículo e/ou uma pessoa por determinado período de tempo. Tal abordagem traz mobilidade e uma melhor abrangência ao sistema como um todo, abrindo uma ampla gama de possibilidades de aplicações no contexto de cidades inteligentes ou espaços inteligentes, com cobertura limitada de câmeras, quando se utilizam dispositivos móveis integrados ao sistema e munidos de sensores.

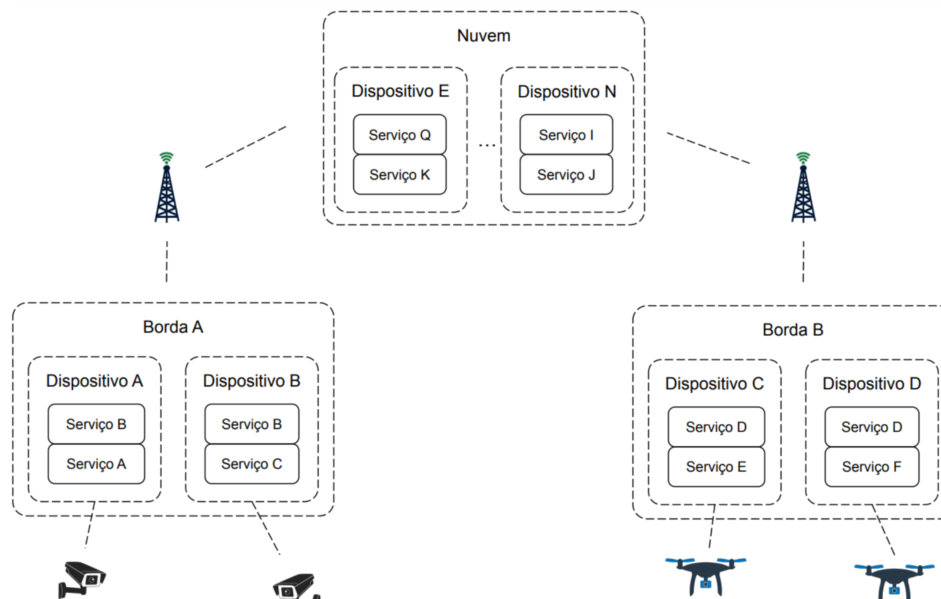
Figura 1 – Representação do campo de visão das câmeras fixas e de um VANT.



Fonte: Produção do próprio autor.

Além disso, os próprios dispositivos robóticos poderiam ser utilizados para processamento, quando necessário para atender aos requisitos impostos pelas aplicações. Na Figura 2, pode-se observar um modelo conceitual de como a arquitetura do sistema é proposta neste trabalho para dar suporte à expansão do espaço inteligente.

Figura 2 – Modelo conceitual da arquitetura do sistema proposto.



Fonte: Produção do próprio autor.

Uma nuvem com ambiente de computação altamente escalável é necessária, pois nela serão conectadas as diversas bordas. Os dispositivos em bordas, tais como VANTs, robôs ou outros dispositivos móveis, podem conter diversos sensores e estarem conectados à nuvem disponibilizando essas informações para o PIS ou realizando o processamento.

Nesse sentido, este trabalho propõe integrar dispositivos robóticos, tais como robôs terrestres e VANTs, na plataforma de orquestração de contêineres, possibilitando a computação em borda das aplicações e a expansão da abrangência através da disponibilização das informações para o PIS. Para isso, pretende-se medir o tempo de resposta através da observabilidade multinível disponibilizada pelo PIS e desenvolver um controlador responsável por avaliar onde as aplicações devem ser executadas, ou em nuvem ou em borda, a fim de atender os requisitos das aplicações com utilização racional dos recursos de infraestrutura. Por fim, por serem duas aplicações muito utilizadas no PIS e que demandam custo computacional variável, de acordo com os requisitos das aplicações, serão utilizadas a detecção de marcadores visuais de realidade aumentada e a detecção de pessoas para ilustrar e validar a abordagem proposta.

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo geral deste projeto pode ser definido como a expansão de um Espaço Inteligente Programável através de dispositivos robóticos, para computação em borda e extensão da sua abrangência.

1.1.2 Objetivos Específicos

- Estender a plataforma de orquestração de contêineres para os dispositivos robóticos;
- Modificar aplicações já desenvolvidas para expor o tempo de resposta total;
- Desenvolver um controlador para o PIS responsável por manipular as aplicações de forma a atender os requisitos do sistema monitorado;
- Testar e validar o controlador através de experimentos realizados no laboratório.

1.2 Estrutura do Texto

O presente trabalho está estruturado da seguinte maneira:

- **Introdução**: o capítulo inicial visa apresentar o tema no qual este projeto está inserido e a sua importância, além de definir os objetivos gerais e específicos.
- **Referencial teórico**: neste capítulo, o referencial teórico e sustentação científica do trabalho são abordados. São apresentadas as tecnologias de virtualização, arquiteturas de softwares, Kubernetes e o PIS. Além disso, características relevantes do PIS para este trabalho são detalhadas.
- **Proposta**: neste capítulo, é apresentada a abordagem de resolução do problema de pesquisa proposto neste trabalho. Assim, são abordados os detalhes de implementação do sistema, medição do tempo de resposta total, comunicação entre nuvem e borda, e desenvolvimento do controlador.
- **Resultados**: neste capítulo, são apresentados os resultados dos estudos de caso realizados para validação deste trabalho. Os tempos de resposta são apresentados em gráficos para visualização da ação do controlador proposto.
- **Conclusão e trabalhos futuros**: no capítulo final deste trabalho são apresentadas as discussões sobre os resultados encontrados, assim como a aplicabilidade e propostas de melhorias para estudos futuros.

2 REFERENCIAL TEÓRICO

Neste capítulo, apresenta-se o referencial teórico deste trabalho. Na Seção 2.1, aborda-se o conceito de virtualização, discutindo as diversas formas de virtualização e suas distinções. Em seguida, na Seção 2.2, o foco é na computação em nuvem e em borda, explorando os modelos e suas vantagens. Na Seção 2.3, é apresentado um comparativo entre arquiteturas monolíticas e de microsserviços, examinando como a utilização de uma arquitetura baseada em microsserviços permite a construção de aplicativos escaláveis e flexíveis. Então, na Seção 2.4, o destaque é o Kubernetes, uma plataforma de orquestração de contêineres para o gerenciamento de microsserviços em ambientes de nuvem. Por fim, na Seção 2.5, é apresentado o PIS e as suas camadas, bem como detalhes de comunicação das aplicações e observabilidade.

2.1 Virtualização

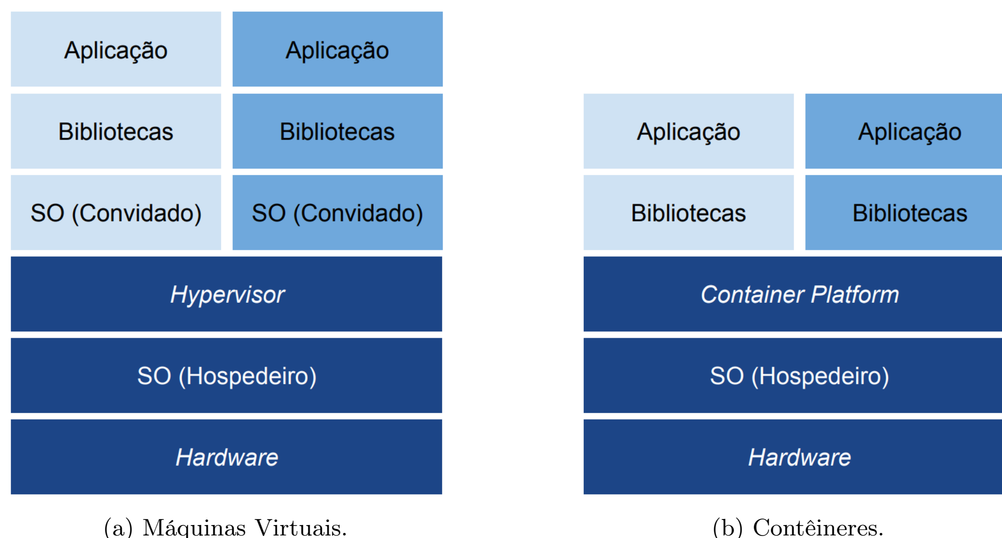
Segundo os autores Jain e Choudhary (2016), o termo "virtualização" pode ser definido como uma abstração dos recursos computacionais. Dessa forma, permite-se que um único recurso físico esteja disponível como múltiplos recursos virtuais ou que múltiplos recursos físicos estejam disponíveis como um único recurso virtual.

Uma das formas de virtualização é através de softwares conhecidos como *Hypervisors*. Os *Hypervisors* podem ser executados diretamente sobre o hardware, possuindo seus próprios *drivers* e, portanto, não necessitando de todo um sistema operacional (SO). Ou, podem ser executados na camada de software, como qualquer outro programa executado pelo computador (EDER, 2016; JAIN; CHOUDHARY, 2016).

Uma outra forma de virtualização, é através do uso de contêineres. Em um sistema operacional, existem algumas características e funcionalidades necessárias para isolar um processo do outro. Softwares de virtualização baseados em contêineres utilizam essas funcionalidades para criar um ambiente isolado para processos, nesse caso, chamado contêiner. Dessa forma, os contêineres compartilham o *kernel* do sistema operacional em que estão sendo executados. Por isso, as aplicações devem ser compatíveis com o *kernel* do sistema operacional e a arquitetura da Unidade de Processamento Central (CPU, do inglês *Central Processing Unit*) (EDER, 2016).

Na Figura 3, observa-se como os softwares de virtualização baseados em contêineres e *Hypervisors* funcionam. O *Hypervisor* separa os recursos da máquina física (hospedeira) para que eles possam ser particionados e dedicados às máquinas virtuais (convidadas). Já em um contêiner, tem-se apenas a aplicação em um ambiente isolado com todos os elementos necessários para executá-lá (POTDAR et al., 2020).

Figura 3 – Camadas de hardware e software em soluções de virtualização baseadas em máquinas virtuais e em contêineres.



(a) Máquinas Virtuais. (b) Contêineres.

Fonte: Produção do próprio autor.

Os autores Potdar et al. (2020) diferenciam as duas tecnologias de virtualização apontadas anteriormente no texto e apresentam algumas vantagens e desvantagens. A criação de máquinas virtuais através de *Hypervisors* demandam mais recursos (CPU, memória e armazenamento), levam mais tempo para inicializar e a customização das imagens que iniciam as máquinas virtuais é mais difícil. No entanto, apresentam a vantagem de isolar os processos a nível de hardware e fornecer um sistema operacional dentro de cada máquina virtual. Por outro lado, a criação de contêineres demanda menos recursos, leva menos tempo para iniciar e a criação de imagens que inicia os contêineres é relativamente fácil. Porém, o sistema operacional da máquina é compartilhado pelos contêineres. A utilização de cada tecnologia depende do cenário em que se deseja utilizar e podem até mesmo ser utilizadas em conjunto.

2.2 Computação em nuvem e borda

De acordo com Mohammed, Zeebaree et al. (2021), o conceito de "computação em nuvem" pode ser definido como um ambiente na Internet que permite utilizar softwares, dados e recursos de qualquer lugar. Portanto, as tecnologias de virtualização desempenham um

papel crítico na criação e na sustentação da infraestrutura de nuvem, tornando-a uma ferramenta essencial. Isso, porque a virtualização oferece flexibilidade, escalabilidade e capacidade de gerenciamento, permitindo que os usuários dimensionem e adaptem suas infraestruturas de nuvem de acordo com as necessidades.

Dessa forma, os serviços de computação em nuvem podem ser agrupados em três modelos essenciais: (i) *Infrastructure as a Service* (IaaS): os usuários são responsáveis por gerenciar recursos computacionais, como máquinas virtuais, endereços IP, armazenamento e rede; (ii) *Platform as a Service* (PaaS): os usuários são responsáveis pelo gerenciamento das plataformas de computação, permitindo a criação e distribuição de aplicativos pela rede; (iii) *Software as a Service* (SaaS): concede aos usuários o acesso a aplicativos mantidos por provedores, disponibilizados pela Internet e eliminando a necessidade de instalação de software em seus próprios computadores pessoais.

Por outro lado, o termo "computação em borda" é um conceito que se refere à estratégia de trazer os serviços e aplicações de computação em nuvem para perto, geograficamente, do usuário final possibilitando um menor tempo de resposta (KHAN et al., 2019). Normalmente, os usuários finais executam as aplicações em dispositivos com capacidade limitada de processamento, o que implica no uso da nuvem para execução dos principais serviços e aplicações. Como resultado, problemas de alta latência e mobilidade são recorrentes.

Segundo os autores Khan et al. (2019), os problemas de computação em nuvem podem ser resolvidos por diferentes modelos de computação em borda. Um deles é a computação em névoa, que estende os recursos computacionais disponíveis na nuvem para a borda da rede através do uso de vários dispositivos interconectados (COUTINHO; CARNEIRO; GREVE, 2016). No entanto, estender uma plataforma altamente virtualizada para borda da rede é um desafio. Isso, porque envolve o uso de diferentes hardwares, protocolos de rede, grande número de dispositivos, entre outras dificuldades.

2.3 Arquitetura de software

Como destacado pelos autores Villamizar et al. (2015), uma das principais razões pelas quais as empresas migram suas aplicações para soluções de computação em nuvem, sejam elas do tipo IaaS ou PaaS, é para o ganho de eficiência em suas operações, buscando a capacidade de dimensionar essas aplicações sob demanda e suportar períodos de pico de uso. Nesse sentido, diversas empresas têm usado arquiteturas de software baseadas em microsserviços no projeto de suas aplicações em nuvem em contrapartida a arquiteturas monolíticas.

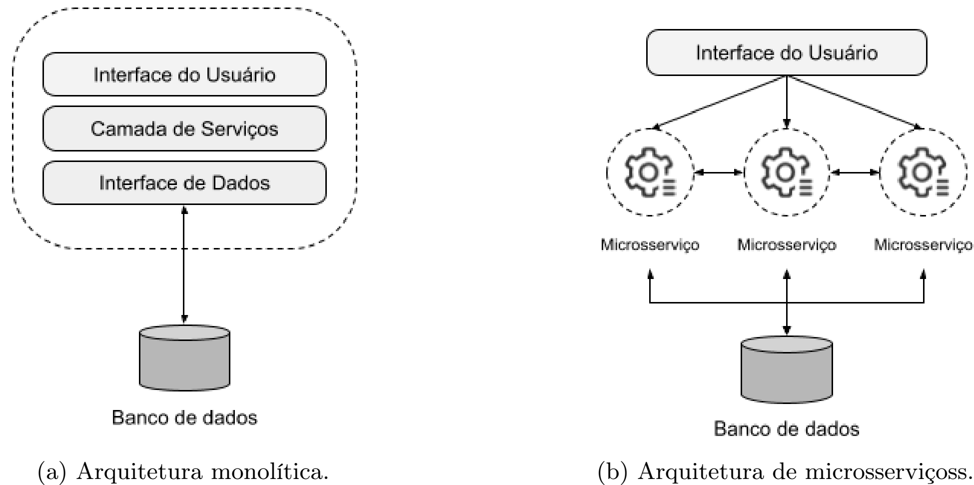
Planejar e desenvolver a arquitetura de um sistema de software significa representar o sistema com um alto grau de abstração, realizando o mapeamento dos componentes que compõem o software (SHARMA; KUMAR; AGARWAL, 2015). Nesse sentido, o padrão de organização da arquitetura de software desempenha um papel importante ao definir a maneira de organizar os componentes do sistema, possibilitando a construção de um sistema completo que atenda às necessidades do cliente. Além disso, a arquitetura de software também é responsável pelo desempenho, a robustez e a capacidade de distribuição da carga de trabalho e manutenção de um sistema (BOSCH, 1999; SHARMA; KUMAR; AGARWAL, 2015).

Segundo Dragoni et al. (2017), uma arquitetura de software monolítica pode ser caracterizada como um software cujos módulos não podem ser executados de forma independente. Embora mais simples de desenvolver inicialmente, os autores Villamizar et al. (2015) destacam que, com o aumento da complexidade da aplicação, torna-se difícil realizar manutenção, localizar problemas e adicionar novas funcionalidades. Além disso, uma pequena mudança em um dos seus módulos resulta na necessidade de reinicialização de toda a aplicação. Também vale ressaltar que uma aplicação monolítica representa um ponto de falha central. Assim, caso a aplicação falhe, todo o conjunto de serviços ficam indisponíveis até a reinicialização da aplicação.

Nesse contexto, arquiteturas de microsserviços têm sido propostas como solução para os problemas encontrados em arquiteturas monolíticas. Os autores Dragoni et al. (2017) definem um microsserviço como um processo coeso e independente que interage com outros microsserviços por meio de mensagens. Nessa definição, utiliza-se o termo "coeso" para indicar que cada microsserviço implementa apenas funcionalidades fortemente relacionadas ao que ele realiza. Esse tipo de arquitetura de desenvolvimento tornou-se amplamente utilizada pela indústria ao possibilitar alta disponibilidade, flexibilidade, escalabilidade e velocidade.

Na Figura 4, pode-se observar as diferenças entre as arquiteturas monolíticas e microsserviços. Como cada microsserviço é um processo independente, pode ser escalado conforme a necessidade, diferentemente de uma arquitetura monolítica onde seria necessário escalar todos os módulos. Além disso, como cada microsserviço é desenvolvido e testado independentemente, uma pequena atualização em um microsserviço não implica na indisponibilidade de toda aplicação. Desta forma, uma aplicação baseada em microsserviços não representa um ponto de falha central. Portanto, caso um microsserviço falhe, apenas uma parte do sistema fica indisponível até a reinicialização do microsserviço que falhou, o que traz uma melhor experiência ao usuário final da aplicação.

Figura 4 – Representação visual de uma arquitetura monolítica e uma arquitetura de microsserviços.



(a) Arquitetura monolítica. (b) Arquitetura de microsserviçoss.

Fonte: Produção do próprio autor.

2.4 Kubernetes

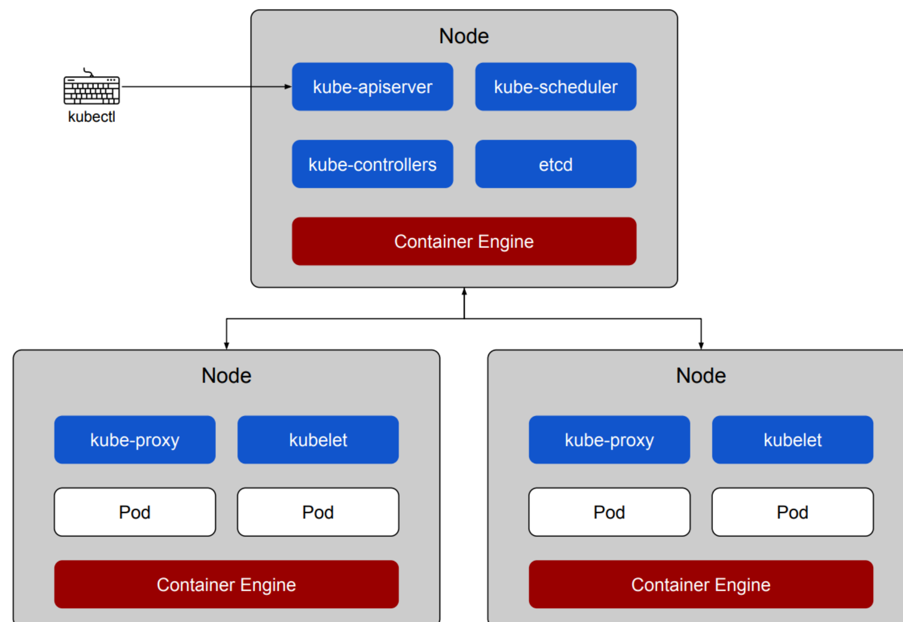
Ao utilizar computação em nuvem aliada a uma arquitetura de software baseada em microsserviços, pode-se dimensionar a infraestrutura necessária para os microsserviços conforme a demanda de maneira eficiente. Nesse cenário, o Kubernetes, como um orquestrador de contêineres, desempenha um papel fundamental, automatizando a implantação, a escalabilidade e a gerência de microsserviços em ambientes de nuvem, tornando a operação desses sistemas altamente eficaz e resiliente.

Kubernetes é uma plataforma de código aberto, portável e extensiva para o gerenciamento de cargas de trabalho e serviços distribuídos em contêineres (GOOGLE, 2023). Um *cluster* Kubernetes pode ser caracterizado como um conjunto de servidores de processamento, chamados de nós de computação, responsáveis por executar os contêineres. Tais servidores de processamento podem ser máquinas virtuais, computadores de alto desempenho ou até mesmo dispositivos com capacidade limitada de processamento.

Obrigatoriamente, em um *cluster*, deve existir um plano de controle a ser executado em um dos nós ou replicado em diversos. O plano de controle é responsável por gerenciar o *cluster*, decidir onde alocar as aplicações, responder a eventos, etc. Os seus principais componentes são: (i) *kube-apiserver*, uma API (do inglês, *Application Programmable Interface*) que expõe as funcionalidades do *cluster*; (ii) *etcd*, um banco de dados distribuído para armazenar o estado do *cluster*; (iii) *kube-scheduler*, aplicação responsável por decidir em que nó cada unidade básica de escalonamento será alocada dentro do *cluster*, levando em conta os recursos e restrições impostas; (iv) *kube-controller-manager*, componente com os diversos controladores do *cluster*.

Além disso, em todos os nós, é necessário que algumas aplicações sejam executadas para que os nós sejam controlados pelo plano de controle. Esses componentes são: (i) *kubelet*, aplicação que garante a execução dos contêineres; (ii) *kube-proxy*, *proxy* de rede executado em cada nó que realiza o encaminhamento de tráfego de rede corretamente dentro do *cluster*. Por fim, a ferramenta de linha de comando do Kubernetes, *kubectl*, permite que o administrador execute comandos nos *cluster*. Na Figura 5, observa-se como esses componentes se integram e formam um *cluster* para o gerenciamento de cargas de trabalho e serviços.

Figura 5 – Representação visual dos componentes de um *cluster* Kubernetes composto por 3 nós.



Fonte: Adaptado de Google (2023).

Através da API disponibilizada pelo plano de controle, ou da ferramenta de linha de comando, é possível criar diversos recursos. Estes recursos podem ser entendidos como objetos ou entidades persistentes. Uma vez criado, os componentes do *cluster* trabalham para garantir a sua existência. Dentre eles, tem-se:

- *Pod*: é o recurso mais simples, a unidade básica de um *cluster* Kubernetes, e pode conter um ou mais contêineres;
- *ReplicaSet*: recurso que garante que um número de réplicas de um determinado *Pod* esteja em execução;
- *DaemonSet*: recurso que garante um *Pod* executando em cada nó do *cluster*;
- *Job*: recurso que garante que um ou mais *Pods* sejam executados com sucesso;

- *Deployment*: recurso que fornece funcionalidades para o gerenciamento do ciclo de vida de um conjunto de *Pods*, tais como atualizações contínuas de *Pods*, a capacidade de reverter e escalar facilmente a quantidade de *Pods*.

Além dos diversos recursos disponíveis por padrão, a API do Kubernetes também permite a criação de recursos personalizados (do inglês, *Custom Resources*). No entanto, pode ser necessário o desenvolvimento de um controlador associado para implementar a lógica de controle e garantir que o estado desejado do recurso definido seja alcançado.

Para restringir execução de *Pods* em nós de processamento específicos, pode-se utilizar o conceito de seleção de nós (do inglês, *nodeSelector*). Então, deve-se atribuir rótulos a cada um dos nós de processamento e, durante a criação dos recursos, adicionar o campo de seleção de nós. Por exemplo, suponha que existam dois nós de processamento, N1 e N2. O nó de processamento N1 encontra-se em uma região chamada *cloud-1*, já o nó de processamento N2 encontra-se em uma outra região chamada *edge-1*. Dessa forma, um rótulo chamado *topology.kubernetes.io/region* pode ser criado e essa informação deverá estar presente em cada nó.

Assim, caso deseja-se criar um recurso do tipo *Deployment* para execução de uma carga de trabalho na região *cloud-1*, pode-se atribuir o rótulo *topology.kubernetes.io/region="cloud-1"* ao nó N1 e usar a seleção de nós durante a criação do recurso. No código abaixo, pode-se observar tal situação onde deseja-se executar um *Deployment* em um conjunto de máquinas que pertecem a região *cloud-1*.

Código 2.1 – Exemplo de um *Deployment* sendo executado em uma região específica utilizando rótulos para atribuição dos *Pods* à nós de processamento específicos.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: nginx
5    labels:
6      app: nginx
7  spec:
8    replicas: 5
9    selector:
10     matchLabels:
11       app: nginx
12   template:
13     metadata:
14       labels:
15         app: nginx
16     spec:
17       nodeSelector:
18         topology.kubernetes.io/region: "cloud-1"
19       containers:
```

```
20        - name: nginx
21          image: nginx:1.24.0
22          ports:
23          - containerPort: 80
24          resources:
25            requests:
26              cpu: 2
```

2.5 Espaço Inteligente Programável

Segundo os autores Lee e Hashimoto (2002), um espaço inteligente pode ser definido como um ambiente equipado com uma rede de sensores que possibilitam a percepção e a compreensão do que ocorre em seu interior. Dessa forma, o espaço inteligente pode tomar decisões por meio de dados coletados. Com essas características, as pessoas ou sistemas no espaço inteligente podem usar funções adicionais proporcionadas pelo espaço.

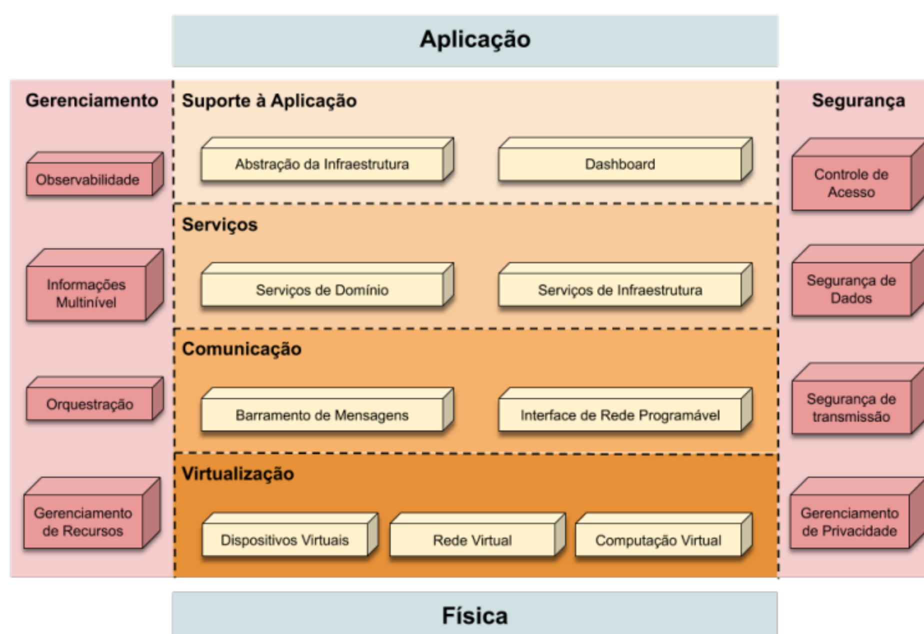
Observe que, os espaços inteligentes podem variar bastante na sua concepção e implementação. Lee e Hashimoto (2002) apontam que, inicialmente, tais espaços eram apenas sistemas que forneciam funcionalidades às pessoas ao compreender eventos que aconteciam no meio. Para atender fisicamente os usuários, foi necessário incorporar agentes físicos, tais como robôs. Um exemplo de tarefa a ser executada por um robô é o deslocamento de um objeto, com o intuito de ajudar alguma pessoa com deficiência a superar algum obstáculo. No entanto, os autores destacam que tais espaços também podem funcionar como uma interface de alto nível tanto para pessoas quanto para robôs.

O autor Carmo (2021) aponta que esses elementos presentes em um espaço inteligente devem ser gerenciados por uma infraestrutura de hardware e software responsável por coletar e analisar os dados, gerar decisões e atuar quando necessário. Para isso, uma arquitetura de software baseada em microsserviços com orquestração multinível centrada na observabilidade e programabilidade granular da infraestrutura é proposta, o PIS. Ao pensar em um espaço inteligente como uma plataforma com arquitetura altamente escalável, o espaço compreendido por um PIS pode ser o ambiente de uma sala ou até mesmo o espaço compreendido por uma cidade. Dessa forma, pode-se atender aos requisitos específicos e altamente rigorosos das aplicações.

2.5.1 Camadas

A arquitetura de software e hardware proposta para o PIS, conforme descrita por Carmo (2021), pode ser visualizada por meio de uma representação estruturada em camadas. Na Figura 6, é possível observar a organização do PIS em quatro camadas horizontais: Virtualização, Comunicação, Serviços e Suporte à Aplicação; e duas camadas transversais: Gerenciamento e Segurança.

Figura 6 – Modelo Conceitual da arquitetura do PIS.



Fonte: Carmo (2021)

A camada de virtualização é responsável por abstrair os recursos de hardware e torná-los disponíveis para o uso. Nesse sentido, utilizam-se softwares como *Hypervisors* e plataformas de criação de contêineres nessa camada. Superior à camada de virtualização, a camada de comunicação é responsável por fornecer conectividade e interação entre as aplicações. As aplicações podem utilizar diferentes protocolos dentro do PIS, tais como HTTP (do inglês, *Hypertext Transfer Protocol*) ou AMQP (do inglês, *Advanced Message Queuing Protocol*). Na Seção 2.5.2, os detalhes de cada protocolo de comunicação são explorados.

A camada de serviços engloba as aplicações desenvolvidas na forma de microsserviços, enquanto a camada de suporte à aplicação corresponde às bibliotecas e ferramentas auxiliares para o desenvolvimento delas. Já a camada de segurança tem como responsabilidade realizar controle de acesso, segurança dos dados, segurança da comunicação e gerenciamento de privacidade. Por isso, essa última está presente ao longo de todas as quatro camadas horizontais.

Por fim, tem-se a camada de gerenciamento, responsável por gerenciar os elementos de todas as quatro camadas horizontais. Para isso, deve prover as seguintes funções: (i) Orquestração e Informações multinível; (ii) Observabilidade; e (iii) Gerenciamento dos recursos. Nesse sentido, o Kubernetes é utilizado como uma solução de software para gerenciar os recursos e realizar a orquestração dos microsserviços que compõe o PIS e, também, o conjunto de microsserviços de várias aplicações que adicionam funções de observabilidade e informações multiníveis para o PIS.

2.5.2 Comunicação

Diversos protocolos de comunicação são empregados no desenvolvimento das aplicações que compõem o PIS. Destacam-se, como principais, os protocolos HTTP e AMQP. Em várias situações, ambos são utilizados de forma simultânea. A seguir, serão explorados os detalhes de cada protocolo de comunicação e como se diferenciam.

O protocolo HTTP é um protocolo de comunicação na camada de aplicação do modelo OSI (do inglês, *Open System Interconnection*) amplamente utilizado e difundido. É baseado no padrão cliente/servidor, onde um servidor é um programa responsável por aceitar conexões de requisições de serviço e enviar respostas. Por outro lado, um cliente é uma aplicação que estabelece conexões com o servidor com o objetivo de enviar requisições (BERNERS-LEE; FIELDING; FRYSTYK, 1996). No contexto do protocolo HTTP, há oito tipos de requisições (também chamadas de métodos HTTP) que o cliente pode iniciar. Assim, o servidor responde adequadamente a cada uma dessas requisições com um código de três dígitos, informando ao cliente o *status* da solicitação. A seguir, tem-se os principais métodos e as ações que realizam:

- GET: usado para ler informações do servidor;
- PUT: usado para modificar completamente um dado no servidor ou gerar um novo dado nele;
- POST: usado para transferir dados ao servidor, na forma de arquivos, formulários ou outros;
- DELETE: usado para remover um dado especificado no servidor.

O protocolo AMQP também é um protocolo de comunicação na camada de aplicação do modelo OSI. No entanto, é um protocolo de comunicação assíncrono orientado a

mensagens baseado no padrão *publish/subscribe*. Como apontado pelos autores Eugster et al. (2003), no padrão *publish/subscribe*, os produtores publicam mensagens em um barramento, enquanto os consumidores se subscrevem para receber mensagens com as informações que desejam.

É relevante mencionar que, de acordo com os autores Eugster et al. (2003), o desenvolvimento de aplicações que empregam comunicações ponto a ponto e síncronas tendem a resultar em aplicações rígidas e estáticas, o que pode prejudicar a escalabilidade do sistema. Portanto, torna-se essencial a escolha de um protocolo de comunicação assíncrono com o objetivo de garantir a escalabilidade dentro do PIS. Dessa forma, o principal protocolo de comunicação utilizado no PIS será o protocolo AMQP. O protocolo HTTP será utilizado de forma auxiliar para prover características de observabilidade às aplicações, tal como abordado na Seção 2.5.3.

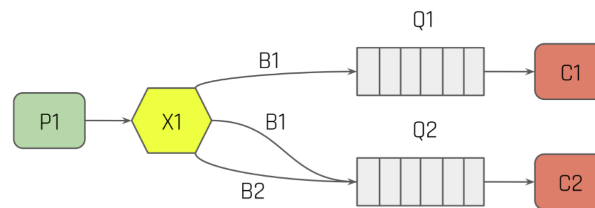
A utilização do protocolo AMQP está associada à execução de uma aplicação conhecida como *broker*, responsável por realizar o roteamento das mensagens dos produtores para os consumidores corretamente. Se necessário, também pode armazenar as mensagens para entregar aos consumidores quando estes estiverem disponíveis. O autor Queiroz (2016) detalha o roteamento de mensagens dentro do PIS através do RabbitMQ (2023) como *broker*. Para implementar o padrão de comunicação *publish/subscribe*, *broker* cria filas para armazenar as mensagens. No entanto, apenas filas não são suficientes para prover todas as funcionalidades. É necessário um elemento de roteamento das mensagens responsável por encaminhar corretamente cada mensagem à fila correta. Este elemento, é denominado *exchange*. Ademais, a relação entre uma fila e uma *exchange* é denominada *binding* e pode ser entendida como uma regra de roteamento. Quando uma mensagem é enviada, esta deve possuir uma *exchange* de destino, bem como uma identificação de roteamento denominada *routing key*.

Considere a Figura 7, que o autor Queiroz (2016) aponta como exemplo. O produtor, P1, envia as mensagens à *exchange* X1. Os consumidores C1 e C2 possuem as suas filas associadas, respectivamente, Q1 e Q2 . Observe que a fila Q1 possui um *binding*, B1, com a *exchange* X1. Por outro lado, a fila Q2 possui dois *bindings*, B1 e B2, com a *exchange* X1. Assim, se uma mensagem for enviada ao *broker* com a *exchange* X1 de destino e *routing key* B1, ambas as filas Q1 e Q2 irão receber a mensagem. No entanto, se uma mensagem for enviada ao *broker* com a *exchange* X1 de destino e *routing key* B2, apenas a fila Q2 irá receber a mensagem.

Existem diversas categorias de *exchanges*. A mencionada no exemplo anterior é do tipo *direct*. Onde, a mensagem é encaminhada para a fila de acordo com o *binding*. No entanto,

este tipo de *exchange* possui limitações. Para realizar um roteamento de mensagens baseado em múltiplos critérios, uma *exchange* do tipo *topic* deve ser utilizada. Nesse tipo de *exchange*, mensagens enviadas ao *broker* não podem ter uma *routing key* arbitrária - devem apresentar uma lista de palavras separadas por pontos.

Figura 7 – Representação da comunicação entre um produtor e dois consumidores, com um *exchange*, duas filas e três diferentes *bindings*



Fonte: Queiroz (2016).

Assim, na camada de comunicação do PIS são utilizadas *exchanges* do tipo *topic* para permitir uma maior flexibilidade de roteamento das mensagens ao construir uma lista de palavras com tipos e subtipos, tais como *CameraGateway.0.Frame*. A primeira palavra representa o serviço, a segunda o identificador da câmera e a terceira o tipo de informação da câmera, no caso uma imagem. Para receber imagens de todas as câmeras que enviam mensagem à *exchange*, basta se subscrever com a seguinte regra: *CameraGateway.*.Frame*.

Por fim, é importante mencionar que existem diversas outras aplicações que usam inúmeros outros protocolos de comunicação com o intuito de prover funcionalidades e suporte às aplicações do PIS. Alguns exemplos são: (i) NFS (do inglês, *Network File System*) sistema de arquivos distribuídos desenvolvido para o compartilhamento distribuído de arquivos na rede; (ii) NTP (do inglês, *Network Time Protocol*) protocolo para sincronização de relógios de computadores; (iii) DNS (do inglês, *Domain Name Services*) sistema hierárquico e distribuído de gestão de nomes.

2.5.3 Observabilidade

A observabilidade é uma das das características mais importante do PIS para este trabalho, pois através das informações disponibilizadas pelas ferramentas de observabilidade será realizado o controle das aplicações no PIS. O termo "observabilidade" pode ser definido como a capacidade de visualizar e compreender o comportamento de um sistema através da coleta e processamento de dados (PICORETI et al., 2018). São utilizados três tipos de dados principais para prover informações de observabilidade de um sistema: *logs*, métricas e *traces*.

Os *logs* são linhas de texto estruturadas e não estruturadas que um sistema gera quando

uma parte específica do código é executada. Em outras palavras, um *log* é um registro de um evento em uma aplicação. Por isso, são comumente utilizados para descobrir problemas e analisar comportamentos imprevisíveis do sistema. Na Figura 8, pode-se observar os *logs* de uma aplicação do PIS chamada de *is-broker-events*, responsável por publicar em forma de mensagens os eventos que acontecem no *broker* RabbitMQ.

Figura 8 – Exemplo de *logs* de uma aplicação do PIS.

```
[I][MainThread][2023-10-09 19:32:43,067][BrokerEvents] BrokerEventsOptions:
broker_uri: "amqp://rabbitmq.default"
management_uri: "http://guest:guest@rabbitmq.default"

[D][MainThread][2023-10-09 19:32:43,067][BrokerEvents] Connecting to RabbitMQ broker at: 'amqp://rabbitmq.default'
[D][MainThread][2023-10-09 19:32:43,129][BrokerEvents] Got list of consumers at: 'http://guest:guest@rabbitmq.default'
[D][MainThread][2023-10-09 19:33:02,549][BrokerEvents] topic='binding.created' metadata={'timestamp_in_ms': 1696879982546
, 'source_name': 'is', 'source_kind': 'exchange', 'destination_name': 'is-frame-transformation-976d9c585-rcp4z/268CECD986
84DB77', 'destination_kind': 'queue', 'routing_key': 'is-frame-transformation-976d9c585-rcp4z/268CECD98684DB77', 'argumen
ts': [], 'vhost': '/', 'user_who_performed_action': 'guest'}
[I][MainThread][2023-10-09 19:33:02,549][BrokerEvents] event='created' topic='is-frame-transformation-976d9c585-rcp4z/268
CECD98684DB77' queue='is-frame-transformation-976d9c585-rcp4z/268CECD98684DB77'
[D][MainThread][2023-10-09 19:33:02,559][BrokerEvents] topic='binding.created' metadata={'timestamp_in_ms': 1696879982558
, 'source_name': 'is', 'source_kind': 'exchange', 'destination_name': 'is-frame-transformation-976d9c585-rcp4z/268CECD986
84DB77', 'destination_kind': 'queue', 'routing_key': '#.FrameTransformations', 'arguments': [], 'vhost': '/', 'user_who_p
erformed_action': 'guest'}
[I][MainThread][2023-10-09 19:33:02,560][BrokerEvents] event='created' topic='#.FrameTransformations' queue='is-frame-tra
nsformation-976d9c585-rcp4z/268CECD98684DB77'
```

Fonte: Produção do próprio autor.

Geralmente, tais *logs* são coletados por uma camada de agregação para serem pré-processados e, em seguida, persistidos para posterior processamento e análise. Existem diversas soluções de software construídas para implementar essa camada de agregação, dentre elas pode-se citar o Grafana Loki (2023): um sistema de agregação de *logs* totalmente escalável e altamente disponível que disponibiliza essas informações em uma interface web para visualização e análise.

Por outro lado, as métricas podem ser definidas como números coletados periodicamente formando uma série temporal. Dessa forma, ajudam a mostrar tendências sobre o comportamento e performance do sistema ao longo do tempo. Em diversos trabalhos, têm-se utilizado métricas como quantidade de uso de CPU ou quantidade de requisições HTTP em determinada API para realizar o escalonamento de aplicações, com o objetivo de prover alta disponibilidade e escalabilidade a fim de suportar picos de uso (NGUYEN et al., 2020; PICORETI et al., 2018). Também tem-se utilizado métricas para geração de alertas, seja a indisponibilidade inesperada de algum recurso ou algo que não deveria estar acontecendo no sistema.

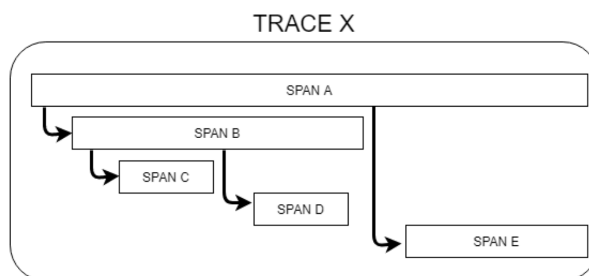
Da mesma forma que os *logs*, as métricas também são coletadas por uma camada de agregação periodicamente para processamento e persistência. Diversas soluções de software foram construídas para implementar essa camada de agregação, dentre elas pode-se citar o Prometheus (2023), um banco de dados de séries temporais e sistema de monitoramento que possibilita a integração com softwares de visualização de métricas e geração de alarmes.

Já os *traces* podem ser entendidos como uma forma de observabilidade do sistema que

permite determinar a causalidade entre eventos de diversas aplicações, bem como extrair medidas de desempenho. Para isso, cada serviço deve contribuir propagando um contexto, que deve ser enviado diretamente a uma ferramenta responsável por relacionar a cadeia de eventos e apresentar aos usuários. Pode-se citar como exemplo de ferramentas de coleta de *traces* o Zipkin (2023) e o Jaeger (2023).

Por ser um método baseado em anotações, nesse tipo de solução há necessidade de instrumentação do código. Ou seja, é necessário informar no código onde a medição começa e termina (SIGELMAN et al., 2010). Na Figura 9, tem-se um exemplo de *trace* e sua respectiva árvore de *spans* (unidade básica de trabalho, um registo temporal com início e fim associado a alguma função ou serviço) indicando a causalidade entre eles.

Figura 9 – Diagrama de exemplo de *trace* e sua respectiva árvore de *spans* no tempo.



Fonte: Cotta (2020).

Assim, através dos *traces*, os usuários podem acompanhar a jornada de uma solicitação ou evento através de um sistema de software devidamente instrumentado, identificando gargalos, erros ou comportamentos indesejados. Com a crescente complexidade dos ambientes de TI modernos, a capacidade de visualizar e analisar *traces* tornou-se essencial para garantir a confiabilidade, o desempenho e a segurança dos sistemas.

2.6 Trabalhos relacionados

No trabalho proposto pelos autores Santoro et al. (2017), uma plataforma de orquestração de aplicações para computação em borda é proposta. De acordo com os requisitos impostos durante o momento de instalação das aplicações, estas eram direcionadas de forma a atender os requisitos geográficos ou de capacidade da rede. Assim, era possível orquestrar aplicações tanto em nuvem quanto em borda (o mais próximo do cliente). Além disso, uma aplicação de detecção facial é apresentada no contexto de cidades inteligentes e alguns cenários são explorados. Ao orquestrar a aplicação de detecção facial perto do cliente, há uma diminuição da largura de banda utilizada na conexão com a nuvem e um menor tempo de resposta. No entanto, os autores não apontam o uso de ferramentas de

observabilidade, que poderiam ser utilizadas para atualizar a instalação dessas aplicações na plataforma dinamicamente. Também não é abordado no artigo aspectos relevantes de como a comunicação ocorre, por exemplo, se é um modelo de comunicação baseado em mensagens.

Já no trabalho proposto pelos autores Mohamed et al. (2017), uma plataforma de computação em borda baseada no uso de VANTs para aplicações IoT é proposta. Um VANT equipado com os recursos de computação pode viajar para um local específico quando necessário e permanecer nesse local para oferecer suporte aos aplicativos IoT locais. Cada VANT possui um *broker* com um conjunto de aplicações disponíveis. Assim, caso o serviço desejado pela aplicação IoT local não esteja disponível a bordo, a aplicação pode solicitar à um *broker* em nuvem através do próprio dispositivo. Nesse contexto, os autores mostram que, ao colocar aplicações no VANT, responde-se de forma mais rápida às requisições ao eliminar a necessidade de consulta em nuvem. No entanto, os autores também não exploram ferramentas de observabilidade e como essas poderiam ser utilizadas dentro da plataforma proposta para uma orquestração inteligente dessas aplicações (em nuvem ou a bordo do VANT).

Um outro trabalho que propõe o uso de VANTs no contexto de computação em borda, foi proposto pelos autores Sanchez-Aguero et al. (2021). O objetivo é gerenciar as operações de VANTs remotamente e prover conectividade aos dispositivos locais. Para isso, esses dispositivos robóticos são capazes de prover uma rede 4G e assim disponibilizar conectividade aos dispositivos IoT locais. Também foi proposta a integração nos VANTs de uma plataforma de virtualização de contêineres para a implantação dos serviços necessários. Assim, algumas aplicações são orquestradas a bordo e outras em nuvem. No entanto, os autores não exploram o uso de ferramentas de observabilidade e possibilidade de realizar algum processamento a bordo para execução de tarefas (por exemplo, seguimento de padrões). A plataforma no dispositivo robótico funciona como uma ponte para nuvem, que é responsável por controlá-lo.

O conceito de observabilidade multinível em Espaços Inteligentes foi introduzido pelo trabalho proposto pelos autores Picoreti et al. (2018), considerando métricas tanto da infraestrutura quanto das aplicações para melhorar o escalonamento e orquestração automática de aplicações em ambientes de nuvem. Os autores obtiveram melhores resultados com o uso de métricas como tempo de resposta, medido pela aplicação, do que com a utilização de métricas como taxa de utilização de CPU. Além do uso de métricas para o escalonamento, foi apontado como trabalhos futuros por Picoreti et al. (2018) o uso de *traces* para a identificação de gargalos no sistema e escalonamento das aplicações como forma de mitigação de problemas relacionados à alta latência. Nessa mesma linha, os

autores Tzanettis et al. (2022) proposeram um *framework* de orquestração baseado nos 3 pilares da observabilidade: métricas, *traces* e *logs*. Ao levar em conta tanto as métricas quanto os *traces*, o *framework* de observabilidade é capaz de responder a problemas de latência e escalonar os serviços adequadamente.

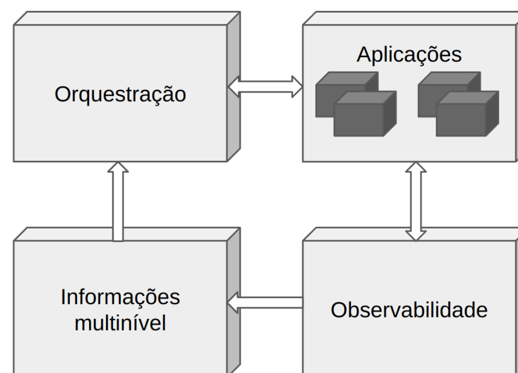
Nesse sentido, o presente trabalho visa estender a plataforma de orquestração de contêineres para os dispositivos robóticos permitindo a computação em borda de aplicações, tal como realizado pelos trabalhos propostos pelos autores Santoro et al. (2017), Mohamed et al. (2017) e Sanchez-Aguero et al. (2021), assim como a expansão da abrangência do espaço inteligente através dos sensores embarcados. No entanto, sua principal diferença está no fato de utilizar as informações multinível, disponibilizadas pelos serviços que fazem parte do PIS, para gerenciar onde devem ser executadas, em nuvem ou em borda, a fim de atender os requisitos das aplicações com utilização racional dos recursos de infraestrutura. Diferentemente dos trabalhos propostos por Picoreti et al. (2018) e Tzanettis et al. (2022), neste trabalho não se pretende utilizar as informações multinível para escalonamento, mas para a correta alocação em nuvem ou em borda.

3 PROPOSTA

Este trabalho tem como objetivo realizar a expansão do PIS através de dispositivos em borda, tanto para computação em borda de aplicações quanto para expansão da abrangência do PIS por meio da disponibilização de informações dos sensores em borda na plataforma. Dessa forma, ao orquestrar as aplicações no *cluster*, pretende-se analisar as informações multiníveis disponibilizadas pelas ferramentas de observabilidade presentes no PIS e desenvolver um controlador responsável por avaliar onde as informações devem ser processadas, em nuvem ou em borda, a fim de atender aos requisitos das aplicações com utilização racional dos recursos de infraestrutura.

Na Figura 10, pode-se observar como este trabalho foi realizado. Tanto as aplicações quanto as ferramentas de observabilidade se comunicam a fim de mensurar as informações desejadas. Por sua vez, as ferramentas de observabilidade disponibilizam essas informações para o PIS, tanto informações características das aplicações, como o tempo de resposta, quanto informações da infraestrutura, como o uso de CPU e memória. A partir dessas informações, desenvolveu-se um controlador integrado à plataforma de orquestração de contêineres para configurar corretamente as aplicações.

Figura 10 – Esquemático de funcionamento do controlador proposto para o PIS.



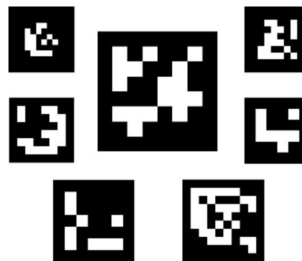
Fonte: Produção do próprio autor.

Primeiramente, foi necessário integrar os dispositivos robóticos dentro do PIS. Ou seja, fez-se necessário adicionar os dispositivos dentro da plataforma de orquestração de contêineres como nós de computação em borda. Assim, obtendo um *cluster* formado por nós de computação em nuvem (*Datacenter* presente no laboratório) e nós de computação em borda. Para isso, utilizou-se uma Raspberry Pi modelo 4B com uma câmera conectada, como exemplo de dispositivo com sensor para computação em borda conectada via rede WiFi.

Para garantir a execução dos conjuntos de *Pods* criados por cada recurso sejam executados em nós de processamento específicos, que pertencem à uma certa região, é necessário utilizar a seleção de nós por meio de rótulos. Em outras palavras, todos os nós de processamento possuem rótulos que indicam a região à qual pertencem, viabilizando a atribuição dos *Pods* a nós de processamento específicos.

Então, foram definidos dois estudos de caso neste trabalho para servirem de base para o desenvolvimento e validação do controlador responsável por definir se o processamento das informações será em borda ou em nuvem. O primeiro estudo de caso consiste na detecção de marcadores visuais, um serviço bastante utilizado em aplicações de realidade aumentada, visão computacional e robótica. Isso, porque permite estimar a postura da câmera em relação ao marcador. Através de um marcador posicionado em um robô, pode-se estimar a posição e orientação do robô em relação à câmera. Na Figura 11, pode-se observar exemplos de marcadores visuais chamados de ArUco, quadrados sintéticos compostos por uma borda preta larga e uma matriz binária interna que determina sua identificação.

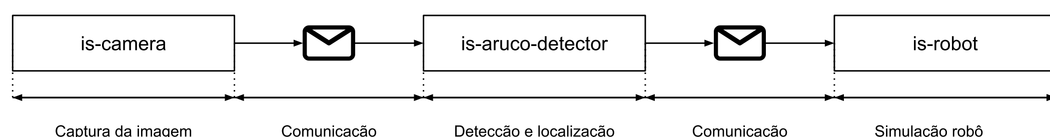
Figura 11 – Exemplos de marcadores visuais ArUco.



Fonte: Produção do próprio autor.

Para realizar este estudo de caso, foi necessário desenvolver um serviço responsável por capturar as imagens da câmera conectada à Raspberry (*is-camera*). Então, modificou-se o serviço de detecção e localização de marcadores visuais (*is-aruco-detector*) previamente desenvolvido para expôr o tempo de comunicação. Por fim, desenvolveu-se um serviço para adquirir as informações de localização dos marcadores visuais na Raspberry, simulando um programa de controle de dispositivo robótico. Na Figura 12, pode-se observar o fluxo de informações e como os serviços se comunicam (a comunicação com o *broker* RabbitMQ foi omitida para simplificação da figura). Observe que, nesse cenário apresentado, apenas o serviço de detecção de marcadores visuais poderia ser executado em nuvem ou em borda.

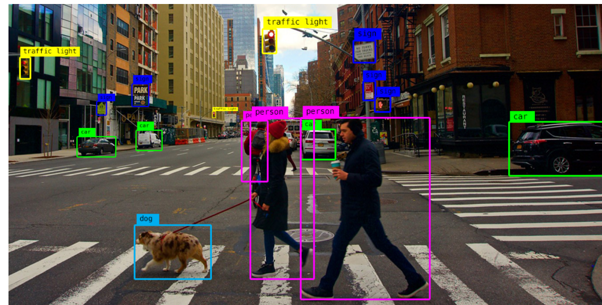
Figura 12 – Esquemático de comunicação entre os serviços do experimento de detecção de marcadores visuais.



Fonte: Produção do próprio autor.

Já o segundo estudo de caso consiste na execução de uma rede neural para detecção de objetos. Entre as diversas redes neurais disponíveis, neste trabalho utilizou-se a YoloV8 (JOCHER; CHAURASIA; QIU, 2023) para detecção de objetos e, através de um parâmetro de configuração, filtrar os objetos de interesse, tais como pessoas, carros, entre muitos outros. Na Figura 13, pode-se observar exemplos de detecção de objetos em uma imagem.

Figura 13 – Exemplos de detecção de objetos.

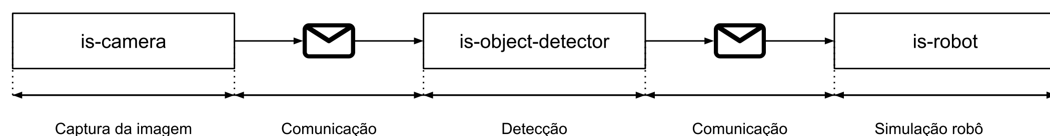


Fonte: Produção do próprio autor.

Assim, além dos serviços já desenvolvidos no experimento anterior, também desenvolveu-se e instrumentou-se o seguinte serviço: *is-object-detector*, serviço responsável por realizar a detecção de objetos.

Na Figura 14, pode-se observar o fluxo de informações e como as aplicações se comunicam (a comunicação com o *broker* RabbitMQ foi omitida para simplificação da figura). Observe que, tal como no cenário apresentado com o detector de marcadores visuais, apenas o serviço de detecção de objetos poderia ser executado em nuvem ou em borda.

Figura 14 – Esquemático de comunicação entre os serviços do experimento de detecção de objetos.



Fonte: Produção do próprio autor.

Todas os serviços citados anteriormente foram desenvolvidos utilizando a linguagem de programação *Python*, com as bibliotecas de suporte ao PIS, e conteinerizados com a ferramenta *Docker*. É importante notar que, como deseja-se obter o tempo de resposta total, é necessário que os serviços sejam instrumentados corretamente. Ou seja, é necessário inserir blocos de códigos responsáveis por enviar as informações desejadas, tais como tempo de comunicação e processamento. Por isso, houve a necessidade de modificação de serviços previamente desenvolvidos. Na Seção 3.1, é explorada com mais detalhes a definição do tempo de resposta, bem como a metodologia de medição.

Uma vez com os serviços desenvolvidos para os experimentos, bem como as aplicações de observabilidade corretamente instaladas, deve-se considerar a forma de comunicação entre

borda e nuvem. Neste trabalho é instalado um *broker* RabbitMQ em cada borda e em nuvem, por isso deve existir um mecanismo de comunicação entre os *brokers*. Assim, na Seção 3.2 é definida uma metodologia para este mecanismo.

Por fim, na Seção 3.3 é abordado o desenvolvimento do controlador responsável por analisar e realizar a configuração do processamento das informações em nuvem ou em borda, bem como configurar corretamente a comunicação entre os *brokers*.

3.1 Medição do tempo de resposta total

Tempo de resposta é definido por Cotta (2020) como a combinação do tempo de processamento dentro dos diversos serviços, somado ao tempo de comunicação entre eles. O tempo de processamento representa o período gasto pelo programa para executar um conjunto específico de tarefas dentro de um serviço, enquanto o tempo de comunicação refere-se ao intervalo de tempo decorrido desde o momento em que a função de envio de dados é acionada até a conclusão da recepção completa dos dados pelo destinatário.

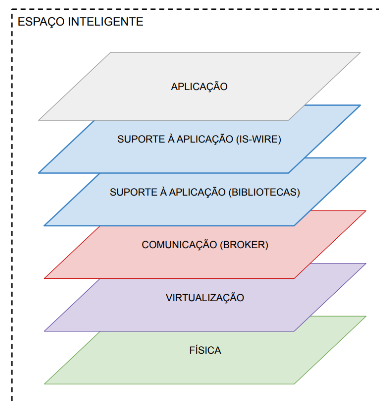
Para realizar a medição do tempo de resposta total, propõe-se a utilização da metodologia definida pelo autor Cotta (2020). Assim, deve-se sincronizar os relógios de todos os dispositivos do sistema para garantir que estejam em conformidade em relação a um mesmo horário. Além disso, utiliza-se uma implementação específica da camada de suporte à aplicação do PIS (*is-wire*) desenvolvida pelo autor, responsável por realizar a comunicação, padronização das mensagens e instrumentação dos serviços de forma a expôr o tempo de comunicação entre eles através de *spans* em *traces*. Na solução proposta pelo autor, realizou-se a divisão da camada de suporte à aplicação original em duas subcamadas, uma contendo todas as bibliotecas e outra contendo a biblioteca de abstração do restante das camadas inferiores. Na Figura 15, pode-se observar como dividiu-se essa camada.

No desenvolvimento de aplicações para o PIS, somente a biblioteca *is-wire* é utilizada pelos desenvolvedores. Assim, todas as bibliotecas da camada inferior, necessárias para o desempenho de qualquer atividade, estão abstraídas. Na Figura 16, pode-se observar em quais instantes de tempo a implementação definida pelo autor Cotta (2020) realiza a medição do tempo de comunicação. Isto é, no instante de tempo t_0 dá-se início a medição do tempo de comunicação com o envio de uma mensagem por um produtor e no instante de tempo t_1 termina-se quando a mensagem é consumida pelo consumidor.

Assim, utilizando tal metodologia de medição do autor Cotta (2020) os *spans* de comunicação são enviados através da camada de suporte à aplicação modificada (*is-wire*)

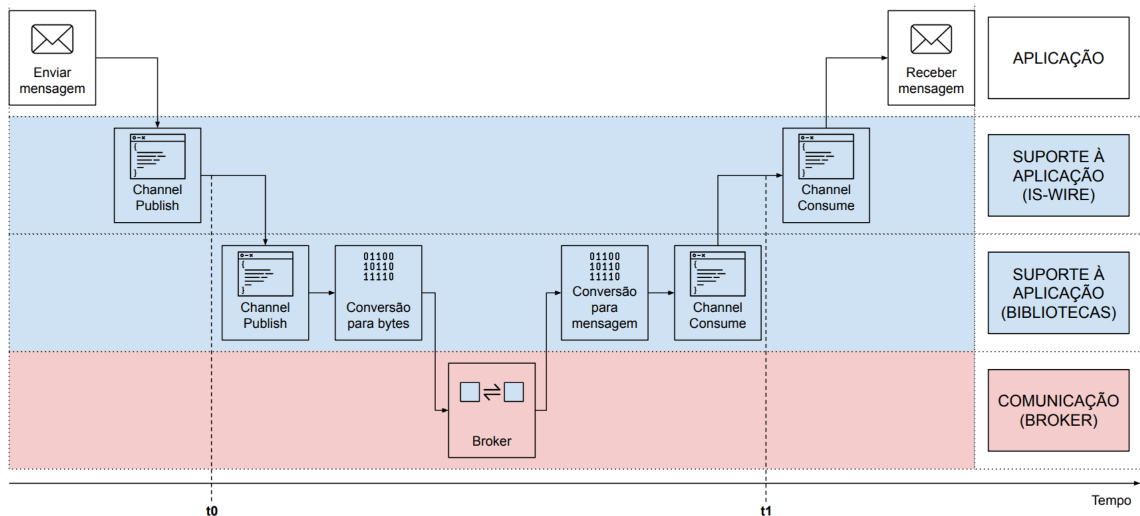
junto com os *spans* de processamento definidos na instrumentação dos serviços. Para realizar a agregação dos *spans* de diversos serviços, os relacionar e armazenar em um banco de dados, a ferramenta Zipkin foi utilizada neste trabalho. Além de uma interface web com a visualização dessas informações na forma de gráficos e diagramas, a ferramenta também disponibiliza uma API para consulta. Nesse sentido, serão explorados a seguir os detalhes de consulta nesta API e a forma de implantação da ferramenta Zipkin no *cluster* Kubernetes, os quais constituem uma das contribuições deste trabalho.

Figura 15 – Divisão da camada de suporte à aplicação do PIS.



Fonte: Adaptado de Cotta (2020).

Figura 16 – Modificação da camada de suporte à aplicação para medição de tempo de comunicação.



Fonte: Adaptado de Cotta (2020).

Esta API do Zipkin desempenha um papel fundamental neste trabalho, sendo empregada para consultar o tempo total de resposta em uma cadeia de serviços. Essa métrica é crucial para embasar decisões relacionadas ao processamento, seja na nuvem ou na borda. Ao realizar uma consulta na API do Zipkin, é essencial fornecer não apenas os nomes dos serviços de interesse, mas também três parâmetros adicionais, cada um desempenhando um papel específico:

- N, este parâmetro determina o número máximo de *traces* a serem retornados.

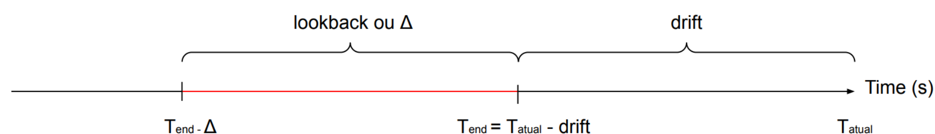
- T_{end}, este parâmetro especifica o instante de tempo final para a consulta. O retorno incluirá apenas *traces* nos quais todos os instantes de tempo dos *spans* estão neste instante final ou antes. Definir T_{end} é essencial para analisar o tempo de resposta em um contexto temporal específico.

- Δ, definindo o *lookback* ou intervalo de tempo, a consulta retorna apenas *traces* nos quais todos os instantes de tempo dos *spans* estão em $T_{end} - \Delta$ ou depois. Isso permite uma análise retrospectiva do tempo de resposta.

A combinação desses três parâmetros oferece uma abordagem abrangente para a obtenção de *traces* e, consequentemente, uma compreensão do tempo de resposta na cadeia de serviços em uma janela de tempo. Além disso, no contexto deste trabalho, o controlador necessita adquirir os *traces* mais recentes para embasar decisões em tempo real. Entretanto, surge uma complexidade devido à necessidade da ferramenta de *tracing* de um intervalo de tempo para efetuar a correlação dos *spans*.

A busca pelos *traces* mais recentes implica que o instante de tempo final, T_{end}, deve ser sincronizado com o instante de tempo atual. No entanto, a ferramenta de *tracing* requer um intervalo de tempo para correlacionar os *spans* provenientes dos diversos serviços. Dessa forma, para obter os *traces* mais recentes o instante de tempo final, T_{end}, será o instante de tempo atual menos uma constante de tempo, denominado neste trabalho como *drift*. A representação gráfica dessa dinâmica pode ser visualizada na Figura 17, onde apenas os *traces* que possuírem os *spans* com instantes de tempo compreendidos pela área em vermelho destacada serão retornados.

Figura 17 – Esquemático dos parâmetros de consulta dos *traces* na API do Zipkin em uma linha temporal.

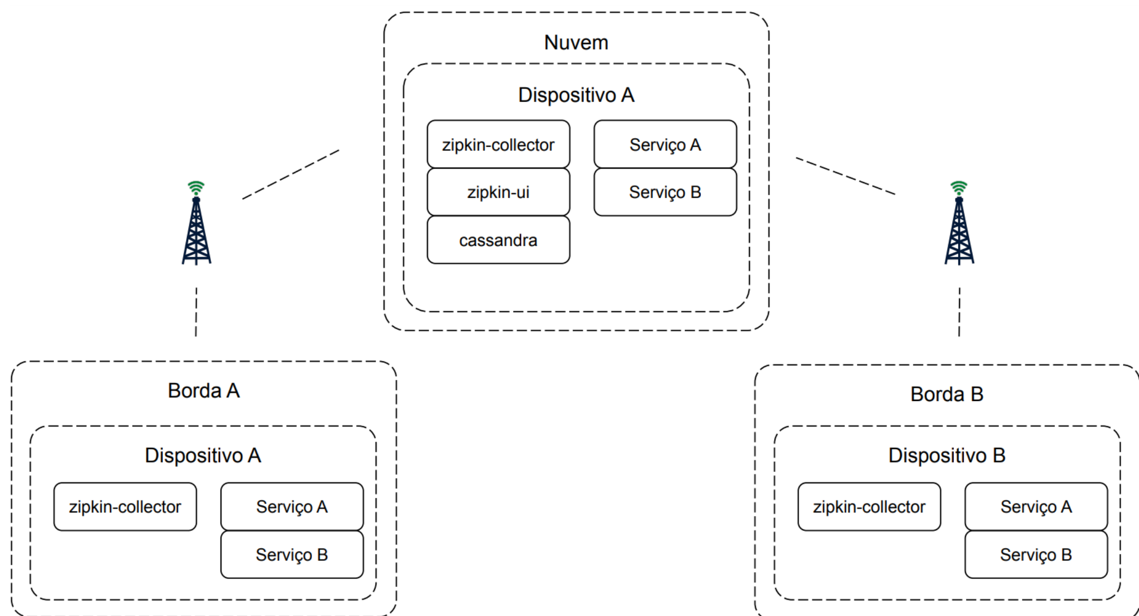


Fonte: Produção do próprio autor.

Além disso, neste trabalho foi necessário planejar a implantação da ferramenta Zipkin no *cluster*, pois tem-se um ambiente com nós em borda e em nuvem. Cada serviço instrumentado para realizar medições, deve enviar essas informações para uma API e, se não planejado corretamente, a perda de informações do sistema pode ocorrer. Dessa forma, foi proposta a divisão dos serviços que compõem toda a ferramenta Zipkin em: (i) *zipkin-collector*, serviço que disponibiliza API para coleta de *traces* enviados pelos serviços do PIS; (ii) *zipkin-ui*, interface web para visualização na forma de gráficos e diagramas; (iii) *cassandra*, banco de dados para o armazenamento e persistência dos dados.

Na Figura 18, pode-se observar a arquitetura de coleta de *traces* proposta com a ferramenta Zipkin. Tanto em borda quanto em nuvem, são executados os *zipkin-collectors* responsáveis por disponibilizar em cada ambiente uma API para coleta dos *traces* dos serviços do PIS, formação de lotes e garantir a inserção dessas informações no banco de dados em nuvem. Apenas em nuvem é executada a interface web de visualização, *zipkin-ui*. Além disso, a API executada em nuvem, através do componente *zipkin-collector*, é habilitada para realizar consultas no banco de dados, enquanto os *zipkin-collector* em borda só podem realizar a inserção de informações no banco de dados.

Figura 18 – Esquemático de implantação da ferramenta Zipkin neste trabalho.



Fonte: Produção do próprio autor.

É importante mencionar que nenhum desses serviços (*zipkin-ui*, *zipkin-collector*, *cassandra*) foram desenvolvidos neste trabalho, apenas analisou-se os executáveis gerados pelo projeto Zipkin (usualmente são executados juntos em um único contêiner) . Além disso, a escolha da ferramenta Zipkin foi realizada devido ao suporte já integrado às aplicações previamente desenvolvidas para o PIS em trabalhos anteriores (PICORETI et al., 2018; COTTA, 2020).

3.2 Conexão entre *brokers*

Às vezes, é necessário mover mensagens de forma confiável e contínua de uma origem em um *broker* RabbitMQ para um destino em outro *broker* RabbitMQ. A origem e o destino podem estar no mesmo *cluster* ou não. Uma *Shovel* é um recurso implementado através do *plugin* do RabbitMQ que move mensagens unidirecionalmente de uma origem para um destino, ou seja, age como um aplicativo cliente bem elaborado, conectando-se à origem e ao destino, consumindo e republicando mensagens e utilizando confirmações de ambos os

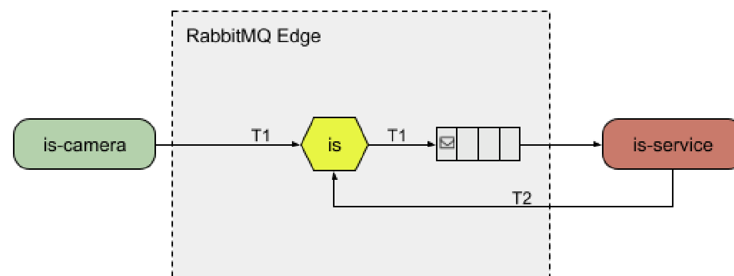
lados para lidar com falhas. Desta forma, tal recurso torna-se uma ferramenta compatível com redes WAN (do inglês, *Wide Area Network*).

Além disso, uma *Shovel* pode ser dinâmica ou estática. Uma *Shovel* estática é criada através de arquivos de configurações que são lidos durante a inicialização do RabbitMQ. Por outro lado, uma *Shovel* dinâmica pode ser configurada através de uma API de gerenciamento do RabbitMQ, iniciada e interrompida a qualquer momento. Dessa forma, as *Shovels* podem ser utilizadas para cargas de trabalho transitórias ou cargas de trabalho permanentes.

Assim, neste trabalho pretende-se ter um *broker* RabbitMQ em nuvem e também um em cada borda. Através da API de gerenciamento do *broker* RabbitMQ em nuvem, criar *Shovels* dinâmicas programaticamente e dessa forma garantir o encaminhamento de mensagens de uma borda para nuvem ou da nuvem para uma borda.

Para melhor compreender a proposta deste trabalho, considere um *broker* RabbitMQ e um grupo de serviços todos os em borda: um serviço genérico (*is-service*) que deseja consumir imagens que são publicadas com um tópico T1 (por exemplo, *CameraGateway.0.Frame*) por um outro serviço (*is-camera*) e então fornecer os resultados através de mensagens com tópico T2 (por exemplo, *Service.0.Results*) na *exchange* IS para que outras aplicações possam consumir estes resultados em borda. A Figura 19 apresenta um esquemático da comunicação dos serviços com o RabbitMQ nessa situação.

Figura 19 – Representação da comunicação de um serviço genérico no PIS em borda.



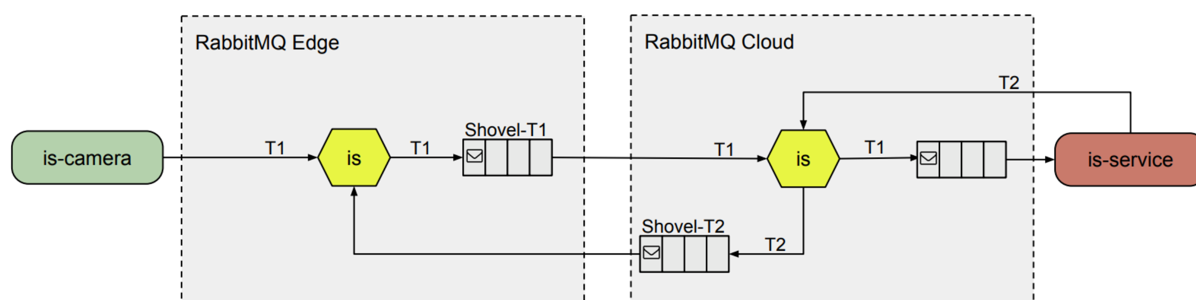
Fonte: Produção do próprio autor.

Este serviço, *is-service*, poderia ser executado em nuvem. No entanto, é necessário que neste caso as mensagens sejam encaminhadas corretamente para o *broker* RabbitMQ em nuvem e as respostas do serviço em nuvem também sejam encaminhadas de volta para o *broker* RabbitMQ em borda. Para isso, através da API de gerenciamento do *broker* RabbitMQ em nuvem, são criadas duas *Shovels* do tipo *exchange-exchange* especificando os tópicos de interesse a serem encaminhados entre as *exchanges*.

Observe a Figura 20, onde o mesmo serviço em borda apresentado na Figura 19 é executado em nuvem. Como deseja-se encaminhar as mensagens com tópico T1 em borda, é necessário

criar uma fila para acumular as mensagens que chegam a *exchange* IS com esse tópico. Então, o RabbitMQ encarrega-se de enviar para *exchange* IS do RabbitMQ em nuvem. Da mesma forma, como deseja-se encaminhar as mensagens com tópico T2 em nuvem, é necessário criar uma fila para acumular as mensagens que chegam na *exchange* IS com esse tópico. Então, o RabbitMQ encarrega-se de enviar para *exchange* IS do RabbitMQ em borda. Todo esse processo de criação de filas é feito automaticamente na criação das *Shovels* através da API de gerenciamento do RabbitMQ em nuvem.

Figura 20 – Representação da comunicação de um serviço genérico orquestrado em nuvem para atender uma aplicação em borda.



Fonte: Produção do próprio autor.

Pode-se criar uma *Shovel* através da API de gerenciamento ou interface web do RabbitMQ. Na Figura 21, observa-se na interface web a criação de uma *Shovel* para o compartilhamento das imagens de uma câmera em borda com o RabbitMQ em nuvem. É importante mencionar que após a criação, apenas as mensagens com tópico de interesse serão compartilhadas entre os RabbitMQ. Ou seja, durante todo o tempo de existência da *Shovel*, todas as mensagens enviadas para *exchange* IS com tópico de interesse serão publicadas na *exchange* IS do RabbitMQ em nuvem.

Figura 21 – Exemplo de criação de uma *Shovel* para o compartilhamento de imagens de uma câmera através da interface web do RabbitMQ.

Add a new shovel

Name: images *
Source: AMQP 0.9.1
URI: ? Exchange: ?
amqp://rabbitmq.edge * is Routing key: CameraGateway.0.Frame
Prefetch count: ?
Auto-delete ? Never
Destination: AMQP 0.9.1
URI ? Exchange: ?
amqp://rabbitmq.default * is Routing key: CameraGateway.0.Frame
Add forwarding headers: ? No
Reconnect delay: ? s
Acknowledgement mode: ? On publish
Add shovel

Fonte: Produção do próprio autor.

Como as *Shovels* podem ser criadas e deletadas programaticamente, a qualquer momento pode-se excluir ou criar novas através da API de gerenciamento, ou pela interface web do RabbitMQ. Nesse sentido, para encaminhar um conjunto de mensagens durante um intervalo de tempo programaticamente, deve-se realizar a criação da *Shovel* através da API de gerenciamento fornecendo os parâmetros necessários, aguardar um intervalo de tempo e, por fim, deletar a *Shovel* também através da API.

3.3 Desenvolvimento do controlador para o PIS

Para realizar o desenvolvimento de um controlador para o PIS, deve-se integrar o programa do controlador à própria plataforma de orquestração de contêineres, ou seja, no *cluster* Kubernetes. Existem diversas formas de alcançar esse objetivo. Neste trabalho, o controlador foi implementado através da definição do padrão Operador (do inglês, *Operator*).

Operadores são extensões de software para o Kubernetes que fazem uso de recursos personalizados (do inglês, *Custom Resources*) para gerir aplicações e os seus componentes. Este padrão de desenvolvimento surgiu com objetivo de automatizar tarefas antes realizadas manualmente por operadores humanos. Pode-se implantar um operador através da definição de um recurso personalizado (do inglês, *Custom Resource Definition*) e o controlador associado ao *cluster*. Este controlador é normalmente executado fora da camada de gerenciamento do *cluster*, ou seja, de forma independente dentro do *cluster* como qualquer outro serviço em um contêiner. Já a definição de recurso personalizado define suas características e todas as opções de configuração acessíveis aos usuários do operador.

Um operador Kubernetes monitora continuamente a aplicação durante sua execução, o que lhe permite executar automaticamente tarefas como cópia de segurança de dados, recuperação após falhas e atualizações ao longo do tempo. Além disso, as atividades desempenhadas por um operador Kubernetes são extremamente versáteis, abrangendo desde o escalonamento de aplicativos complexos até a realização de atualizações de versão. Dessa forma, pode-se resumir o comportamento do controlador para aplicações do PIS em dois modos de processamento:

- Modo de processamento em borda. Neste caso, deve-se testar a conexão com a nuvem periodicamente para avaliar a possibilidade de troca para o modo de processamento em nuvem. Assim, configura-se uma *Shovel* durante um intervalo de tempo para encaminhar mensagens para a nuvem. Por isso, é necessário que o serviço em nuvem exista para geração de *traces*. Caso o tempo de resposta, especificado através de uma

lista de *spans*, for maior que um limiar deve-se manter o modo de processamento em borda. No entanto, se o tempo de resposta for menor deve-se mudar o modo de processamento para nuvem.

- Modo de processamento em nuvem. Neste caso, como todas as mensagens são encaminhadas para nuvem e de volta, é necessário persistir a conexão de teste e adicionar outras. Como *traces* são gerados continuamente permitindo avaliar a qualidade da conexão com a nuvem, pode-se escalonar o serviço em borda para zero réplicas. Caso o tempo de resposta, especificado através de uma lista de *spans*, seja menor que um limiar, deve-se manter o modo de processamento em nuvem. No entanto, se o tempo de resposta for maior, deve-se mudar o modo de processamento para borda. O que implica no escalonamento de volta da aplicação em borda para a quantidade de réplicas que haviam anteriormente em borda, bem como a exclusão das *Shovels* nos *brokers* RabbitMQ.

Para consulta dos *traces* mais recentes na ferramenta Zipkin, afim de decidir o modo de processamento, a quantidade máxima de *traces* é configurada para 1000, o instante de tempo final será o instante de tempo atual menos uma constante de tempo de 2min (*drift*) e o intervalo de tempo (*lookback*) foi de 2min. Dessa forma, o controlador terá um ação de controle atrasada em 2min no sistema. Consequentemente, em cada consulta do controlador na API do Zipkin, tem-se os resultados dos tempos de resposta entre 4min e 2min atrás. Então, o controlador realiza a média destes *traces* para decidir o modo de processamento. Pode-se pensar neste valor como uma média móvel durante todo o experimento.

A Figura 22 oferece uma visão da lógica de controle incorporada no projeto do controlador, neste trabalho, através de um fluxograma simplificado. É importante mencionar que, para considerar o tempo de resposta em nuvem baixo, é necessário que a média de dez consultas (este parâmetro pode ser configurado durante a inicialização do controlador) consecutivas à API do Zipkin se mantenha abaixo de um limiar especificado na definição do controlador no PIS. Assim, assegura-se que o controlador evite alternar repetidamente entre os modos de processamento.

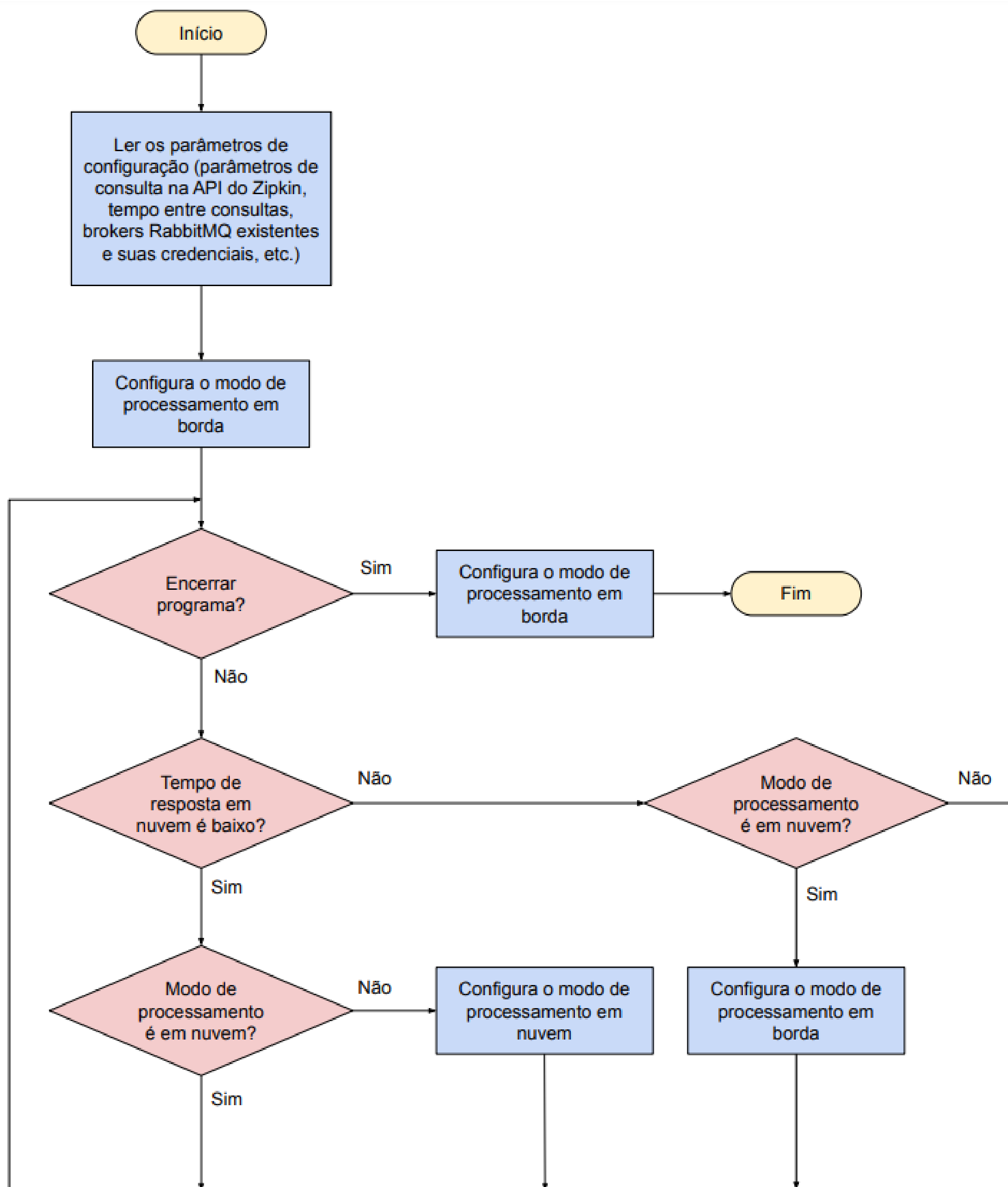
Dessa forma, o controlador para um serviço genérico no PIS foi pensado tal como um recurso personalizado tal como definido no Código 3.1. A definição de recurso personalizado pode ser encontrada no Apêndice .1. Assim, é necessário apontar os serviços em nuvem ou em borda através dos campos *spec.cloud.deployment* (linhs 8 à 10) e *spec.edge.deployment* (linhas 13 à 15). Além disso, deve-se apontar os *spans* dentro dos *traces* que devem ser contabilizados para encontrar o tempo de resposta para o controle (linhas 17 à 22). O

valor de limiar para o controle é dado pelo campo *tracesTargetValue* (linha 23, caso seja menor que este limiar, deve-se processar em nuvem). Por fim, lista-se as *Shovels* que devem ser criadas através da API de gerenciamento do RabbitMQ em nuvem. Observe que, há dois campos: *connections.test* (linha 25) e *connections.stable* (linha 30). Isso ocorre porque é necessário que algumas *Shovels* sejam estabelecidas de tempos em tempos para gerar *traces* e avaliar a qualidade da conexão com a nuvem. Assim, as conexões do campo *connections.test* (linha 26 à 29) são realizadas de tempos em tempos quando o modo de processamento é em borda e continuamente quando o modo de processamento é em nuvem. Já as conexões do campo *connections.stable* (linha 31 à 34) são configuradas somente quando o modo de processamento é em nuvem.

Código 3.1 – Exemplo de um controlador para um serviço genérico no PIS.

```
1  apiVersion: labvisio.ufes.br/v1
2  kind: IsController
3  metadata:
4    name: is-service-controller
5  spec:
6    cloud:
7      deployment:
8        name: is-service-cloud
9        namespace: default
10       apiVersion: apps/v1
11   edge:
12     deployment:
13       name: is-service-edge
14       namespace: default
15       apiVersion: apps/v1
16   spans:
17   - name: frame
18     service: cameragateway
19   - name: commtime
20     service: service.commtime
21   - name: service
22     service: service
23   tracesTargetValue: 100000
24   connections:
25     test:
26     - name: images
27       src: edge
28       dest: default
29       topic: "CameraGateway.0.Frame"
30     stable:
31     - name: tfs
32       src: default
33       dest: edge
34       topic: "Service.0.Results"
```

Figura 22 – Esquemático com o fluxograma de funcionamento do controlador.



Fonte: Produção do próprio autor.

4 EXPERIMENTOS E RESULTADOS

Neste capítulo serão apresentados os experimentos realizados, bem como os resultados obtidos. Para isso, o capítulo inicia apresentando os recursos computacionais, de hardware e software, utilizados para realização deste trabalho e depois os estudos de casos abordados para validação do controlador proposto.

Este trabalho está disponível na plataforma de hospedagem de código-fonte e arquivos com controle de versão chamada GitHub, acessível no repositório chamado *luizcarloscf/tcc*[1]. Além dos serviços desenvolvidos, todos os arquivos necessários para executar as demais aplicações utilizadas neste projeto também podem ser encontrados lá.

Inicialmente, estava planejada a utilização de VANTs ou robôs terrestres para a realização de experimentos reais, como o seguimento de alvo móvel. Contudo, devido à indisponibilidade desses dispositivos para utilização no laboratório, optou-se por conduzir experimentos físicos em situação de rede controlada. Nestes experimentos, utilizou-se Raspberrys com uma câmera conectada para emular um dispositivo robótico e realizar a aquisição de dados. Dessa maneira, foi possível controlar as condições de rede e inserir atrasos na transmissão de dados para a nuvem afim de observar a ação do controlador para o PIS proposto neste trabalho.

4.1 Recursos Computacionais

Nesta seção é apresentada uma visão geral dos recursos computacionais, software e hardware, utilizados para o desenvolvimento deste trabalho. É importante mencionar que todos os experimentos apresentados neste trabalho foram realizados no PIS presente no Lab VISIO - Laboratório de Visão Computacional e Robótica do PPGEE da UFES.

Recursos de Software

Na etapa de criação e configuração da plataforma de orquestração de contêineres, utilizou-se o software Kubernetes 1.25.0, além de todo o conjunto de softwares, tais como RabbitMQ 3.7.6 e Zipkin 2.24.1, que compõem a plataforma do PIS. Como requisito para a medição do tempo de resposta e a correta correlação dos eventos na ferramenta de *tracing*, utilizou-se

[1] <https://github.com/luizcarloscf/tcc>

o software chamado Chrony 4.2 para a sincronização dos relógios dos computadores. Já durante as etapas de adaptação das aplicações do sistema, desenvolvimento e validação do controlador proposto foi utilizada a linguagem de programação em Python 3.10 e o conjunto de ferramentas de suporte à aplicação do PIS previamente desenvolvidas. Além disso, todas as aplicações foram empacotadas na forma de contêineres utilizando o software Docker 20.10.

Para simular alterações das condições de rede durante os experimentos, como aumento do tempo de transmissão de uma imagem da borda para nuvem, foi utilizado o comando *tc* do sistema operacional Linux. Nos experimentos realizados, foram inseridos atrasos a todos os pacotes que são enviados pela interface de rede Wi-Fi do dispositivo em borda (Raspberry).

Recursos de Hardware

Como dispositivo em borda conectado via WiFi em uma rede de 5 Ghz, foi utilizada uma Raspberry Pi modelo 4B com as seguintes configurações: (i) sistema operacional Linux, distribuição Ubuntu Server 22.04; (ii) processador Cortex-A72 (ARM v8) 64-bit SoC, 1,5 GHz com 4 núcleos físicos; (iii) memória SDRAM de 4 GB LPDDR4 3200 MHz; (iv) conexão WiFi de 2,4 GHz e 5,0 GHz IEEE 802.11ac; (v) cartão de memória de 32 GB (classe 10, até 80 MB/s); (v) Raspberry Pi Camera Module 2.

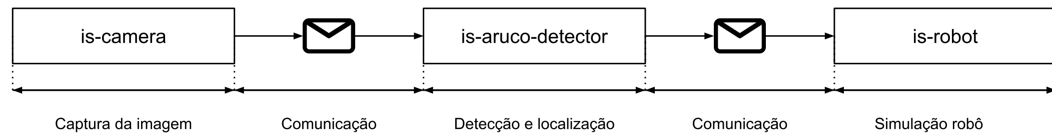
Além disso, foi utilizado o *datacenter* presente no laboratório como ambiente de computação em nuvem para realização dos testes de validação do sistema proposto. Nesse *datacenter*, a máquina com as seguintes configurações foi utilizada como máquina de computação de nuvem: (i) sistema operacional Linux, distribuição Ubuntu Server 22.04; (ii) processador Intel Xeon, 3,2 GHz com 28 núcleos físicos; (iii) memória DDR4 de 64 GB; (iv) conexão Ethernet de 1 GHz; (v) SSD de 480 GB; (iv) 3 placas NVIDIA GeForce RTX 3060 e 1 placa NVIDIA GeForce RTX 3090.

4.2 Estudo de caso I - Detecção e localização de marcadores visuais

Tal como destacado na Figura 12 (repetida abaixo como Figura 23), os serviços que fazem parte desse primeiro estudo de caso são: *is-camera*, *is-aruco-detector*, *is-robot*. Nesse caso, feita a correta configuração de roteamento de mensagens entre os *brokers* em nuvem e em borda, o serviço *is-aruco-detector* poderia ser executado tanto em nuvem quanto em borda. Enquanto, os serviços *is-camera* e *is-robot* podem ser executados apenas em borda. Isso, porque o serviço *is-camera* precisa ter acesso à câmera conectada na placa Raspberry e o

serviço *is-robot* simula a aquisição de informações da nuvem para o controle de um robô.

Figura 23 – Esquemático de comunicação entre os serviços do experimento de detecção de marcadores visuais.

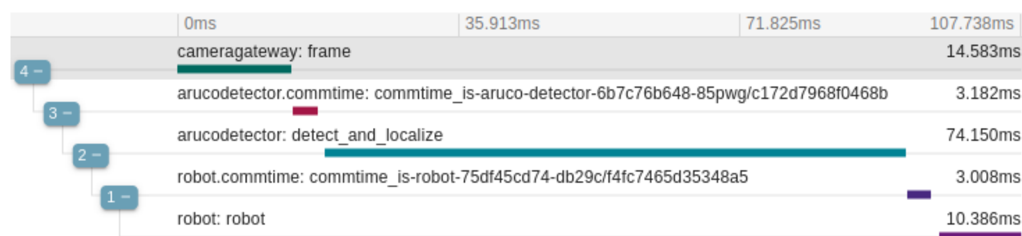


Fonte: Produção do próprio autor.

Inicialmente, será considerado que todos os serviços são executados em borda, ou seja, na Raspberry - dispositivo de computação em borda neste experimento. Na Figura 24, pode-se observar um exemplo de *trace* dos serviços que fazem parte da detecção e localização de marcadores visuais do tipo ArUco. A soma de todos os *spans* que compõem este *trace* formam o tempo de resposta total, uma vez que é a combinação do tempo de processamento dos diversos serviços, somado ao tempo de comunicação entre eles.

Devido à capacidade limitada de processamento, a tarefa de detecção e localização dos marcadores visuais do tipo ArUco é computacionalmente custosa para o dispositivo em borda. Apesar disso, o dispositivo consegue processar todas as imagens a uma taxa de dez imagens por segundo, sem a necessidade de descartar nenhuma imagem. Além disso, o tempo de comunicação é extremamente baixo já que não há transmissão da imagem via rede.

Figura 24 – Exemplo de *trace* com processamento de imagens em borda.

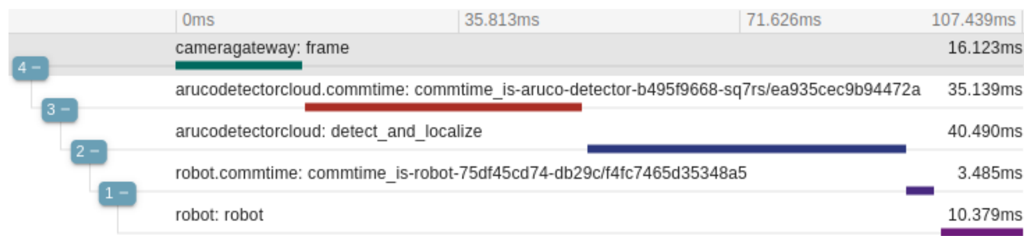


Fonte: Produção do próprio autor.

Por outro lado, quando o processamento é realizado em nuvem, observa-se um menor tempo de processamento, devido à maior capacidade de processamento do servidor presente em nuvem, e um maior tempo de comunicação, devido à necessidade de transmissão da imagem via rede. Na Figura 25, observa-se tal situação.

Dessa forma, analisou-se a média e desvio padrão de 1.000 *traces* das situações apresentadas na Figura 24 e 25. Estes resultados estão apresentados na Tabela 1. Observa-se que o tempo de resposta total com o processamento de imagens em nuvem é um pouco menor que o tempo de resposta total com o processamento de imagens em borda, quando não há nenhum atraso na transmissão das imagens para a nuvem. No entanto, caso ocorra

Figura 25 – Exemplo de *trace* com processamento de imagens em nuvem.



Fonte: Produção do próprio autor.

um atraso na transmissão da imagem na situação com o processamento em nuvem, o tempo de resposta irá aumentar e poderá ser bem maior ao tempo de resposta total com o processamento de imagens em borda.

Tabela 1 – Tempo de resposta total na tarefa de detecção e localização de marcadores com o processamento de imagens em borda e em nuvem.

	Nuvem	Borda (Raspberry)
Média (ms)	80,6855	115,4512
Desvio padrão (ms)	15,6016	16,4924

Fonte: Produção do próprio autor.

Para controlar o processamento de imagens, o controlador do Código 4.1 é proposto. Observe que é necessário que já existam os *deployments* tanto em nuvem (linhas 8 à 10) quanto em borda (linhas 13 à 15), devidamente configurados. Assim, especificou-se os *spans* nos *traces* que devem ser contabilizados para decisão de processamento (linhas 17 à 22), bem como o valor do limiar de decisão (linha 23). Dessa forma, caso o tempo de captura da imagem, comunicação e localização de ArUcos em nuvem seja maior que 85.000 μs, o processamento será realizado em borda. Entretanto, se esse tempo for menor ou igual a tal limiar, o processamento será realizado todo em nuvem.

Código 4.1 – Exemplo de um controlador para o serviço de detecção de ArUco.

```
1  apiVersion: labvisio.ufes.br/v1
2  kind: IsController
3  metadata:
4    name: is-aruco-detector-controller
5  spec:
6    cloud:
7      deployment:
8        name: is-aruco-detector-cloud
9        namespace: default
10       apiVersion: apps/v1
11   edge:
12     deployment:
13       name: is-aruco-detector-edge
14       namespace: default
15       apiVersion: apps/v1
16   spans:
```

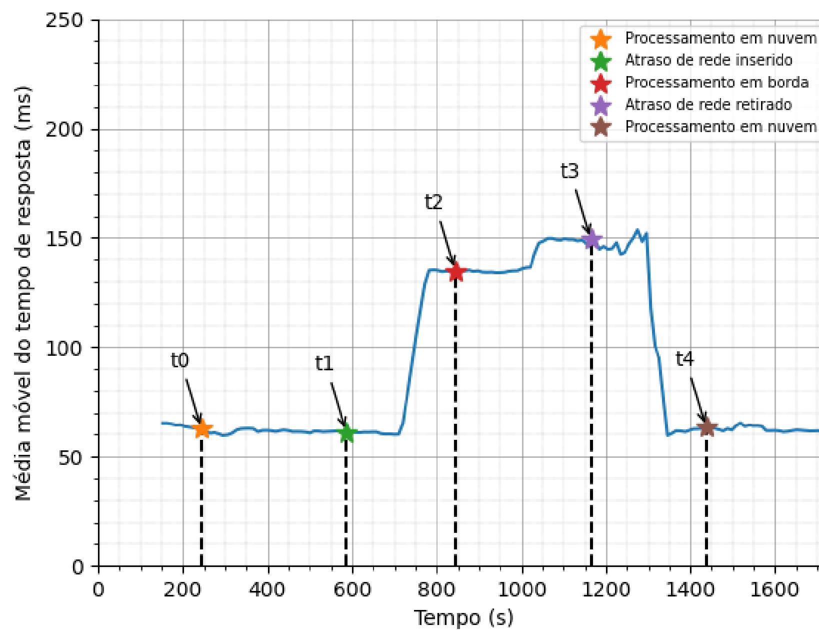
```
17    - name: frame
18      service: cameragateway
19    - name: commtime
20      service: arucodetectorcloud.commtime
21    - name: detect_and_localize
22      service: arucodetectorcloud
23    tracesTargetValue: 85000
24    connections:
25      test:
26      - name: images
27        src: edge
28        dest: default
29        topic: "CameraGateway.0.Frame"
30      stable:
31      - name: tfs
32        src: default
33        dest: edge
34        topic: "ArUco.0.FrameTransformations"
```

O valor do limiar de decisão foi determinado através de uma análise dos valores apresentados na Tabela 1. Quanto menor o limiar do tempo de resposta em nuvem, maior a susceptibilidade a mudanças para o processamento em borda. Ou seja, menor será o atraso na transmissão das mensagens para levar ao modo de processamento em borda. Através desse limiar, é possível estabelecer a sensibilidade do controlador. Em cada aplicação, é crucial que o usuário pondere esse valor ao criar o recurso. Em situações em que o processamento em borda é desejado apenas em circunstâncias críticas, faz sentido estabelecer um valor de limiar mais elevado. No entanto, se a capacidade computacional do dispositivo em borda atende aos requisitos da aplicação, um valor de limiar mais baixo pode ser mais apropriado. Como neste caso, o dispositivo em borda possui capacidade computacional para a detecção e localização dos marcadores visuais, utilizou-se um valor de 85.000 μs de limiar.

Na Figura 26, observa-se o tempo de resposta especificado na definição do controlador durante todo o experimento. Isto é, o tempo de resposta com o serviço de detecção de marcadores visuais em nuvem. Inicialmente, o modo de processamento sempre começa em borda. Antes do instante de tempo t_0, o controlador verificou que a média do tempo de resposta é favorável ao processamento em nuvem, através da amostragem de imagens periodicamente. Logo, no instante de tempo t_0, o modo de processamento é configurado para nuvem pelo controlador. No entanto, no instante de tempo t_1, um atraso de rede de 75 ms é inserido manualmente na interface de rede do dispositivo em borda. Assim, ocorre um aumento do tempo de comunicação, o que força o controlador a realizar o processamento em borda no instante de tempo t_2. Já no instante de tempo t_3, o atraso de rede é retirado manualmente, implicando em uma redução do tempo de resposta e levando o controlador

a realizar o processamento em nuvem novamente no instante de tempo t_4.

Figura 26 – Média móvel do tempo de resposta em nuvem ao longo do experimento.



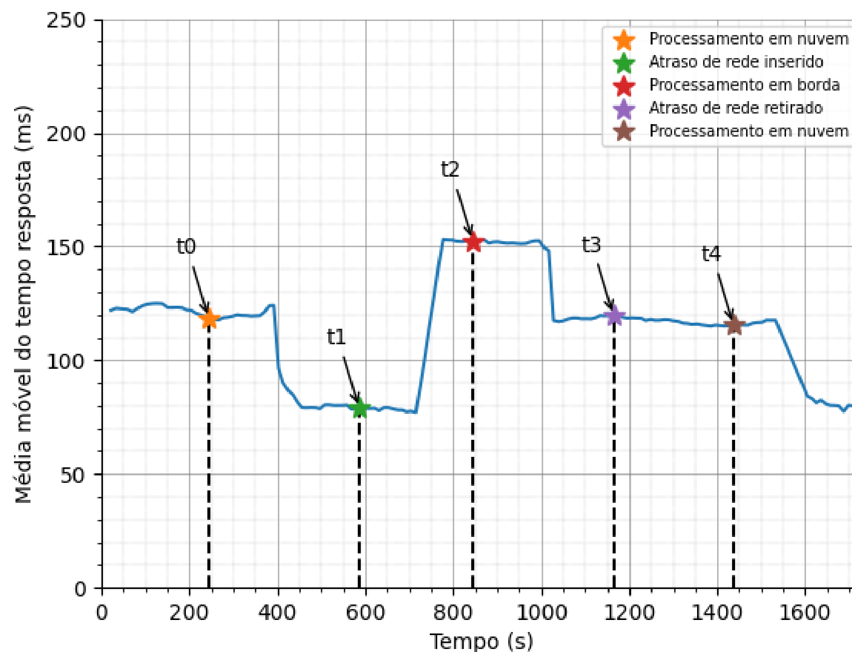
Fonte: Produção do próprio autor.

Para observar se o controle realizado através da curva apresentada na Figura 26 foi efetivo, é necessário observar o tempo de resposta total da aplicação. Isto é, deve-se verificar se o controle realizado permite a minimização do tempo de resposta total. Define-se o tempo de resposta total como a soma do tempo de processamento para captura da imagem, tempo de comunicação e processamento com o serviço de detecção e localização de marcadores visuais (seja em nuvem ou em borda), tempo de comunicação e processamento com o serviço de simulação de controle do robô.

Portanto, na Figura 27, mede-se o tempo de resposta total da mesma forma que o controlador realiza durante todo o experimento. Então, obteve-se a curva da média móvel do tempo de resposta total da aplicação. É possível visualizar que antes do instante de tempo t_0, o tempo de resposta total era em torno 125 ms. Quando o processamento mudou para nuvem, o tempo de resposta total diminuiu para algo em torno de 80 ms e permaneceu. No instante de tempo t_1, quando o atraso de rede é inserido, observa-se que o tempo de resposta total aumenta e, consequentemente, no instante de tempo t_2, o modo de processamento mudou para a borda. Assim, levando o tempo de resposta total ao patamar de 125 ms novamente. Entre os instantes de tempo t_3 e t_4, o tempo de resposta total não mudou. Entretanto, observa-se que na curva da Figura 26 a conexão com a nuvem melhorou, devido à retirada do atraso de rede. Como neste momento o modo de processamento está em borda, não nota-se nenhuma mudança no tempo de resposta total.

Entretanto, como o controle é realizado através da curva apresentada na Figura 26, no instante de tempo t_4 todo o processamento muda para nuvem, levando o tempo de resposta total ao patamar de 80 ms novamente.

Figura 27 – Média móvel do tempo de resposta total ao longo do experimento.



Fonte: Produção do próprio autor.

É importante notar que, durante todo o experimento, o controlador agiu sobre os serviços, escalonando e configurando a comunicação, com o objetivo de minimizar o tempo de resposta total. No entanto, é necessário conhecimento do desenvolvedor durante a configuração do controlador para especificar corretamente o tempo de resposta que será contabilizado para o controle.

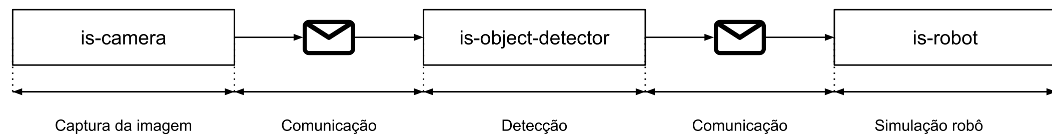
Além disso, observa-se nos gráficos apresentados nas Figuras 26 e 27 um intervalo de tempo de 2 minutos entre qualquer ação realizada e o efeito observado nos gráficos. Esse intervalo decorre do fato de que, em cada consulta do controlador na API do Zipkin, os resultados dos tempos de resposta estão compreendidos entre 4 minutos e 2 minutos atrás.

4.3 Estudo de caso II - Detecção de pessoas

Tal como destacado na Figura 14 (repetida abaixo como Figura 28), os serviços que fazem parte desse segundo estudo de caso são: *is-camera*, *is-object-detector*, *is-robot*. Nesse caso, uma vez feita a configuração de roteamento de mensagens entre os *brokers* em nuvem e

em borda, o serviço *is-object-detector* pode ser orquestrado tanto em nuvem quanto em borda. Entretanto, os serviços *is-camera* e *is-robot* só podem ser executados em borda, tal como visto no primeiro estudo de caso.

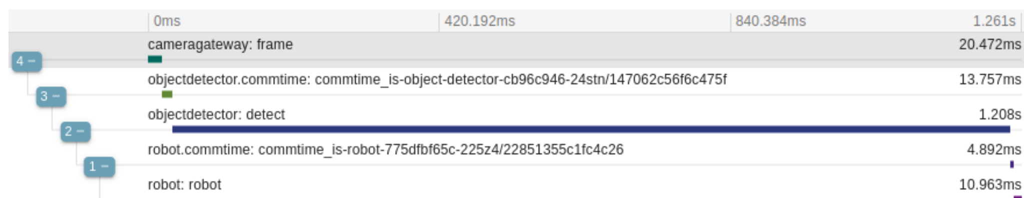
Figura 28 – Esquemático de comunicação entre os serviços do experimento de detecção de objetos.



Fonte: Produção do próprio autor.

A Figura 29 mostra um exemplo de *trace* dos serviços que fazem parte da detecção de pessoas quando todos são executados em borda. Da mesma forma que o apresentado na Figura 24, a soma de todos os *spans* que compõe este *trace* formam o tempo de resposta total, uma vez que é a combinação do tempo de processamento dentro dos diversos serviços, somado ao tempo de comunicação entre eles. Devido à capacidade limitada de processamento, a tarefa de detecção de objetos é computacionalmente pesada para o dispositivo em borda. Assim, algumas mensagens são enfileiradas no *broker* em borda. Isso implica em um tempo de comunicação maior e equivalente, ou pior, à transmissão da imagem via rede para nuvem em condições favoráveis.

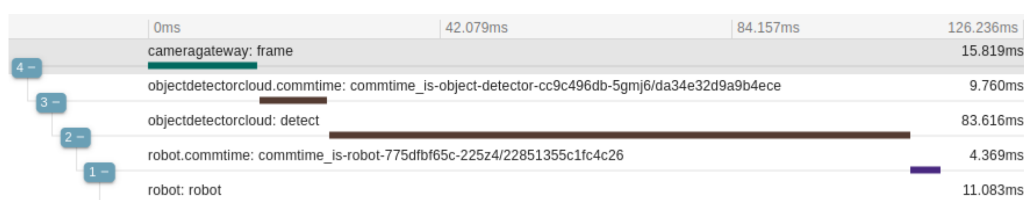
Figura 29 – Exemplo de *trace* com processamento de imagens em borda.



Fonte: Produção do próprio autor.

Por outro lado, quando o processamento é realizado em nuvem, observa-se um menor tempo de processamento e um menor tempo de comunicação, devido à maior capacidade de processamento do servidor. Assim, o serviço de detecção de objetos consegue lidar com o fluxo de imagens. Na Figura 30, pode-se observar tal situação.

Figura 30 – Exemplo de *trace* com processamento de imagens em nuvem.



Fonte: Produção do próprio autor.

Dessa forma, analisou-se a média e desvio padrão de 1.000 *traces* das situações apresentadas na Figura 29 e 30. Estes resultados estão apresentados na Tabela 2. Pode-se observar que o

tempo de resposta total com o processamento de imagens em nuvem é aproximadamente 10 vezes menor que o tempo de resposta total com o processamento de imagens em borda. No entanto, caso ocorra um atraso na transmissão da imagem na situação com o processamento em nuvem, o tempo de resposta irá aumentar e poderá se igualar ao tempo de resposta total com o processamento de imagens em borda.

Tabela 2 – Tempo de resposta total na tarefa de detecção de objetos com o processamento de imagens em borda e em nuvem.

	Nuvem	Borda (Raspberry)
Média (ms)	124,73	968,84
Desvio padrão (ms)	39,9	157,16

Fonte: Produção do próprio autor.

Para controlar o processamento de imagens, o controlador do Código 4.2 é proposto. É necessário que já existam os *deployments* tanto em nuvem (linhas 8 à 10) quanto em borda (linhas 13 à 15), devidamente configurados. Assim, especificaram-se os *spans* nos *traces* que devem ser contabilizados para decisão de processamento (linhas 17 à 22), bem como o valor do limiar de decisão (linha 23). Dessa forma, caso o tempo de captura da imagem, comunicação e detecção de pessoas for maior que 1.000.000 μs (1 s), o processamento será realizado em borda. Entretanto, se esse tempo for menor que 1.000.000 μs (1 s), o processamento será realizado todo em nuvem. Tal como foi possível observar no estudo de caso envolvendo a detecção e localização de marcadores visuais, o valor do limiar de decisão foi determinado através de uma análise dos valores apresentados na Tabela 2. Como neste caso, o processamento em borda é desejado apenas em circunstâncias críticas, faz sentido estabelecer um valor de limiar mais elevado. Além disso, como o dispositivo em borda não possui uma alta capacidade computacional para a detecção de objetos, utilizou-se um valor alto como limiar.

Código 4.2 – Exemplo de um controlador para o serviço de detecção de pessoas.

```
1  apiVersion: labvisio.ufes.br/v1
2  kind: IsController
3  metadata:
4    name: is-object-detector-controller
5  spec:
6    cloud:
7      deployment:
8        name: is-object-detector-cloud
9        namespace: default
10       apiVersion: apps/v1
11   edge:
12     deployment:
13       name: is-object-detector-edge
14       namespace: default
15       apiVersion: apps/v1
16   spans:
```

```
17    - name: frame
18      service: cameragateway
19    - name: commtime
20      service: objectdetectorcloud.commtime
21    - name: detect
22      service: objectdetectorcloud
23  tracesTargetValue: 1000000
24  connections:
25    test:
26    - name: images
27      src: edge
28      dest: default
29      topic: "CameraGateway.0.Frame"
30    stable:
31    - name: objs
32      src: default
33      dest: edge
34      topic: "Object.0.Detections"
```

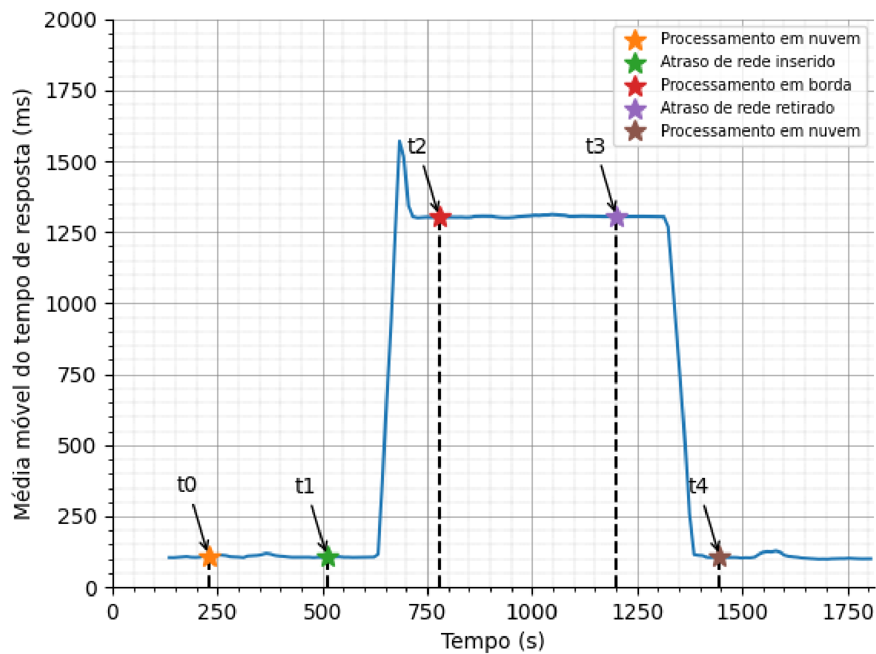
Na Figura 31, observa-se o tempo de resposta especificado na definição do controlador durante todo o experimento. Isto é, o tempo de resposta com o serviço de detecção de objetos em nuvem. Assim, iniciou-se o experimento com o processamento em borda. Antes do instante de tempo t_0, verifica-se que a média do tempo de resposta é favorável para o processamento em nuvem, através do encaminhamento periódico de mensagens para nuvem. Então, no instante de tempo t_0, o modo de processamento é configurado para nuvem. No entanto, no instante de tempo t_1 o atraso de rede de 1 s é inserido na interface de rede do dispositivo em borda. Isso implica em um aumento do tempo de comunicação e força o controlador a realizar o processamento em borda no instante de tempo t_2. Já no instante de tempo t_3, o atraso de rede é retirado, causando uma redução do tempo de resposta e levando o controlador a realizar o processamento em nuvem novamente no instante de tempo t_4.

Assim como no estudo de caso anterior, para observar se o controle realizado através da curva apresentada na Figura 31 foi efetivo, deve-se verificar se o controle realizado permite a minimização do tempo de resposta total. Vale relembrar que o tempo de resposta total é a soma do tempo de processamento para captura da imagem, tempo de comunicação e processamento com o serviço de detecção de objetos (seja em nuvem ou em borda), tempo de comunicação e processamento com o serviço de simulação de controle do robô.

Portanto, na Figura 32, mostra-se o tempo de resposta total durante todo experimento. Então, obteve-se a curva da média móvel do tempo de resposta total da aplicação. Observa-se que antes do instante de tempo t_0, o tempo de resposta total era em torno 1.000 ms. Quando o processamento mudou para nuvem, o tempo de resposta total diminuiu

para aproximadamente 125 ms e permaneceu. No instante de tempo t_1, quando o atraso de rede é inserido, observa-se que o tempo de resposta total aumenta para 1.250 ms e, consequentemente, no instante de tempo t_2, o modo de processamento mudou para a borda. Assim, levando o tempo de resposta total ao patamar de 1.000 ms novamente. Entre os instantes de tempo t_3 e t_4, o tempo de resposta total não mudou. Entretanto, observa-se que na curva da Figura 31 a conexão com a nuvem melhorou, devido a retirada do atraso de rede. Como neste momento o modo de processamento está em borda, não nota-se nenhuma mudança no tempo de resposta total. Entretanto, como o controle é realizado através da curva apresentada na Figura 31, no instante de tempo t_4 todo o processamento muda para nuvem, levando o tempo de resposta total ao patamar de 125 ms novamente.

Figura 31 – Média móvel do tempo de resposta em nuvem ao longo do experimento.

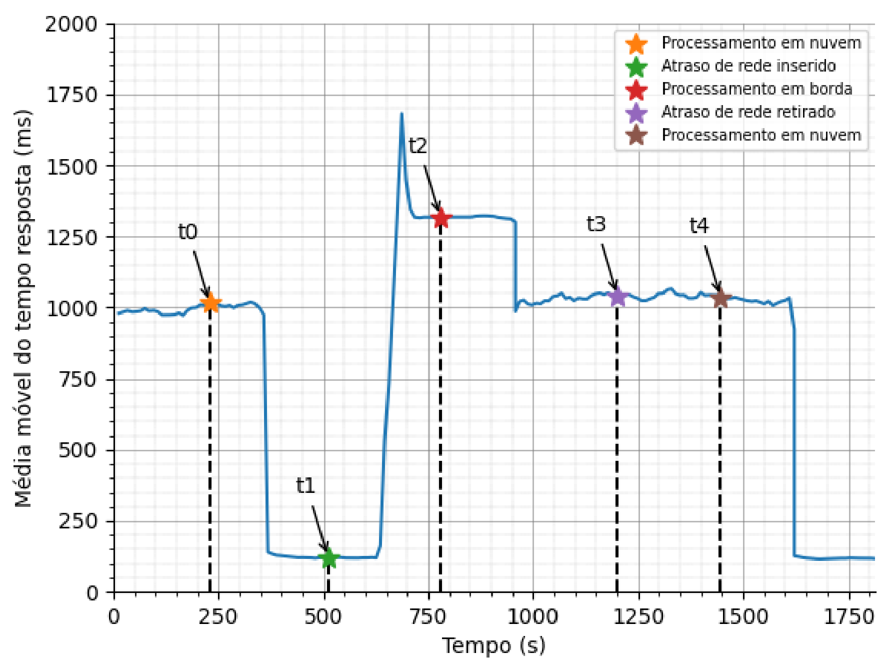


Fonte: Produção do próprio autor.

Assim, da mesma forma que no estudo de caso de detecção e localização de marcadores visuais, o controlador agiu sobre os serviços durante todo o experimento, orquestrando e configurando a comunicação, com o objetivo de minimizar o tempo de resposta total.

Além disso, observa-se nos gráficos apresentados nas Figuras 31 e 32 um intervalo de tempo de 2 minutos entre qualquer ação realizada e o efeito observado nos gráficos, semelhante ao que foi observado no experimento anterior. Esse intervalo decorre do fato de que, em cada consulta do controlador na API do Zipkin, os resultados dos tempos de resposta estão compreendidos entre 4 minutos e 2 minutos atrás.

Figura 32 – Média móvel do tempo de resposta total ao longo do experimento.



Fonte: Produção do próprio autor.

5 CONCLUSÃO E TRABALHOS FUTUROS

5.1 Conclusão

O objetivo principal deste trabalho foi propor a expansão do PIS através de dispositivos em borda, tanto para computação em borda de aplicações quanto para expansão da abrangência do PIS por meio da disponibilização das informações dos sensores em borda na plataforma. Ao integrar estes dispositivos na plataforma de orquestração de contêineres e desenvolver aplicações para o PIS que utilizem dados dos sensores conectados neles, foi possível disponibilizar estas informações tanto em borda quanto em nuvem.

Também foi proposto um controlador responsável por avaliar as condições dos serviços especificados, configurando o processamento em borda ou em nuvem, quando favorável, a fim de atender os requisitos impostos com utilização racional dos recursos de infraestrutura. Foi possível observar, através dos experimentos realizados, que o controlador proposto agiu sobre o conjunto de serviços especificados procurando sempre minimizar o tempo de resposta quando possível.

O estudo de caso de detecção e localização de marcadores visuais apresentou um cenário menos pesado computacionalmente, enquanto o estudo de caso de detecção de objetos com redes neurais levou o problema de processamento em borda ao extremo. Assim, evidenciou-se que realizar ações apenas nos recursos de computação, como o aumento do número de instâncias ou quantidade de recursos disponíveis, não mitigam os problemas encontrados no PIS. A programabilidade granular da infraestrutura foi essencial para propôr uma solução capaz de lidar com o problema, configurando a comunicação entre os serviços e realizando a migração entre nuvem e borda.

A forma como o controlador foi implementado permite adaptá-lo para diversas situações sem a necessidade de modificação do código-fonte, pois todos os parâmetros podem ser configurados, seja na definição do controlador através do recurso personalizado criado neste trabalho ou através dos parâmetros de configuração do controlador. Em ambos os estudos de casos, tanto na detecção de objetos através de redes neurais quanto na detecção e localização de marcadores visuais do tipo ArUco, foi possível adaptar o controlador para atender os requisitos específicos das aplicações. Caso seja preciso integrar novas informações para o controle, tais como métricas customizadas, pode-se realizar de forma relativamente simples modificando o código-fonte.

Além disso, a proposta de integração de dispositivos móveis traz ao PIS uma grande flexibilidade e mobilidade, tanto para a sua infraestrutura quanto para a sua abrangência. Dessa forma, tem-se um espaço inteligente que pode ser moldado para atender diversos cenários com diferentes requisitos, desde um espaço abrangido por uma sala ou andar de um prédio até o espaço monitorado em uma cidade.

Por fim, espera-se que este trabalho possa contribuir para o avanço das pesquisas na plataforma do PIS e na área de computação em borda. Ao possibilitar atender os requisitos das aplicações com utilização racional dos recursos de infraestrutura, espera-se que este trabalho possa ajudar na implantação de aplicações em cidades inteligentes visando otimizar o funcionamento da cidade, melhorar a mobilidade urbana, a segurança e a qualidade de vida dos cidadãos.

5.2 Trabalhos futuros

O controlador proposto nesse trabalho utiliza técnicas bem simples para configurar o modo de processamento, em nuvem ou em borda. Também não considera cenários mais complexos, com um conjunto maior de serviços a serem controlados. Nesse sentido, um sugestão de trabalho futuro seria testar o controlador em cenários mais complexos e adaptá-lo conforme a necessidade. É possível que, com o desenvolvimento em cenários mais complexos, seja necessário a criação de outros operadores Kubernetes para automatização de outras tarefas no PIS.

Adicionalmente, é importante observar que este trabalho não abordou a situação em que ocorre a perda total de conexão com a nuvem e como o sistema deveria reagir nesse cenário. A premissa adotada foi a existência contínua de conexão com o ambiente de computação em nuvem, sob diferentes condições de atraso de comunicação. Portanto, em trabalhos futuros, seria pertinente considerar e implementar melhorias no controlador para lidar com a ausência temporária ou completa de conexão com a nuvem.

Como o foco deste trabalho estava na expansão da infraestrutura do PIS e de sua abrangência, os estudos de casos apenas simulam cenários reais. Uma outra proposta de trabalho, seria realizar a integração de VANTs ou robôs no PIS e desenvolver aplicações com eles. Isso permitiria verificar como as ações realizadas pelo controlador, com o objetivo de minimização do tempo de resposta, levariam a uma maior eficiência em certos cenários. Considere, por exemplo, que deseja-se seguir um alvo móvel com um VANT. Em baixas taxas de imagens por segundo mais facilmente perde-se o rastreamento do alvo móvel, quando este se locomover mais rápido que o VANT. Assim, minimizando o tempo de res-

posta e maximizando a taxa de imagens por segundo, dificilmente perderia-se o alvo móvel. Nesse sentido, tal proposta de trabalho poderia avaliar estes aspectos em experimentos reais mostrando como a ação do controlador leva a resultados melhores.

Por último, ao integrar VANTs ou robôs no PIS, seria possível avaliar a escalabilidade do sistema. Poderia-se aumentar a carga no ambiente de computação em borda por meio da introdução de mais serviços para processamento de informações em borda ou incorporar mais dispositivos robóticos.

REFERÊNCIAS

ALMONFREY, D.; CARMO, A. P. do; QUEIROZ, F. M. de; PICORETI, R.; VASSALLO, R. F.; SALLES, E. O. T. A flexible human detection service suitable for intelligent spaces based on a multi-camera network. International Journal of Distributed Sensor Networks, v. 14, n. 3, 2018. Disponível em: <https://doi.org/10.1177/1550147718763550>. Citado na página 13.

BERNERS-LEE, T.; FIELDING, R.; FRYSTYK, H. RFC 1945: Hypertext transfer protocol - HTTP/1.0. Internet Engineering Task Force, 1996. Disponível em: <https://www.rfc-editor.org/rfc/rfc1945.html>. Citado na página 26.

BOSCH, J. Design and use of software architectures. In: . [S.l.: s.n.], 1999. p. 404–404. ISBN 0-7695-0275-x. Citado na página 20.

CARMO, A. P. do. Uma Arquitetura de Microsserviços centrada na Observabilidade Multinível para Espaços Inteligentes baseados em Visão Computacional. Tese (Doutorado) — UFES, 2021. Disponível em: <https://engenhariaeletrica.ufes.br/pt-br/pos-graduacao/PPGEE/detalhes-da-tese?id=14968>. Citado 4 vezes nas páginas 12, 13, 24 e 25.

COTTA, W. A. A. Medição de tempo de comunicação e exibição de tempo de resposta para Espaços Inteligentes Programáveis. Dissertação (Mestrado) — UFES, 2020. Disponível em: <https://engenhariaeletrica.ufes.br/pt-br/pos-graduacao/PPGEE/detalhes-da-tese?id=13368>. Citado 5 vezes nas páginas 12, 30, 36, 37 e 39.

COUTINHO, A.; CARNEIRO, E. O.; GREVE, F. G. P. Computação em névoa: Conceitos, aplicações e desafios. Minicursos do XXXIV SBRC, p. 266–315, 2016. Citado na página 19.

CUSTODIO, P.; QUEIROZ, F. de; ALMONFREY, D.; COTTA, W.; CARMO, A. do; VASSALLO, R. Proposta de um ambiente interacional baseado em um espaço inteligente programável. In: Anais do XII Simpósio Brasileiro de Computação Ubíqua e Pervasiva. Porto Alegre, RS, Brasil: SBC, 2020. p. 181–190. ISSN 2595-6183. Disponível em: <https://sol.sbc.org.br/index.php/sbcup/article/view/11224>. Citado na página 13.

DRAGONI, N.; GIALLORENZO, S.; LAFUENTE, A. L.; MAZZARA, M.; MONTESI, F.; MUSTAFIN, R.; SAFINA, L. Microservices: yesterday, today, and tomorrow. Present and ulterior software engineering, Springer, p. 195–216, 2017. Citado na página 20.

EDER, M. Hypervisor- vs. container-based virtualization. Chair for Network Architectures and Services, Department of Computer Science, Technische Universität München, 2016. Disponível em: <http://www.net.in.tum.de/fileadmin/TUM/NET/NET-2016-07-1/NET-2016-07-1_01.pdf>. Citado na página 17.

EUGSTER, P.; FELBER, P.; GUERRAOUI, R.; KERMARREC, A.-M. The many faces of publish/subscribe. ACM Comput. Surv., v. 35, p. 114–131, 06 2003. Citado na página 27.

GOOGLE. Kubernetes: an open-source system for automating deployment, scaling, and management of containerized applications. 2023. Disponível em: <https://kubernetes.io>. Acesso em: 10 mar. 2023. Citado 2 vezes nas páginas 21 e 22.

GRAFANA LOKI. Grafana Loki OSS: log aggregation system. 2023. Disponível em: <https://grafana.com/oss/loki/>. Acesso em: 09 out. 2023. Citado na página 29.

IZABEL, V. H.; OLIVEIRA, M. D. de; MENINES, M.; VASSALLO, R. F.; SILVA, L.; CARMO, A. P. do. Implantação do sistema mobilysa em espaços inteligentes programáveis. In: Anais do XIV Simpósio Brasileiro de Computação Ubíqua e Pervasiva. Porto Alegre, RS, Brasil: SBC, 2022. p. 61–70. ISSN 2595-6183. Disponível em: <https://sol.sbc.org.br/index.php/sbcup/article/view/20611>. Citado na página 13.

JAEGER. Jaeger: open source, distributed tracing platform. 2023. Disponível em: <https://www.jaegertracing.io/>. Acesso em: 09 out. 2023. Citado na página 30.

JAIN, N.; CHOUDHARY, S. Overview of virtualization in cloud computing. In: 2016 Symposium on Colossal Data Analysis and Networking (CDAN). Indore, India: IEEE, 2016. p. 1–4. Citado na página 17.

JOCHER, G.; CHAURASIA, A.; QIU, J. YOLO by Ultralytics. 2023. Disponível em: <https://github.com/ultralytics/ultralytics>. Citado na página 35.

KHAN, W. Z.; AHMED, E.; HAKAK, S.; YAQOOB, I.; AHMED, A. Edge computing: A survey. Future Generation Computer Systems, v. 97, p. 219–235, 2019. ISSN 0167-739X. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167739X18319903>. Citado na página 19.

LEE, J.-H.; HASHIMOTO, H. Intelligent space—concept and contents. Advanced Robotics, Taylor & Francis, v. 16, n. 3, p. 265–280, 2002. Citado 2 vezes nas páginas 12 e 24.

MOHAMED, N.; AL-JAROODI, J.; JAWHAR, I.; NOURA, H.; MAHMOUD, S. UAVfog: A UAV-based fog computing for internet of things. In: 2017 IEEE SmartWorld, Ubiquitous Intelligence and Computing, Advanced and Trusted Computed, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI). [S.l.: s.n.], 2017. p. 1–8. Citado 2 vezes nas páginas 31 e 32.

MOHAMMED, C. M.; ZEEBAREE, S. R. et al. Sufficient comparison among cloud computing services: Iaas, paas, and saas: A review. International Journal of Science and Business, IJSAB International, v. 5, n. 2, p. 17–30, 2021. Citado na página 18.

NGUYEN, T.-T.; YEOM, Y.-J.; KIM, T.; PARK, D.-H.; KIM, S. Horizontal pod autoscaling in kubernetes for elastic container orchestration. Sensors, v. 20, n. 16, 2020. ISSN 1424-8220. Disponível em: <https://www.mdpi.com/1424-8220/20/16/4621>. Citado na página 29.

PICORETI, R.; CARMO, A. P. do; QUEIROZ, F. M. de; GARCIA, A. S.; VASSALLO, R. F.; SIMEONIDOU, D. Multilevel observability in cloud orchestration. In: 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology

Congress(DASC/PiCom/DataCom/CyberSciTech). IEEE, 2018. p. 776–784. Disponível em: <https://doi.org/10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00134>. Citado 5 vezes nas páginas 28, 29, 31, 32 e 39.

POTDAR, A.; NARAYAN, D.; KENGOND, S.; MULLA, M. Performance evaluation of docker container and virtual machine. Procedia Computer Science, v. 171, p. 1419–1428, 01 2020. Citado na página 18.

PROMETHEUS. Prometheus: monitoring system and time series database. 2023. Disponível em: <https://prometheus.io/>. Acesso em: 09 out. 2023. Citado na página 29.

QUEIROZ, F. M. de. Desenvolvimento da Infraestrutura de um Espaço Inteligente baseado em Visão Computacional e IoT. Monografia (Graduação) — UFES, Vitória, 2016. Citado 2 vezes nas páginas 27 e 28.

QUEIROZ, F. M. de; PICORETI, R.; CANUTO, C.; RAMPINELLI, M.; VASSALLO, R. F. Estimating tridimensional coordinates of skeleton joints in a multicamera system. In: Anais do XIV Workshop de Visão Computacional. Ilhéus, BA, Brasil: SBC, 2018. p. 108–115. Citado na página 13.

RABBITMQ. Messaging that just works. 2023. Disponível em: <https://www.rabbitmq.com/>. Acesso em: 10 mar. 2023. Citado na página 27.

SANCHEZ-AGUERO, V.; GONZALEZ, L. F.; VALERA, F.; VIDAL, I.; SILVA, R. A. López da. Cellular and virtualization technologies for UAVs: An experimental perspective. Sensors, v. 21, n. 9, 2021. ISSN 1424-8220. Disponível em: <https://www.mdpi.com/1424-8220/21/9/3093>. Citado 2 vezes nas páginas 31 e 32.

SANTORO, D.; ZOZIN, D.; PIZZOLLI, D.; PELLEGRINI, F. D.; CRETTI, S. Foggy: A platform for workload orchestration in a fog computing environment. In: 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). Hong Kong, China: IEEE, 2017. p. 231–234. Citado 2 vezes nas páginas 30 e 32.

SATYANARAYANAN, M. Pervasive computing: vision and challenges. IEEE personal communications, IEEE, NEW YORK, v. 8, n. 4, p. 10–17, 2001. ISSN 1070-9916. Citado na página 12.

SHARMA, A.; KUMAR, M.; AGARWAL, S. A complete survey on software architectural styles and patterns. Procedia Computer Science, v. 70, p. 16–28, 2015. ISSN 1877-0509. Proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems. Disponível em: <https://www.sciencedirect.com/science/article/pii/S187705091503183X>. Citado na página 20.

SIGELMAN, B. H.; BARROSO, L. A.; BURROWS, M.; STEPHENSON, P.; PLAKAL, M.; BEAVER, D.; JASPAN, S.; SHANBHAG, C. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. [S.l.], 2010. Disponível em: <https://research.google.com/archive/papers/dapper-2010-1.pdf>. Citado na página 30.

TZANETTIS, I.; ANDRONA, C.-M.; ZAFEIROPOULOS, A.; FOTOPOULOU, E.; PAPAVASSILIOU, S. Data fusion of observability signals for assisting orchestration of distributed applications. Sensors, v. 22, n. 5, 2022. ISSN 1424-8220. Disponível em: <https://www.mdpi.com/1424-8220/22/5/2061>. Citado na página 32.

VILLAMIZAR, M.; GARCéS, O.; CASTRO, H.; VERANO, M.; SALAMANCA, L.; CASALLAS, R.; GIL, S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: 2015 10th Computing Colombian Conference (10CCC). [S.l.: s.n.], 2015. p. 583–590. Citado 2 vezes nas páginas 19 e 20.

WEISER, M. The computer for the 21 st century. Scientific american, JSTOR, v. 265, n. 3, p. 94–105, 1991. Citado na página 12.

ZIPKIN. Zipkin: a distributed tracing system. 2023. Disponível em: <https://zipkin.io/>. Acesso em: 10 mar. 2023. Citado na página 30.

Apêndices

.1 Definição de recurso personalizado

A definição de recurso personalizado para o recurso *IsController* pode ser visualizada abaixo.

Código 1 – Definição de recurso personalizado para o recurso *IsController*.

```
1 apiVersion: apiextensions.k8s.io/v1
2 kind: CustomResourceDefinition
3 metadata:
4   name: iscontrollers.labvisio.ufes.br
5 spec:
6   scope: Namespaced
7   group: labvisio.ufes.br
8   names:
9     kind: IsController
10    plural: iscontrollers
11    singular: iscontroller
12    shortNames:
13      - ic
14      - ics
15  versions:
16    - name: v1
17      served: true
18      storage: true
19      schema:
20        openAPIV3Schema:
21          type: object
22          properties:
23            spec:
24              type: object
25              properties:
26                cloud:
27                  type: object
28                  properties:
29                    deployment:
30                      type: object
31                      properties:
32                        name:
33                          type: string
34                        namespace:
35                          type: string
36                        apiVersion:
37                          type: string
38                      required:
39                      - name
40                      - namespace
41                      - apiVersion
42                  required:
43                  - deployment
44                edge:
45                  type: object
46                  properties:
47                    deployment:
```

```
48            type: object
49            properties:
50              name:
51                type: string
52              namespace:
53                type: string
54              apiVersion:
55                type: string
56            required:
57            - name
58            - namespace
59            - apiVersion
60          required:
61          - deployment
62        spans:
63          type: array
64          items:
65            type: object
66            properties:
67              name:
68                type: string
69              service:
70                type: string
71            required:
72            - name
73            - service
74        tracesTargetValue:
75          type: number
76        connections:
77          type: object
78          properties:
79            test:
80              type: array
81              items:
82                type: object
83                properties:
84                  name:
85                    type: string
86                  src:
87                    type: string
88                  dest:
89                    type: string
90                  topic:
91                    type: string
92                required:
93                - name
94                - src
95                - dest
96                - topic
97            stable:
98              type: array
99              items:
100               type: object
101               properties:
102                 name:
103                   type: string
```

```yaml
104                           src:
105                             type: string
106                           dest:
107                             type: string
108                           topic:
109                             type: string
110                         required:
111                         - name
112                         - src
113                         - dest
114                         - topic
115                   required:
116                   - test
117                   - stable
118               required:
119               - edge
120               - cloud
121               - spans
122               - tracesTargetValue
123           required:
124           - spec
```