INSTITUTO FEDERAL DO ESPÍRITO SANTO
BACHARELADO ENGENHARIA ELÉTRICA

VITOR BERMOND SOUZA

DESENVOLVIMENTO DE UMA APLICAÇÃO PARA MONITORAMENTO DE RECURSOS EM UM ESPAÇO INTELIGENTE PROGRAMÁVEL

Guarapari
2025

VITOR BERMOND SOUZA

DESENVOLVIMENTO DE UMA APLICAÇÃO PARA MONITORAMENTO DE RECURSOS EM UM ESPAÇO INTELIGENTE PROGRAMÁVEL

Trabalho de Conclusão de Curso apresentado à Coordenadoria do Curso de Engenharia Elétrica do Instituto Federal de Educação, Ciência e Tecnologia do Espírito Santo, como requisito parcial para a obtenção do título de Engenheiro Eletricista.

Orientador: Prof. Dr. Alexandre Pereira do Carmo

Guarapari

2025

Dados Internacionais de Catalogação na Publicação (CIP)
Instituto Federal do Espírito Santo – *Campus* Guarapari

S729d Souza, Vitor Bermond.
Desenvolvimento de uma aplicação para monitoramento de recursos em um espaço inteligente programável / Vitor Bermond Souza. – 2025.
87 f. : il.

Orientador (a): Alexandre Pereira do Carmo.
Monografia (Graduação) – Instituto Federal do Espírito Santo, *Campus* Guarapari, Engenharia Elétrica, 2025.

1. Sistemas de controle inteligente. 2. Espaço inteligente programável. 3. Engenharia elétrica I. Carmo, Alexandre Pereira do. II. Instituto Federal do Espírito Santo. III. Título.

CDD: 621.3

Elaborado por Karla Matos Curto Valle CRB6-ES / 769

VITOR BERMOND SOUZA

DESENVOLVIMENTO DE UMA APLICAÇÃO PARA MONITORAMENTO DE RECURSOS EM UM ESPAÇO INTELIGENTE PROGRAMÁVEL

Trabalho de Conclusão de Curso apresentado à Coordenadoria do Curso de Engenharia Elétrica do Instituto Federal de Educação, Ciência e Tecnologia do Espírito Santo, como requisito parcial para a obtenção do título de Engenheiro Eletricista.

Aprovado em 30 de Junho de 2025

COMISSÃO EXAMINADORA

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

gov.br Documento assinado digitalmente
ALEXANDRE PEREIRA DO CARMO
Data: 17/12/2025 19:22:13-0300
Verifique em https://validar.iti.gov.br

Prof. Me. Pedro Paulo Pecolo Filho
Instituto Federal do Espírito Santo
Examinador

gov.br Documento assinado digitalmente
PEDRO PAULO PECOLO FILHO
Data: 18/12/2025 10:08:32-0300
Verifique em https://validar.iti.gov.br

gov.br Documento assinado digitalmente
WALBER ANTONIO RAMOS BELTRAME
Data: 18/12/2025 08:43:44-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Walber Antonio Ramos Beltrame
Instituto Federal do Espírito Santo
Examinador

AGRADECIMENTOS

Agradeço, primeiramente, à minha mãe, por ter sido a principal força e incentivo que me impulsionaram a iniciar e concluir a faculdade. Sou grato também pelas amizades construídas ao longo dessa jornada, que tornaram os desafios mais leves e os dias mais significativos.

“A vida só é dura pra quem é mole”

RESUMO

A disseminação dos conceitos de espaços e cidades inteligentes tem promovido avanços significativos, possibilitando a automação de tarefas e a realização de funções de difícil execução. Um espaço inteligente caracteriza-se por sua capacidade de observar o ambiente por meio de sensores, comunicar-se com entidades presentes e interagir com base em dados coletados. Contudo, à medida que esses ambientes se expandem, o gerenciamento de seus recursos e aplicações torna-se um desafio crescente. Atualmente é possível monitorar o impacto dos serviços implantados individualmente, mas não existe uma abstração de uma aplicação, que é a composição de vários serviços. Além disso toda gestão existente requer comandos específicos em terminais. Dessa forma, surge a necessidade de uma interface de monitoramento onde seja possível gerenciar aplicações individualmente de forma prática. Este projeto visa criar uma aplicação para monitoramento e gerenciamento de recursos em um Espaço Inteligente Programável (PIS) localizado no IFES - Campus Guarapari. O foco do projeto está no front-end, com ênfase na recepção, padronização e tratamento das informações para exibição adequada ao usuário final, auxiliando em tomadas de decisão dentro do espaço. Inicialmente, a interface será focada no monitoramento de uma aplicação específica, com possibilidade de expansão para outras no futuro. A implementação da interface pretende explorar tecnologias como Flutter, Grafana, Power BI, RabbitMQ, Banco de dados e Kubernetes, buscando oferecer uma solução eficiente e escalável. Este trabalho descreve as etapas do desenvolvimento dessa aplicação.

Palavras-chave: Espaço Inteligente. Espaço Inteligente Programável (PIS). Front-End. Monitoramento. Flutter. RabbitMQ

ABSTRACT

The concepts of intelligent spaces and intelligent cities has promoted significant advances, enabling the automation of tasks and the performance of difficult-to-execute functions. A intelligent space is characterized by its ability to observe the environment through sensors, communicate with present entities and interact based on collected data. However, as these environments expand, the management of their resources and applications becomes an increasing challenge. It is currently possible to monitor the impact of services implemented individually, but there is no abstraction of an application, which is the composition of several services. In addition, all existing management requires specific commands in terminals. Thus, the need arises for a monitoring interface where it is possible to manage applications individually in a practical way. This project aims to create an application for monitoring and managing resources in a Programmable Intelligent Space (PIS) located at IFES - Guarapari Campus. The focus of the project is to develop the front-end, with an emphasis on receiving, standardizing and processing information for appropriate display to the end user, assisting in decision-making within the space. Initially, the interface will focus on monitoring a specific application, with the possibility of expanding to others in the future. The implementation of the interface aims to explore technologies such as Flutter, Grafana, Power BI, RabbitMQ, Database and Kubernetes, seeking to offer an efficient and scalable solution. This work describes the steps of the development of this application.

Keywords: Smart Space. Programmable Intelligent Space (PIS). Front-End. Monitoring. Flutter. RabbitMQ

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

IFES	Instituto Federal do Espírito Santo
PIS	Programmable Intelligent Space
IS	Intelligent Space
CPU	Central Processing Unit
GPU	Graphics Processing Unit
TP	Tempo de Processamento
ET	Erro de Trajetória
NI	Número de Instâncias
FPS	Frames Per Second
SQL	Structured Query Language
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
JSON	JavaScript Object Notation
CSV	Comma-Separated Values
UI	User Interface
HMI	Human-Machine Interface
HTTP	Hypertext Transfer Protocol

SUMÁRIO

1 INTRODUÇÃO

Espaços inteligentes têm ganhado popularidade devido à sua utilidade e à relativa facilidade de implantação. Esse conceito se estende até mesmo às cidades inteligentes, que podem ser vistas como uma ampliação em escala dos espaços inteligentes. A principal diferença entre um espaço inteligente e um espaço comum está na presença de múltiplos sensores, atuadores e serviços computacionais que interagem diretamente com as pessoas que nele estão inseridas. Esses sensores e atuadores variam conforme a aplicação, indo desde dispositivos de detecção de luz, que permitem o acionamento automático de postes durante a noite, até câmeras utilizadas para videomonitoramento (CARMO, 2021).

O presente projeto tem como objeto de estudo o Espaço Inteligente Programável (PIS) do IFES - Campus Guarapari, que já conta com diversas aplicações de videomonitoramento, baseadas em câmeras distribuídas pelo espaço. Esse tipo de aplicação exige elevado poder de processamento, uma vez que os serviços associados realizam o processamento contínuo de imagens (CARMO, 2021). Contudo, os recursos computacionais disponíveis no espaço são finitos, e o gerenciamento eficiente dessa limitação torna-se um desafio significativo. O PIS utiliza tecnologias modernas que permitem a divisão e o compartilhamento de recursos, como o uso de GPUs para diferentes serviços. Porém, a operação e o monitoramento desses recursos não são intuitivos, especialmente para novos desenvolvedores ou usuários menos experientes (CARMO, 2021).

Uma aplicação, no contexto do PIS, pode ser entendida como um conjunto interligado de serviços computacionais que trabalham em conjunto para atingir um objetivo específico. Atualmente, a gestão de recursos no PIS é realizada de forma fragmentada, considerando os serviços de forma individual, sem uma abstração que represente a aplicação como um todo. Essa abordagem dificulta o monitoramento e o gerenciamento, especialmente quando é necessário identificar falhas ou otimizar recursos para atender aos requisitos de desempenho de uma aplicação completa. Assim, torna-se evidente a necessidade de uma solução que facilite a visualização e o acompanhamento dos recursos utilizados por aplicações no PIS, promovendo uma visão unificada e simplificada

desse ambiente complexo.

Espaços inteligentes têm como objetivo principal facilitar a interação entre a tecnologia e as pessoas. Entretanto, novos desenvolvedores que se deparam com o PIS podem sentir-se desorientados, considerando a quantidade de conceitos necessários para compreender o funcionamento e o desenvolvimento dentro desse espaço. Além disso, usuários que não estão familiarizados com o uso de comandos no terminal podem enfrentar dificuldades consideráveis na interação com o sistema.

Com base nesse contexto, a solução proposta é o desenvolvimento de uma interface de monitoramento de recursos utilizados por uma aplicação. Interfaces gráficas são mais intuitivas e acessíveis do que linhas de comando, possibilitando que qualquer usuário, técnico ou não, consiga visualizar o estado dos recursos e da saúde do espaço. Essa interface poderá auxiliar na tomada de decisão em alocação de recursos para aplicações e no diagnóstico de falhas ou mau funcionamento, tornando o PIS mais eficiente e acessível para diferentes perfis de usuários.

Neste projeto, serão descritas as pesquisas realizadas para o desenvolvimento dessa interface, abordando desde as possíveis ferramentas que podem ser utilizadas para a sua construção até as diferentes formas de comunicação possíveis dentro do espaço inteligente.

1.1 JUSTIFICATIVA

Toda a pesquisa realizada neste projeto tem como objeto de estudo o PIS do IFES - Campus Guarapari, onde já existem diversas aplicações e microsserviços voltados para visão computacional. Esse tipo de aplicação geralmente requer grande poder computacional para o processamento das imagens coletadas no ambiente. Dado que os recursos do PIS são limitados, torna-se crucial monitorá-los de forma eficiente em tempo real, tanto para preservar sua saúde quanto para obter um feedback claro sobre o impacto de cada aplicação em funcionamento sobre os recursos do PIS e, consequentemente, sobre o desempenho de outras aplicações e da própria infraestrutura do espaço.

Um exemplo de como a disponibilidade de recursos pode impactar o funcionamento de uma aplicação pode ser observado em um serviço que realiza o controle de um robô através de imagens. Uma alta taxa de FPS (Frames Per Second) pode ser essencial para um controle adequado, mas isso pode ser bastante exigente para o PIS, pois um FPS mais elevado exige o processamento de um maior número de imagens em um intervalo de tempo reduzido. Caso o tempo de resposta ultrapasse um limite predefinido, a funcionalidade da aplicação pode ser comprometida. Dessa forma, surge a necessidade de um sistema onde seja possível monitorar o desempenho e uso de recursos de uma aplicação por inteiro, incluindo cada serviço que a compõe, desde a coleta até o processamento das imagens. É de grande importância coletar os dados de desempenho e de comunicação entre os serviços que compõem a aplicação.

Com uma visão clara destes dados, será possível tomar melhores decisões quanto à alocação de recursos e identificação de problemas. Uma das premissas de um PIS é a capacidade de ter uma observação multinível, o que reforça a motivação para a elaboração deste projeto, que visa facilitar o monitoramento desses recursos.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

O objetivo deste projeto é desenvolver uma interface de monitoramento para análise do uso de recursos de uma aplicação de domínio dentro do PIS do IFES - Campus Guarapari, visando facilitar a tomada de decisões.

1.2.2 Objetivos Específicos

Para alcançar o objetivo geral, alguns objetivos específicos devem ser atingidos:

- Extrair metricas de desempenho dos serviços da aplicação alvo.
- Transmitir as métricas para a interface de monitoramento.
- Desenvolver a interface de monitoramento, de forma a exibir os dados eficientemente.

- Containerizar a aplicação de monitoramento utilizando Docker e integrá-la ao PIS.

2 REVISÃO BIBLIOGRÁFICA/REFERENCIAL TEÓRICO

Nesta seção, serão discutidos assuntos que servirão como base teórica para o desenvolvimento deste trabalho. Os temas abordados na pesquisa incluem: espaço inteligente, espaço inteligente programável, interfaces gráficas, ferramentas para desenvolvimento da interface e meios e padrões de comunicação.

2.1 ESPAÇO INTELIGENTE (IS)

A premissa de um espaço inteligente surge do conceito de tecnologias inteligentes ou do inglês “smart”. Dispositivos inteligentes podem realizar funções e atividades rotineiras de forma autônoma, sem que isso necessariamente envolva o uso de inteligência artificial. Isso é possível graças à conexão com a internet, permitindo que realizem tarefas simples e se comuniquem sem intervenção humana (Positivo Casa Inteligente, 2024).

Para que dispositivos inteligentes possam desempenhar sua função de forma autônoma é necessário que haja alguma instrumentação que o possibilite observar o seu alvo ou ambiente em que ele está inserido. Essa instrumentação normalmente ocorre através de sensores que podem ser os mais variados possíveis a depender do dispositivo. E além disso, para que tudo funcione corretamente é necessário uma conexão e uma infraestrutura que realize e padronize a comunicação entre os dispositivos, sensores e pessoas. Dessa necessidade surgem os espaços inteligentes.

> Um espaço inteligente pode ser definido como um espaço físico equipado com uma rede de sensores, que obtém informações sobre o mundo que observa, e uma rede de atuadores, que permite sua interação com os usuários e a modificação do próprio ambiente através de serviços de computação. Dessa forma, sensores, atuadores e serviços de computação são governados por uma infraestrutura capaz de coletar e analisar as informações obtidas pelos sensores e tomar decisões. (COTTA, 2020, p. 22)

Um espaço inteligente deve ser capaz de observar o meio onde está inserido através de sensores, conseguir se comunicar com as entidades nele presentes e interagir com base nas variáveis observadas.

Em ambientes internos o conceito de espaço inteligente já é explorado. Sensores de movimento e câmeras inteligentes são amplamente utilizados para otimizar o uso de recursos energéticos, ajustando automaticamente a iluminação e o ar-condicionado conforme a ocupação dos espaços. Esse tipo de sistema contribui para a economia de energia e melhora o conforto dos ocupantes (ZLATANOVA *et al.*, 2020).

Em ambientes externos, o conceito de cidades inteligentes se desdobra, representando espaços inteligentes em uma escala ampliada. Essas cidades incorporam tecnologias avançadas e soluções integradas para melhorar a qualidade de vida dos cidadãos, a eficiência dos serviços urbanos e a sustentabilidade ambiental. Elas utilizam redes de sensores e sistemas de informação para monitorar e gerenciar recursos como água, energia e tráfego, promovendo um uso mais racional e eficiente do espaço urbano.

Em várias cidades, postes de iluminação estão sendo equipados com sensores de presença e câmeras, ajustando automaticamente a intensidade da luz conforme a necessidade, reduzindo o consumo de energia (ZLATANOVA *et al.*, 2020).

Em (JHA *et al.*, 2022) sensores espalhados por áreas urbanas monitoram a qualidade do ar, níveis de poluição e condições climáticas, fornecendo dados em tempo real que podem ser usados para intervenções na saúde pública e melhorias ambientais

2.2 ESPAÇO INTELIGENTE PROGRAMÁVEL (PIS)

Existe uma classe de espaços inteligentes chamada espaço inteligente programável (do inglês Programmable Intelligent Space - PIS). Enquanto um espaço inteligente tradicional busca observar o ambiente e tomar decisões para atender às necessidades dos usuários, um PIS introduz a capacidade de orquestração multinível centrada na observabilidade e programabilidade granular da infraestrutura, permitindo ajustes dinâmicos e um uso mais racional dos recursos (CARMO, 2021).

Essas novas características surgem a partir da construção da infraestrutura do PIS que é feita a partir de microsserviços e aplicações independentes e isoladas. Essa abordagem traz a vantagem da facilidade de implementar novas funções para o espaço através dos serviços, uma vez que os serviços estarão isolados e não será necessário

se preocupar com a infraestrutura como um todo. Geralmente os serviços que são desenvolvidos para um PIS são construídos de forma que possam ser reaproveitados em outras aplicações e outros PIS em locais diferentes, seguindo um padrão de construção e comunicação (CARMO, 2021).

O padrão de construção é feito com o uso de Docker para armazenar os serviços. O Docker é a ferramenta padrão para implementar uma aplicação ou serviço usando containers. Em outras palavras, o Docker se baseia no formato mais popular para empacotar serviços e é a container engine mais usada atualmente. A grande vantagem de um container é encapsular todas as dependências necessárias para o funcionamento do script, como bibliotecas, o runtime e o código da aplicação. Tudo isso em um único pacote chamado de imagem, que pode ser versionado e fácil de distribuir (Alura, 2024).

Os serviços podem ser divididos em duas classes:

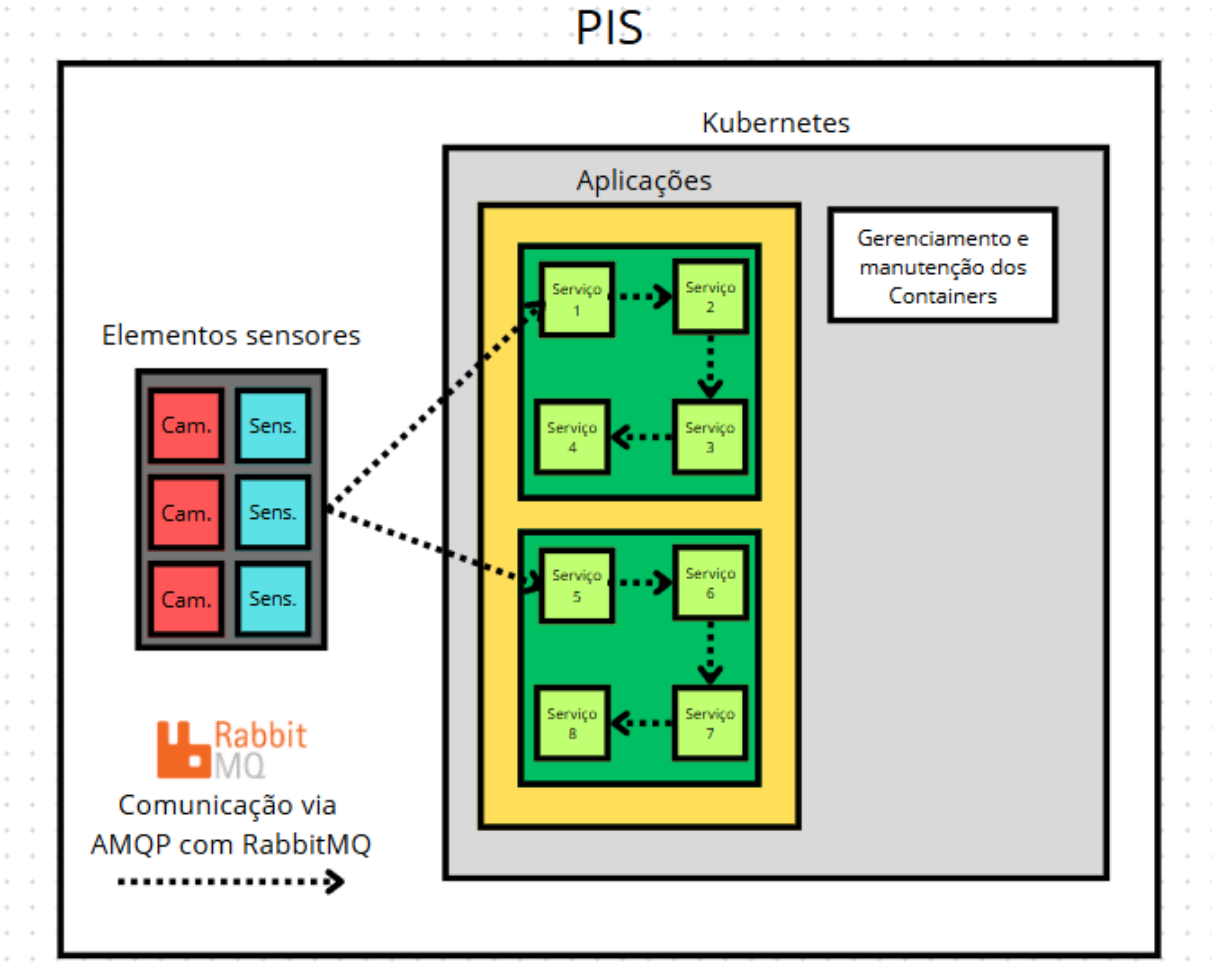
- **Serviços de infraestrutura:** Os serviços de infraestrutura envolvem a manutenção da conectividade, segurança e comunicação entre os dispositivos, mantendo a estrutura do PIS.
- **Serviços de Domínio:** Os serviços de domínio lidam com a lógica central e interações específicas com o ambiente, como controle de um robô, iluminação inteligente ou tratamento de imagens.

A programabilidade granular oferecida pelo PIS permite que o espaço seja escalado quase indefinidamente, sendo a única limitação real o desempenho e a capacidade do hardware subjacente. Essa flexibilidade é uma grande vantagem em relação aos espaços inteligentes tradicionais, cuja arquitetura é mais rígida e enraizada, tornando a implementação de novas funcionalidades ou serviços um processo complexo e trabalhoso. No PIS, por outro lado, a modularidade e a capacidade de orquestração facilitam a adição de novos microsserviços de forma mais simplificada, permitindo que o sistema evolua conforme novas demandas surjam (CARMO, 2021; COTTA, 2020).

As características que tornam o PIS diferente de um espaço comum são proporcionadas principalmente pelo uso do Kubernetes, um orquestrador de containers que automatiza

a implantação, o gerenciamento e a escalabilidade das aplicações. Ele permite a orquestração multinível anteriormente citada, gerindo containers onde os serviços da aplicação são executados. Kubernetes organiza e isola os serviços, enquanto facilita a comunicação entre eles. Ele monitora continuamente as aplicações e, ao detectar uma falha em algum serviço, substitui-o automaticamente, garantindo a continuidade da operação e a alta disponibilidade de cada componente essencial (Alura, 2024). A figura 1 demonstra de forma simplificada a estrutura do PIS.

Figura 1 – Representação da funcionalidade do PIS e suas entidades.

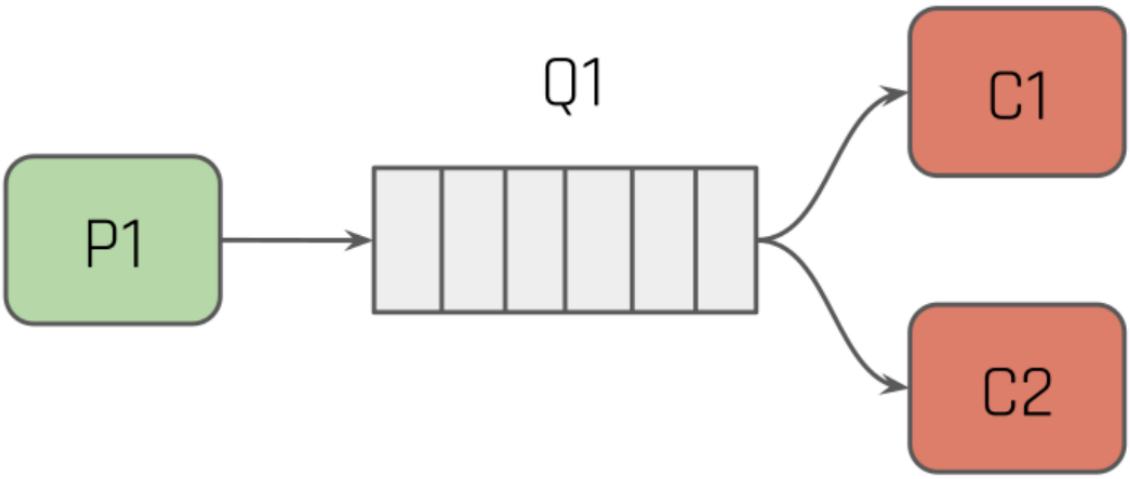


Fonte: Elaborado pelo autor.

O protocolo Advanced Message Queuing Protocol (AMQP) é o padrão escolhido para a comunicação dentro do PIS, e junto a ele é utilizado o broker RabbitMQ. Essa entidade é responsável por lidar com a camada de comunicação do PIS. A função do broker é receber e encaminhar mensagens. Para isso, o RabbitMQ implementa filas onde essas mensagens ficam armazenadas para que possam posteriormente consumidas

e lidas pelos destinatários, conforme a Figura 2. Essa fila pode ser configurada de várias formas para atender as diferentes necessidades dos serviços dentro do PIS. Ela pode ser configurada para ser persistente e ter tamanho fixo, por exemplo, nesse caso se uma mensagem for entregue à fila e ninguém consumi-la, ela ficará lá até que o próximo consumidor se conecte e consuma essa mensagem (CARMO, 2021).

Figura 2 – Representação das filas do RabbitMQ.



Fonte: (QUEIROZ, 2016).

2.3 INTERFACES GRÁFICAS

A digitalização crescente do mundo moderno gera uma enorme quantidade de dados continuamente. Esse volume elevado de informações pode dificultar a análise eficiente e, como consequência, criar obstáculos na tomada de decisões. Nesse cenário, o uso de gráficos de séries temporais e alarmes se destaca como uma solução eficaz para simplificar a interpretação dos dados.

Em ambientes computacionais complexos, os dados muitas vezes estão distribuídos entre diferentes aplicações e serviços, o que pode complicar ainda mais o processo de análise. As interfaces gráficas surgem como uma ferramenta essencial para integrar esses dados e facilitar sua visualização de forma clara. Elas são especialmente valiosas em operações críticas, onde informações precisas e acessíveis podem ser determinantes para garantir segurança e eficiência. Além disso, um bom design de interface pode aumentar a produtividade dos usuários em 25% a 40% (GALITZ, 2007).

Essas interfaces permitem a criação de dashboards, que agrupam os dados mais relevantes em um formato amigável ao usuário, mesmo que ele não tenha um conhecimento técnico profundo do sistema. Isso possibilita uma comunicação clara entre o homem e a máquina, essencial para operações que demandam rápida compreensão do que está ocorrendo.

Estudos sobre interfaces gráficas destacam que elas ajudam significativamente usuários leigos a interagir com sistemas de maneira mais eficiente. Isso ocorre porque as interfaces simplificam a apresentação de informações, eliminando a necessidade de memorizar comandos complexos, o que é comum em interfaces de linha de comando. As interfaces gráficas utilizam metáforas visuais e elementos interativos que imitam objetos do mundo físico, facilitando a compreensão do usuário e a navegação intuitiva pelos sistemas (HUNT, 2023).

Na indústria, as interfaces gráficas desempenham um papel crucial no monitoramento e controle de ativos por meio de sistemas supervisórios. Estes sistemas combinam elementos de hardware e software para permitir que os operadores controlem e monitorem processos industriais em tempo real, utilizando dados de sensores, válvulas, bombas, e outros equipamentos de campo. As HMI (Human-Machine Interface), por exemplo, permitem a coleta de dados, o controle de processos locais ou remotos, e a interação direta com dispositivos industriais através de uma interface gráfica (DEVASIA, 2020).

No contexto de um espaço inteligente, a visualização de determinadas variáveis pode ser interessante, devido à complexidade dos sistemas envolvidos. Num cenário onde os recursos computacionais são limitados, o seu monitoramento se torna importante. Nesse sentido, uma interface gráfica bem projetada torna-se uma solução interessante para proporcionar uma visão clara e acessível das informações importantes, otimizando a experiência do usuário e auxiliando na gestão eficiente dos recursos do PIS, visto que esse recurso ainda não foi implementado.

2.4 FERRAMENTAS PARA DESENVOLVIMENTO DA INTERFACE

Para o monitoramento é interessante a criação de diferentes telas com possibilidade de desenvolvimento de gráficos para ilustrar séries temporais para o usuário final poder ter uma noção do histórico dos dados para tomar uma decisão sensata, alarmes para indicar situações críticas que precisam de intervenções imediatas e um design simples que permita até que usuários leigos entendam o que está acontecendo.

Atualmente, algumas ferramentas se destacam como opções viáveis para o desenvolvimento da interface. Entre as principais, destacam-se o Flutter, o Grafana e o Power BI, que serão discutidas nesta seção.

2.4.1 Flutter

O Flutter é um framework de desenvolvimento multiplataforma criado pelo Google, projetado para facilitar a criação de interfaces que funcionam em diferentes plataformas a partir de um único código-fonte. Tecnicamente, o Flutter é um conjunto de bibliotecas pré-configuradas que simplifica o desenvolvimento de projetos, oferecendo componentes e ferramentas para criar interfaces visuais interativas e personalizáveis. Sua compatibilidade e praticidade têm atraído diversas empresas, incluindo o próprio Google, que o utiliza em vários de seus aplicativos, além de outras empresas como Nubank, Groupon, eBay e desenvolvedoras de jogos como a Supercell (Flutter, 2024).

No contexto deste projeto, o uso do Flutter pode ser vantajoso por permitir a criação de interfaces altamente customizáveis, proporcionando uma exibição adaptável das informações monitoradas. Ele garante a possibilidade de um alto nível de personalização para consolidar dados de forma clara para o usuário final, além de garantir a escalabilidade e expansibilidade da aplicação no futuro.

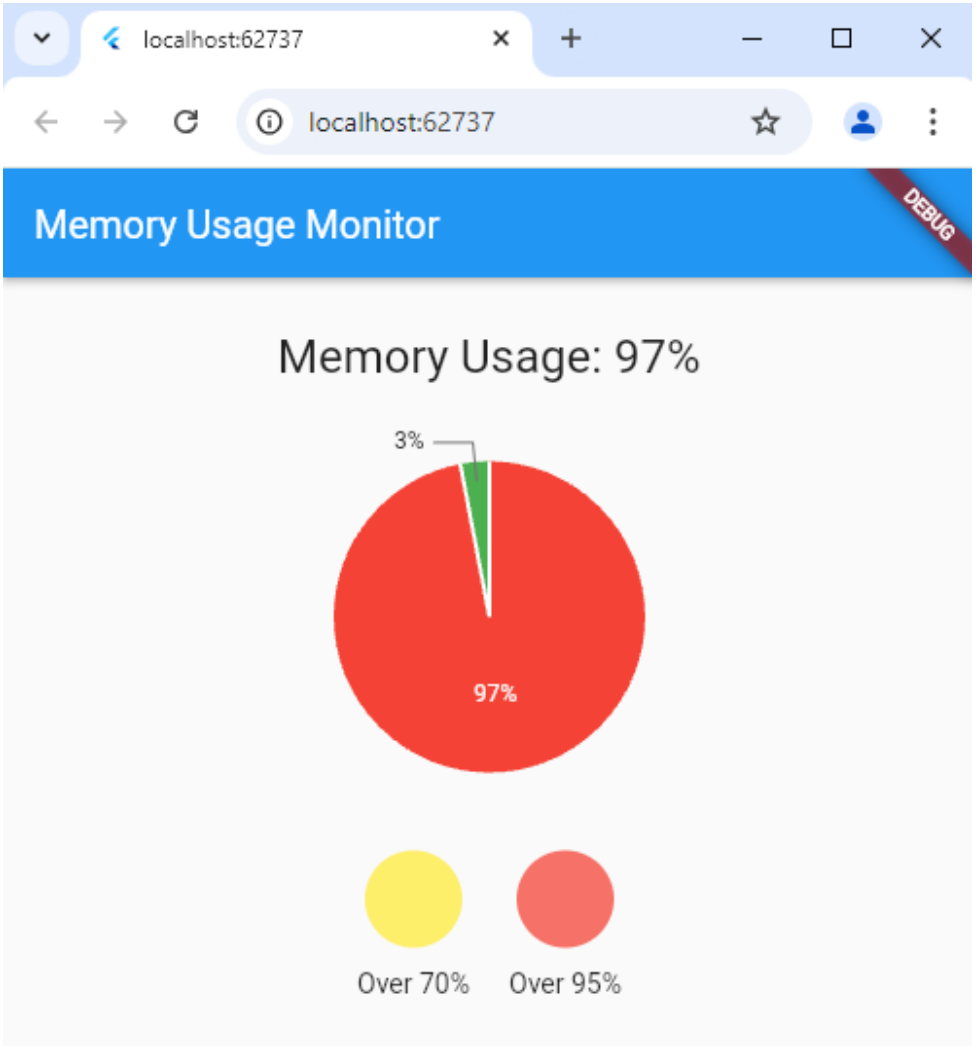
Além disso, o Flutter, em conjunto com sua linguagem base Dart, possui gama de bibliotecas úteis para o desenvolvimento do projeto, incluindo:

- **fl_charts**: Facilita a criação de gráficos para visualização de dados, essenciais para exibir dados de séries temporais e indicadores de forma visual.

- **dart_amqp**: Possibilita a comunicação com brokers RabbitMQ, amplamente utilizado no PIS para a comunicação entre serviços e aplicações.
- **cloud_firestore** e **sqflite**: Oferecem suporte para comunicação com bancos de dados, como Firebase e SQL, permitindo armazenamento e recuperação de dados de forma prática.

A Figura 3 exemplifica um gráfico desenvolvido que ilustra o uso de memória em um momento específico, com alarmes que indicam situações adversas.

Figura 3 – Gráfico de pizza elaborado com Flutter.



Fonte: Elaborado pelo autor.

No entanto, uma das desvantagens do Flutter é o uso do Dart como linguagem principal, a qual, embora eficiente, é menos conhecida e possui uma curva de aprendizado

mais acentuada devido à sua sintaxe própria. Essa característica pode impactar a familiarização de novos desenvolvedores com a plataforma.

Outra limitação é o tamanho final do aplicativo, que tende a ser um pouco maior em comparação a outras soluções, o que pode ser um fator importante em projetos que exigem otimização extrema de recursos.

2.4.2 Grafana

Grafana é uma plataforma de monitoramento e visualização de dados em tempo real, ideal para sistemas que requerem supervisão constante de métricas. Criado para facilitar a visualização de métricas por meio de dashboards altamente customizáveis, o Grafana permite que desenvolvedores e equipes monitorem o desempenho de seus sistemas e aplicações. Ele é projetado para transformar dados complexos em visualizações que facilitam a tomada de decisões em tempo real, com opções para personalizar cada visualização para que os dados sejam apresentados da forma mais relevante possível para cada uso específico (Grafana Labs, 2024).

No contexto deste projeto, o Grafana se destaca por ser compatível com o Prometheus, que já é utilizado no PIS do IFES - Campus Guarapari para a coleta de métricas em algumas aplicações. O Prometheus é um sistema de monitoramento para serviços e aplicações. Ele coleta as métricas de seus alvos em determinados intervalos, avalia expressões de regras, exibe os resultados e também pode acionar alertas se alguma condição for observada como verdadeira. Ele utiliza um modelo de dados multidimensional como uma série temporal identificados pelo nome da métrica e pares chave/valor (PROMETHEUS, 2020).

O Prometheus coleta dados de desempenho e os organiza em um formato de série temporal, e o Grafana, por sua vez, apresenta essas informações em dashboards, facilitando o monitoramento em tempo real. Essa integração permite monitorar uma ampla gama de métricas e adaptar os dashboards às necessidades de cada aplicação.

Além do Prometheus, o Grafana oferece suporte para diversas outras fontes de dados, como o MySQL, aumentando sua flexibilidade para consolidar informações de múltiplos

sistemas em uma única interface. A Figura 4 exemplifica um painel criado com Grafana em uso no IFES para monitoramento de métricas do PIS.

Figura 4 – Dashboard Grafana

Fonte: Elaborado pelo autor.

Por outro lado, o Grafana apresenta algumas limitações para o objetivo específico deste projeto. Embora seja excelente para visualização de dados e monitoramento, o Grafana não é adequado para desenvolvimento de interfaces interativas ou para aplicações que exijam um front-end altamente personalizável. Sua funcionalidade é otimizada para dashboards informativos, o que restringe seu uso para construção de interfaces complexas ou de alta interatividade.

Além disso, a configuração do Grafana geralmente exige uma infraestrutura de back-end

robusta, com ajustes específicos para conectar e armazenar dados de fontes externas. Mesmo com a utilização do Prometheus, seria necessário um esforço considerável de configuração para obter todas as funcionalidades desejadas para uma aplicação expansível e customizável, como a do projeto proposto.

2.4.3 Power BI

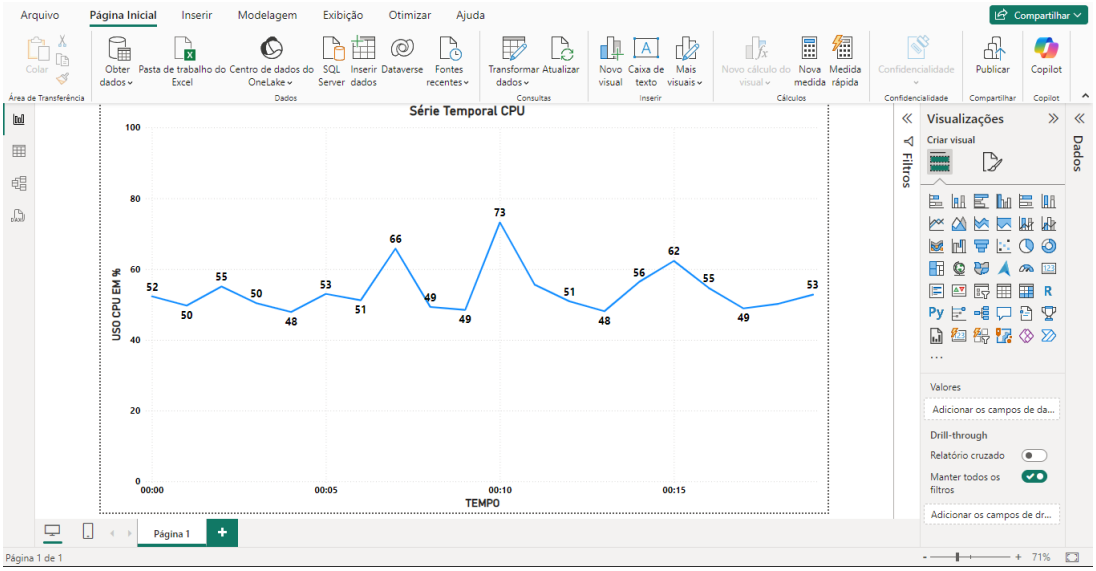
O Power BI é uma plataforma de análise de dados e criação de visualizações desenvolvida pela Microsoft, projetada para transformar dados complexos em insights visuais de fácil compreensão. Com uma interface intuitiva e personalizável, o Power BI permite a criação de relatórios interativos e dashboards que possibilitam a exploração de dados em tempo real. Esses dashboards são acessíveis em dispositivos móveis e desktops, permitindo que os usuários monitorem indicadores de desempenho e tomem decisões informadas de maneira ágil e eficaz (Microsoft Power BI Blog, 2024).

Uma das principais vantagens do Power BI é sua integração com o ecossistema Microsoft, incluindo ferramentas como Excel, SharePoint e Azure. Essa conectividade o torna uma escolha popular entre empresas que utilizam soluções Microsoft, facilitando a consolidação de dados provenientes de diferentes fontes em um ambiente de visualização centralizado. Além disso, o Power BI suporta conexões com múltiplos bancos de dados e serviços externos, como SQL Server e Google Analytics, possibilitando uma análise ampla e integrada dos dados (Microsoft Learn, 2024).

A Figura 5 ilustra um exemplo de gráfico criado com o Power BI para visualização de dados em séries temporais.

Embora o Power BI seja uma ferramenta poderosa para visualização e análise de dados, ele apresenta algumas limitações para o projeto proposto. Seu uso é vantajoso apenas se os dados a serem monitorados estiverem em bancos de dados com os quais o Power BI possa se comunicar diretamente, como o SQL Server ou planilhas Excel. No entanto, como o PIS utiliza AMQP como padrão de comunicação, o Power BI não oferece suporte direto a este protocolo, exigindo integrações externas para conectar-se a brokers de mensagens como o RabbitMQ, o que aumenta a complexidade do projeto.

Figura 5 – Gráfico de linha elaborado com Power BI.



Fonte: Elaborado pelo autor.

Além disso, o Power BI não pode ser containerizado com Docker, ao contrário de outras aplicações do PIS, o que pode comprometer a padronização da infraestrutura. Outra limitação é que muitos dos recursos avançados do Power BI estão disponíveis apenas na versão paga, o que pode gerar custos adicionais caso o uso dessas funcionalidades se tornem necessárias durante o desenvolvimento e a implementação do projeto.

2.4.4 Comparação

Para avaliar a melhor ferramenta para o desenvolvimento da interface de monitoramento do PIS, foram analisadas as principais vantagens e desvantagens de cada uma das opções mais viáveis: Flutter, Grafana e Power BI. Essas ferramentas se destacam por sua capacidade de visualização de dados, conectividade e personalização, mas apresentam limitações que podem impactar sua escolha para o projeto proposto. A Tabela 1 resume os principais pontos fortes e fracos de cada uma dessas soluções, ajudando a identificar a mais adequada para atender aos requisitos específicos do projeto.

Tabela 1 – Comparação entre as ferramentas de desenvolvimento.

Ferramenta	**Vantagens**	**Desvantagens**
Flutter	- Desenvolvimento multiplataforma com único código-fonte (Web, PC e Mobile); - Interface altamente customizável; - Grande biblioteca de plugins e pacotes; - Possibilidade de expansibilidade de funcionalidades da aplicação no futuro;	- Utiliza Dart, uma linguagem menos conhecida e complicada; - Aplicativos tendem a ser maiores em tamanho;
Grafana	- Especializado em monitoramento e visualização de métricas em tempo real; - Suporte para múltiplas fontes de dados; - Compatível com Prometheus, facilitando integração com o PIS;	- Focado em dashboards informativos, com pouca flexibilidade para criar interfaces interativas; - Requer configuração de backend robusta;
Power BI	- Interface intuitiva e customizável para relatórios e dashboards; - Forte integração com o ecossistema Microsoft; - Suporte a múltiplas fontes de dados externas;	- Não pode ser containerizado com Docker; - Integrações complexas para AMQP e RabbitMQ; - Funções avançadas apenas na versão paga;

Fonte: Elaborado pelo autor.

2.5 MEIOS E PADRÕES DE COMUNICAÇÃO

Nesta seção serão discutidas as possíveis formas de comunicação que a interface de monitoramento pode adotar para a recepção dos dados de desempenho almejados. Os dados devem ser recepcionados para a exibição em gráficos de séries temporais e alarmes. Dentro do contexto do PIS, há duas formas que se destacam destes dados serem transmitidos: através de uma consulta a um banco de dados que será atualizado constantemente ou através de uma transmissão direta com a aplicação, que neste caso irá tratar e armazenar as informações de forma adequada internamente.

Há algumas questões que deverão ser tratadas dependendo da forma de comunicação que for adotada. Essas questões serão discutidas individualmente para cada meio de comunicação. Os meios de comunicação mais promissores para uso neste projeto são:

- Comunicação direta usando o protocolo AMQP junto com o broker RabbitMQ, que é o padrão de comunicação dentro do PIS, entre a interface de monitoramento e um serviço de coleta de dados;
- Consulta a um banco de dados SQL, que seria atualizado constantemente com os novos valores medidos para cada variável, permitindo à interface de monitoramento realizar uma consulta desses dados conforme necessário.

Vale ressaltar que, atualmente, o PIS não possui um banco de dados ou um serviço que realize a coleta e o envio dos dados no formato necessário para o projeto. Portanto, nesta seção, discutiremos os padrões e processos ideais para que a interface de monitoramento seja capaz de receber os dados.

2.5.1 Comunicação com RabbitMQ

O RabbitMQ funciona como um "correio" que receberia o dados de diversos serviços e os encaminharia para a interface de monitoramento, onde serão processados e exibidos. Ele utiliza um sistema de comunicação baseado em filas, onde as filas armazenam as mensagens até que sejam consumidas pela interface de monitoramento. Isso assegura que os dados estarão disponíveis mesmo se a interface estiver temporariamente desconectada, pois as mensagens aguardam até serem lidas.

A arquitetura da comunicação entre a aplicação e a interface de monitoramento pode ser descrita como um modelo de publicação-subscrição. O serviço de coleta de dados seria implementado em Python utilizando a biblioteca is-wire, que abstrai e simplifica o uso do RabbitMQ para implementação das filas e a publicação dos dados nela. Essa biblioteca é utilizada como padrão no PIS nos serviços desenvolvidos a ele, mas nada impediria de se utilizar uma biblioteca de baixo nível como a pika, que permite mudar parâmetros diretamente ligados ao RabbitMQ, se necessário.

O serviço de coleta de dados iria publicar em um tópico respectivo ao dado que ele coleta. E além do dado coletado, é essencial publicar junto a ele a informação de qual serviço o dado se refere e o horário que foi medido. Com estes dados se torna possível criar gráficos de séries temporais para se observar o comportamento do

serviço em um determinado tempo. Por exemplo, um serviço que mede o uso de GPU, publicaria a sua lista no tópico “Data.GPU”, onde poderá ser consumida pela interface de monitoramento.

Assim podemos definir que o serviço de coleta de dados atua como um publicador, enviando informações para o RabbitMQ. Já a interface de monitoramento é o consumidor, que recebe esses dados, os processa e exibe para o usuário.

O uso do RabbitMQ nesse projeto apresenta algumas vantagens como:

- Assincronicidade: Como as mensagens são armazenadas nas filas até serem consumidas, a interface de monitoramento pode acessar as informações em seu próprio ritmo, sem comprometer o fluxo dos dados.
- Escalabilidade: Com RabbitMQ, é possível integrar novos serviços de coletas de dados sem alterações complexas na arquitetura, apenas com a implementação de novas filas que podem ser consumidas pela interface de monitoramento em locais diferentes no código.
- Confiabilidade: O sistema suporta técnicas de persistência e confirmações de mensagens, garantindo que nenhuma informação seja perdida no processo de transmissão.
- Integração: O broker já é utilizado no PIS, o que facilitaria a sua implementação em uma nova aplicação.

2.5.2 Comunicação com banco de dados SQL

O banco de dados SQL (Structured Query Language) é uma tecnologia amplamente utilizada para armazenar e gerenciar dados de maneira estruturada e persistente. Ele organiza as informações em tabelas, o que permite consultas eficientes e complexas, ideais para o monitoramento e a análise de grandes volumes de dados. No contexto deste projeto, o uso de um banco de dados SQL serviria como um repositório centralizado onde as medições e dados de desempenho seriam armazenados continuamente.

Para a coleta de dados, um serviço de coleta seria configurado para registrar periodicamente os valores de cada variável monitorada no banco de dados SQL. Este serviço atualizaria o banco de dados com as novas leituras de desempenho, armazenando informações como a identificação da aplicação a ser monitorada, o tipo de variável medida e a data e hora da coleta (ELMASRI; NAVATHE, 2017).

A interface de monitoramento, por sua vez, realizaria consultas SQL diretamente no banco de dados para obter os dados históricos e em tempo real. Essas consultas seriam configuradas para extrair informações específicas com base nos requisitos da interface gráfica da aplicação, por exemplo, filtrando as medições por data e hora.

Para otimizar o desempenho, seria recomendável que a interface de monitoramento executasse consultas periódicas, evitando consultas contínuas que sobrecarregariam o banco de dados (DATE, 2004). Alternativamente, a interface de monitoramento poderia utilizar uma técnica de polling, onde verificaria o banco de dados em intervalos regulares para buscar novas informações.

Das vantagens do uso de um banco de dados SQL, vale citar:

- Persistência e Histórico Completo: Como os dados são armazenados em um banco de dados, o histórico de medições pode ser facilmente mantido e acessado para análises de longo prazo, sem a necessidade de um armazenamento separado. Isso permite a criação de relatórios históricos e análises de tendências (SILBERSCHATZ; KORTH; SUDARSHAN, 2010).
- Estruturação e Flexibilidade nas Consultas: O SQL oferece uma linguagem poderosa para realizar consultas complexas, permitindo que a interface de monitoramento filtre, ordene e agrupe dados de forma eficiente. Isso é útil para gerar gráficos personalizados ou visualizações detalhadas (ELMASRI; NAVATHE, 2017).
- Familiaridade e Padronização: Bancos de dados SQL são amplamente conhecidos e suportados por uma variedade de tecnologias e plataformas, facilitando

a integração com outras ferramentas de análise e visualização de dados (DATE, 2004).

- Centralização dos Dados: O banco de dados funciona como um repositório central, permitindo que diferentes aplicações acessem os mesmos dados sem a necessidade de duplicação ou sincronização (CONNOLLY; BEGG, 2014).

Por outro lado, o uso de um banco de dados SQL pode oferecer algumas desvantagens no contexto desse projeto:

- Latência nas Consultas: Dependendo do volume de dados e da frequência das consultas, pode haver um atraso perceptível no tempo de resposta, especialmente se muitas leituras forem realizadas em um curto período. Esse fator pode impactar a atualização em tempo real dos gráficos e alarmes (SILBERSCHATZ; KORTH; SUDARSHAN, 2010).
- Sobrecarga do Banco de Dados: Em sistemas de monitoramento de alta frequência, o banco de dados pode ficar sobrecarregado com um grande volume de operações de escrita e leitura, o que pode exigir otimizações ou escalabilidade na infraestrutura. (CONNOLLY; BEGG, 2014).
- Dificuldade de Escalabilidade para Assincronia: Ao contrário do RabbitMQ, que permite comunicação assíncrona, o banco de dados SQL exige que a aplicação esteja constantemente consultando para verificar novas medições, o que não é ideal para sistemas que precisam processar dados em tempo real (ELMASRI; NAVATHE, 2017).
- Complexidade de Manutenção: Bancos de dados SQL requerem manutenção periódica, como a indexação de tabelas e otimizações de consulta, especialmente em sistemas que acumulam grandes quantidades de dados (DATE, 2004).

Por fim, a comunicação via banco de dados SQL é uma alternativa viável para a aplicação de monitoramento, oferecendo armazenamento de dados e flexibilidade nas consultas. No entanto, seu uso pode não ser o mais eficiente em contextos que exigem

atualização constante em tempo real, devido à latência nas consultas e ao potencial de sobrecarga do banco de dados. É uma solução que traz vantagens para análises de dados históricos e acesso centralizado, mas, para o monitoramento em tempo real, pode exigir ajustes na infraestrutura para evitar problemas de desempenho.

2.5.3 Comparação

Para fins de comparação, a tabela 2 indica um resumo entre as vantagens e desvantagens de se utilizar o RabbitMQ ou um banco de dados SQL para comunicação dos dados na aplicação de monitoramento.

Tabela 2 – Comparação entre RabbitMQ e SQL.

Método	**Vantagens**	**Desvantagens**
RabbitMQ	- Alta responsividade e baixa latência, ideal para atualização em tempo real; - Comunicação assíncrona permite que a aplicação consuma dados em seu próprio ritmo; - Suporta escalabilidade, permitindo fácil integração de novos serviços de coletas; - Confiabilidade com suporte à persistência e confirmação de mensagens, reduzindo o risco de perda de dados; - Já é implementado no PIS, facilitando integração de novos publicadores e consumidores;	- Requer uma infraestrutura dedicada para gerenciamento de filas e monitoramento (o que não se torna um problema, pois já é implementado no PIS); - Não armazena os dados permanentemente, a menos que configurado para isso, o que dificulta a criação de históricos detalhados;
SQL	- Armazenamento persistente de dados, permitindo históricos completos, facilitando a visualização de muitos dados; - Familiaridade com o SQL facilita a integração e manutenção por uma ampla gama de desenvolvedores; - Não requer uma infraestrutura de mensageria, reduzindo a complexidade em sistemas onde a persistência é mais importante que a latência;	- Latência maior em relação ao RabbitMQ; - Limitações de escalabilidade em sistemas com alta taxa de leitura e escrita simultâneas, podendo exigir otimizações; - Consumo elevado de recursos em consultas frequentes e atualização constante dos dados, especialmente em tempo real; - Menos adequado para comunicação assíncrona em tempo real, onde há necessidade de resposta imediata sem espera por consultas;

Fonte: Elaborado pelo autor.

3 METODOLOGIA

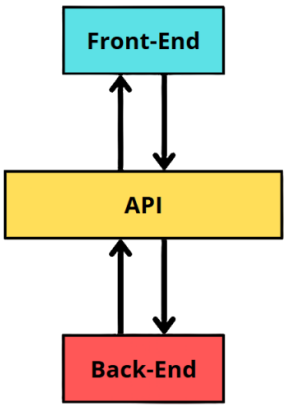
Nesta seção serão explicitados as escolhas e métodos adotados para o desenvolvimento da interface, tanto quanto a motivação para as mesmas.

O sistema proposto é estruturado em três componentes principais: o back-end, o front-end e um mecanismo de comunicação intermediário (API).

O back-end consiste em um conjunto de serviços simulados, responsáveis por gerar métricas sintéticas que emulam dados reais oriundos de ferramentas amplamente utilizadas em ambientes distribuídos, como Zipkin, Prometheus, Kubernetes e Docker. Essas simulações visam replicar cenários típicos de monitoramento de aplicações, incluindo métricas de uso de CPU, tempo de processamento, número de instâncias, entre outras. O back-end não será desenvolvido de fato neste projeto, limitando-se a simulações.

O front-end, por sua vez, é responsável pela interface com o usuário, projetada para exibir os dados de forma clara, responsiva e personalizável, por meio de gráficos interativos. Para possibilitar a comunicação entre essas camadas, será utilizada uma API que atua como elo entre os produtores (serviços simulados) e os consumidores (interface de visualização). Essa arquitetura modular garante flexibilidade, desacoplamento entre as partes e maior escalabilidade para futuras integrações no ecossistema do PIS. A Figura 6 ilustra a estrutura descrita.

Figura 6 – Estrutura do Projeto



Fonte: Elaborado pelo autor.

O foco do projeto está na construção do front-end da interface, responsável por receber e tratar as informações de desempenho, de forma a exibi-las adequadamente ao usuário final.

A interface será construída com foco na flexibilidade e usabilidade, permitindo ao usuário selecionar o tipo de dado a ser monitorado. Serão desenvolvidas diferentes telas com gráficos e informações personalizadas para garantir uma visualização clara e intuitiva dos dados selecionados.

O processo de desenvolvimento seguirá uma abordagem incremental e iterativa. Inicialmente, será implementado o monitoramento completo de uma única métrica, contemplando desde a coleta simulada de dados até a exibição na interface. Após a validação desta etapa, o mesmo processo será repetido para as demais métricas, garantindo que, ao menos, um fluxo de monitoramento esteja completamente funcional e validado ao final de cada ciclo.

A interface será projetada de forma genérica para suportar diversas aplicações no PIS, porém será selecionada uma aplicação-alvo para fins de simulação da extração e análise dos dados de desempenho.

Após o desenvolvimento, serão realizados testes funcionais para validar a consistência da coleta e exibição dos dados, bem como a estabilidade da comunicação entre back-end simulado e o front-end.

Concluído o desenvolvimento e os testes, a aplicação será empacotada com Docker e implantada na rede do PIS. Isso garantirá sua acessibilidade para os usuários do espaço e serviços de coleta de dados ja existentes, seguindo os padrões de containerização utilizados na infraestrutura do PIS.

Por fim, será conduzida uma avaliação com usuários típicos do PIS. A pesquisa terá como foco a navegabilidade da interface, clareza na apresentação das informações e satisfação geral. Essa análise será realizada por meio de questionários aplicados a usuários já familiarizados com o sistema atual, baseado em terminal. A comparação entre as abordagens permitirá avaliar os ganhos em acessibilidade, eficiência e

facilidade de uso promovidos pela solução desenvolvida.

3.1 MATERIAIS

Para o desenvolvimento da interface proposta, as tecnologias a serem utilizadas foram definidas de acordo com as pesquisas feitas nas seções 2.4 e 2.5.

A interface de monitoramento será desenvolvida utilizando o framework Flutter. Essa escolha se justifica principalmente pela capacidade do Flutter de gerar interfaces modernas e customizáveis a partir de um único código-fonte, o que traz vantagens no aspecto de flexibilidade. Além disso, o Flutter junto ao Dart possuem bibliotecas úteis para a finalidade deste projeto, como a `fl_chart`, que permite a construção de gráficos interativos e personalizados.

Outro fator determinante para a adoção do Flutter foi a disponibilidade de bibliotecas que possibilitam a comunicação com serviços externos, como a dart_amqp, que permite a comunicação via RabbitMQ. O uso dessa biblioteca possibilita o recebimento assíncrono dos dados de desempenho, viabilizando a exibição em tempo real sem comprometer a responsividade da interface além de mantér o padrão de comunicação já existente no PIS.

Em relação à comunicação dos dados, o RabbitMQ mostrou-se a ferramenta mais adequada para a comunicação em tempo real, devido à sua baixa latência, comunicação assíncrona, persistência de mensagens, facilidade de escalabilidade e pela facilidade de integração no PIS, visto que essa tecnlogia já é utilizada no espaço como padrão de comunicação. Por meio dele, a interface pode receber dados continuamente publicados por serviços de coleta, sem a necessidade de consultas constantes, o que reduz o estresse computacional causado pela interface e melhora a experiência do usuário.

A partir do RabbitMQ também poderão ser feitas requisições de dados para que a interface possa ter acesso a serviços intermediários que façam consultas a bancos de dados. Dessa forma dados históricos poderão ser recebidos por serviços que fazem uma ponte entre a interface de monitoramento e banco de dados reais. Essa arquitetura garante que dados antigos possam ser acessados conforme a necessidade

da aplicação.

Essa estratégia híbrida permite à aplicação obter o melhor dos dois mundos: a velocidade e eficiência do RabbitMQ para a recepção e visualização de novos dados em tempo real, e a persistência e organização do banco de dados para os dados históricos. Dessa forma, a interface garante um monitoramento em tempo real, sem abrir mão da capacidade de realizar análises de dados históricos.

Para o desenvolvimento dos serviços simulados de coleta e consulta de dados, será utilizada a linguagem Python em conjunto com a biblioteca Pika, que fornece uma interface para comunicação com o RabbitMQ por meio do protocolo AMQP. Essa escolha se deve à simplicidade e flexibilidade do Python, bem como à ampla adoção da biblioteca Pika em sistemas de mensageria assíncrona. Com isso, será possível simular o comportamento de serviços reais de monitoramento, publicando mensagens em tempo real e respondendo a requisições da interface. Esses serviços permitirão validar de forma prática o funcionamento completo do fluxo de comunicação, desde a emissão de dados no back-end até a recepção e exibição no front-end, assegurando a aderência ao modelo proposto e a compatibilidade com a infraestrutura de mensageria do PIS.

3.2 EXPERIMENTOS E DESENVOLVIMENTO

Esta seção apresenta os principais experimentos realizados ao longo do processo de desenvolvimento da aplicação de monitoramento. O objetivo é descrever a construção progressiva da solução, desde os protótipos iniciais até a consolidação de funcionalidades mais robustas e integradas. A abordagem experimental adotada permitiu validar escolhas técnicas, testar diferentes estratégias de comunicação e visualização, bem como ajustar o comportamento da interface com base em resultados obtidos. A seguir, são detalhadas as etapas práticas conduzidas durante o projeto.

3.2.1 Protótipo Inicial

O primeiro experimento realizado teve como objetivo validar a comunicação entre um serviço publicador de dados simulados e um algoritmo Flutter/Dart. Para isso, definiu-

se como variável inicial o consumo de CPU de uma aplicação, visto que é um dado frequentemente utilizado como métrica de desempenho em ambientes computacionais, especialmente no contexto do PIS.

Considerando um cenário real de monitoramento, esse tipo de dado poderia ser extraído com o uso de ferramentas como o Zipkin ou Prometheus, sendo posteriormente publicado em uma fila RabbitMQ. Tendo esse conceito como base, foi implementado um protótipo de serviço em Python, atuando como publicador de dados simulados de uso de CPU. O algoritmo gera, em intervalos de um segundo, valores aleatórios entre 0 e 100, representando o percentual de uso de CPU, e os publica na fila `Data.CPU` do RabbitMQ. A mensagem é estruturada em formato JSON, contendo a chave `cpu_usage`, que armazena o valor simulado.

Do lado da interface gráfica, desenvolvida em Flutter, foi implementado um consumidor que se conecta à fila `Data.CPU` utilizando a biblioteca `dart_amqp`. Ao receber cada mensagem, a aplicação decodifica o JSON, extrai o valor da CPU e o armazena localmente em uma lista de pontos (do tipo `FlSpot`), utilizada para exibição em um gráfico de linha. Para isso, foi empregada a biblioteca `fl_chart`, escolhida por sua popularidade e praticidade na construção de gráficos no Flutter.

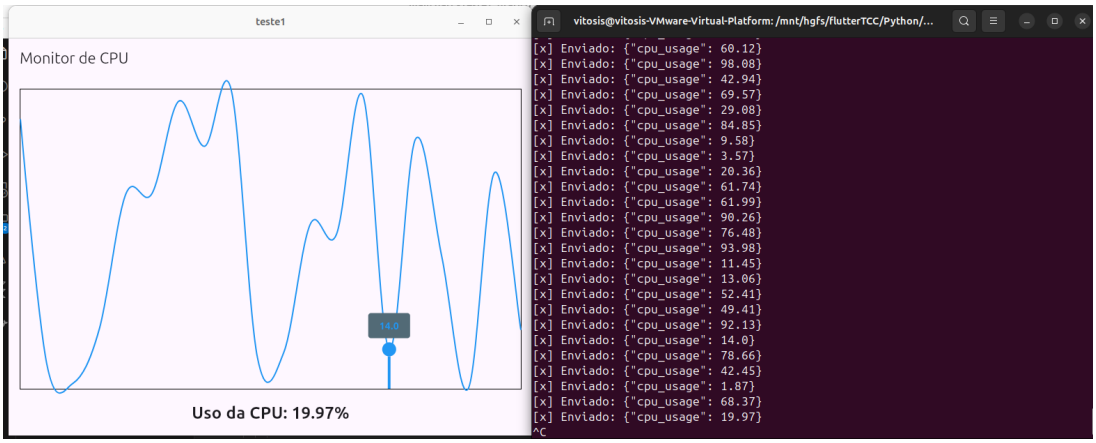
O gráfico de linha foi configurado para exibir os dados de CPU em tempo real. Cada novo valor recebido é atribuído a um ponto `FlSpot`, com coordenadas (x, y), onde o eixo x representa o índice temporal (incrementado a cada nova leitura) e o eixo y representa o valor de uso de CPU recebido.

A cada novo ponto `FlSpot` adicionado, o gráfico renderiza novamente para ilustrar todos os pontos em tempo real. Isso é feito através da função `setState()`.

Além da plotagem dos dados, o gráfico foi configurado com o eixo y fixo entre 0 e 100, impedindo a variação automática de máximos e mínimos padrão da `fl_chart` e proporcionando uma visualização melhor dos dados. A Figura 7 ilustra o funcionamento do protótipo, com o serviço publicador e a interface gráfica em execução.

Com esse experimento, foi possível validar com sucesso a comunicação em tempo

Figura 7 – Publicação e recepção simples de CPU simulada



Fonte: Elaborado pelo autor.

real entre o publicador em Python e o consumidor no Flutter, bem como a correta visualização dos dados na interface gráfica.

Os algoritmos até aqui apresentados estão disponíveis no repositório GitHub do projeto:[1]

Esse protótipo representa a primeira etapa prática de desenvolvimento da aplicação de monitoramento. A partir dele, outras funcionalidades foram planejadas para aprimorar a visualização e o controle dos dados, possibilitando novos experimentos descritos nas próximas seções.

3.2.2 Histórico e banco de dados

A partir do algoritmo desenvolvido na seção anterior, tornou-se possível acompanhar os dados em tempo real por meio de um gráfico. No entanto, esses dados não apresentavam informações sobre o tempo em que foram medidos, o que deixava sua interpretação limitada. Além disso, uma vez ultrapassado o número máximo de pontos permitidos no gráfico ou com o fechamento da aplicação, os dados eram perdidos. Diante disso, surgiu a necessidade de implementar, na interface de monitoramento, uma funcionalidade de consulta a dados históricos, o que exigiu também o desenvolvimento de um banco de dados para armazenamento.

Para atender a esse objetivo, o algoritmo publicador em Python foi aprimorado com a

[1] <https://github.com/VitorBermond/Monitoramento_Flutter_IS/tree/master/1_cpuSimple>

adição de registros de tempo. Para isso, utilizou-se a biblioteca `datetime`, que fornece suporte para geração de timestamps no padrão ISO 8601, um formato amplamente aceito por diferentes bibliotecas e sistemas. A cada geração de um dado simulado de uso de CPU, é atribuído a ele um timestamp que representa o momento da medição. Esse par de informações é publicado na fila `Data.CPU`, com as chaves `cpu_usage` e `timestamp`.

Na interface de monitoramento, o timestamp é extraído e formatado no padrão `HH:mm:ss` utilizando a biblioteca `intl` do Flutter. Esse horário é armazenado na lista `timeLabelsReal`, em paralelo à lista de valores `cpuDataReal`, de modo que ambos compartilham o mesmo índice. Essa estrutura garante que cada ponto no gráfico esteja associado ao instante exato em que a medição foi realizada através do índice das duas listas.

Os rótulos de tempo são utilizados na configuração do eixo horizontal (eixo x) do gráfico, por meio da propriedade do gráfico de linha `bottomTitles`. A função responsável por exibir os títulos do eixo exibe os rótulos correspondentes a cada ponto. Essa estratégia proporciona ao usuário uma visão temporal clara do uso da CPU ao longo do tempo, mesmo durante a execução em tempo real, ao invés de mostrar index que só se incrementam com o tempo. Os rótulos de tempo também aparecem quando o cursor do usuário passa por cima de algum ponto no gráfico, facilitando a visualização.

Para manter a legibilidade do gráfico, os rótulos são exibidos em intervalos controlados, e o algoritmo ajusta dinamicamente o índice a ser mostrado com base na posição relativa do ponto na lista. Isso evita sobreposição de textos e mantém a interface visualmente organizada, mesmo com variações na densidade dos dados

Além de publicar os dados, o algoritmo publicador foi aprimorado para os armazenar em um banco de dados local no formato de planilha (arquivo CSV). Essa abordagem foi adotada para ser possível simular algum banco de dados, sendo que em uma situação real os dados seriam armazenados externamente. Cada linha da planilha contém o timestamp da medição e o valor de uso de CPU correspondente. Para evitar o crescimento descontrolado da planilha e garantir a eficiência do sistema, foi implementada uma rotina de manutenção que exclui registros com mais de 30 dias de idade. Essa janela de retenção pode ser facilmente ajustada de acordo com as

necessidades futuras da aplicação.

Com o banco de dados estabelecido, a interface de monitoramento passou a contar com dois modos de operação: **modo tempo real** e **modo histórico**. No modo tempo real, os dados recebidos da fila `Data.CPU` são armazenados nas variáveis locais `cpuDataReal` (uma lista de objetos `FlSpot`) e `timeLabelsReal` (uma lista de `Strings` representando os horários das medições). Cada medição ocupa a mesma posição (índice) em ambas as listas, permitindo a associação entre o valor numérico e o seu horário correspondente.

Exemplo de estrutura dos dados:

- `cpuDataReal = [FlSpot(0.0, 72.5), FlSpot(1.0, 65.2), FlSpot(2.0, 84.3)]`
- `timeLabelsReal = ["15:01:22", "15:01:23", "15:01:24"]`

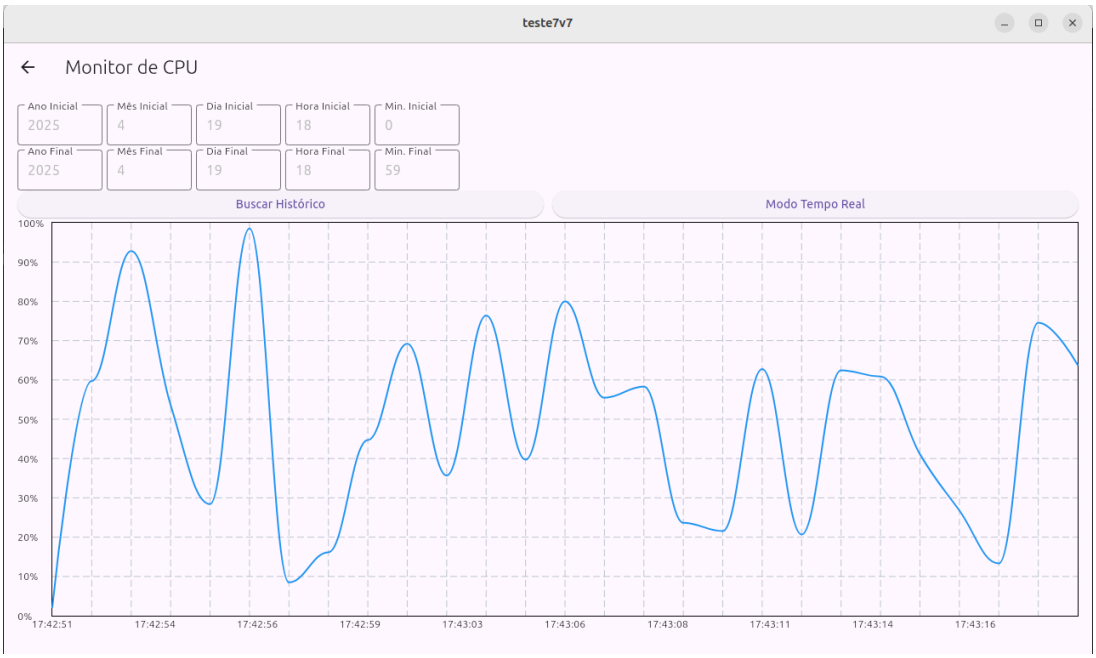
Essas variáveis são utilizadas como parâmetros de entrada no gráfico, para então serem devidamente exibidas.

O modo histórico funciona de forma similar ao modo tempo real, mas os dados são recebidos da fila `Hist.CPU`, onde são publicados sob demanda. Nesse modo, os dados são armazenados nas listas `cpuDataHist` e `timeLabelsHist`, que seguem a mesma lógica de associação entre índices e valores/horários.

Para que o usuário possa ter acesso ao histórico, foram adicionados elementos de interface (widgets) para inserção de um intervalo de data e hora, bem como dois botões: um para alternar entre os modos (histórico/tempo real) e outro para solicitar os dados históricos. A Figura 8 mostra disposição dos novos widgets na interface.

Os campos de texto (`TextField`) foram programados para permitir que o usuário insira manualmente um intervalo de tempo para consulta de dados históricos. Esses campos foram configurados para minimizar a ocorrência de erros de entrada que pudessem comprometer a execução dos algoritmos. Por exemplo, o campo destinado ao ano aceita exclusivamente quatro dígitos numéricos, enquanto os campos de mês, dia, hora e minuto permitem apenas dois dígitos numéricos.

Figura 8 – Novos Widgets na tela de monitoramento de CPU



Fonte: Elaborado pelo autor.

Os campos exibem um valor sugerido (placeholder) correspondente ao horário atual, em uma tonalidade de cinza mais clara. Isso orienta o usuário sobre o formato esperado e, ao mesmo tempo, indica entradas em branco. Essa característica é importante porque, ao construir a requisição de histórico, o algoritmo trata qualquer campo não preenchido como um valor padrão, assumindo os valores atuais do sistema para ano, mês, dia e hora, e adotando os valores de 0 (início) e 59 (fim) para os minutos. Essa abordagem faz com que o sistema possa processar uma requisição mesmo quando alguns campos forem omitidos pelo usuário. Essa foi uma decisão de projeto que evita falhas por valores nulos e, nesse caso específico, resulta em uma busca pelos dados da hora corrente.

Ao pressionar o botão de buscar histórico, a construção da requisição é feita no formato JSON, contendo as chaves que representam os componentes da data e hora de início e de fim do intervalo. Essa estrutura é então publicada na fila `HistRequest.CPU`, de onde será consumida pelo serviço responsável pela busca no banco de dados.

Essa estrutura de requisição foi projetada para ser interpretada por um serviço de back-end implementado em Python. Esse serviço escuta a fila `HistRequest.CPU`, extrai o intervalo de tempo da mensagem, e realiza a conversão para objetos `datetime`

válidos que podem ser utilizados em comparações de datas e horas. Em seguida, percorre as linhas da planilha CSV e filtra os registros cujos timestamps estejam dentro do intervalo especificado. Os dados selecionados são inseridos na lista `historico`, com cada entrada contendo as chaves `cpu_usage`, `timestamp` e `timeIndex`, sendo esta última utilizada como coordenada x na plotagem do gráfico. A Figura 9 ilustra a parte do algoritmo responsável por essa filtragem utilizando os parâmetros inseridos pelo usuário dentro da interface.

Figura 9 – Busca por intervalos no banco de dados

```
if start_datetime <= row_timestamp <= end_datetime:
    historico.append({
        "timestamp": row_timestamp.strftime("%H:%M:%S"),
        "cpu_usage": cpu_usage,
        "timeIndex": time_index
    })
    time_index += 1
```

Fonte: Elaborado pelo autor.

O índice `timeIndex` é atribuído de forma incrementativa, iniciando em 0, e representa a ordem dos dados do mais antigo para o mais recente, permitindo que o gráfico seja plotado de forma temporalmente coerente.

Um problema identificado durante a exibição de históricos com grande quantidade de dados má visualização da curva e o impacto no desempenho da aplicação, causando travamentos e lentidão perceptível ao usuário.

Para mitigar esse problema, foi implementada a técnica de otimização downsampling por média. Essa técnica reduz o número de pontos a serem plotados, condensando as informações de forma representativa e garantindo que o gráfico continue leve e com uma visualização menos poluída.

O algoritmo de downsampling segue as seguintes etapas:

1. Ao receber o pacote, verifica se o número total de pontos históricos (`cpuDataHist.length`) ultrapassa o limite máximo estabelecido de pontos no gráfico (`maxPoints = 60`);

2. Divide a lista de pontos em blocos de tamanho fixo, calculado com:

$$\text{step} = \left\lceil \frac{n}{N} \right\rceil \tag{3.1}$$

onde n é o número total de pontos e N é o número máximo de pontos permitido (Definido como 60);

3. Para cada bloco, calcula-se a média dos valores x (índice temporal) e y (uso de CPU), com as fórmulas:

$$\bar{x} = \frac{1}{k}\sum_{i=1}^{k} x_i, \quad \bar{y} = \frac{1}{k}\sum_{i=1}^{k} y_i \tag{3.2}$$

onde k é o número de elementos no bloco;

4. A média de cada bloco $(\bar{x}, \bar{y})$ é adicionada como novo ponto em uma lista de dados reduzida (`downsampledData`);

5. Para fins de exibição no eixo temporal do gráfico, também é armazenado o rótulo de tempo correspondente ao ponto central do intervalo original (`downsampledLabels`), obtido pelo índice médio:

$$\text{midpointIndex} = \left\lfloor \frac{i+j}{2} \right\rfloor \tag{3.3}$$

6. Ao final, substituem-se os dados históricos e rótulos originais pelas versões reduzidas.

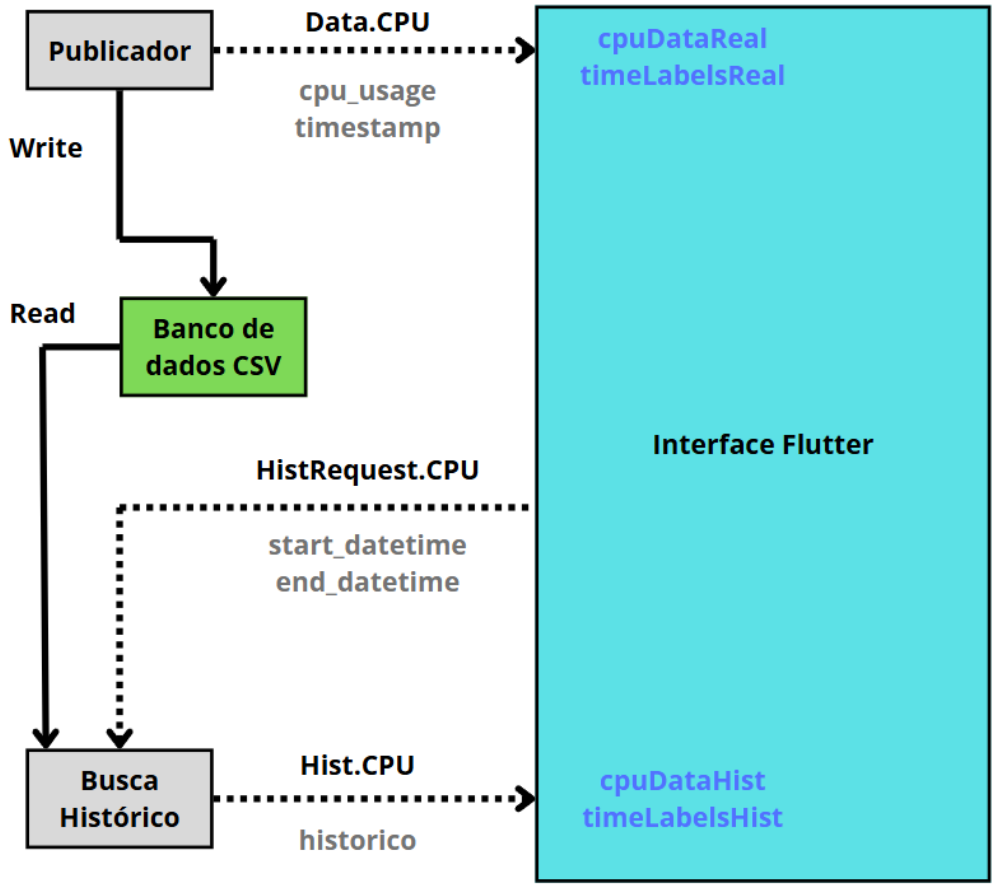
Esse procedimento garante que o gráfico final nunca ultrapasse o limite pre-estabelecido de pontos no gráfico `maxPoints`. Isso proporciona uma visualização mais estável, com uma percepção contínua da tendência geral do uso de CPU, mesmo com perda de detalhes pontuais.

Embora esse processo possa eliminar variações menores e picos de curto prazo, ele é eficaz para garantir a performance da aplicação. Para visualizar dados com maior resolução, o usuário pode selecionar intervalos de tempo menores, nos quais o downsampling será menos agressivo ou sequer necessário.

Por fim, o botão de alternância entre modos altera uma variável booleana interna que determina se os dados a serem plotados devem vir de `cpuDataReal` e `timeLabelsReal` (modo tempo real) ou de `cpuDataHist` e `timeLabelsHist` (modo histórico).

Com essas funcionalidades implementadas, a aplicação passou a oferecer tanto o monitoramento em tempo real com baixa latência quanto a possibilidade de análise histórica de dados. A Figura 10 ilustra um fluxograma do funcionamento da interface e seus dois serviços integrados.

Figura 10 – Fluxograma de funcionamento da interface



Fonte: Elaborado pelo autor.

Os algoritmos até aqui apresentados estão disponíveis no repositório GitHub do projeto.[2]

3.2.3 Múltiplos serviços

Nas seções anteriores, foram implementadas as funcionalidades de visualização em tempo real e de consulta a dados históricos de uso de CPU, considerando apenas um único serviço simulado. No entanto, o objetivo deste projeto é monitorar uma aplicação como um todo, e uma aplicação pode ser conceituada como um conjunto de serviços interdependentes. Diante disso, nesta etapa foi simulada a coleta de dados de múltiplos serviços distintos, com o intuito de representar o comportamento de uma aplicação em um cenário real.

Para isso, o serviço publicador (responsável pela geração dos dados simulados) foi modificado para gerar quatro instâncias distintas de dados, cada uma representando a carga de CPU de um serviço independente. Essa modificação exigiu uma decisão importante no que diz respeito à estratégia de transmissão das mensagens via RabbitMQ. Foram consideradas duas abordagens principais:

1. **Publicação em uma única fila com identificação de serviço:** nesta abordagem, todas as medições de CPU são publicadas na mesma fila (`Data.CPU`), e cada mensagem contém um identificador do serviço correspondente, por meio de um campo adicional no JSON, como `service_name`. Essa centralização simplifica a lógica de publicação no lado produtor, reduz a quantidade de filas gerenciadas pelo broker e facilita a leitura por consumidores que desejam obter um panorama geral de todos os serviços.

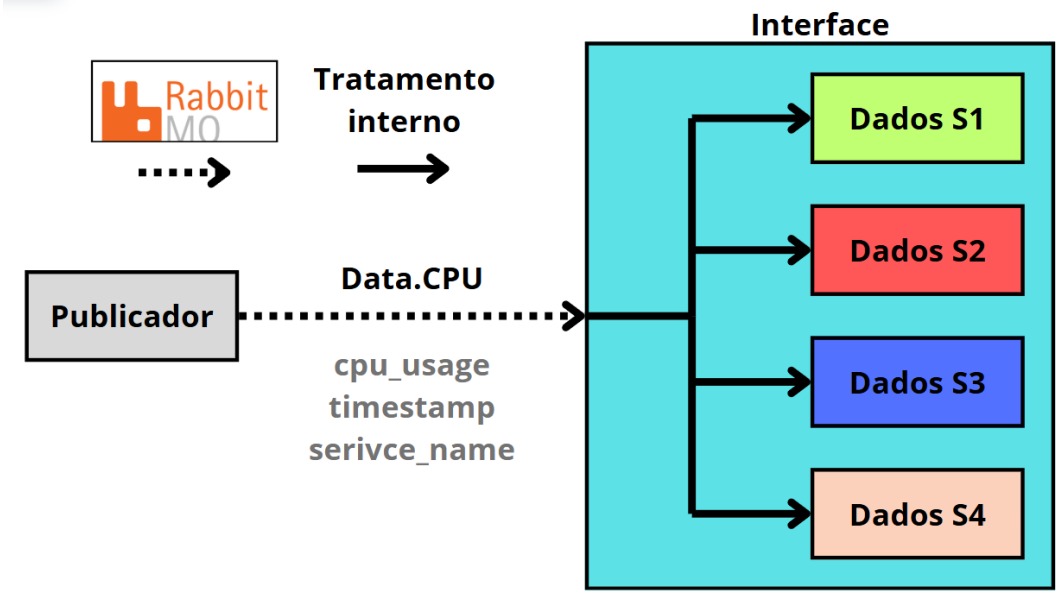
   Entretanto, essa abordagem exige que a interface de monitoramento implemente uma lógica adicional para classificar e separar os dados conforme o identificador do serviço, o que aumenta a complexidade do código no front-end. Além disso, em cenários com alto volume de dados e muitos serviços sendo monitorados simultaneamente, a centralização em uma única fila pode gerar gargalos, aumento de latência e possíveis perdas de desempenho, especialmente se o ritmo de

[2] <https://github.com/VitorBermond/Monitoramento_Flutter_IS/tree/master/2_cpuHist>

consumo não acompanhar a taxa de publicação. Porém isso não seria um problema real no cenário desta aplicação, visto que os dados a serem transmitidos não são de grande volume. A situação seria diferente se fossem transmitidas imagens, por exemplo.

A Figura 11 ilustra o fluxo de dados utilizando a estratégia de fila única.

Figura 11 – Fluxo de fila única no RabbitMQ



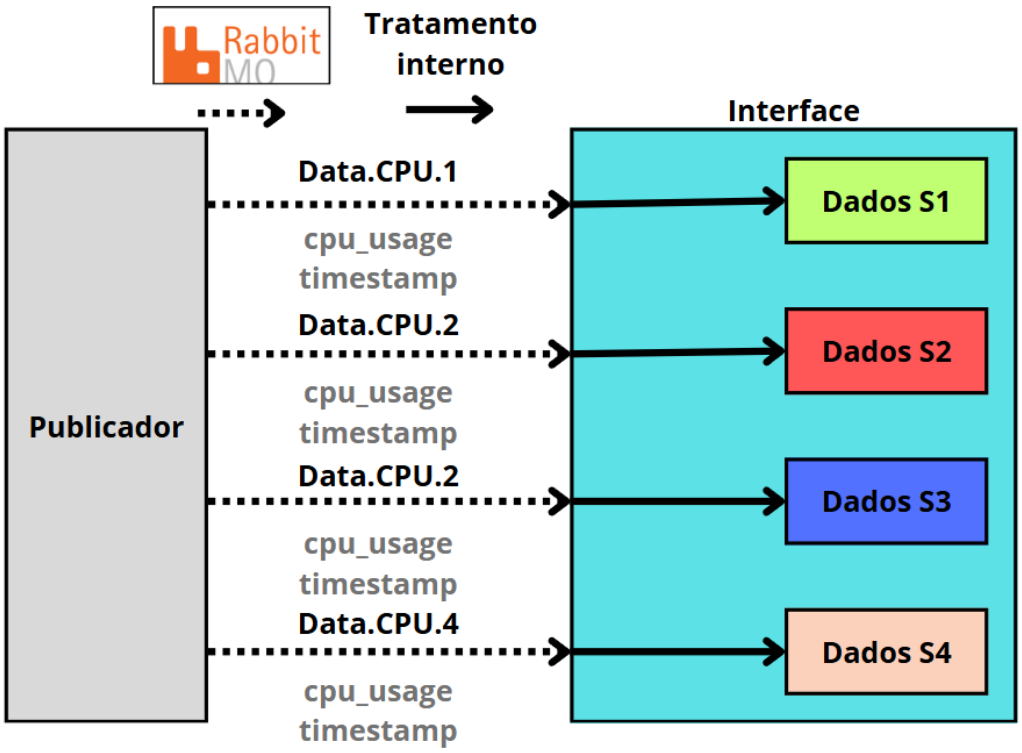
Fonte: Elaborado pelo autor.

2. **Publicação em múltiplas filas segmentadas por serviço:** nesta alternativa, cada serviço simulado tem sua própria fila exclusiva, como `Data.CPU.0`, `Data.CPU.1`, `Data.CPU.2`, e assim por diante. Essa segmentação permite um controle mais granular das conexões e assinaturas da interface, possibilitando, por exemplo, que o sistema visualize ou consuma apenas os serviços selecionados pelo usuário. Além disso, essa abordagem tende a escalar melhor, uma vez que distribui a carga de mensagens entre várias filas, minimizando o risco de sobrecarga em um único canal.

   Como pontos negativos, essa estratégia aumenta a complexidade do lado publicador, que precisa lidar com múltiplas filas, e do lado da infraestrutura de mensageria, que deve gerenciar e manter diversas filas ativas. Em ambientes com número variável de serviços (por exemplo, instâncias criadas dinamicamente), a criação e remoção de filas também pode exigir uma lógica adicional de gerenciamento.

   A Figura 12 ilustra o fluxo de dados utilizando a estratégia de multiplas filas.

Figura 12 – Fluxo de multiplas filas no RabbitMQ



Fonte: Elaborado pelo autor.

Ambas as estratégias são válidas, mas para os fins deste projeto foi adotada a abordagem de fila única, por favorecer a simplicidade da estrutura de comunicação e trazer a complexidade de tratamento para o Front-end. A separação dos dados por serviço passa a ser feita exclusivamente na interface de monitoramento.

Com essa decisão, o serviço publicador foi ajustado para iterar sobre uma lista fixa de identificadores de serviços: `"1A"`, `"2B"`, `"3C"` e `"4D"`. A cada segundo, o sistema gera um valor aleatório de uso de CPU para cada serviço e publica uma mensagem JSON contendo os campos `timestamp`, `cpu_usage` e `service_name`, conforme a Figura 13.

Figura 13 – Log do serviço publicador

```
python3 pubunix.py
[x] Serviço: 1A | Enviado: {"timestamp": 1747140433, "cpu_usage": 5.63, "service_name": "1A"}
[x] Serviço: 2B | Enviado: {"timestamp": 1747140433, "cpu_usage": 71.37, "service_name": "2B"}
[x] Serviço: 3C | Enviado: {"timestamp": 1747140433, "cpu_usage": 16.31, "service_name": "3C"}
[x] Serviço: 4D | Enviado: {"timestamp": 1747140433, "cpu_usage": 73.88, "service_name": "4D"}
[x] Serviço: 1A | Enviado: {"timestamp": 1747140434, "cpu_usage": 36.08, "service_name": "1A"}
[x] Serviço: 2B | Enviado: {"timestamp": 1747140434, "cpu_usage": 32.54, "service_name": "2B"}
[x] Serviço: 3C | Enviado: {"timestamp": 1747140434, "cpu_usage": 9.31, "service_name": "3C"}
[x] Serviço: 4D | Enviado: {"timestamp": 1747140434, "cpu_usage": 70.85, "service_name": "4D"}
^C
```

Fonte: Elaborado pelo autor.

O formato do `timestamp` foi adaptado para o padrão Unix Epoch, no qual o tempo é representado como um valor inteiro correspondente ao número de segundos ou milissegundos decorridos desde 1º de janeiro de 1970 (UTC). Essa abordagem facilita o tratamento e filtragem do tempo, que será tratado como um número inteiro, e se aproxima do formato extraído de sistemas reais de coleta de métricas, como o Zipkin e Prometheus.

Na interface Flutter, foi necessário adaptar o código para lidar com múltiplas séries temporais no gráfico de linha. O widget `LineChart` permite a exibição simultânea de várias curvas, onde cada uma seria associada a um serviço diferente. Para organizar esses dados de maneira eficiente, foram utilizadas estruturas do tipo `Map`. Em Dart, um `Map` é uma estrutura de dados que associa chaves únicas a variáveis do mesmo tipo. Neste caso, o nome do serviço é utilizado como chave, e a lista de valores representa os dados daquele serviço. Dessa forma, novos maps são criados dinamicamente de acordo com a chegada de novos serviços.

Todos os dados são centralizados em uma instância da classe `CPUUsageData`, represen-

tada pela variável `dataStore`. Nela são armazenados mapas que guardam os dados em tempo real e históricos, os rótulos de tempo e outras informações necessárias à renderização dos gráficos. O uso da classe dessa forma garante que os dados fiquem centralizados e armazenados na aplicação mesmo quando o usuário trafega sobre diferentes telas.

Durante o consumo das mensagens, o código realiza verificações com métodos como `putIfAbsent` e operadores nulos seguros, a fim de evitar exceções. Essas precauções são necessárias, pois em Dart o acesso a uma chave inexistente em um `Map` pode causar falhas na aplicação.

Além disso, foi criado o Map `servicesList`, que associa identificadores numéricos incrementais aos nomes dos serviços detectados dinamicamente, conforme novos nomes de serviços chegam. Essa estrutura auxilia em diversas funcionalidades da interface, como a geração de botões de controle de visibilidade das linhas no gráfico, a construção das requisições de dados históricos e a associação visual entre cada serviço e uma cor específica.

Para permitir essa associação visual por serviço, foi criada uma variável global e constante `listaCores`, que armazena uma lista fixa de cores predefinidas, conforme a Figura 14.

Figura 14 – Cores pre-definidas para serviços

```
const List<Color> listaCores = [
  Color(0xFF2196F3),
  Color(0xFFFF9800),
  Color(0xFFF44336),
  Color(0xFF8BC34A),
  Color(0xFF3F51B5),
  Color(0xFFF48FB1),
  Color(0xFF00E676),
  Color(0xFF009688),
  Color(0xFFFFC107),
  Color(0xFF4CAF50),
  Color(0xFFF44336),
  Color(0xFF673AB7),
  Color(0xFFFF5722),
  Color(0xFF00BCD4),
  Color(0xFF9C27B0),
];
```

Fonte: Elaborado pelo autor.

A seleção da cor correspondente a cada serviço é feita com base na posição do serviço na lista `servicesList`. Assim, cada serviço possui uma cor única e consistente em todos os elementos gráficos, facilitando a identificação visual pelo usuário.

A função responsável por requisitar dados históricos também foi atualizada para lidar com múltiplos serviços. Embora a lógica geral da requisição tenha se mantido, agora, além do intervalo de tempo desejado, é incluída na requisição a lista de serviços presentes em `servicesList`. Dessa forma, o serviço que processa os pedidos históricos saberá exatamente de quais serviços e planilhas deve buscar os dados, com base nas chaves identificadas pela interface.

Essa abordagem, no entanto, impõe uma limitação funcional: a interface precisa primeiro receber ao menos uma mensagem em tempo real de um determinado serviço para que ele seja adicionado à `servicesList`. Apenas após esse reconhecimento inicial, torna-se possível realizar uma requisição de histórico para tal serviço.

O serviço responsável por processar as requisições de dados históricos também foi adaptado para operar com múltiplos serviços. Ele atua como um consumidor da fila HistRequest.CPU, onde a interface Flutter publica mensagens de requisição no formato JSON contendo três campos principais: `services_list`, `start_datetime` e `end_datetime`. O campo `services_list` contém a lista de identificadores de serviços que devem ter seus dados históricos consultados, enquanto os campos de data definem o intervalo de tempo em que os dados devem ser buscados, ambos no formato Unix Epoch.

Ao receber uma requisição, o serviço percorre a lista de serviços solicitados e, para cada um, constrói o nome do arquivo CSV correspondente (como cpu_usage_1A.csv, cpu_usag_2B.csv, etc.). Em seguida, chama a função que abre o arquivo indicado, lê as linhas e filtra os dados que se encontram dentro do intervalo solicitado. Cada linha válida é transformada em um objeto contendo timestamp e cpu_usage.

Após processar os dados de cada serviço, o sistema envia uma resposta individualizada para a fila Hist.CPU, contendo o nome do serviço e a lista dos pontos históricos encontrados. Isso permite que a interface receba os dados já categorizados por serviço,

simplificando seu tratamento exibição no gráfico.

A renderização do gráfico de séries temporais é realizada com base nos dados organizados nos mapas mencionados anteriormente, que armazenam os dados recebidos no modo tempo real e histórico. Para isso, é utilizada uma estrutura de repetição que percorre todos os serviços armazenados em `servicesList` e gera uma instância de `LineChartBarData` para cada um. Essas instâncias são então agrupadas na lista `lineBarsData`, que é fornecida como entrada ao widget `LineChart`. Dessa forma, todos os Maps criados tendo como chave o nome dos serviços são varridos e plotados no gráfico.

A visibilidade de cada linha correspondente a um serviço no gráfico é gerenciada dinamicamente por meio do mapa `lineVisibility`, no qual cada chave representa o nome do serviço e o valor associado é um booleano que indica se a linha deve ser exibida (true) ou ocultada (false). Antes de renderizar as linhas, o sistema verifica este mapa: se o serviço estiver marcado como invisível ou não possuir dados disponíveis, sua respectiva linha não é incluída na lista `lineBarsData` e, portanto, não será desenhada no gráfico.

Essa lógica permite maior controle sobre a exibição das informações, tornando o gráfico mais limpo e adaptável às preferências do usuário. Para dar acesso a esse controle, foram adicionados à interface botões para cada serviço. Esses botões permitem alternar a visibilidade de forma individual e interativa, atualizando o valor correspondente no mapa lineVisibility sempre que são acionados. Dessa forma, o usuário pode personalizar a visualização dos dados conforme sua necessidade, ocultando temporariamente informações irrelevantes ou focando em serviços específicos.

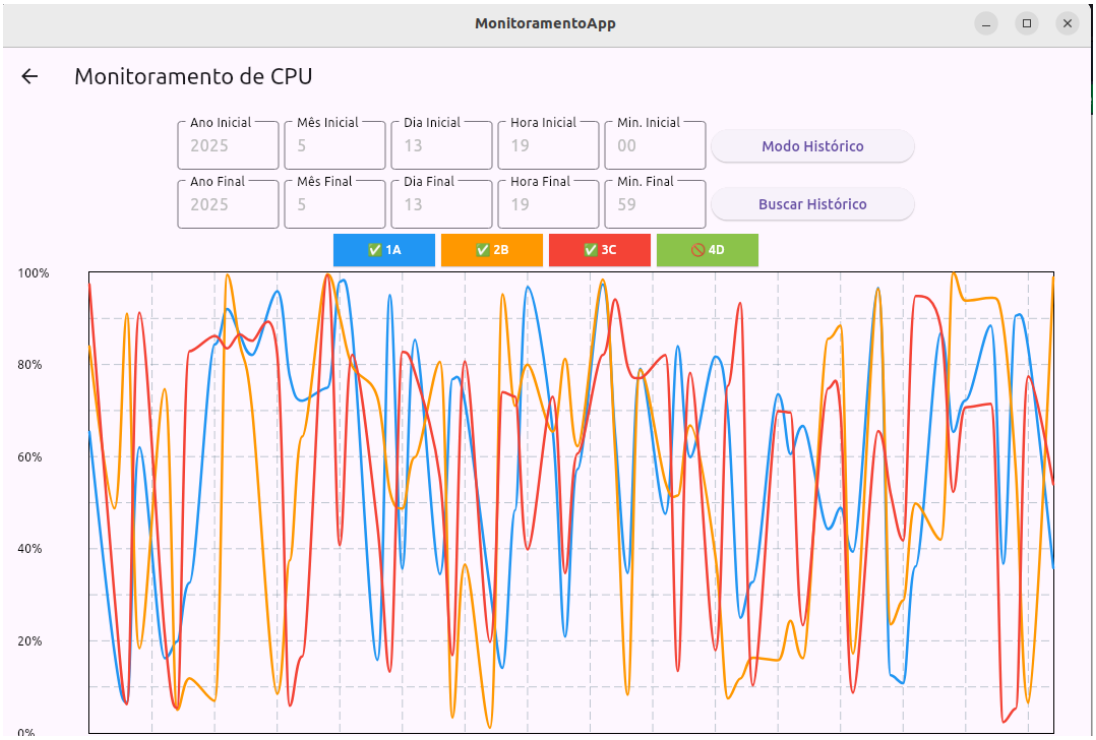
Os botões são gerados dinamicamente a partir da estrutura `servicesList`, utilizando o widget `Wrap` para organizar sua disposição na interface. Cada botão exibe o nome do serviço correspondente, acompanhado de um ícone visual que indica seu estado atual . A cor de fundo de cada botão segue o mesmo padrão de cores utilizado para as linhas no gráfico, reforçando a associação entre elementos visuais.

A ação de alternar a visibilidade é realizada por meio de um comando `setState()`, que

inverte o valor booleano associado ao serviço no mapa `lineVisibility`. Essa ação força a reconstrução do gráfico, atualizando a renderização para refletir o novo estado do botão. A implementação é feita de forma a garantir que, mesmo com a adição dinâmica de novos serviços, os botões correspondentes sejam gerados corretamente e mantenham a coerência visual com o restante da interface.

Com essas mudanças, a aplicação dá o poder para o usuário gerenciar as métricas de cada serviço de forma clara e interativa. A tela de monitoramento de CPU após essas mudanças está ilustrada na Figura 15. Nota-se que dados já foram recebidos do serviço publicador e a visibilidade do serviço "4D" foi desativada.

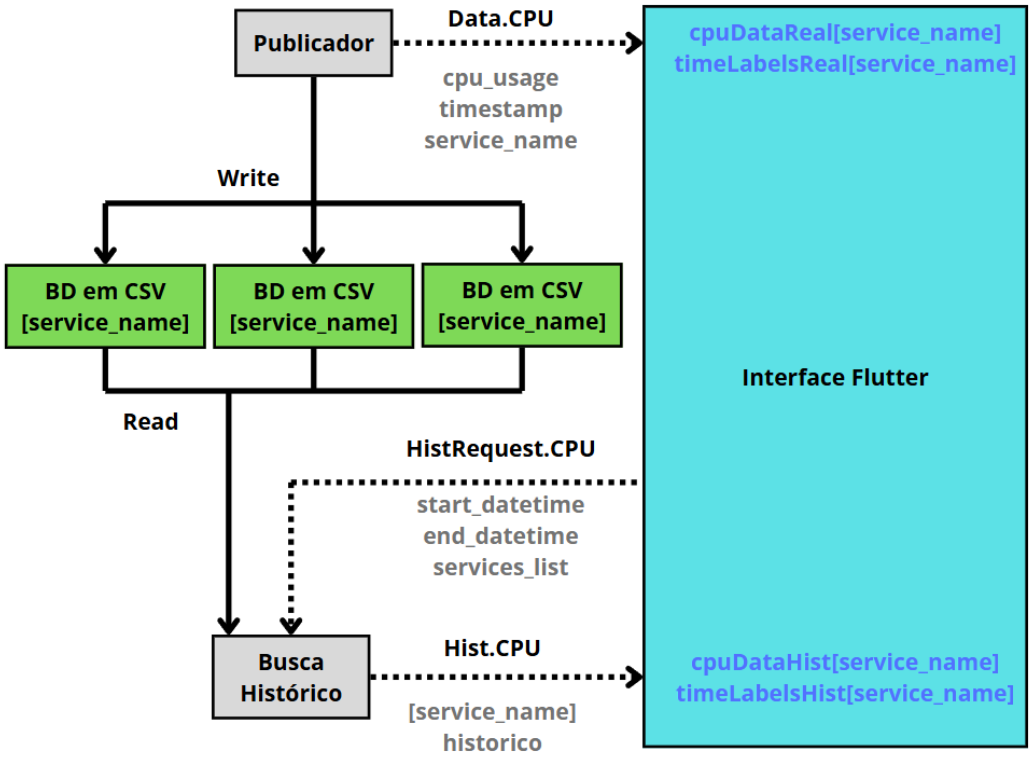
Figura 15 – Monitoramento de CPU de multiplos serviços



Fonte: Elaborado pelo autor.

A Figura 16 ilustra um fluxograma de funcionamento da aplicação após as mudanças para receber e ilustrar os dados de CPU de múltiplos serviços.

Figura 16 – Fluxograma de funcionamento para multiplos serviços



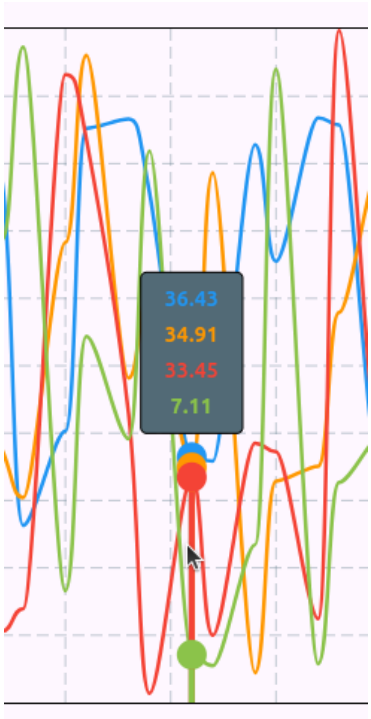
Fonte: Elaborado pelo autor.

3.2.4 Melhorias visuais e de funcionalidade

Com as funções de coleta, histórico e suporte a múltiplos serviços já estabelecidos, nessa seção, o esforço concentrou-se na qualidade da interação com a interface.

O comportamento padrão do widget `LineChart` mostrou-se insuficiente: ao aproximar-se o cursor nos pontos do gráfico, o balão informativo (*tooltip*) exibia apenas o valor numérico de cada ponto, sem unidade, sem horário de coleta. Dessa forma, os *tooltips* são pobres em informações. Além disso, em cenários com duas ou mais séries, toques no gráfico resultavam em um toque em uma coluna inteira de pontos. Esse comportamento padrão pode ser visualizado na Figura 17.

Figura 17 – Tooltip padrão fl_chart



Fonte: Elaborado pelo autor.

A estratégia adotada foi personalizar o objeto `LineTouchData`, que lida com animações ao toque no gráfico, para entregar ao usuário uma experiência melhor ao analisar os dados plotados.

1. **Raio de sensibilidade reduzido.** O parâmetro `touchSpotThreshold` foi fixado em 12 pixels. Assim o gráfico só considera "toque" quando o cursor realmente se aproxima do ponto, deixando de acionar o *tooltip* para toda a coluna de pontos.
2. **Indicador minimalista.** A linha vertical que acompanha a coluna foi removida e manteve-se apenas a elevação dos pontos tocados, evitando poluição visual.
3. **Tooltip enriquecido.** A rotina `getTooltipItems`, que lida com o conteúdo dos *tooltips*, agora:
    - localiza o serviço pelo `barIndex`, que retorna qual linha está sendo tocada, e obtém sua cor e colore o texto com a mesma tonalidade da série, reforçando a associação cromática por serviço e impedindo que todos os *tooltips* fiquem da mesma cor;

- utiliza o recurso de extrair o `spotIndex` dos pontos tocados para consultar o Map `timeLabelsReal` ou `timeLabelsHist` (dependendo se está no modo histórico ou tempo real), resgatar os rótulos de tempo já formatado em datetime e com emojis para identificação de hora e data;
- concatena unidade (%) ao valor da métrica e acrescenta um emoji identificador;

4. **Mapeamento índice ↔ rótulo de tempo.** Cada mensagem recebida via RabbitMQ contém um campo de timestamp no formato Unix Epoch, que representa o instante da coleta em segundos desde 01/01/1970. Esse formato, embora prático para manipulação computacional, não é legível para usuários humanos. Por isso, a cada novo pacote recebido, o timestamp é convertido para o formato `dd/MM/yy HH:mm:ss`, com acréscimo de emojis que indicam data e hora para facilitar a identificação visual.

   Os rótulos formatados são armazenados no `Map <String, List<String» timeLabelsReal` ou `Map <String, List<String» timeLabelsHist`, onde cada chave representa um serviço e a lista associada contém os rótulos de tempo, organizados na mesma ordem dos dados gráficos (`FlSpot`). Assim, o índice de cada elemento no mapa de rótulos é sincronizado com os índices nos demais mapas, permitindo que a função de exibição do tooltip, ao identificar o `spotIndex` de um ponto, acesse diretamente o rótulo de tempo correspondente.

   A Figura 18 apresenta o trecho responsável por essa conversão e formatação de data e hora.

Figura 18 – Tratamento de timestamp

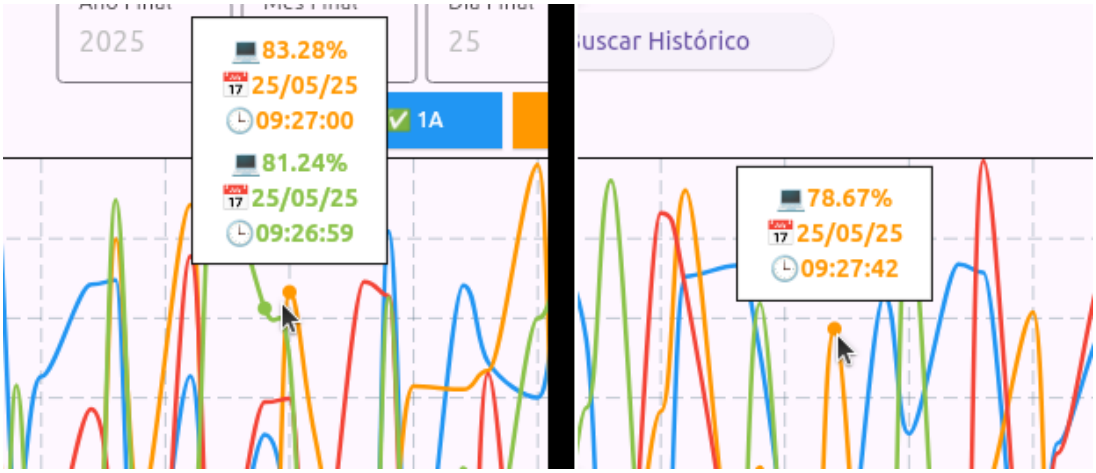
```
// Transforma Unix Epoch em string formatada
String formattedTime = DateFormat("📅dd/MM/yy\n🕒HH:mm:ss").format(
  DateTime.fromMillisecondsSinceEpoch(timestampEpoch * 1000)
      .toLocal(),
);
```

Fonte: Elaborado pelo autor.

Após a customização, o usuário visualiza, com um único hover, valor, serviço, data e hora do ponto de interesse, sem sobrecarga de informações adjacentes. O resultado

final das configurações de tooltips pode ser visualizado na Figura 18 que ilustra o tooltip quando o usuário aproxima o mouse de um e de dois pontos diferentes no gráfico.

Figura 19 – Tooltips configurados

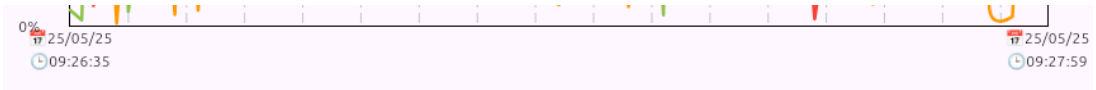


Fonte: Elaborado pelo autor.

O objeto `BottomTitles` que trata dos rótulos do eixo inferior do gráfico também foi personalizado para apresentar as mesmas informações de tempo armazenadas nos mapas `timeLabelsReal` ou `timeLabelsHist`, já utilizados nos tooltips. No entanto, ao exibir múltiplas séries simultaneamente, manter os rótulos para todos os pontos causaria sobreposição visual e poluição no gráfico.

Para evitar esse problema e manter uma apresentação mais limpa, optou-se por exibir apenas dois rótulos: o correspondente ao ponto mais à esquerda (isto é, o mais antigo) e o ponto mais à direita (o mais recente) dentre todas as séries visíveis. Esses dois valores são obtidos por uma função que percorre os dados de todos os serviços e identificando os extremos com base nos valores de `x`. A Figura 20 mostra os novos rótulos do eixo inferior do gráfico.

Figura 20 – Rótulos do eixo inferior



Fonte: Elaborado pelo autor.

Essa solução é suficiente para situar o usuário no intervalo de tempo representado no gráfico e, ao mesmo tempo, preserva a legibilidade. Como os detalhes de cada ponto podem ser consultados ao passar o cursor sobre o gráfico (como descrito

anteriormente), não há necessidade de sobrecarregar o eixo inferior com múltiplos rótulos intermediários.

Os botões de visibilidade dos serviços também foram aprimorados para fornecer uma informação adicional relevante ao usuário. Além de indicarem o nome do serviço e seu estado de visibilidade (ativo ou oculto no gráfico), cada botão agora exibe a média dos valores registrados para aquele serviço. Essa média é calculada com base nos valores de Y presentes nos mapas de pontos `cpuDataReal` ou `cpuDataHist`, a depender do modo atual do gráfico. A Figura 21 ilustra a nova disposição dos botões com a média do seu respectivo serviço.

Figura 21 – Botões de visibilidade com média

Fonte: Elaborado pelo autor.

Essa modificação oferece ao usuário uma visão estatística imediata sobre o comportamento da métrica monitorada. A média é um indicador útil para compreender o padrão geral de consumo de cada serviço, especialmente em cenários com oscilações pontuais que poderiam mascarar tendências.

Outra melhoria implementada na interface foi a adição de um campo de texto e um botão que permitem ao usuário inserir manualmente o nome de um serviço na lista de serviços monitorados. Essa funcionalidade surgiu como resposta a uma limitação observada no processo de requisição de dados históricos: o sistema só permitia solicitar históricos de um serviço após o recebimento prévio de algum dado em tempo real relacionado a ele.

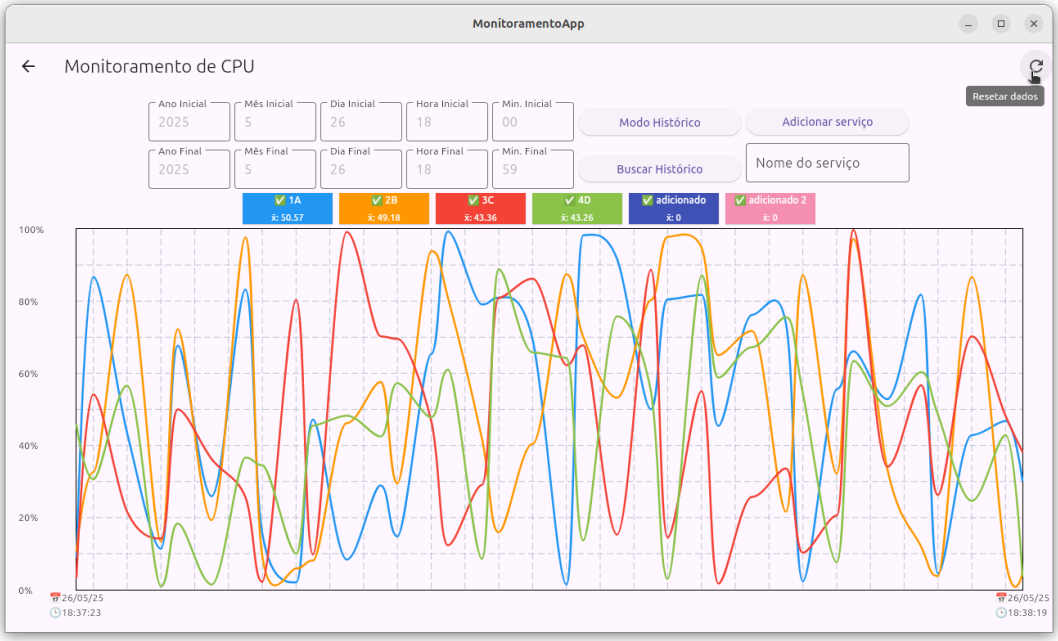
Isso ocorria porque, como mencionado anteriormente, a requisição de dados históricos depende da lista `servicesList`, que é enviada ao serviço responsável por consultar os bancos de dados. Sem a inclusão prévia do serviço nessa lista, a requisição não pode ser processada corretamente. Com a nova funcionalidade, o usuário pode inserir o nome de qualquer serviço desejado, mesmo que ele ainda não tenha gerado nenhum dado em tempo real, viabilizando a consulta imediata ao seu histórico, sem limitar-se a esperar a chegada de uma métrica de certo serviço para então poder realizar

requisições. A adição bem-sucedida do serviço pode ser confirmada visualmente pelo surgimento do botão de visibilidade correspondente na interface, visto que os botões são gerados dinamicamente com os nomes contidos em `servicesList`.

Para complementar essa funcionalidade e possibilitar correções por inserções incorretas ou testes durante o uso, foi incluído um botão de redefinição no canto superior direito da tela. Esse botão limpa todos os dados atuais da interface, incluindo os mapas de dados, os rótulos de tempo e a lista de serviços visíveis, permitindo que o usuário reinicie a visualização com um estado limpo.

A Figura 22 ilustra a interface após todas as melhorias feitas nesta seção.

Figura 22 – Tela final de CPU



Fonte: Elaborado pelo autor.

Com isso, conclui-se o desenvolvimento do monitoramento da métrica de CPU. Neste ponto, todas as funcionalidades relacionadas à visualização, armazenamento, consulta e interação com os dados estão plenamente implementadas e refinadas, oferecendo uma experiência completa e eficaz ao analisar dados.

3.2.5 Generalização e menu inicial

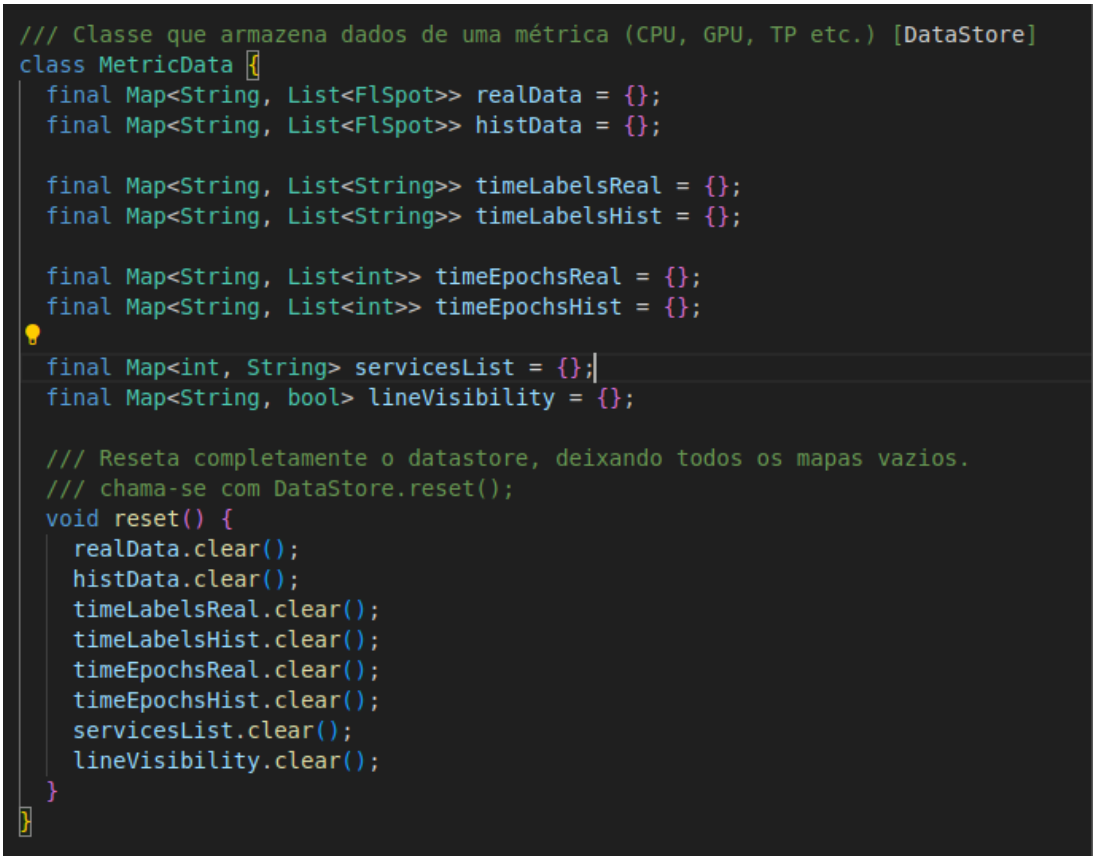
Nesta etapa, o objetivo é preparar a estrutura do algoritmo para permitir a visualização de diferentes métricas reaproveitando os componentes desenvolvidos nos experimentos anteriores. Como a maioria das métricas de desempenho podem ser representadas de forma eficiente por gráficos de linha de séries temporais, o núcleo da lógica de visualização pode permanecer o mesmo. Será implementada uma tela inicial que serve como menu de navegação entre os diferentes gráficos, permitindo que o usuário escolha a métrica que deseja monitorar.

Para evitar duplicação de código e facilitar futuras manutenções e extensões da interface, os widgets anteriormente criados foram encapsulados em classes reutilizáveis, com nomes genéricos. Com isso, é possível instanciar componentes similares, como o gráfico de linha, os botões de visibilidade ou os elementos de inserção de datas, com apenas uma mudança de parâmetro, como o nome da métrica ou os limites do eixo Y. A estrutura modular resultante permite dividir os elementos gráficos e lógicos em classes armazenadas em arquivos separados, promovendo maior organização e reusabilidade dos algoritmos.

O primeiro elemento a ser generalizado foi a classe que armazenava os dados em maps. Para isso, foi criada a classe `MetricData`, anteriormente referenciada como `DataStore`, responsável por armazenar todos os mapas utilizados na construção dos widgets: dados em tempo real e histórico, rótulos de tempo formatados, timestamps em Unix Epoch e controle de visibilidade das linhas. A Figura 23 apresenta a definição dessa classe, incluindo o método que realiza a reinicialização completa de seus dados.

Para que diferentes métricas possam ter seus próprios conjuntos de dados isolados, foi implementado um gerenciador de instâncias denominado `MetricDataManager`. Esse gerenciador segue o padrão *singleton por chave*, onde cada métrica é identificada por uma string (como `"cpu"` ou `"gpu"`), e associada a uma única instância da classe `MetricData`. Dessa forma, ao solicitar uma instância por nome, o sistema garante que os dados daquela métrica sejam sempre recuperados ou manipulados de forma consistente. A Figura 24 ilustra essa estrutura.

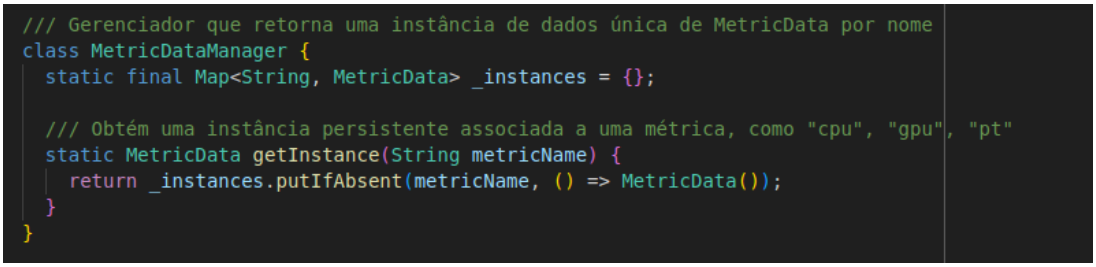
Figura 23 – Classe MetricData



```dart
/// Classe que armazena dados de uma métrica (CPU, GPU, TP etc.) [DataStore]
class MetricData {
  final Map<String, List<FlSpot>> realData = {};
  final Map<String, List<FlSpot>> histData = {};

  final Map<String, List<String>> timeLabelsReal = {};
  final Map<String, List<String>> timeLabelsHist = {};

  final Map<String, List<int>> timeEpochsReal = {};
  final Map<String, List<int>> timeEpochsHist = {};

  final Map<int, String> servicesList = {};
  final Map<String, bool> lineVisibility = {};

  /// Reseta completamente o datastore, deixando todos os mapas vazios.
  /// chama-se com DataStore.reset();
  void reset() {
    realData.clear();
    histData.clear();
    timeLabelsReal.clear();
    timeLabelsHist.clear();
    timeEpochsReal.clear();
    timeEpochsHist.clear();
    servicesList.clear();
    lineVisibility.clear();
  }
}
```

Fonte: Elaborado pelo autor.

Figura 24 – Gerenciador de instâncias de MetricData



```dart
/// Gerenciador que retorna uma instância de dados única de MetricData por nome
class MetricDataManager {
  static final Map<String, MetricData> _instances = {};

  /// Obtém uma instância persistente associada a uma métrica, como "cpu", "gpu", "pt"
  static MetricData getInstance(String metricName) {
    return _instances.putIfAbsent(metricName, () => MetricData());
  }
}
```

Fonte: Elaborado pelo autor.

O gráfico de linha desenvolvido e aprimorado nas seções anteriores também foi encapsulado em uma classe reutilizável denominada `MetricChart`. Essa classe recebe como parâmetros de entrada uma instância de `MetricData`, o sufixo de unidade da métrica (como “%” ou “°C”) e, opcionalmente, os valores mínimo e máximo do eixo Y. Caso os limites não sejam definidos, o widget `LineChart`, da biblioteca `fl_chart`, ajusta automaticamente os extremos com base nos dados plotados.

Essa abordagem permite construir gráficos de diferentes métricas com o mesmo código-base, bastando alterar os parâmetros de entrada. A Figura 25 exemplifica o uso da classe `MetricChart` para visualizar os dados da instância de CPU, obtida por meio do `MetricDataManager`. O trecho também ilustra como é feita a associação entre o gráfico e o respectivo `dataStore`.

Figura 25 – Uso da classe MetricChart

```
// Cria o DataStore (metrics.dart)
final cpuDataStore = MetricDataManager.getInstance("cpu");
```

```
// Gráfico de linha (linechart.dart)
MetricChart(
  dataStore: cpuDataStore,
  unitSuffix: '%',
  minY: 0,
  maxY: 100,
), // MetricChart
```

Fonte: Elaborado pelo autor.

Outros componentes passaram pelo mesmo processo de encapsulamento, como os botões de visibilidade e os campos de entrada de intervalo de datas. Já elementos mais simples (por exemplo, o campo de texto e o botão para inclusão manual de serviços) permaneceram declarados de forma direta, pois seu baixo grau de complexidade não justificava a criação de uma classe própria.

As rotinas de comunicação com o RabbitMQ para dados em tempo real, histórico e requisição de histórico mantiveram a lógica anterior. Como ainda não há serviços de coleta definitivos, preservar as funções “especializadas” facilita a adaptação futura: cada conexão pode ser ajustada para o formato exato de `JSON` que vier a ser adotado pelo serviço responsável pela métrica em questão, uma vez que as métricas podem

ser extraídas de diferentes fontes como o Docker, Kubernetes, Zipkin, Prometheus e etc que disponibilizarão diferentes formatos de JSON.

Por fim, a maior parte do código originalmente dedicado à tela de CPU foi refatorada para utilizar as novas classes genéricas. A reconstrução da interface, agora baseada nesses componentes reutilizáveis, produz exatamente o mesmo resultado apresentado na Figura 22. A partir desse ponto, criar uma nova tela de monitoramento para qualquer outra métrica torna-se um processo essencialmente repetitivo: obter a instância de `MetricData`, configurar as filas correspondentes e ajustar os limites do eixo Y. Isso diminui significativamente o esforço de desenvolvimento e garante consistência visual em todo o sistema.

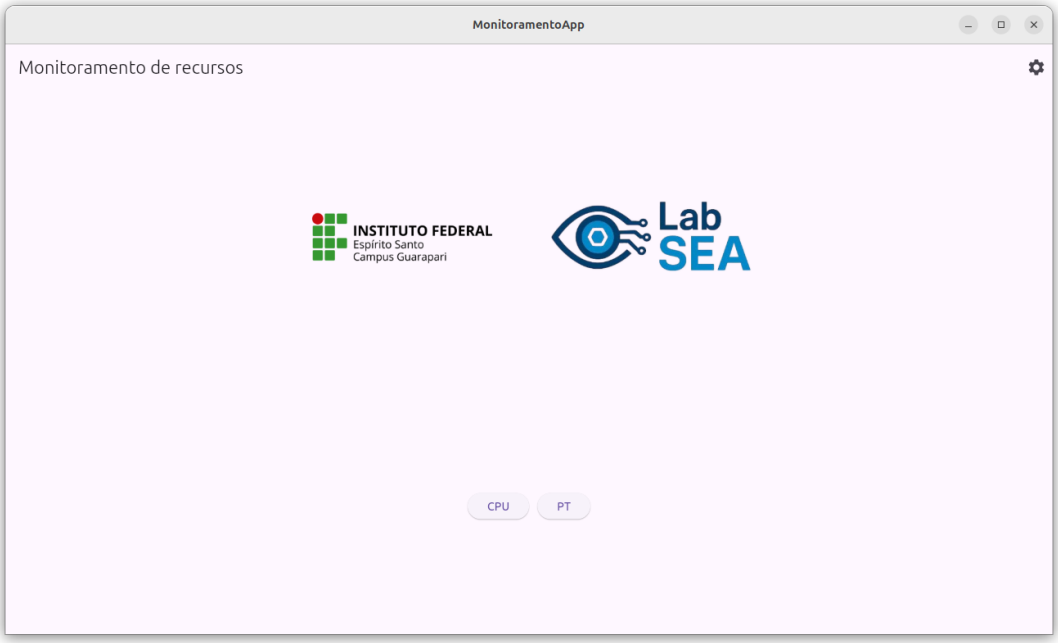
Antes de iniciar a criação das telas dedicadas ao monitoramento de outras métricas, foi desenvolvida uma tela inicial para a aplicação, funcionando como um menu de navegação. Essa tela permite ao usuário escolher qual métrica deseja visualizar. A composição da tela é simples e direta: ela exibe a logo do IFES - Campus Guarapari, uma imagem representando o LabSEA (laboratório onde o PIS está localizado), um botão de acesso à tela de configurações e, logo abaixo, uma linha com os botões de navegação entre as diferentes métricas disponíveis. A Figura 26 ilustra a tela de menu descrito.

A tela de configurações permite modificar o endereço IP que será utilizado nas conexões com o servidor RabbitMQ. Essa funcionalidade oferece maior flexibilidade e autonomia para testes em diferentes ambientes, sem a necessidade de alteração do código-fonte para essa finalidade. O valor inserido é utilizado diretamente pelas funções responsáveis pela comunicação com as filas de dados.

Também é possível ajustar o parâmetro `maxPoints`, que define o número máximo de pontos exibidos simultaneamente nos gráficos. Esse controle é útil para adaptar a visualização conforme a necessidade.

Além dos campos de entrada, a tela conta com dois botões adicionais: um para confirmar e aplicar as alterações realizadas, e outro para redefinir os parâmetros para seus valores padrão, restaurando o IP para `localhost` e o `maxPoints` para 60. A figura

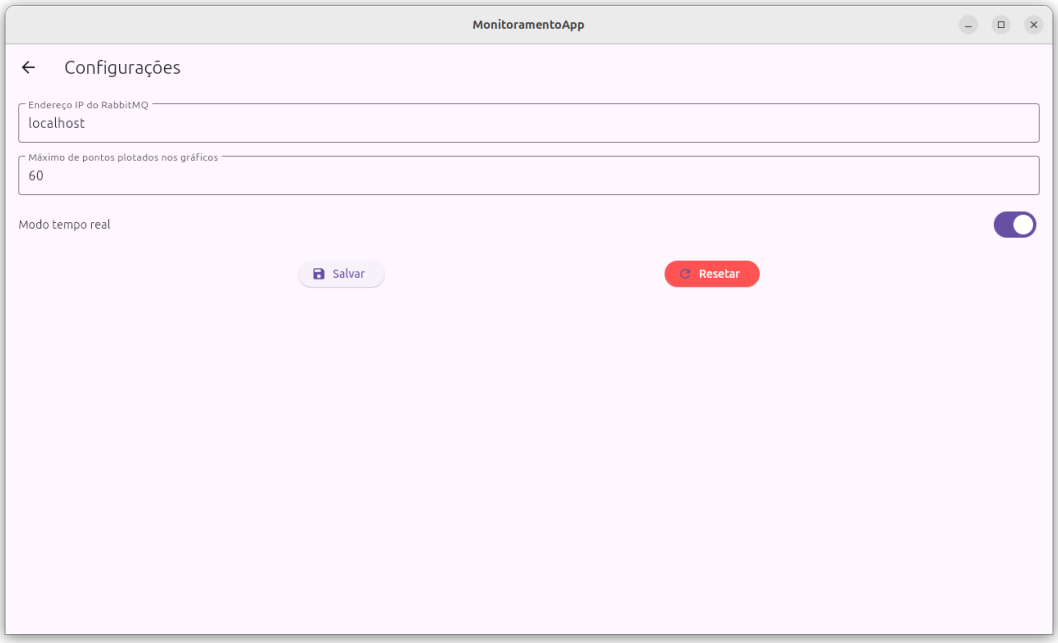
Figura 26 – Tela do menu inicial



Fonte: Elaborado pelo autor.

27 ilustra a tela de configurações desenvolvida.

Figura 27 – Tela de configurações



Fonte: Elaborado pelo autor.

Por fim, os botões exibidos na tela inicial permitem ao usuário navegar diretamente para as telas de monitoramento específicas de cada métrica. Cada uma dessas telas será baseada na mesma estrutura desenvolvida para a CPU, mas adaptada para as

particularidades da métrica em questão, como nomes das filas RabbitMQ, limites do eixo Y, unidade de medida e DataStore utilizado. Essa arquitetura modular torna o processo de expansão do sistema mais rápido e o código mais simples de se entender.

3.2.6 Múltiplas métricas

Com os componentes encapsulados na seção anterior, criar novas telas de monitoramento tornou-se essencialmente um exercício de configuração. Para cada nova métrica, basta duplicar o código-base da tela de CPU e ajustar poucos parâmetros, mantendo toda a infraestrutura de widgets, navegação e armazenamento de dados. As adaptações necessárias são listadas a seguir.

1. **Filas RabbitMQ.** O esquema de nomes segue o padrão `Data.`*`SUFIXO`*, `Hist.`*`SUFIXO`* e `HistRequest.`*`SUFIXO`*. Para GPU as filas tornam-se `Data.GPU`, `Hist.GPU` e assim por diante; para tempo de processamento, `TP`; para frames por segundo, `FPS`, etc.

2. **Instância exclusiva de** `MetricData`**.** Cada métrica recebe seu próprio datastore, obtido por `MetricDataManager.getInstance("gpu")`, `"tp"`, `"fps"`... O nome da variável segue o mesmo padrão (`gpuDataStore`, `tpDataStore`...). A Figura 25 mostra, como exemplo, a declaração da instância para CPU.

3. **Tratamento do JSON no modo tempo real.** A conexão de dados em tempo real (`Data.`*`SUFIXO`*) pode ser ajustada ao formato bruto emitido pelo serviço de coleta (Docker, Zipkin, Kubernetes ou outro). Como as duas filas de histórico são atendidas por serviços intermediários próprios, estas já obedecerão ao mesmo esquema de chaves definido para CPU, dispensando adaptações adicionais.

4. **Título da *AppBar*.** Deve indicar claramente a métrica monitorada, por exemplo ‘‘`Monitoramento de GPU`’’.

5. **Parâmetros do** `MetricChart`**.** É necessário informar:

   - o datastore correspondente;
   - o sufixo de unidade (“%”, “°C”, “ms”, “Hz”, etc.);

- `minY` e `maxY` quando apropriado (por exemplo, 0–100 % para CPU/GPU).

Fora esses itens, todo o restante do código permanece inalterado, assegurando consistência visual e reduzindo o esforço de desenvolvimento.

Foram selecionadas seis métricas consideradas importantes para aplicações no PIS: uso de CPU, uso de GPU, tempo de processamento (TP), *frames per second* (FPS) para aplicações de visão computacional, erro de trajetória (ET) e número de instâncias ativas do serviço (NI). Para validar as novas telas, criaram-se serviços simuladores análogos ao descrito na Seção 3.2.3, emitindo dados sintéticos para cada fila correspondente. As interfaces resultantes reproduzem o mesmo comportamento funcional demonstrado anteriormente para CPU, comprovando a eficácia da abordagem generalizada.

Para dar mais liberdade ao usuário, foi criada uma tela personalizada (tela CUSTOM) que permite configurar dinamicamente os parâmetros de monitoramento, como nomes das filas RabbitMQ, campos relevantes do JSON e limites do eixo Y do gráfico. Embora o armazenamento de dados ainda utilize uma instância genérica de `MetricData`, essa interface oferece um nível elevado de flexibilidade, tornando possível monitorar métricas não previstas, desde que compatíveis com o formato esperado. Essa funcionalidade é especialmente útil durante fases de testes e desenvolvimento, quando é comum experimentar diferentes serviços de coleta de métricas, nomes de filas e estruturas de dados. Assim, o próprio usuário pode ajustar a aplicação para consumir e visualizar qualquer métrica disponível, sem necessidade de alterações no código-fonte. Além de expandir o escopo da ferramenta, essa abordagem reduz o tempo necessário para validar novas fontes de informação no ambiente do PIS.

Algumas métricas, como o uso de CPU, podem ser geradas por serviços que monitoram a aplicação como um todo (Docker, por exemplo, o qual enxerga apenas containers inteiros), sem distinguir entre diferentes componentes ou serviços internos. Nesses casos, o campo `service_name` no JSON recebido pode estar ausente ou nulo. Para lidar com essa possibilidade e evitar erros de execução, as funções responsáveis pelo recebimento de dados em tempo real foram ajustadas para atribuir automaticamente o valor `"APP"` sempre que esse campo não estiver presente ou for inválido. Esse

tratamento garante a robustez do sistema frente a formatos de mensagens incompletos, mantendo a consistência da visualização e evitando falhas na associação dos dados ao gráfico correspondente.

3.2.7 Docker

Com a aplicação devidamente construída e funcional para o monitoramento de múltiplas métricas, o próximo passo foi sua containerização. Essa etapa tem como objetivo preparar a interface para integração ao ecossistema do PIS, garantindo portabilidade, escalabilidade e facilidade de implantação, especialmente em ambientes baseados em orquestração como o Kubernetes.

Para isso, a aplicação Flutter foi adaptada para rodar em sua versão Web, utilizando como base um *Dockerfile* dividido em duas fases. A primeira fase, baseada em uma imagem Debian, realiza o processo de instalação do Flutter, checkout da versão específica (3.27.3), cópia do código-fonte da aplicação e execução da compilação para Web usando o comando `flutter build web`. Já a segunda fase utiliza uma imagem leve do Nginx para servir os arquivos estáticos gerados, copiando o conteúdo da pasta `build/web` para o diretório padrão do servidor HTTP.

A escolha pelo Nginx como Proxy Web garante compatibilidade com práticas comuns de distribuição e facilita o roteamento interno no PIS, além de possibilitar testes locais e implantação em qualquer máquina com Docker instalado.

O Dockerfile utilizado para a construção da imagem pode ser visualizado na Figura 28.

Figura 28 – Dockerfile para construção da interface Flutter em Web

Dockerfile > ...

```
# Environemnt to install flutter and build web
FROM debian:latest AS build-env
# install all needed stuff
RUN apt-get update
RUN apt-get install -y curl git unzip
# define variables
ARG FLUTTER_SDK=/usr/local/flutter
ARG FLUTTER_VERSION=3.27.3
ARG APP=/app/
#clone flutter
RUN git clone https://github.com/flutter/flutter.git $FLUTTER_SDK
# change dir to current flutter folder and make a checkout to the specific version
RUN cd $FLUTTER_SDK && git fetch && git checkout $FLUTTER_VERSION
# setup the flutter path as an enviromental variable
ENV PATH="$FLUTTER_SDK/bin:$FLUTTER_SDK/bin/cache/dart-sdk/bin:${PATH}"
# Start to run Flutter commands
# doctor to see if all was installes ok
RUN flutter doctor -v
# create folder to copy source code
RUN mkdir $APP
# copy source code to folder
COPY . $APP
# stup new folder as the working directory
WORKDIR $APP
# Run build: 1 - clean, 2 - pub get, 3 - build web
RUN flutter clean
RUN flutter pub get
RUN flutter build web
# once heare the app will be compiled and ready to deploy
# use nginx to deploy
FROM nginx:1.25.2-alpine
#FROM nginx:alpine
# copy the info of the builded web app to nginx
COPY --from=build-env /app/build/web /usr/share/nginx/html
# Expose and run nginx
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]

```

Fonte: Elaborado pelo autor.

Com a imagem devidamente construída e executada, a aplicação pôde ser acessada via navegador. A interface manteve algumas funcionalidades esperadas, permitindo a visualização dos gráficos e interação com os controles e navegação entre telas.

A próxima etapa do experimento consistiu em estabelecer a comunicação entre a aplicação containerizada e o serviço RabbitMQ, a fim de permitir o recebimento de dados em tempo real diretamente no ambiente do Docker. Para isso, foi criada uma rede virtual Docker de modo a garantir que os contêineres envolvidos pudessem se comunicar internamente. A aplicação Flutter Web foi então executada nessa rede permitindo o acesso externo via navegador. Em paralelo, foi iniciado um contêiner com o RabbitMQ utilizando a imagem oficial `rabbitmq:4-management` na mesma rede.

A conectividade entre os dois contêineres foi validada com sucesso por meio de testes de `ping` entre os containers, confirmando que os serviços estavam visíveis entre si dentro da rede. No entanto, apesar da comunicação de baixo nível estar operante, a aplicação Flutter Web não conseguiu estabelecer conexão com o RabbitMQ via protocolo AMQP, resultando em falhas na publicação e consumo de mensagens.

Após uma análise mais aprofundada, foi identificado que essa limitação decorre de uma restrição inerente ao modelo de segurança dos navegadores modernos: aplicações Web executadas no navegador não possuem acesso direto a protocolos de rede como AMQP (porta 5672), que operam fora do escopo do protocolo HTTP/S. Isso significa que, mesmo com a aplicação corretamente containerizada e operando em ambiente Docker, qualquer tentativa de conexão direta ao RabbitMQ, como foi programado, falhará em Web.

Tendo o problema reconhecido, foram identificadas três soluções possíveis para contornar a limitação imposta pelos navegadores Web quanto à comunicação direta com o protocolo AMQP:

1. Criação de um serviço intermediário HTTP-AMQP. A solução mais robusta e compatível com a arquitetura Web consiste na criação de um serviço intermediário (também chamado de *bridge* ou *backend gateway*), que se comunique diretamente com o RabbitMQ utilizando o protocolo AMQP e, simultaneamente, ofereça uma interface HTTP ou WebSocket para a aplicação Flutter Web. Esse serviço atuaria como ponte entre o front-end e o broker de mensagens, traduzindo as requisições HTTP da interface para comandos AMQP, e vice-versa. Na prática, esse intermediário pode ser desenvolvido em linguagens como Python (utilizando bibliotecas como `aio-pika` e `FastAPI`) ou Node.js (com `amqplib` e `Express/WebSocket`), e hospedado como um microserviço adicional no ambiente Docker ou Kubernetes. Essa abordagem exige uma reformulação parcial da aplicação Flutter, substituindo a lógica de comunicação baseada em AMQP por chamadas HTTP (REST ou WebSocket), porém garante compatibilidade total com navegadores e mantém o acesso seguro e controlado aos dados.

2. Construção da interface como aplicação nativa em Docker (Flutter Desktop). Outra alternativa é abandonar a versão Web e compilar a aplicação Flutter como um

executável nativo (por exemplo, para Linux), de modo que ela possa ser empacotada como uma imagem Docker padrão sem as restrições impostas pelos navegadores. Nesse cenário, o contêiner da aplicação teria acesso pleno ao protocolo AMQP, e a lógica atual de comunicação com o RabbitMQ poderia ser mantida exatamente como está, sem necessidade de adaptações no código. A imagem resultante conteria o executável compilado, as bibliotecas necessárias e eventualmente um servidor X virtual (como `xvfb`) para permitir execução gráfica. Embora tecnicamente viável, essa abordagem exige mais recursos do sistema, pois o contêiner precisa suportar interfaces gráficas. Além disso, pode ser necessário configurar permissões adicionais para acesso ao ambiente gráfico da máquina host, caso o contêiner não seja executado em modo privilegiado. Ainda assim, essa solução é vantajosa em testes locais e protótipos avançados em que o uso do navegador não é uma exigência.

3. Distribuição do executável nativo para execução manual. É possível compilar a aplicação localmente gerando um executável para Linux, Windows ou outra plataforma, utilizando o comando `flutter build linux` (ou equivalente). Nesse modelo, o usuário final executaria o programa diretamente em sua máquina, sem necessidade de contêiner, e a comunicação com o RabbitMQ seria feita de forma direta, sem qualquer restrição de protocolo. Essa abordagem é a mais simples em termos de execução da lógica existente, pois aproveita todo o código da aplicação atual sem modificações. No entanto, traz consigo desafios de configuração manual: o usuário precisará inserir o endereço IP do RabbitMQ na interface, garantir que o broker esteja acessível na rede local e configurar manualmente permissões e dependências. Essa opção pode ser útil em ambientes controlados, como laboratórios de testes ou uso interno por desenvolvedores, mas é menos indicada para implantação em larga escala ou ambientes heterogêneos.

Considerando o contexto da infraestrutura do PIS, a solução 1 configura-se como a alternativa mais adequada, pois permite manter a aplicação acessível via navegador, característica essencial para ambientes multiusuário e distribuídos como o do PIS. No entanto, essa abordagem exige uma reestruturação significativa da lógica de comunicação da interface, implicando em tempo adicional de estudo, desenvolvimento e validação do novo fluxo baseado em HTTP ou WebSocket.

Diante das limitações de tempo previstas para a conclusão deste projeto, optou-se por adotar a solução 3, baseada na distribuição manual do executável gerado pelo Flutter Desktop. Embora apresente limitações em termos de usabilidade e praticidade, especialmente por exigir configuração manual de comunicação entre o RabbitMQ e execução local pelo usuário, essa abordagem se mostrou viável do ponto de vista técnico e plenamente funcional com a arquitetura atual da interface. A adoção dessa solução viabilizou a entrega de uma interface funcional dentro do cronograma, ao mesmo tempo em que mantém aberto o caminho para futuras evoluções baseadas em arquitetura web com intermediário HTTP.

4 RESULTADOS

Como resultado do processo de desenvolvimento descrito neste trabalho, foi construída uma interface gráfica de monitoramento modular e funcional, projetada especificamente para o ecossistema do PIS, mas com aplicabilidade geral para qualquer sistema que emita dados de séries temporais em formato JSON. A aplicação é totalmente personalizável e independente da métrica monitorada, sendo capaz de adaptar-se a diferentes cenários sem necessidade de alterações no código-fonte. Isso foi possível graças à abstração da lógica de monitoramento em componentes reutilizáveis, configuráveis por parâmetros e ao tratamento prevendo diferentes fontes de dados.

A comunicação com os serviços de coleta foi viabilizada por meio do RabbitMQ, já utilizado na infraestrutura do PIS, o que garantirá compatibilidade com o sistema existente. A interface foi projetada para operar alternativamente com dados JSON brutos vindos de serviços de extração de métricas dentro do sistema distribuído, dispensando a necessidade obrigatória de um banco de dados ou formatações intermediárias. Para cada mensagem recebida contendo métricas, a aplicação identifica automaticamente o nome do serviço, a métrica e o instante de tempo, convertendo essas informações em pontos de um gráfico de linha. A arquitetura foi estruturada para permitir o recebimento simultâneo de dados de múltiplos serviços ou não, com a renderização de uma linha por serviço monitorado, e com o suporte à visualização histórica mediante integração com serviços externos que realizam consultas a banco de dados.

A aplicação suporta dois modos principais de operação: monitoramento em tempo real e visualização de histórico. Embora os serviços de histórico devam ser desenvolvidos separadamente, a interface já possui suporte para consumi-los de forma transparente, desde que sigam o mesmo padrão de mensagens utilizado nos serviços simulados de consulta a banco de dados. Essa separação de responsabilidades entre interface e provedores de dados garante maior flexibilidade.

Além disso, foi implementada uma tela com um gráfico totalmente personalizável (CUSTOM), que permite ao usuário alterar os parâmetros de monitoramento, como nomes de filas RabbitMQ, campos do JSON a serem interpretados e limites dos eixos dos gráficos. Com isso, é possível visualizar métricas que não foram previstas

inicialmente no projeto, confirmando a genericidade. A interface não impõe restrições quanto à nomenclatura ou estrutura interna dos dados, desde que respeitado o formato base (JSON contendo uma métrica, timestamp e nome de serviço). Esse recurso torna a aplicação útil não apenas como solução final, mas também como ferramenta de prototipação e testes para novos serviços e métricas.

Por fim, apesar do sucesso funcional da aplicação, algumas limitações foram observadas no processo de implantação. A versão Web foi corretamente construída e empacotada via Docker, com uso do Nginx como servidor HTTP. No entanto, não foi possível estabelecer comunicação direta com o RabbitMQ por meio do protocolo AMQP, devido às restrições impostas por navegadores modernos, que impedem conexões diretas com protocolos não baseados em HTTP/S. Com isso, a aplicação não pôde ser executada de forma acessível por outros dispositivos via navegador, limitando sua utilização prática no contexto do PIS. A solução adotada foi a execução manual do binário nativo em ambiente local, o que garante pleno funcionamento e possibilita a validação da arquitetura proposta, ainda que com menor nível de acessibilidade para usuários finais.

Em síntese, a interface construída se mostrou funcional, flexível e extensível. Seu projeto permite a integração simples com novos serviços de coleta, uso direto de dados emitidos em JSON, e configuração dinâmica de todos os parâmetros relevantes. Ainda que não tenha sido implantada em ambiente Web de produção, sua arquitetura modular está preparada para isso, restando apenas adequações pontuais de comunicação para viabilizar sua operação em larga escala.

4.1 ESTUDOS DE CASO

A fim de validar a utilidade da aplicação desenvolvida, foram simulados estudos de caso onde a interface seria potencialmente eficaz no monitoramento de dados.

4.1.1 Erro de trajetória x FPS

Com o objetivo de demonstrar a aplicabilidade da interface desenvolvida em um cenário representativo de aplicações no PIS, foi conduzido um estudo de caso inspirado no

experimento descrito por (CARMO, 2021). No experimento original, um robô móvel percorre uma trajetória pré-definida (lemniscata de Bernoulli) com o tempo de execução fixado em 25 segundos por volta, e sob diferentes taxas de quadros por segundo (FPS) da câmera utilizada para o controle visual da movimentação.

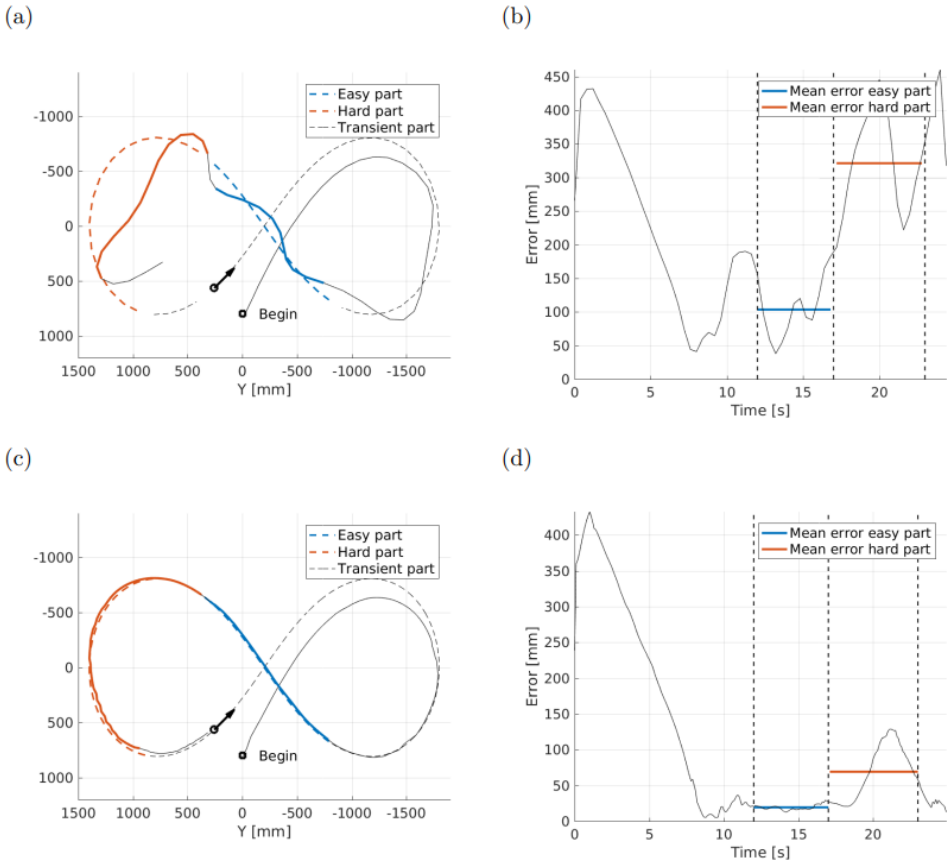
A proposta central do experimento era avaliar a influência do FPS no erro de trajetória do robô. Foram testadas duas taxas: 2,5 FPS e 10 FPS. Observou-se que, sob uma frequência menor de captura (2,5 FPS), o robô apresentava maiores desvios em relação à trajetória ideal, sobretudo nos trechos mais curvos do percurso. Isso ocorre porque uma menor frequência de amostragem aumenta a janela temporal entre quadros (400 ms), reduzindo a capacidade do sistema de reagir rapidamente a mudanças de posição. Com a elevação para 10 FPS (intervalo de 100 ms), a resposta do sistema de controle torna-se mais eficiente, resultando em um erro médio inferior a 50 mm contra cerca de 100 mm com 2,5 FPS e picos de erro reduzidos de 330 mm para 80 mm nas regiões mais complexas da trajetória. Os resultados desse experimento são apresentados na Figura 29.

Com base nesse contexto, foi elaborado um experimento complementar utilizando a interface de monitoramento desenvolvida neste trabalho. Para isso, foi criado um serviço em Python responsável por simular a publicação, via RabbitMQ, de dados sintéticos representando coletas de FPS, erro de trajetória, tempo de processamento, velocidade do robô, uso de CPU e de GPU em tempo real. Esses dados foram então consumidos e visualizados através da interface, permitindo a observação dessas variáveis sob diferentes configurações operacionais.

A visualização em tempo real permitiu analisar, de forma clara, a correlação entre o FPS e outras métricas do sistema. Ficou evidente que a redução do FPS tende a aumentar o erro de trajetória, conforme esperado, mas também afeta o tempo de processamento e o uso de GPU. A interface ainda permite verificar se o tempo de processamento por quadro extrapola o intervalo entre capturas (1/FPS), o que indicaria um gargalo computacional. A análise dessas variáveis é essencial para a definição do melhor FPS para a aplicação, de forma a garantir a performance do sistema.

Embora a métrica de velocidade do robô não estivesse prevista inicialmente no escopo

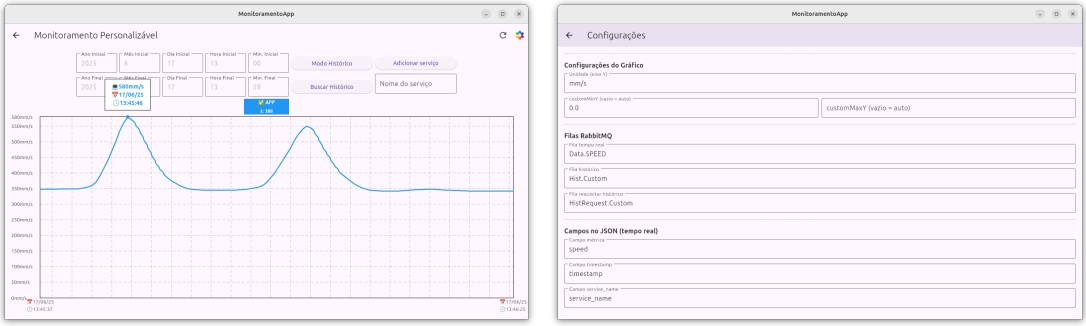
Figura 29 – Trajetória realizada e o erro para 2.5 FPS (a,b) e 10 FPS (c,d).



Fonte: (CARMO, 2021).

da interface, foi possível monitorá-la por meio da tela CUSTOM, apresentada na Seção 3.2.6. Essa tela permite configurar dinamicamente a métrica desejada, definindo parâmetros como unidade de medida (ex.: mm/s), fila de escuta (ex.: Data.SPEED) e formato de mensagem. A Figura 30 ilustra a configuração e exibição dos dados de velocidade pela interface, inclusive demonstrando o tratamento de campos nulos para service_name, que por padrão assume o valor "APP".

Figura 30 – Tela CUSTOM monitorando velocidade.



Fonte: Elaborado pelo autor.

Durante os testes, foram fixadas diferentes taxas de FPS para observar seus impactos sobre as demais variáveis. A análise gráfica revelou que taxas mais altas de FPS exigem maior poder computacional, com aumento significativo no uso de GPU. Essa elevação está diretamente relacionada à demanda por processamento mais frequente de imagens, refletindo no uso intensivo da GPU. As Figuras 31 e 32 comparam o comportamento do sistema sob 2,5 FPS e 10 FPS, com destaque para a redução média do erro de trajetória e o aumento proporcional do uso de GPU.

Figura 31 – Comparativo das métricas em 2.5FPS

Adicionalmente, foi realizada uma simulação com FPS fixado em 20 quadros por segundo. Nesse caso, observou-se que o sistema não conseguiu sustentar essa taxa. O tempo de processamento, que se mantinha estável, passou a superar o limite de 1/FPS (50 ms), o que levou a uma queda na taxa real de quadros para aproximadamente 16,7 FPS. Com essa taxa observa-se que há uma redução do erro de trajetória em comparação a 10 FPS após o robô se estabilizar no trajeto da lemniscata. Esse comportamento foi acompanhado de um pico de uso da GPU, que atingiu 100%, caracterizando a saturação computacional do sistema. A Figura 33 ilustra esse cenário.

Com base nesses resultados, determinou-se que uma taxa de 15 FPS pode ser o ideal no sistema utilizado. Nessa configuração, a GPU é amplamente utilizada, mas não chega à saturação completa, o tempo de processamento permanece dentro do limite

Figura 32 – Comparativo das métricas em 10FPS

Figura 33 – Tentativa de operação a 20 FPS e saturação da GPU.

de 1/FPS (66,6 ms) e o erro de trajetória é minimizado ao se estabilizar no percurso. A Figura 34 demonstra o monitoramento das métricas nesse cenário de 15 FPS.

Figura 34 – Comparativo das métricas em 10FPS

Este estudo de caso evidencia o potencial da interface como ferramenta de suporte à análise de desempenho em sistemas de controle visual. Ao permitir a visualização em tempo real e a comparação simultânea de múltiplas métricas, a aplicação se mostra não apenas como um painel de monitoramento passivo, mas como um recurso ativo no auxílio à tomada de decisões em relação a configurações e alocação de recursos para as aplicações em cenários reais no PIS.

4.1.2 Análise de Histórico

Em um cenário hipotético, levantou-se a necessidade de monitorar, ao longo de um dia completo, o comportamento do uso de GPU de uma aplicação de reconhecimento facial implantada no PIS. Essa aplicação, baseada em visão computacional, é alimentada por quatro câmeras distribuídas em diferentes pontos do espaço, cada uma associada a um serviço específico e independente, mas com estrutura e função idênticas: capturar imagens continuamente e processá-las em busca de rostos humanos conhecidos, comparando-os com uma base de dados predefinida.

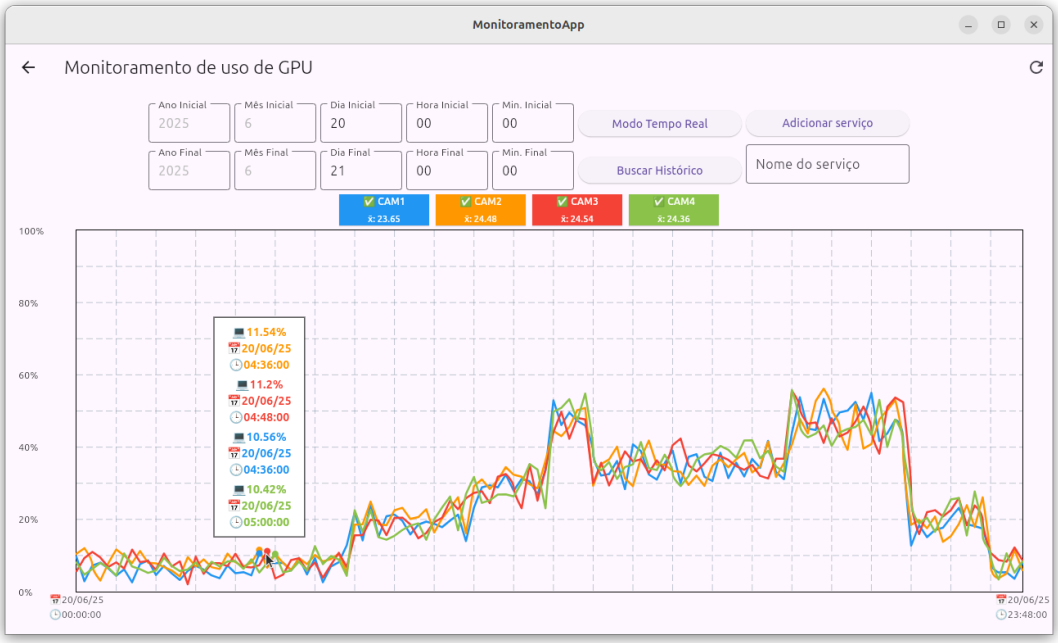
O experimento buscou observar como o uso de GPU variava em diferentes períodos do dia, refletindo a complexidade das imagens recebidas. Esperava-se, por exemplo, que durante a madrugada, quando o ambiente monitorado encontra-se vazio e com pouca ou nenhuma iluminação, o processamento fosse menos exigente, resultando em menor utilização da GPU. Por outro lado, durante os períodos de maior circulação de pessoas, como nos turnos matutino e vespertino, seria natural observar picos de uso, devido ao aumento no número de rostos capturados, rastreados e comparados pela aplicação.

Para viabilizar essa análise, foi implementado um serviço responsável por armazenar periodicamente os valores de uso de GPU de cada um dos quatro serviços em um banco de dados. A estrutura do banco adotada para este experimento foi baseada em arquivos CSV, formato já apresentado na seção 3.2.2. Os dados coletados foram mantidos com resolução suficiente para permitir a análise histórica de um dia.

Complementando essa estrutura, um serviço intermediário foi desenvolvido para atuar como ponte entre a interface de monitoramento e o repositório de dados. Esse serviço escuta requisições publicadas pela interface na fila HistRequest.GPU, contendo um JSON com três campos: timestamp_inicial, timestamp_final e uma lista com os identificadores dos serviços que se deseja consultar. Ao receber a requisição, o serviço filtra o banco de dados e publica a resposta formatada na fila Hist.GPU, da qual a interface consome os dados, trata e os exibe ao usuário.

A Figura 35 apresenta um exemplo do gráfico gerado pela interface ao se requisitar o histórico de um dia completo para os quatro serviços associados às câmeras.

Figura 35 – Histórico diário de uso da GPU por quatro serviços de reconhecimento facial.



O gráfico revela padrões semelhantes de uso entre os serviços, com aumentos pela manhã e à tarde, horários que coincidem com os períodos letivos dos usuários do PIS. Isso sugere uma correlação direta entre a movimentação de pessoas no espaço e a demanda computacional das aplicações. Durante a noite, observa-se uma leve redução no uso de GPU, reflexo da diminuição da circulação no laboratório. Já na madrugada, o uso se estabiliza em patamares mínimos, o que pode ser atribuído à ausência de movimento e ao fato de as imagens capturadas serem predominantemente escuras e de fácil descarte pela lógica da aplicação.

A interface aplica automaticamente a técnica de downsampling ao receber o grande volume de dados do histórico de um dia inteiro. Essa técnica calcula médias por blocos de dados, suavizando variações abruptas e melhorando a legibilidade do gráfico. No entanto, é possível configurar a quantidade de pontos limite na tela de configurações da aplicação. A Figura 35 ilustra os dados de um dia inteiro com o número de pontos configurado para 120, permitindo uma visualização mais detalhada das oscilações ao longo do dia.

Além de identificar tendências gerais, a análise histórica também pode revelar comportamentos anômalos, como um uso de GPU incompatível com o horário ou divergente

entre serviços idênticos. A possibilidade de monitoramento individual por serviço, associada à visualização comparativa simultânea, contribui para diagnósticos e decisões de manutenção, redistribuição de carga ou ajuste na lógica de funcionamento da aplicação.

Esse experimento reforça a utilidade da interface desenvolvida como ferramenta analítica não apenas para observação em tempo real, mas também para investigações retroativas. A flexibilidade para consulta de períodos customizados, aliada à capacidade de agregar e visualizar múltiplas métricas, torna a solução proposta particularmente útil para aplicações que exigem controle de desempenho e otimização contínua de recursos computacionais.

5 CONCLUSÃO

O desenvolvimento da aplicação para monitoramento de recursos no PIS mostrou-se uma solução viável e eficaz para enfrentar os desafios relacionados à visualização e à gestão de métricas computacionais em sistemas distribuídos. A proposta apresentou uma interface gráfica modular, capaz de exibir dados em tempo real e históricos de múltiplos serviços de forma interativa, responsiva e personalizável.

A estrutura da aplicação, baseada na arquitetura do PIS do IFES - Campus Guarapari, foi projetada com foco em reutilização de componentes, flexibilidade na configuração e compatibilidade com tecnologias já adotadas no ecossistema, como RabbitMQ. A interface permite que diferentes métricas sejam monitoradas simultaneamente, promovendo um alto grau de abstração e independência quanto ao tipo de dado analisado e sua fonte.

Os experimentos realizados, incluindo o estudo de caso que relaciona erro de trajetória e FPS, e a análise histórica do uso de GPU ao longo do dia, evidenciaram o potencial da ferramenta como suporte à tomada de decisões. A possibilidade de ajustar parâmetros, consultar métricas específicas por serviço, aplicar técnicas de downsampling e exibir dados de maneira visualmente limpa e rica em informações, reforça a aplicabilidade da solução em ambientes reais que demandam supervisão e tomadas de decisão em relação à otimização de desempenho.

Apesar de restrições impostas pelo tempo e pelos recursos disponíveis, foi possível entregar uma aplicação funcional, mesmo sendo necessário ser distribuída via download para os usuários. A estrutura modular construída permite que, com ajustes pontuais na comunicação, a solução possa ser adaptada para implantação em larga escala via web ou integração com outros protocolos, como HTTP ou WebSocket.

Como proposta de trabalho futuro, destaca-se a criação de serviços reais de coleta de dados integrados às ferramentas utilizadas no PIS, como Zipkin, Prometheus, Docker e Kubernetes. Essa integração permitirá a obtenção de métricas diretamente das aplicações em execução, substituindo os dados simulados utilizados neste projeto. Com essa implementação, seria possível monitorá-los em tempo real através das

extrações brutas.

A implementação de serviços que consultem bancos de dados gerados pelas fontes das métricas poderá aprimorar ainda mais a funcionalidade histórica da aplicação, ampliando sua utilidade para análises de longo prazo e detecção de anomalias. Este serviço deverá seguir o padrão de requisição e entrega de histórico realizado pelo serviço de simulação de histórico apresentado na seção 3.2.2.

Além disso, a implementação de um sistema de alertas configuráveis poderá ampliar significativamente a utilidade da ferramenta. Essa funcionalidade permitiria que os usuários definissem limiares personalizados para cada métrica monitorada, acionando notificações automáticas quando esses limites forem ultrapassados. Os alertas poderiam ser enviados por diferentes canais, como e-mail, notificações web ou integrações com webhooks, contribuindo para antecipar falhas, automatizar respostas rápidas e aumentar a acessibilidade para usuários que não acompanham a interface continuamente. A configuração desses alertas, se realizada via interface gráfica, também manteria a proposta de usabilidade e flexibilidade da aplicação.

Conclui-se, portanto, que os objetivos propostos foram parcialmente atingidos. A aplicação desenvolvida representa uma contribuição relevante para o monitoramento e análise de sistemas computacionais em espaços inteligentes, oferecendo uma base sólida para futuras pesquisas e aprimoramentos.

O código-fonte da versão final do projeto está disponível no repositório GitHub ([1]) junto ao seu executável distribuído ([2]).

[1] <https://github.com/VitorBermond/Monitoramento_Flutter_IS/tree/master/3_final>
[2] <https://github.com/VitorBermond/Monitoramento_Flutter_IS/tree/master/4_executavelFinal>

REFERÊNCIAS

Alura. **O que é o projeto Kubernetes e para que ele serve?** 2024. Disponível em: <https://www.alura.com.br/artigos/o-que-e-kubernetes>.

CARMO, A. P. d. **Uma arquitetura de microsserviços centrada na observabilidade multinível para espaços inteligentes baseados em visão computacional**. Tese (Tese (Doutorado em Engenharia Elétrica)) — Universidade Federal do Espírito Santo, Vitória, 2021.

CONNOLLY, T. M.; BEGG, C. E. **Database Systems: A Practical Approach to Design, Implementation, and Management**. 6ª. ed. [S.l.]: Pearson, 2014.

COTTA, W. A. A. **Medição de tempo de comunicação e exibição de tempo de resposta para espaços inteligentes programáveis**. Dissertação (Dissertação (Mestrado em Engenharia Elétrica)) — Universidade Federal do Espírito Santo, Vitória, 2020.

DATE, C. J. **An Introduction to Database Systems**. 8ª. ed. [S.l.]: Addison-Wesley, 2004.

DEVASIA, A. **An Introduction to Supervisory Control and Data Acquisition (SCADA)**. 2020. Disponível em: <https://control.com/technical-articles/an-introduction-to-supervisory-control-and-data-acquisition-scada>.

ELMASRI, R.; NAVATHE, S. B. **Fundamentals of Database Systems**. 7ª. ed. [S.l.]: Pearson, 2017.

Flutter. **Build apps for any screen**. 2024. Disponível em: <https://flutter.dev/>.

GALITZ, W. O. **The essential guide to user interface design: an introduction to gui design principles and techniques**. [S.l.]: John Wiley & Sons, 2007.

Grafana Labs. **Panels and visualizations**. 2024. Disponível em: <https://grafana.com/docs/grafana/latest/panels-visualizations/>.

HUNT, J. Graphical user interfaces. In: ____. **Advanced Guide to Python 3 Programming**. Springer, 2023. Disponível em: <https://doi.org/10.1007/978-3-031-40336-1_16>.

JHA, R. *et al.* Artificial intelligence for waste management in smart cities: a review. **Environmental Chemistry Letters**, 2022. Disponível em: <https://link.springer.com/article/10.1007/s10311-023-01604-3>.

Microsoft Learn. **Connecting to external data in Power BI**. 2024. Disponível em: <https://learn.microsoft.com>.

Microsoft Power BI Blog. **Announcing new AI Capabilities for Power BI**. 2024. Disponível em: <https://powerbi.microsoft.com>.

Positivo Casa Inteligente. **Tecnologias smart: trazendo a combinação perfeita entre o ambiente físico e digital**. 2024. Disponível em: <https://blog.positivocasainteligente.com.br/fisico-digital-tecnologias-smart>.

PROMETHEUS. **Overview | Prometheus**. 2020. Acesso em: 02 nov. 2024. Disponível em: <https://prometheus.io/docs/introduction/overview/>.

QUEIROZ, F. M. d. **Desenvolvimento da Infraestrutura de um Espaço Inteligente baseado em Visão Computacional e IoT**. Monografia (Graduação) — Universidade Federal do Espírito Santo, Vitória, 2016.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. **Database System Concepts**. 6ª. ed. [S.l.]: McGraw-Hill, 2010.

ZLATANOVA, S. *et al.* Spaces in spatial science and urban applications—state of the art review. **ISPRS International Journal of Geo-Information**, v. 9, n. 1, p. 58, 2020.