UNIVERSIDADE DE CAXIAS DO SUL
ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

GUILHERME SIMIONI WASKIEVICZ

CONSTRUÇÃO DE UMA ARQUITETURA PARA INTEGRAÇÃO E ESCALABILIDADE DO SISTEMA EDUCACIONAL WEBALGO EM UM AMBIENTE WEB

BENTO GONÇALVES

2025

GUILHERME SIMIONI WASKIEVICZ

CONSTRUÇÃO DE UMA ARQUITETURA PARA INTEGRAÇÃO E ESCALABILIDADE DO SISTEMA EDUCACIONAL WEBALGO EM UM AMBIENTE WEB

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientador: Prof. Dr. Leonardo Pellizzoni

BENTO GONÇALVES

2025

GUILHERME SIMIONI WASKIEVICZ

CONSTRUÇÃO DE UMA ARQUITETURA PARA INTEGRAÇÃO E ESCALABILIDADE DO SISTEMA EDUCACIONAL WEBALGO EM UM AMBIENTE WEB

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Aprovado em 25/11/2025

BANCA EXAMINADORA

Prof. Dr. Leonardo Pellizzoni
Universidade de Caxias do Sul - UCS

Prof. Me. Alexandre Erasmo Krohn Nascimento
Universidade de Caxias do Sul - UCS

Prof. Dr. Ricardo Vargas Dorneles
Universidade de Caxias do Sul - UCS

AGRADECIMENTOS

A realização deste Trabalho de Conclusão de Curso representa o encerramento de uma jornada de muito aprendizado, dedicação e crescimento pessoal e profissional. Nada disso seria possível sem o apoio e a presença de pessoas que caminharam ao meu lado ao longo deste percurso.

Agradeço primeiramente aos meus pais, Jocemar e Elissandra, pelo amor incondicional, pelos ensinamentos e pela base sólida que sempre me ofereceram. São o exemplo de esforço, honestidade e perseverança que me inspira diariamente. À minha irmã, Manuela, por ser uma presença constante de alegria e ternura, lembrando-me, mesmo nos dias mais corridos, da importância das pequenas coisas. À minha namorada, Rafaela, pelo carinho, pela paciência e pelo incentivo constante, sua compreensão e apoio foram fundamentais para que eu seguisse firme até o fim desta etapa. Aos meus amigos, por cada palavra de encorajamento, pelas risadas e pela leveza que tornaram essa caminhada mais agradável.

Expresso também minha gratidão aos professores que compuseram a banca avaliadora, Prof. Me. Alexandre Erasmo Krohn Nascimento e Prof. Dr. Ricardo Vargas Dorneles, pela disponibilidade, pelas observações criteriosas e pelas contribuições que enriqueceram este trabalho. Um agradecimento especial ao meu orientador, Prof. Dr. Leonardo Pellizzoni, pelos valiosos aconselhamentos, pela dedicação e pela confiança depositada em mim durante todo o processo.

Agradeço, ainda, às turmas que participaram dos testes e contribuíram diretamente para a validação do projeto. A colaboração de cada participante foi essencial para que esta pesquisa se tornasse concreta e significativa.

Por fim, deixo meu sincero agradecimento a todos que, de alguma forma, contribuíram para esta conquista. Cada gesto de apoio e cada palavra de incentivo tiveram um papel importante na realização deste sonho.

“A educação é a arma mais poderosa que você pode usar para mudar o mundo.”

Paulo Freire

RESUMO

O constante avanço tecnológico e a crescente demanda por soluções flexíveis, acessíveis e alinhadas ao uso em plataformas *web* tem fortalecido a modernização de sistemas, principalmente os desenvolvidos em ambiente *desktop*. O presente trabalho propõe uma arquitetura que permite integrar um novo sistema *web* ao sistema *backend* do WebAlgo, utilizado no ensino de programação na Universidade de Caxias do Sul (UCS). A proposta busca viabilizar a comunicação entre a interface e o servidor consolidado, sem alterar a estrutura atual do servidor ou expor diretamente dados sensíveis. Dessa forma, a base da construção baseou-se na realização de pesquisas, levantamento de referências, análise de trabalhos similares e na definição de uma arquitetura composta por contêineres, utilizando Docker e NGINX, além da implementação do *middleware* responsável pela intermediação da comunicação entre o *front-end* e o servidor legado.

Os testes realizados demonstraram que a arquitetura proposta apresentou desempenho estável, mantendo baixa latência, alta disponibilidade e comportamento consistente mesmo sob diferentes níveis de carga. Observou-se que o uso do NGINX como *proxy* reverso e balanceador de carga contribuiu para uma distribuição eficiente das requisições, enquanto o *middleware* mostrou-se resiliente e capaz de intermediar a comunicação de forma segura e padronizada. Nos cenários de uso real em sala de aula, a aplicação manteve funcionamento contínuo e responsivo, evidenciando que a solução desenvolvida é adequada para o ambiente acadêmico e está preparada para ambientes locais e em nuvem.

Palavras-chave: Integração.Middleware.Modernização.NGINX.Docker.

ABSTRACT

The constant technological evolution and the growing demand for flexible, accessible, and web-aligned solutions have reinforced the modernization of systems, especially those originally developed for desktop environments. This work proposes an architecture that enables the integration of a new web-based system with the WebAlgo backend, used in programming education at UCS. The solution aims to establish communication between the new interface and the consolidated legacy server without modifying its structure or exposing sensitive data. The design was based on research, reference analysis, related work, and the definition of a container-based architecture using Docker and NGINX, in addition to implementing a middleware responsible for mediating communication between the front-end and the legacy system.

The conducted tests demonstrated that the proposed architecture delivered stable performance, maintaining low latency, high availability, and consistent behavior under different load levels. The use of NGINX as a reverse proxy and load balancer enabled efficient request distribution, while the middleware proved resilient and capable of securely and consistently handling communication. In real classroom scenarios, the application remained responsive and fully operational, indicating that the solution is suitable for academic environments and ready for both local and cloud deployments.

Keywords: Integration.Middleware.Modernization.NGINX.Docker.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

ACR Registro de Contêiner do Azure
ACI Azure Container Instances
AKS Azure Kubernetes Service
AWS Amazon Web Services
C3E Código de Três Endereços
CORS Cross-Origin Resource Sharing
CORBA Arquitetura de Corretor de Objetos de Requisição Comum
CPU Central Processing Unit
CSS Folhas de Estilo em Cascata
DMZ Zona Desmilitarizada
EC2 Amazon Elastic Compute Cloud
ECS Amazon Elastic Container Service
GUI Interface Gráfica do Usuário
HTML Linguagem de Marcação de Hipertexto
HTTP Protocolo de Transferência de Hipertexto
IaC Infraestrutura como Código
IDE Ambiente de Desenvolvimento Integrado
IDL Linguagem de Definição de Interface
JSON Notação de Objetos JavaScript
LIBRAS Língua Brasileira de Sinais
LTS Long-Term Support
MVC Modelo-Visão-Controlador
OCI Oracle Cloud Infrastructure
OKE Container Engine for Kubernetes
SOAP Protocolo Simples de Acesso a Objetos
SSL Secure Sockets Layer
UCS Universidade de Caxias do Sul
UML Linguagem de Modelagem Unificada
URL Localizador Uniforme de Recursos
XML Linguagem de Marcação Extensível
OPC UA Open Platform Communications Unified Architecture
CPS Sistemas Ciberfísicos
VUs Usuários virtuais simultâneos

SUMÁRIO

1 INTRODUÇÃO

Um dos principais desafios para estudantes que estão ingressando em cursos de Computação é o aprendizado de programação. Para melhorar esse processo, é essencial identificar e corrigir equívocos comuns dos alunos, exigindo dos professores, abordagens didáticas eficazes (QIAN; LEHMAN, 2017). Para auxiliar nesse processo, existem diversas plataformas, como o *Beecrowd*[1], *CodeHS*[2] e *Codecademy*[3], onde os alunos podem praticar e aprimorar suas habilidades de lógica e resolução de problemas (CRUZ, 2022; REIS *et al.*, 2015). Ainda, destacam-se sistemas que atuam no auxílio dos alunos de instituições de ensino, oferecendo um ambiente adaptado às necessidades dos estudantes (LIMA *et al.*, 2024; CAVALCANTE; SILVA; VITORINO, 2020; JARAMILLO-ALCAZAR *et al.*, 2018).

As disciplinas introdutórias de programação da Universidade de Caxias do Sul (UCS) utilizam o sistema Webalgo, desenvolvido por professores da universidade e amplamente empregado desde 2009. Essa ferramenta, originalmente desenvolvida para uso via web, utilizava a tecnologia Java Applet no cliente, a qual foi posteriormente descontinuada nos navegadores, preservando suas funcionalidades. Em razão dessa obsolescência, o sistema foi adaptado para o formato *desktop*. O WebAlgo permite que os alunos resolvam exercícios e compreendam conceitos fundamentais da linguagem C e português estruturado, oferecendo *feedbacks* relevantes durante a resolução das atividades (DORNELES; JR; ADAMI, 2010). O programa consolidou-se, ao longo do tempo, como um importante aliado no ensino dessas disciplinas. Todavia, à medida que os hábitos tecnológicos da população evoluíram, surgiram novos desafios. Segundo dados da PNAD Contínua (IBGE, 2023), o acesso à internet por dispositivos móveis tornou-se predominante no Brasil. Esse novo panorama, aliado à crescente expectativa por flexibilidade e acesso remoto, motivou a transição da versão *desktop* para uma interface *web* atualizada.

No intuito de aprimorar a ferramenta, algumas melhorias foram implementadas, como o desenvolvimento de um compilador e uma máquina virtual para um subconjunto da linguagem Python (MIOTTO, 2019). Embora a ferramenta tenha estabelecido as bases técnicas, o sistema ainda permitia o aprimoramento da sua interação com o usuário. Para isso, foi desenvolvida uma nova interface web, proporcionando uma nova experiência aos usuários.

Buscando modernizar a solução e torná-la mais acessível, um projeto recente reformulou a aplicação para um novo ambiente, preservando sua proposta educacional e introduzindo uma nova interface *web*, a qual não possui qualquer conexão com as APIs já existentes do WebAlgo. A atualização da arquitetura trouxe amplo acesso via navegadores *web*, eliminando a necessidade de instalação ou configuração de ambientes locais. Nesse contexto, o desenvolvi-

[1] <https://beecrowd.com/>
[2] <https://codehs.com/>
[3] <https://www.codecademy.com/>

mento do compilador abrangeu as três fases sequenciais de análise, a geração de código intermediário no formato Código de Três Endereços (C3E), atuando como um *bytecode*, e a definição da máquina virtual responsável por sua execução (SUSIN, 2024).

Atualmente, a versão *desktop* do WebAlgo realiza a comunicação com o servidor interno da UCS por meio de chamadas Protocolo de Transferência de Hipertexto (HTTP), através de um cliente HTTP utilizando Java, responsável pelo gerenciamento das sessões e requisições para ações de login, cadastro, criação e alteração de problemas, possibilitando a integração direta com as APIs do servidor. Dessa forma, torna-se viável a implementação de um *middleware* que viabilize a integração entre o novo sistema web e o servidor. Essa solução tem como finalidade garantir a continuidade do uso da ferramenta, preservar os dados históricos e assegurar a compatibilidade entre as plataformas, proporcionando um fluxo de dados eficiente e uma transição suave para o ambiente modernizado, a fim de otimizar a comunicação entre os sistemas sem expor os dados sensíveis do servidor interno da UCS.

1.1 QUESTÃO DA PESQUISA

Como integrar uma interface web a um *backend* consolidado não expondo dados do servidor diretamente?

1.2 OBJETIVO GERAL

Objetiva-se construir uma solução de integração entre um sistema *backend* consolidado e uma nova interface *web*, permitindo que ambos funcionem em conjunto, buscando estabelecer um canal de comunicação, sem que as informações importantes do servidor sejam acessíveis diretamente pela camada de apresentação. Desta forma, será adotada uma abordagem que permita a mediação entre os sistemas, mantendo a autonomia do *backend* e filtrando as múltiplas requisições.

1.2.1 Objetivos Específicos

1. Preservar as funcionalidades da ferramenta.
2. Planejar a autenticação dos usuários na nova interface.
3. Adotar estratégias de integração seguras
4. Implementar um mecanismo de mediação que faça a ponte entre o sistema legado e a nova interface

1.3 ESTRUTURA DO TRABALHO

O presente trabalho está organizado da seguinte forma:

- Este capítulo apresentou uma introdução sobre o trabalho, incluindo seu cenário, motivações, objetivo geral e objetivos específicos.
- O Capítulo 2 tem por objetivo apresentar conceitos teóricos sobre o desenvolvimento *client-server* e o uso de *proxy* como forma de integração entre plataformas heterogêneas.
- O Capítulo 3 apresenta uma pesquisa de revisão sistemática realizada para compreender o cenário do tema e quais os principais meios já utilizados para resolver problemas relacionados.
- O Capítulo 4 apresenta a proposta de solução e aborda os testes realizados para validar o funcionamento e aderência aos requisitos da arquitetura.
- O Capítulo 5 apresenta o desenvolvimento da solução proposta, descrevendo detalhadamente a arquitetura implementada, as tecnologias utilizadas e as etapas do processo de construção do sistema.
- O Capítulo 6 apresenta a conclusão do trabalho, destacando os resultados obtidos e possíveis direções para trabalhos futuros.

2 REFERENCIAL TEÓRICO

Este capítulo apresentará uma descrição do modelo metodológico adotado e das principais tecnologias utilizadas ao longo do desenvolvimento do projeto. A apresentação desses elementos é utilizada para explicar as decisões técnicas e garantir o embasamento necessário para a compreensão do trabalho.

2.1 METODOLOGIA

Considerando o que foi discutido, o trabalho tem como premissa desenvolver uma aplicação mediadora, com o propósito de integrar uma nova interface de aprendizado de programação ao banco de dados de uma aplicação existente. Com o objetivo de atingir esse propósito, a aplicação pretende empregar ferramentas associadas ao *back-end*. Nesse sentido, a linguagem Java e o uso de *proxy*, foram considerados como uma opção viável para a construção da aplicação.

Serão adotadas como etapas metodológicas o modelo ICONIX, que auxilia nos processos de modelagem e desenvolvimento do sistema. Esse modelo se divide em dois fluxos: estático e dinâmico. O fluxo estático é responsável por gerar artefatos ligados aos dados, como o modelo de domínio e o diagrama de classes. Por outro lado, o fluxo dinâmico foca na definição dos processos relacionados à interação dos usuários com o sistema, gerando os diagramas de casos de uso, robustez e sequência. Esses diagramas são representados por meio da Linguagem de Modelagem Unificada (UML) (ROSENBERG; STEPHENS, 2007). Por essa razão, o ICONIX foi escolhido, pois, além de gerar todos os artefatos necessários em cada fase da implementação, ele proporciona um processo iterativo e prático, o que facilita o trabalho do desenvolvedor. A Figura 1 ilustra o fluxo da metodologia ICONIX e os artefatos gerados.

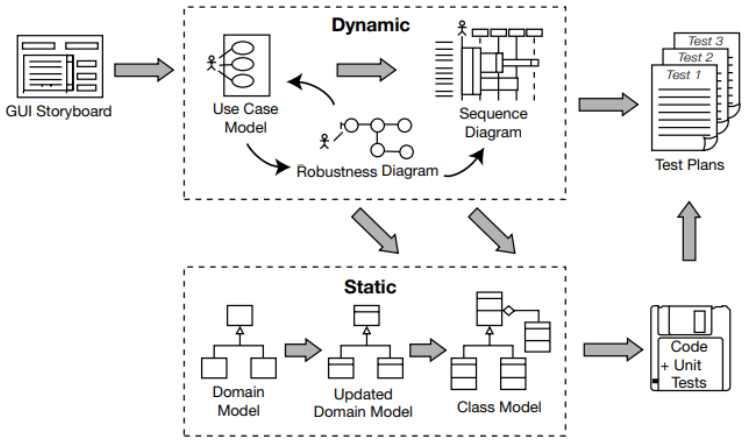


Figura 1 – Visão geral do processo ICONIX

Fonte: (ROSENBERG; STEPHENS, 2007)

2.2 MODELO CLIENTE-SERVIDOR

O termo cliente-servidor, no ambiente computacional, é um sistema no qual um cliente e um servidor interagem para viabilizar a comunicação entre si. Essa comunicação ocorre onde um agente, denominado cliente, realiza requisições para o outro agente, o servidor, este executa a demanda e retorna os resultados para o cliente (OLUWATOSIN, 2014). Esse padrão corresponde a uma arquitetura amplamente utilizada na *web*.

No contexto *web*, o *software* cliente, geralmente um navegador, se torna responsável por requisitar os recursos necessários por meio de uma Localizador Uniforme de Recursos (URL), aguardando, portanto, a resposta para apresentá-la adequadamente ao usuário final. Esse *software* pode estar presente em dispositivos como *notebooks* e *smartphones* (CONNOLLY; HOAR, 2015).

Já o servidor, por sua vez, pode ser visto como componente central do modelo, responsável pelo armazenamento de dados e pela hospedagem de aplicações *web*, entre outras funções. No entanto, sua principal característica está no recebimento de requisições, no processamento dessas e no retorno ao requisitante com algum recurso, como, por exemplo, um arquivo Linguagem de Marcação de Hipertexto (HTML). Atualmente existem diversos tipos de servidores, *web*, de aplicação, de *e-mail* e de banco de dados (CONNOLLY; HOAR, 2015).

Podemos dividir as ferramentas e necessidades do sistema em duas partes principais: *front-end* e *back-end*. O *front-end* corresponde ao Cliente, geralmente executado em um navegador *web*, enquanto o *back-end* está associado aos artefatos do Servidor. Na Seção 2.2.1 serão discutidas as características do *back-end*, enquanto na Seção 2.2.2 serão apresentados os aspectos relacionados ao *front-end*.

2.2.1 Back-end

Segundo Adam et al. (2019), o *back-end* diz respeito a *scripts* e aplicações que operam diretamente nos servidores, de forma transparente ao usuário final. Ele tem a responsabilidade de processar, armazenar e entregar dados ou serviços requisitados pelas aplicações *front-end*, certificando que essa comunicação entre o usuário e o sistema aconteça de forma organizada e segura. Em resumo, o *back-end* trabalha como núcleo lógico dos *softwares*, englobando o acesso ao banco de dados, integração de dados, lógica de negócios e também da implementação de APIs. Sua principal função é receber as requisições do *front-end*, operar sobre elas conforme as regras específicas, interagir com serviços ou banco de dados, e enviar as respostas adequadas de volta à interface de usuário.

Entretanto, é importante destacar que, o desenvolvimento *back-end* é composto por diferentes linguagens, como Java, PHP, Python, C++ e JavaScript, e os *frameworks* comumente utilizados como Express, Rails, Laravel, Django e Spring, que dão apoio às funcionalidades do sistema. É possível encontrar ampla variedade tecnológica para o desenvolvimento *back-end*,

evidenciando a complexidade e a abrangência no suporte à comunicação entre sistemas, suas interfaces e a manipulação de dados (DALMIA; CHOWDARY, 2020).

Conforme discutido pelos autores, o *back-end* realiza um papel importante na construção de sistemas, sendo responsável por comunicar com o banco de dados, servidores e pelo processamento da lógica de negócio e dados, sustentando de forma eficiente, segura e integra às funcionalidades da aplicação para o usuário (ADAM; BESARI; BACHTIAR, 2019; DALMIA; CHOWDARY, 2020).

2.2.2 Front-End

O desenvolvimento de interfaces *web*, chamado de *front-End*, corresponde à implementação da camada iterativa e visual das aplicações *web*. Diferentemente do *back-end*, explicado na seção anterior, o *front-end* trata-se da parte do sistema na qual o usuário final interage diretamente, incluindo *design*, estrutura, conteúdo, dados e funcionalidades. O *front-end* é responsável por traduzir o projeto visual de uma página *web* ou aplicação para o ambiente digital, por meio de Interface Gráfica do Usuário (GUI) e comandos em linha, agregando textos, menus de navegação, imagens, vídeos e outros componentes importantes de uma interface (DALMIA; CHOWDARY, 2020).

A criação de interfaces para usuários envolve o uso de diferentes linguagens, como HTML, Folhas de Estilo em Cascata (CSS) e JavaScript. Normalmente, trabalham junto de *frameworks* e bibliotecas que facilitam o desenvolvimento, melhorando a experiencia do desenvolvedor e que aprimoram a do usuário final, alguns deles são: AngularJS, jQuery, ReactJS e SASS. Esses recursos disponibilizam para o desenvolvedor formas mais dinâmicas de criação de páginas, melhorando a iteratividade e responsividade, propriedades fundamentais para atender as necessidades atuais dos usuários e dos dispositivos modernos (DALMIA; CHOWDARY, 2020).

Assim, o *front-end* desempenha um papel fundamental no desenvolvimento de sites, focando na interação funcional e na apresentação visual que conecta o usuário final à interface de forma acessível, funcional e clara.

2.3 MIDDLEWARE

O termo *middleware* é amplamente conhecido no desenvolvimento de sistemas, destacando-se como uma camada intermediaria dinâmica e versátil, capaz de resolver problemas heterogêneos e que tenham uma baixa compatibilidade tecnológica (GAZIS; KATSIRI, 2022). Em essência, trata-se de um software que atua como mediador entre as aplicações, garantindo a compatibilidade entre as interfaces e descomplicando a integração de componentes. Além disso, o *middleware* assume tarefas críticas, como a abstração da comunicação entre aplicativos, redes, *hardware* e a gestão de recursos entre aplicações e sistemas operacionais (EMMERICH, 2000).

Ao centralizar essas operações, especialmente em sistemas distribuídos, esse desenvolvimento reduz a carga dos desenvolvedores, eliminando assim a necessidade de conexões complexas e necessárias no futuro (STEEN; TANENBAUM, 2023).

2.4 PROXY

O *proxy* é um serviço que age como um intermediário entre um dispositivo de um usuário e um servidor. Ele atua roteando as solicitações e respostas, ocultando informações importantes do usuário e oferecendo funcionalidades de segurança, filtragem de conteúdos e otimização de desempenho. A Seção 2.4.1 e a Seção 2.4.2 apresentam dois dos principais tipos de *proxy* atualmente existentes.

2.4.1 Proxy de Encaminhamento

O *proxy* de encaminhamento, comumente chamado apenas de *proxy*, está presente normalmente na grande maioria das redes domésticas e corporativas, sendo uma solução tradicional de segurança de rede atuando como intermediário entre os clientes e a internet. A sua principal função é representar o usuário diante dos servidores externos, proporcionando o controle de acesso a recursos e auxiliando na otimização de tráfego. Essa estratégia possibilita, por exemplo, limitar o acesso a determinados conteúdos, esconder o endereço IP dos clientes e otimizar o desempenho da rede ao guardar respostas frequentes. Apesar de sua eficácia em ambientes de menor escala, o *proxy* de encaminhamento pode apresentar algumas limitações e baixa eficiência em ambientes com grande volume de tráfego e demandas mais complexas (KOVACS, 2024).

2.4.2 Proxy Reverso

O *proxy* reverso pode ser considerado o oposto do *proxy* de encaminhamento, atuando como um servidor intermediário que recebe as solicitações dos clientes e as destina para os servidores internos, sem que o usuário perceba essa intermediação. Diferentemente do proxy de encaminhamento, que é configurado no navegador, o proxy reverso atua no lado do servidor de forma transparente, conforme ilustrado na Figura 2. O autor aborda três padrões principais para uso do proxy reverso: Protection Reverse Proxy, realiza a proteção dos servidores contra ataques de nível do protocolo da aplicação, isolando a infraestrutura em uma Zona Desmilitarizada (DMZ) e filtrando as requisições; Integration Reverse Proxy permite a integração de diversos servidores através de um único ponto de entrada, garantindo a estabilidade de URLs para os usuários e ocultando a estrutura interna do servidor; Front Door centraliza as autenticações dos usuários e atua no gerenciamento das sessões, permitindo login único para múltiplas aplicações. Destaca-se também outros benefícios que o proxy reverso pode oferecer, como flexibilidade na gestão da infraestrutura, aumento da segurança e redução de custos com certificados Secure

Sockets Layer (SSL). Em contrapartida, aponta desvantagens como a criação de um ponto único de falha, inclusão de latência e cuidados reforçados na configuração e manutenção do sistema (SOMMERLAD, 2003).

Outra função importante que o *proxy* reverso pode desempenhar é na implementação de estratégias de balanceamento de carga entre múltiplos servidores *back-end*. A forma mais simples de realizar o balanceamento é através do revezamento das requisições de maneira sequencial entre os servidores, abordagem conhecida como *round robin*. Estratégias mais aprimoradas podem contar com análises de estatísticas, como por exemplo a carga de processamento ou o tempo de resposta de cada servidor, para orquestrar de forma mais eficiente as requisições. O autor evidencia que, em aplicações que preservam as sessões de usuário, é importante garantir a nomeada "*session stickiness*", enviando todas as requisições de um mesmo usuário ao mesmo servidor, com a intenção de preservar a integridade da sessão. Deste modo, o *proxy* reverso além de integrar os sistemas, também contribui com a escalabilidade e a disponibilidade da infraestrutura (SOMMERLAD, 2003).

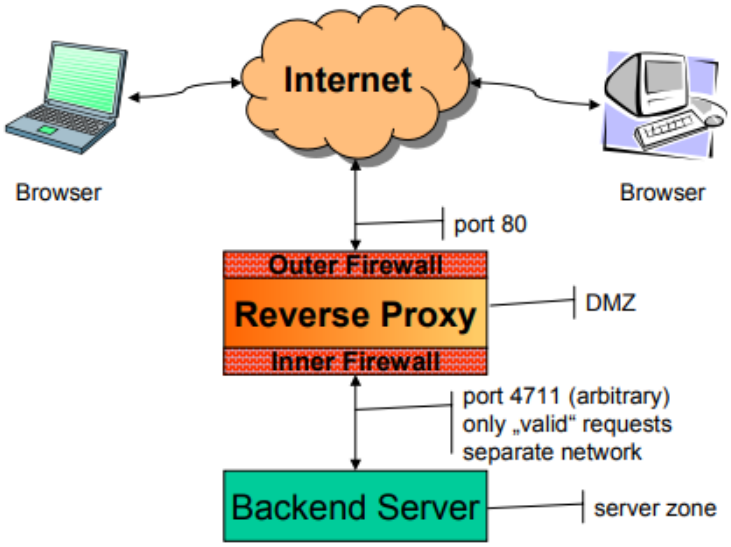


Figura 2 – Exemplo de um *proxy* reverso

Fonte: (SOMMERLAD, 2003)

3 TRABALHOS CORRELATOS

Esta seção aborda trabalhos cujas problemáticas e soluções estejam relacionados com a proposta deste trabalho, ainda que aplicadas em domínios distintos do conhecimento.

Foi realizada uma pesquisa bibliográfica sistemática com o objetivo de identificar trabalhos relacionados à integração de sistemas legados ou ao desenvolvimento de aplicações para o ensino. A investigação priorizou trabalhos publicados nos últimos 5 anos, abrangendo as seguintes bases de dados: Google Acadêmico, ResearchGate e ScienceDirect. Os termos de busca utilizados incluíram *Legacy system*, *interoperability*, *integration*, *middleware*, *development of educational software*, *reverse proxy* e *load balance*, visando recuperar artigos que abordassem tanto aspectos técnicos de interoperabilidade entre sistemas legados quanto soluções tecnológicas aplicadas ao contexto educacional. A Tabela 1 exibe a relação dos artigos selecionados, que serão abordados a seguir.

Tabela 1 – Artigos selecionados que de alguma forma se relacionam com o presente trabalho

Título	**Referência**	**Fonte**	**Seção**
A taxonomy of service identification approaches for legacy software systems modernization.	(ABDELLATIF *et al.*, 2021)	ScienceDirect	Seção 3.1
Upgrading of legacy systems to cyber-physical systems.	(KUTSCHER *et al.*, 2020)	ResearchGate	Seção 3.1
Integração de sistemas legados e atuais em instituição de registro público utilizando um Gateway para comunicação de diferentes plataformas	(OLIVEIRA *et al.*, 2022)	Brazilian Journal of Development	Seção 3.1
Integração de Sistemas Legados para Aprendizado a Distância: Estudo de Caso em Planejamento de Sistemas Móveis Celulares.	(MIRANDA, 2003)	LEA UFPA	Seção 3.1
Software modernization powered by dynamic language product lines	(CAZZOLA; FAVALLI, 2024)	ScienceDirect	Seção 3.1
Evaluation of technical approaches for real-time data transfer from electronic health record systems.	(KIRILOV; DUGAS, 2024)	ScienceDirect	Seção 3.2
Cost-effective server-side redeployment for web-based online laboratories using nginx reverse proxy.	(LEI *et al.*, 2020)	ScienceDirect	Seção 3.2
A new remote web-based msplus data visualization system for east.	(ZHANG *et al.*, 2023)	ResearchGate	Seção 3.2
ALGO+ Uma ferramenta para o apoio ao ensino de Algoritmos e Programação para alunos iniciantes.	(AMARAL *et al.*, 2017)	ScienceDirect	Seção 3.3
LIBRAS Game: Trabalhando o ensino da matemática com alunos surdos dos anos iniciais através do uso de aplicativo educacional.	(PRATES, 2018)	Repositório UTFPR	Seção 3.3

Fonte: O Autor (2025).

A Seção 3.1 apresenta cinco estudos que exploram as abordagens existentes para a integração de sistemas legados, com foco em estratégias que permitem a disponibilidade de sistemas antigos e novas soluções, garantindo a continuidade dos serviços e a evolução tecnológica.

A Seção 3.2 contém três artigos, onde os autores apresentam soluções com a utilização de *proxy* reverso, permitindo o funcionamento simultâneo de diferentes versões, unificando o acesso, proporcionando segurança e escalabilidade.

A Seção 3.3, por sua vez, possui dois estudos que trazem ferramentas desenvolvidas para auxiliar no processo de aprendizagem dos estudantes, demonstrando como a tecnologia é fundamental no processo de ensino, para torna-lo mais convidativo e eficiente.

3.1 INTEGRAÇÃO DE SISTEMAS LEGADOS

De forma resumida, os artigos abordados nessa seção tiveram como objetivo apresentar estudos com foco na integração de sistemas já consolidados. Abdellatif et al (2021) desenvolveram uma revisão sistemática da literatura na qual analisaram primeiramente 3246 publicações e selecionaram 41 trabalhos mais relevantes, focadas em sistemas legados. Os autores criaram uma taxonomia multicamadas que realiza a classificação em quatro dimensões principais: grau de usabilidade, processos de identificação adotados, artefatos de entradas utilizados, como casos de uso, código fonte, banco de dados e etc, e os tipos de serviços gerados como saída. Alguns trabalhos analisados pelos autores apresentam estratégias de integração com sistemas existentes, analisando técnicas como a geração de interfaces REST, padrões de projetos *wrappers* e reutilização de código orientado a objeto, preservando a lógica de negócio testada e validada ao longo dos anos, evitando os riscos, esforços e custos da reescrita completa. Dessa forma, possibilitando que sistemas antigos continuem operando enquanto são progressivamente integrados a novas arquiteturas.

O estudo realizado por Kutscher et al., (2020) propõe uma metodologia de cinco etapas para modernização de sistemas legados e validou essa proposta integrando um CNC laser plotter para um Sistemas Ciberfísicos (CPS) da Indústria 4.0. Na primeira etapa foi feita a análise do estado inicial, onde os autores começaram identificando a estrutura e os recursos da máquina, por exemplo a máquina não possuía qualquer capacidade de comunicação, interação ou virtualização. Para o segundo passo foi realizada a identificação das necessidades de integração, para poder definir o que precisaria ser adicionado, como comunicação com outros sistemas, criação de uma interface de acesso remoto e padronização dos dados da máquina. Já no terceiro passo constituiu a escolha da tecnologia, a escolhida foi o Open Platform Communications Unified Architecture (OPC UA), criando um servidor embarcado conectado à máquina para a comunicação e modelagem dos dados. No quarto passo aconteceu o desenvolvimento da integração, onde os autores instalaram o servidor OPC UA em Python conectado a CNC, criando um modelo virtual da máquina no servidor, permitindo que os dados dos sensores e comandos da máquina fossem monitorados e passados para o cliente externo OPC UA. Posteriormente, no quinto e último passo, validou-se os resultados obtidos em um cenário real, com um destaque positivo. Após realizar todas as etapas da metodologia, o autor ressalta que a migração de sistemas legados a CPS é eficaz e oferece um caminho prático e reutilizável, mantendo a estrutura

do sistema legado com a nova integração moderna.

Oliveria et al. (2022) abordou um estudo sobre a integração entre sistemas legados e web modernos de uma instituição pública estadual. O objetivo do artigo foi realizar a aplicação de uma arquitetura baseada em *gateway* para garantir a interoperabilidade de plataformas heterogêneas, evitando a reescrita ou a necessidade de substituição completa do sistema antigo, por intermédio de um *middleware*, essa abordagem dialoga com a proposta do presente trabalho.

O desenvolvimento foi composto por dois componentes principais, o módulo nomeado JS e o módulo CTG (CICS Transaction Gateway), fornecida pela IBM. O módulo JS é constituído por duas camadas: uma em .NET(C#) e outra em Java, que se comunicavam através de *web services* com os dados em Linguagem de Marcação Extensível (XML). Essa comunicação é o elemento central da integração entre os sistemas, pois é por ela que os dados trafegam da nova interface *web* até o *backend*. A primeira camada da estrutura, em .NET(C#), fica responsável por receber os dados da aplicação *web*, padronizando-os e convertendo para o formato XML, após são transmitidos para a segunda camada do *middleware* em Java, sobre o servidor Jboss. Nesta etapa, os *web services* são importantes para garantir a interoperabilidade entre os diferentes ambientes e linguagens, realizando validações adicionais e delegando as chamadas para o CICS. As respostas obtidas por ele são convertidas novamente em XML e percorrem o caminho inverso, retornando por todas as camadas anteriores até alcançar a interface *web*, conforme exibido na Figura 3. Dessa maneira, os *web services* garantem a conexão entre as diferentes plataformas mantendo o isolamento das camadas, que favorece a manutenção, escalabilidade e segurança (OLIVEIRA *et al.*, 2022).

A relevância deste trabalho está na forma que o projeto lida com a integração dos sistemas legados, funcionando paralelamente com os sistemas modernos. A ideia de preservar o sistema já consolidado e adicionar uma nova camada, por meio de um *gateway* de integração com *web services*, apresenta a viabilidade e os benefícios do modelo.

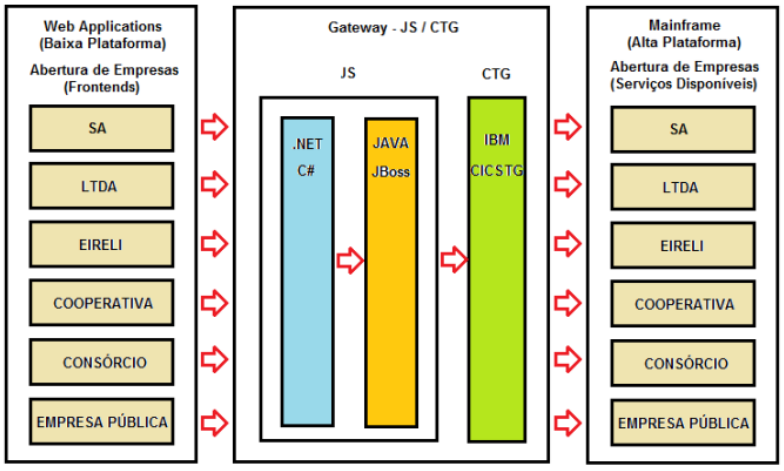


Figura 3 – Middleware que permite a integração entre os sistemas

Fonte: (OLIVEIRA *et al.*, 2022)

Miranda (2003) visou unificar os quatro módulos do sistema CELLP, originalmente divididos em ambientes heterogêneos como Delphi (Windows) e Kylix (Linux), utilizando uma solução de *middleware* baseada na arquitetura Arquitetura de Corretor de Objetos de Requisição Comum (CORBA) para a integração dos sistemas. A estratégia constituiu no encapsulamento de cada módulo como um objeto distribuído, por meio de interfaces padronizadas chamadas Linguagem de Definição de Interface (IDL), responsáveis pela comunicação entre os módulos, descrevendo os métodos e parâmetros necessários, independente da linguagem ou sistema operacional. Para tornar o sistema acessível via web, foi acrescentado ao *middleware* os *web services* baseados no protocolo Protocolo Simples de Acesso a Objetos (SOAP). Isso permitiu que as requisições HTTP, do navegador, fossem convertidas em chamadas CORBA e posteriormente direcionadas aos servidores, que após o processamento dos dados, respondiam através do formato XML, possibilitando assim a manipulação e visualização dos dados diretamente da nova interface *web*. O resultado dessa integração foi o WebCELLP, uma plataforma unificada que auxiliou os alunos no aprendizado das disciplinas, passando a interagir com todos os módulos integrados a apenas uma interface *web*, executando projetos práticos de planejamento de redes celulares de forma semipresencial. A modernização realizada por Miranda manteve os sistemas legados em pleno funcionamento, enquanto adicionava uma camada intermediária para comunicação distribuída, preservando suas funcionalidades originais e garantindo a continuidade do serviço durante a transição. A comunicação via XML poderia ser trocada pelo Notação de Objetos JavaScript (JSON), mais leve e muito utilizado atualmente. A arquitetura seguiu o padrão de desenvolvimento Modelo-Visão-Controlador (MVC), separando as camadas de apresentação, controle e negócios. Essa escolha facilitou a manutenção, a modularidade e a expansão do sistema.

Ademais, Cazzola e Favalli (2024) propuseram a utilização de um método baseado em Dynamic Language Product Lines (DLPL), que sem interromper o funcionamento do sistema, permitiu a modernização progressiva do sistema legado. Utilizando uma ferramenta de desenvolvimento de linguagens, o compilador do sistema legado em COBOL foi reestruturado para micro-linguagens modulares, permitindo a construção de variantes que interpretavam o código antigo e o novo, essa estratégia criou três variantes: uma para o novo código Java, uma para o código legado e uma híbrida, responsável por permitir a interoperabilidade dos sistemas. Dessa forma, se tornou possível executar os trechos atualizados enquanto o restante do sistema continuava operando. Alguns testes foram feitos para demonstrar que a transição entre as linguagens COBOL e Java ocorriam de maneira concisa, como por exemplo a implementação de múltiplas versões do sistema, cada uma com uma etapa da modernização, garantindo assim um processo de migração contínuo, seguro e sem a reescrita total do sistema.

Os estudos apresentados nessa seção demonstram diferentes formas de realizar integrações e modernizações de sistemas legados, conservando suas funcionalidades essenciais enquanto introduzem novas camadas de interoperabilidade e comunicação. Essa forma de evolução, utilizando um *middleware*, *web services* ou *gateways*, interage diretamente com a proposta

do presente trabalho, que visa integrar dois sistemas heterogêneos, um legado de interface *desktop* com uma nova interface *web*, preservando a operação contínua do ambiente existente. Dessa forma, como nos trabalhos analisados, os projetos buscam garantir a disponibilidade entre as plataformas heterogêneas, reduzindo riscos e custos de reconstrução, utilizando soluções de integração que permitem a convivência harmônica entre o antigo e as novas tecnologias do mercado.

3.2 MEDIADOR PARA INTEGRAÇÕES DE SISTEMAS HETEROGÊNEOS

Sob distintas perspectivas, autores apresentam a reestruturação e modernização de sistemas voltados a experimentos científicos, fazendo o uso de novas tecnologias como o servidor NGINX. Um estudo recente conduzido por Kirilov e Dugas (2024), realizou uma análise comparativa para quatro métodos distintos de entrega de dados em tempo real (RTD) em sistemas de prontuário eletrônico (EHR): *proxy* reverso, WebSocket Notifications, REST Hooks e database triggers. O estudo possibilita uma avaliação técnica e detalhada sobre viabilidade de cada abordagem, por fornecer as implementações práticas e mensurações precisas dos desempenhos.

O foco do presente trabalho recai sobre o uso de *proxy* reverso como uma camada intermediária de comunicação entre o sistema legado e a nova interface web. O método testado pelo artigo demonstra que o *proxy* reverso viabiliza a comunicação entre sistemas de maneira transparente e também apresenta um considerável desempenho em termos de variabilidade e latência.

Diante dos testes realizados nos quatro métodos, o *proxy* reverso teve latência média de 14,43 ms e um desvio padrão de apenas 4,58 ms, indicando um bom desempenho e estabilidade. Embora o método de *database triggers* tenha apresentado menor latência média (13,52 ms), o *proxy* reverso apresentou valores máximos mais baixos e menor variabilidade, mostrando que pode ser mais adequado em cenários que a previsibilidade e estabilidade são fatores tão importantes quanto velocidade. Ao simular com condições realistas o sistema de produção, com até 100 clientes simultâneos e crescente volume de dados, o uso do *proxy* reverso continuou oferecendo um comportamento firme. Nos cenários de pacotes com 160 recursos transferidos, a latência média foi de 268,04 ms, vencendo REST hooks (8172,83 ms) e WebSockets (388,96 ms) em confiabilidade e escalabilidade. O motivo desse desempenho pode estar lincado à capacidade do *proxy* em desacoplar a entrega de notificações em tempo real da lógica do servidor, permitindo otimizações como cache, balanceamento, segurança e distribuindo carga de maneira positiva (KIRILOV; DUGAS, 2024).

Explorando a arquitetura, o *proxy* foi implementado utilizando a linguagem Golang, com a biblioteca *httputil*, fazendo papel de um *middleware*. Ele intercepta as requisições para o servidor FHIR, verifica as respostas e, ao identificar a criação ou alteração de dados, envia notificações ao sistema consumidor, conforme apresentado na Figura 4. Dessa forma, o mo-

delo garante que o sistema *backend* permaneça separado, evitando assim a exposição direta e simplificando a aplicação de políticas de autenticação, auditoria e filtragem (KIRILOV; DUGAS, 2024).

Além da performance, o *proxy* reverso também apresenta alguns pontos positivos relacionados à segurança e manutenção, atuando como ponto de controle centralizado, limitando acessos a *endpoints* internos, realizando o balanceamento de carga do servidor, monitoramento de uso e isolamento de falhas, recursos que podem ser críticos em ambientes com vários sistemas concorrendo por um mesmo acesso ao *backend*. Em conclusão, os experimentos práticos realizados por Kirilov e Dugas comprovam a eficiência e estabilidade que pode ser alcançada utilizando *proxy* reverso, e também o posicionam como uma alternativa viável equilibrada entre performance, segurança e escalabilidade para sistemas que exigem entrega para múltiplos consumidores. Devido ao contexto do seguinte trabalho, que aborda a integração entre um sistema legado e um novo sistema web, o uso do *proxy* reverso se mostra tecnicamente amparado e comprovadamente eficiente, sendo uma base firme para o uso em uma arquitetura de um *middleware*.

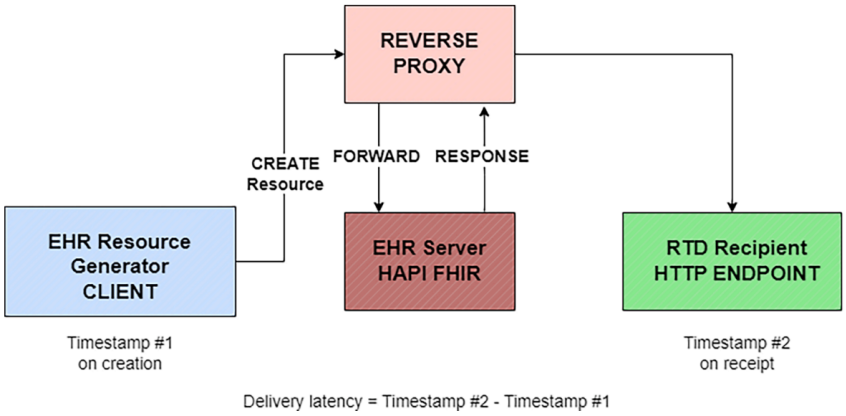


Figura 4 – Arquitetura do *proxy* reverso para a realização dos testes. O tempo é medido pelos *timestamps*

Fonte: (KIRILOV; DUGAS, 2024)

LEI *et al.* (2020) apresentam a reconstrução do servidor para o laboratório remoto NCS-Lab, unificando os dois sistemas existentes, o antigo sistema (YUI/Tomcat) era acessado através da url "/ncslab"e o novo desenvolvido (React/NGINX+PHP) com "/react". Estes domínios distintos geravam alguns problemas operacionais, como duplicação de certificados HTTPS, maior exposição a ameaças de segurança e conflitos de recursos web. A solução apresentada utilizou NGINX como *proxy* reverso para realizar a unificação dos acessos através de um único domínio.

Conforme apresentado na Figura 5, a arquitetura implementada ilustra como o NGINX atua como intermediário entre os usuários e o sistema *backend* consolidado, após a implementação as requisições são direcionadas de forma transparente, delegando as chamadas de cada URL para seus respectivos servidores, simplificando a infraestrutura e auxiliando na redução de custos. Os resultados obtidos pelo autor, mostram que a utilização do *proxy* reverso é eficaz

para a unificação de sistemas, redução do tempo de implementação e de custos com infraestrutura, mantendo a compatibilidade dos sistemas legados conforme as novas ferramentas vão evoluindo (LEI *et al.*, 2020).

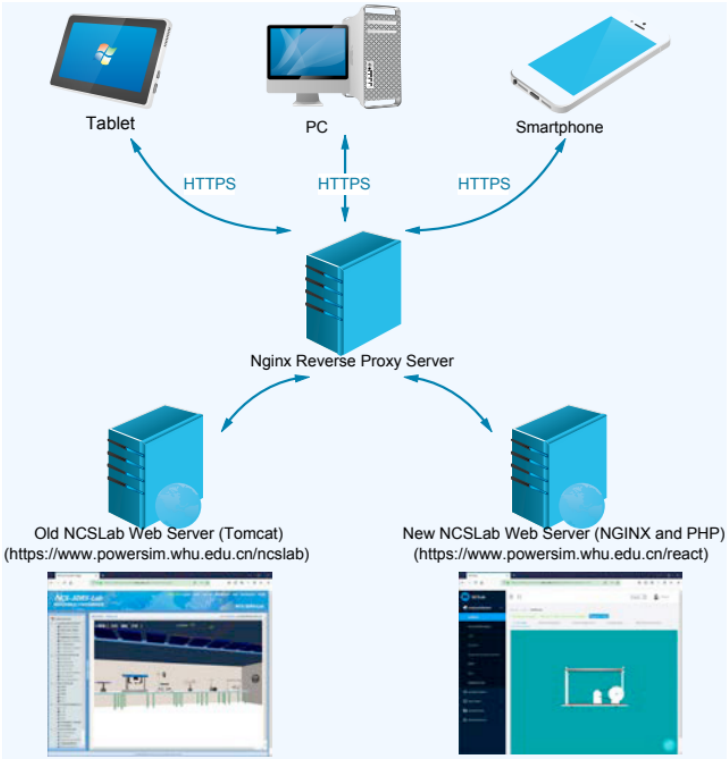


Figura 5 – Ilustração da reimplantação de servidores NCSLab

Fonte: (LEI *et al.*, 2020)

Ainda, ZHANG *et al.* (2023) discutem sobre a criação de um novo sistema web para leitura das informações do Tokamak EAST, projeto de fusão nuclear, que gera um grande volume de dados a cada disparo experimental. Objetivando promover uma interface moderna e acessível a qualquer dispositivo, visando superar as limitações de soluções anteriores, permitindo o acesso de múltiplos usuários simultaneamente e possibilitando a integração com novos componentes e tornando o projeto escalável.

O início do processo envolve o armazenamento dos dados coletados no sistema MD5Plus, acessados através do protocolo MDSip pelo *backend* Java, que aproveita a tecnologia de *multithreading* para garantir que as múltiplas requisições sejam tratadas corretamente, sem prejudicar o desempenho da ferramenta. O Java entrega o serviço de clusterização, disponibilizando várias instâncias de servidor, garantindo a escalabilidade necessária para suportar o grande número de requisições, que por sua vez são encaminhadas ao servidor correto através do *proxy* reverso (ZHANG *et al.*, 2023).

Atuando como *proxy* reverso, o NGINX trabalha de forma intermediária entre os usuários e os servidores *backend*, permitindo que o balanceamento de carga ocorra através do algoritmo *Weighted Round-Robin*, no qual cada servidor recebeu um peso de acordo com a sua capa-

cidade de processamento, ilustrado na Figura 6. O NGINX distribuiu cada requisição de forma proporcional a esses pesos, priorizando os servidores mais potentes. Dessa forma, ocorreu uma distribuição equilibrada do tráfego, garantindo que nenhum servidor ficasse ocioso enquanto outros estivessem sobrecarregados. Alguns fatores importantes mencionados pelos autores no ponto de vista de segurança, dão conta que o NGINX foi utilizado também para realizar a separação de recursos estáticos e dinâmicos, possibilitando que os usuários acessem apenas os recursos públicos, atuou no encaminhamento de requisições controladas, evitando acessos não autorizados ou fora do escopo previsto, e na segregação das comunicações, permitindo que as requisições fossem feitas apenas por portas de rede confiáveis e seguras. A efetividade do estudo apresentado foi comprovada nos 21 sinais pulsados nos experimentos, onde o sistema evidenciou o baixo consumo de memória e uma alta taxa de resposta mesmo com um grande volume de dados, conseguindo exibir os sinais com menor latência, maior escalabilidade e eficiência (ZHANG *et al.*, 2023).

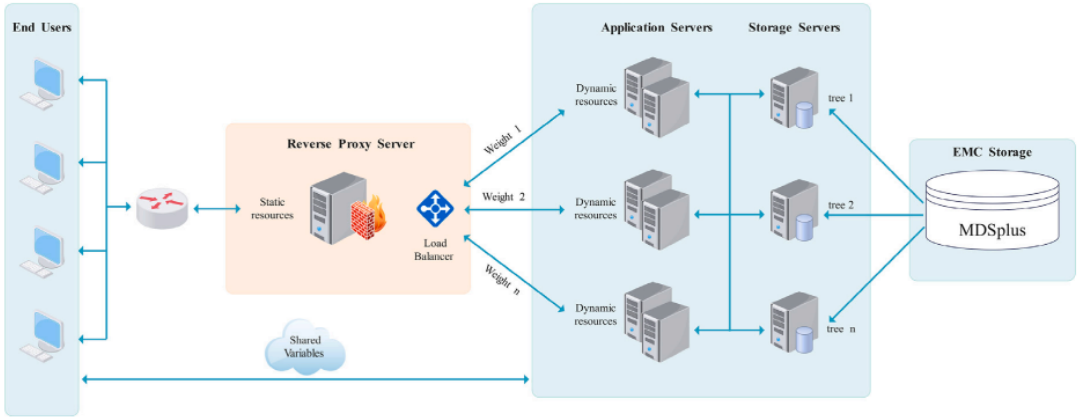


Figura 6 – Diagrama geral dos serviços EAST

Fonte: (ZHANG *et al.*, 2023)

Os trabalhos citados anteriormente dão conta que a utilização de um *proxy* reverso é uma solução eficaz, auxiliando no reforço da segurança dos sistemas e no controle das requisições. Pelos testes realizados, é interessante notar a capacidade positiva de centralizar o acesso por um único meio, controlando o fluxo dos dados e protegendo os servidores internos de acesso diretos não esperados, aspectos que se destacaram nas abordagens dos diferentes autores. Desta forma, é possível observar que adotar o uso de um *proxy* reverso faz sentido no desenvolvimento do presente trabalho, uma vez que é necessário manter dois sistemas rodando sobre um mesmo servidor, sem problemas operacionais. Além disso, a utilização de um *proxy* reverso permitirá ocultar as informações sensíveis do servidor, acrescentando uma camada a mais de proteção.

3.3 APLICAÇÕES VOLTADAS AO ENSINO

Amaral et al. 2017 apresentam o desenvolvimento de um sistema web Algo+, focado em apoiar o processo de ensino e aprendizagem de lógica e programação. O principal objetivo foi

construir uma ferramenta que auxiliasse os alunos iniciantes a superar as tradicionais barreiras das disciplinas introdutórias da área da computação, diminuindo a alta taxa de evasão e reprovação. Com esse objetivo, a construção foi fundamentada em teorias de aprendizagem, como por exemplo o Ciclo de Aprendizagem Experiencial de Kolb (KOLB, 2014), com a intenção de atender diferentes estilos cognitivos, com uma estrutura adaptativa, motivadora e clara.

As etapas de desenvolvimento iniciaram com a definição do problema, seguido por uma revisão de literatura, responsável por sustentar a proposta. Em seguida, foi conduzida a escolha da tecnologia, o projeto, a implementação e a validação da ferramenta. O Algo+ foi desenvolvido como um sistema web, disponível através do navegador, consequentemente eliminando a instalação de qualquer software pelo usuário. A ferramenta contempla duas interfaces principais, uma destinada à administração e controle do ambiente e outra ao uso comum para alunos e professores (AMARAL *et al.*, 2017).

Ainda, a lista com as tecnologias utilizadas na implementação do portal e em seguida a Figura 7 ilustra a arquitetura projetada e desenvolvida.

- HTML5 como base estrutural das páginas *web*.
- CSS utilizado para definir estilos visuais da interface de usuário.
- jQuery como *framework* JavaScript para manipulação dinâmica da interface e dos formulários.
- PHP no lado do servidor, para realizar o processamento das informações, notas, atividades e controle de usuários.
- MySQL foi utilizado como banco de dados para garantir o armazenamento das informações de forma segura e eficiente.

Para obter o melhor desempenho de aprendizagem, a organizações dos conteúdos apresentados pelo Algo+ foi estruturada em módulos, seguindo uma sequencia lógica e alinhada às etapas do ciclo de Kolb, que visam sentir, observar, pensar e fazer. Cada um dos módulos possuem o material teórico, atividades formativas, multimídia e avaliações automáticas, que permitem a construção do conhecimento dos alunos de maneira progressiva. Alguns paradigmas de programação foram implementados, como o desenvolvimento visual com Scratch, programação estruturada com a linguagem C e o desenvolvimento para dispositivos móveis com AppInventor (AMARAL *et al.*, 2017).

Já o processo de validação foi realizado por meio de experimentos com duas turmas das disciplinas de algoritmos e programação envolvendo 47 estudantes. Os resultados mostraram que o desempenho da nova ferramenta foi similar ao desempenho em exames presenciais, indicando a efetividade do ambiente no suporte ao aprendizado independente. Com isso, a aplicação

evidenciou seu potencial como uma solução híbrida, atuando no apoio de professores e no estudo autônomo dos alunos que desejam aprimorar suas habilidades em programação e lógica (AMARAL *et al.*, 2017).

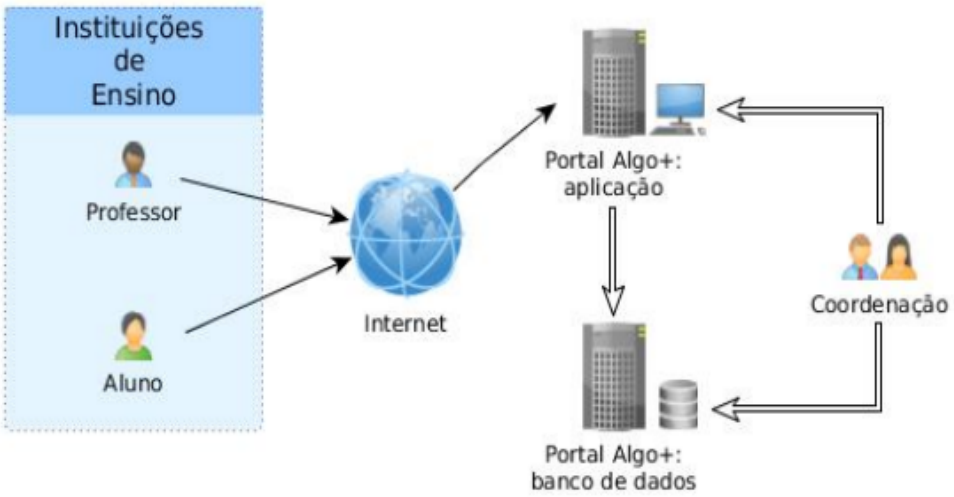


Figura 7 – Arquitetura do portal Algo+

Fonte: (AMARAL *et al.*, 2017)

Outro trabalho que vale destacar é o LIBRAS Game, que teve como objetivo desenvolver um aplicativo mobile educacional que auxiliasse o ensino de números de 1 a 9, para crianças surdas dos anos iniciais. O projeto desenvolvido foi dividido em etapas, como levantamento de requisitos, projeto, implementação, testes e validação. O aplicativo busca facilitar a associação entre numerais, quantidades e sua representação em Língua Brasileira de Sinais (LIBRAS). O desenvolvimento utilizou a Ambiente de Desenvolvimento Integrado (IDE) do Android Studio como ambiente principal, e fez uso do Java como linguagem de programação, XML para construção de interfaces e utilizado o *software* livre GIMP 2.8 para a criação e edição das imagens do *app*. Todos os testes do aplicativo foram realizados em um computador com sistema operacional Debian e um celular Samsung J5 utilizando Android. Ao realizar os testes com os usuários que não possuíam domínio da LIBRAS, o aplicativo demonstrou êxito, visto que os participantes conseguiram reconhecer os números no final das atividades (PRATES, 2018).

Assim como o Algo+ e o LIBRAS Game, o presente trabalho busca atuar no processo de aprendizagem de alunos por meio de desenvolvimentos tecnológicos, propondo igualmente o uso de ferramentas digitais para tornar o ensino mais acessível, eficiente e motivador. Ambas iniciativas apresentam preocupação em reduzir as dificuldades padrões enfrentadas no processo de formação dos alunos, apenas diferenciam-se nos públicos alvo e nas áreas do conhecimento, porém evidencia como a integração entre a tecnologia e educação pode ser aplicada em diferentes contextos pedagógicos.

4 PROPOSTA DE SOLUÇÃO

O objetivo deste trabalho foi propor e desenvolver uma solução para integrar uma aplicação web a um sistema legado, de forma a permitir a comunicação entre a nova interface *web* e o *backend* já consolidado da ferramenta WebAlgo. Essa solução preservou a lógica de negócio do sistema de APIs existente, ao mesmo tempo em que proporcionou uma nova experiência de uso por meio da nova interface *web*.

Optou-se por utilizar um *middleware* para realizar a integração, atuando como uma ponte entre o *frontend* e o *backend*, encapsulando a lógica de negócio. Dessa forma, a nova interface web não interagiu diretamente com o servidor, mas com o *middleware*, responsável por tratar e direcionar as requisições conforme necessário. Além disso, foi utilizado um *proxy* reverso como parte da arquitetura da solução, com papel fundamental na organização do tráfego entre os componentes, funcionando como ponto único e controlado de entrada para requisições externas. Com isso, foi possível ocultar detalhes da infraestrutura interna e proteger o *backend* de acessos diretos, além de viabilizar a configuração de um balanceador de carga para cenários com mais de um servidor interno.

A Figura 8 ilustra a arquitetura implementada, evidenciando o papel do *middleware* na mediação entre os componentes da solução e o NGINX como *proxy* reverso, responsável também pelo balanceamento de cargas.

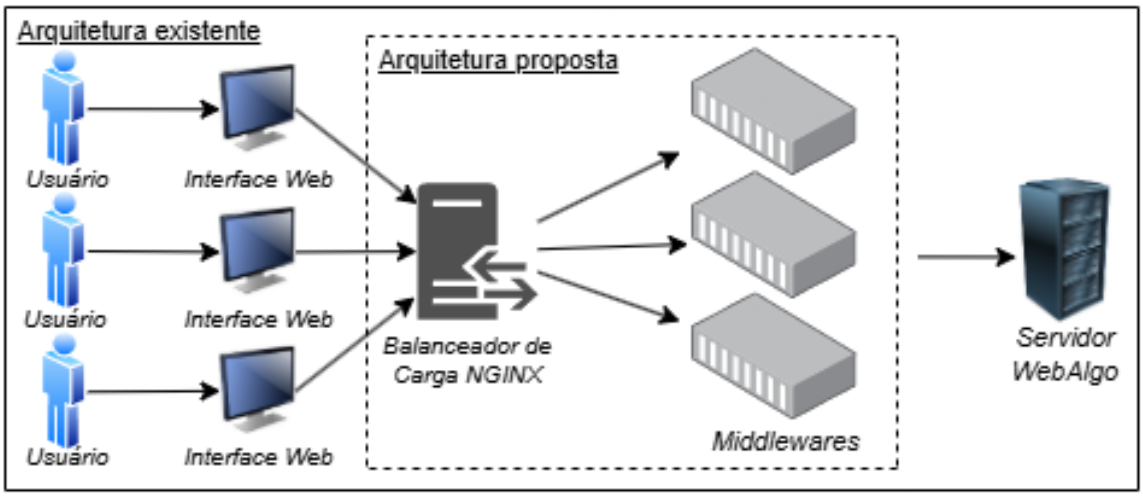


Figura 8 – Arquitetura da solução

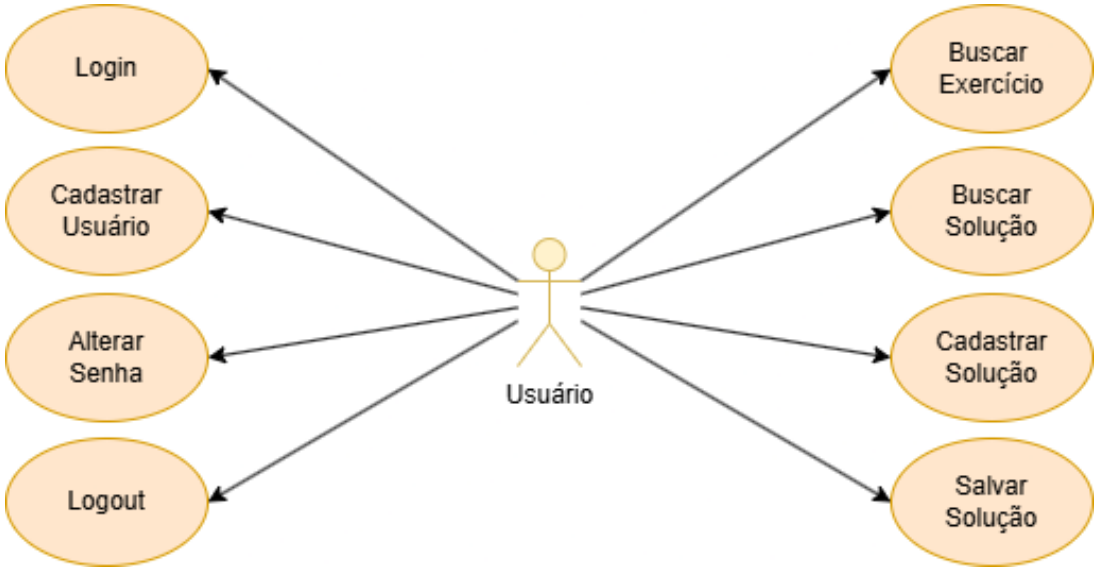
Fonte: O Autor (2025)

4.1 DIAGRAMA DE CASOS DE USO

Um caso de uso define como um ator interage com o sistema por meio de determinadas ações. Os atores são agentes externos, como pessoas ou outros sistemas, que interagem com o sistema executando um caso de uso (WIEGERS; BEATTY, 2013). Os diagramas são agrupadores de casos de uso, apresentando o título dos casos mapeados e sua relação com um ou mais atores (ROSENBERG; STEPHENS, 2007).

Nesta seção, foram apresentados três casos de uso conforme previsto no processo ICONIX, a fim de exemplificar o funcionamento das APIs e demonstrar a integração entre a nova interface *web* moderna e o *backend* consolidado. Está presente no detalhamento de casos de uso apenas o viés do usuário, pois o serviço HTTP do *backend* consolidado não possuía uma distinção clara entre usuários administradores e alunos. O levantamento dos casos de uso apresentados na Figura 9 foi realizado com base na análise das funcionalidades disponíveis na nova interface *web*, em conjunto com as APIs existentes no servidor interno. Dessa maneira, as funcionalidades que não estavam presentes no diagrama refletiram a ausência de implementação na interface *web* naquele momento e, por esse motivo, não foram mapeadas como casos de uso neste contexto.

Figura 9 – Diagrama de caso de uso do sistema



Fonte: O Autor (2025)

4.1.1 DETALHAMENTO DE CASOS DE USO

Nesta etapa, foram detalhados os casos de uso relacionados ao domínio de usuário, contemplando as funcionalidades essenciais para autenticação e gerenciamento de acessos. A Figura 10 apresenta os *mockups* das telas de login e cadastro de usuários, ilustrando a interface

inicial do sistema. As interações representadas refletiram os fluxos definidos nos casos de uso e serviram como base para a implementação efetiva da interface *web*.

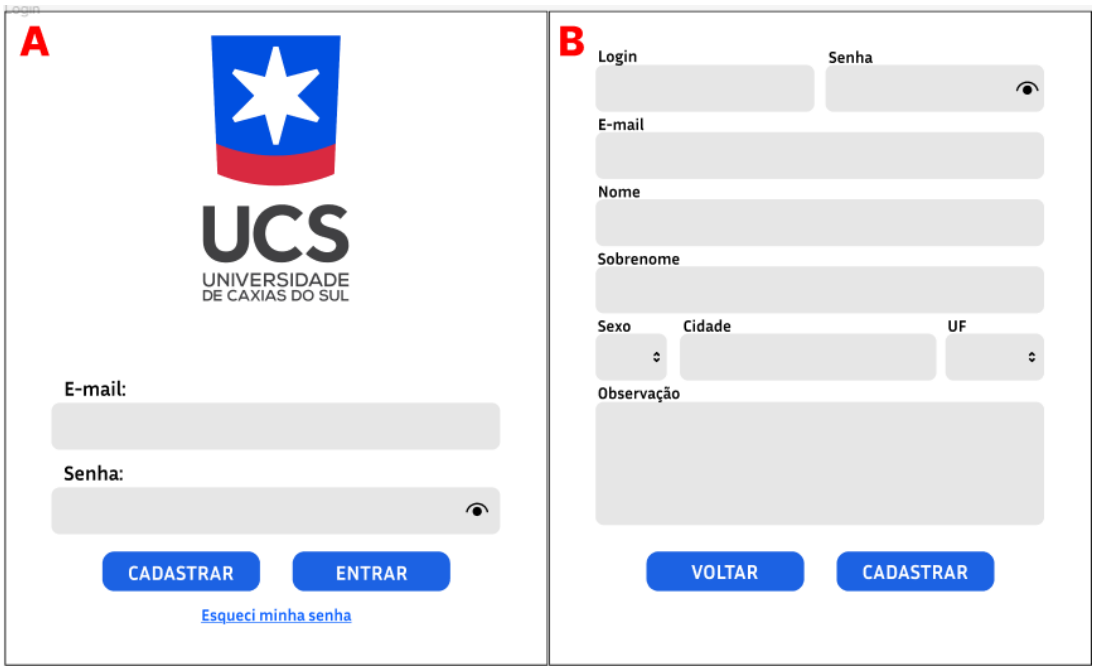


Figura 10 – Lado A: Tela de *login*. Lado B: Tela de cadastro

Fonte: O Autor (2025)

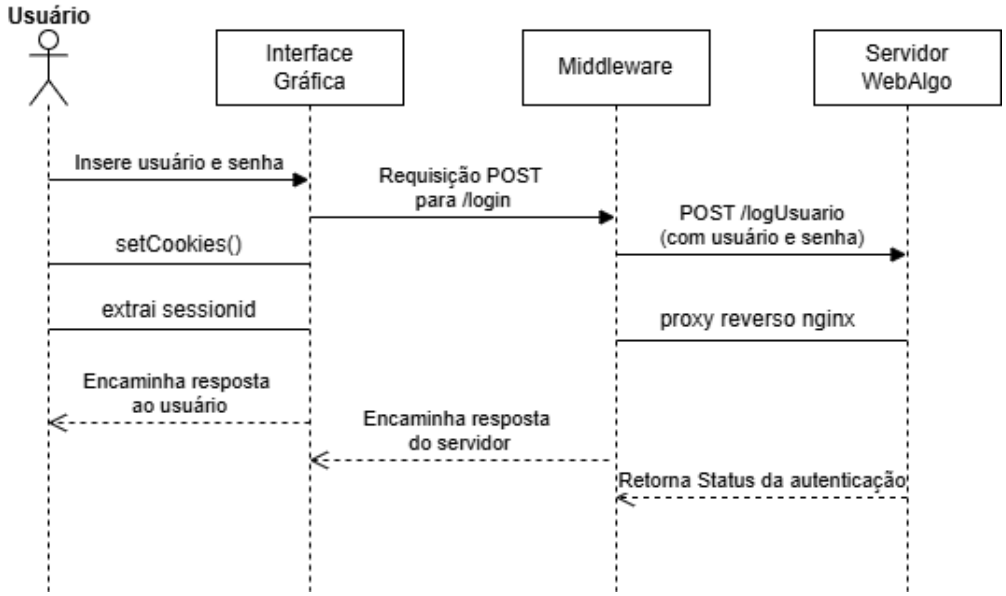
4.1.1.1 Caso de Uso UC-01 realizar *Login*

No lado **A** da Figura 10, encontra-se o modelo de interface gráfica referente ao caso de uso UC-01 – Realizar Login. O detalhamento deste caso de uso foi apresentado no Quadro 1 e a Figura 11 exibiu o respectivo diagrama de sequências.

4.1.1.2 Caso de Uso UC-02 realizar cadastro

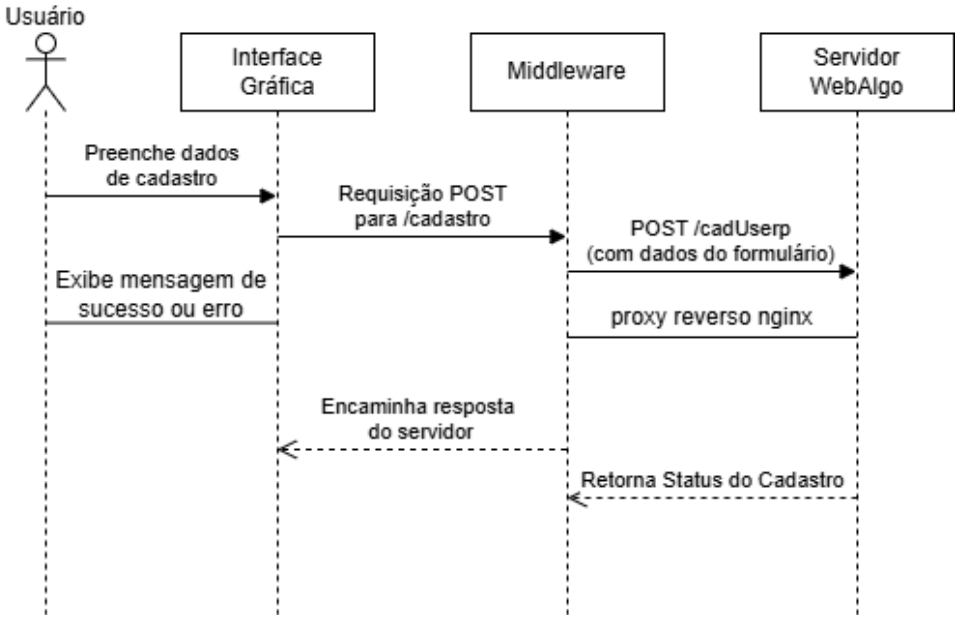
No lado **B** da Figura 10, encontra-se o modelo de interface gráfica referente ao caso de uso UC-02 – Realizar cadastro. O detalhamento deste caso de uso foi apresentado no Quadro 2 e a Figura 12 exibiu o seu diagrama de sequências.

Figura 11 – Diagrama de sequência do *Login*



Fonte: O Autor (2025)

Figura 12 – Diagrama de sequência do cadastro



Fonte: O Autor (2025)

Quadro 1 – UC-01 Realizar *Login*

Código e Descrição	UC-01 Realizar *Login*
Ator	Usuário
Descrição	Permite que um usuário realize a autenticação no sistema por meio do envio de nome de usuário e senha.
RF Associados	RF-01
Pré-Condições	O usuário deve estar previamente cadastrado no sistema.
Pós-Condições	Usuário autenticado, com *sessionid* e nome armazenados localmente.
Fluxo principal	1. Usuário acessa a tela de *login*. 2. Usuário informa nome de usuário e senha. 3. Sistema envia requisição POST para o *endpoint* `/logUsuario`. 4. Sistema recebe a resposta do servidor. 5. Sistema armazena o *cookie* de sessão e extrai o *sessionid*. 6. Sistema autentica o usuário.
Fluxos alternativos	**Falha na autenticação:** 4.a) Servidor retorna erro de autenticação. - Sistema informa que os dados estão incorretos. **Sem resposta do servidor:** 4.b) Ocorre erro de comunicação (*timeout* ou falha HTTP). - Sistema informa falha de conexão. ***Cookies* não recebidos:** 4.c) Sistema não consegue recuperar `Set-Cookie`. - Usuário é informado sobre falha ao autenticar.

Fonte: Autor (2025)

Quadro 2 – UC-02 Realizar cadastro

Código e Descrição	UC-02 Realizar *Cadastro*
Ator	Usuário
Descrição	Permite que um usuário realize o cadastro no sistema, fornecendo seus dados pessoais e de acesso.
RF Associados	RF-02
Pré-Condições	O usuário **não** deve estar previamente cadastrado no sistema.
Pós-Condições	Usuário cadastrado, podendo realizar *login* com o usuário e senha informados.
Fluxo principal	1. Usuário acessa a tela de cadastro. 2. Usuário preenche os campos: *login*, senha, nome, sobrenome, e-mail, sexo, cidade, estado (UF) e observação. 3. Sistema envia requisição POST para o *endpoint* `/cadUserp`. 4. Sistema recebe a resposta do servidor. 5. Sistema informa ao usuário que o cadastro foi realizado com sucesso.
Fluxos alternativos	**Falha no cadastro:** 4.a) Servidor retorna erro. – Sistema informa o erro ao usuário (ex: *login* já existente ou dados inválidos). **Sem resposta do servidor:** 4.b) Ocorre erro de comunicação (*timeout* ou falha HTTP). – Sistema informa falha de conexão.

Fonte: O Autor (2025)

4.1.1.3 Caso de Uso UC-03 buscar exercício

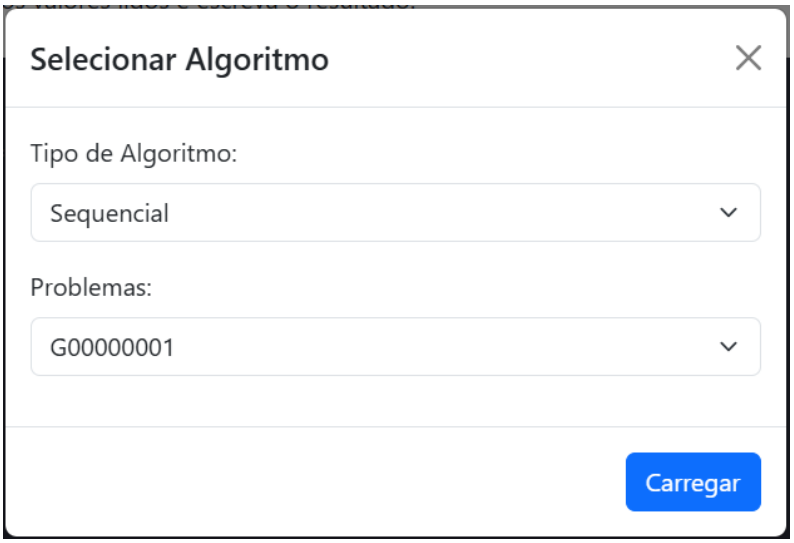
A Figura 13 ilustra a interface gráfica referente ao caso de uso UC-03 – Buscar exercício, responsável por permitir que os alunos localizassem os exercícios de algoritmos disponíveis para desenvolvimento. O detalhamento deste caso de uso foi apresentado no Quadro 3 e a Figura 14 exibiu o respectivo diagrama de sequências.

Quadro 3 – UC-03 Buscar exercícios

Código e Descrição	UC-03 Buscar exercícios
Ator	Usuário
Descrição	Permite selecionar um tipo de algoritmo e visualizar os problemas disponíveis, a fim de carregá-los na página.
RF Associados	RF-03
Pré-Condições	Estar na interface de seleção de algoritmos.
Pós-Condições	Algoritmo selecionado carregado na página.
Fluxo principal	1. Usuário acessa a interface de seleção. 2. Sistema exibe tipos de algoritmos. 3. Usuário seleciona o tipo. 4. Sistema envia POST para `/buscaProblemasChave`. 5. Recebe e exibe a lista de problemas. 6. Usuário escolhe um problema e clica em **Carregar**. 7. Sistema envia POST para `/dadosProblema`. 8. Sistema retorna os dados e carrega o algoritmo.
Fluxos alternativos	**Erro ao buscar problemas:** 4.a) Erro na busca – Sistema informa "Não foi possível carregar os problemas". **Falha de conexão:** 4.b) Timeout ou erro HTTP – Sistema informa falha de conexão.

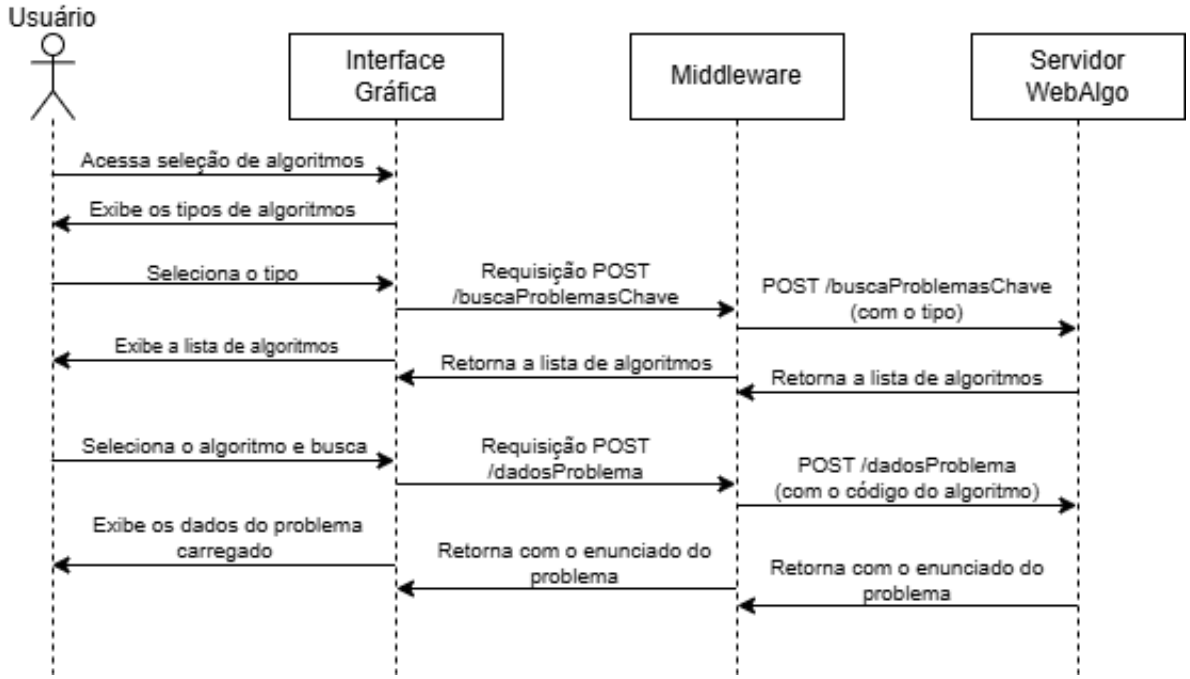
Fonte: O Autor (2025)

Figura 13 – Menu para busca de exercícios



Fonte: O Autor (2025)

Figura 14 – Diagrama de sequência para busca de exercícios



Fonte: O Autor (2025)

4.2 VALIDAÇÃO DA PROPOSTA DE ARQUITETURA

Com a finalidade de avaliar o funcionamento das requisições HTTP do sistema de APIs existente e identificar possíveis problemas nas chamadas realizadas, foram conduzidos testes utilizando a ferramenta *curl*, que permite simular chamadas HTTP diretamente via terminal, acessando o endereço exposto *endereco.com.br*. O levantamento das chamadas foi realizado por meio da análise do código-fonte do WebAlgo, complementado pelas informações obtidas no trabalho desenvolvido por Adriano (MARGARIN, 2018). A seguir, são descritas as etapas executadas durante o experimento.

Inicialmente, foi realizada uma requisição *POST* para o *endpoint /logUsuario*, contendo os parâmetros *username* e *password* no corpo da requisição. Também foram adicionados cabeçalhos HTTP importantes para simular chamadas do tipo *AJAX*, como *Content-Type*, *Referer* e *X-Requested-With*, conforme apresentado no exemplo a seguir:

```
curl -v --insecure \
  -H "Referer: https://endereco.xx.xx/" \
  -H "X-Requested-With: XMLHttpRequest" \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "username=test&password=test" \
  https://endereco.xx.xx/logUsuario
```

Como resposta, o servidor retornou o código *HTTP/1.1 200 OK* e o cabeçalho de resposta contendo o *cookie* de sessão, denominado *sessionId*, evidenciando o sucesso na autenticação do *login*. Esse identificador foi utilizado para validar as requisições subsequentes do usuário, permitindo que ele permanecesse autenticado enquanto interagia com o sistema. No entanto, não foi possível constatar se o *cookie* possuía um tempo de expiração, permanecendo válido até que o usuário realizasse o *logout* ou ocorresse a reinicialização do servidor. Assim, para que a nova interface *web* mantivesse a sessão ativa, a solução implementada consistiu em armazenar temporariamente o *sessionId* no navegador, por meio do *Session Storage*, preservando os dados enquanto a aba permanecia aberta. Dessa forma, a cada requisição realizada ao servidor interno, o valor do *cookie* era recuperado do *Session Storage* e enviado no cabeçalho HTTP, garantindo que a comunicação permanecesse autenticada durante o uso da aplicação. Além do *cookie* de sessão, o retorno também continha um objeto JSON, responsável por identificar o usuário ativo, conforme apresentado a seguir:

```
Set-Cookie: sessionid=19f1f5065f8ec5cf1429a2d8797b9669;
{"respostas": "test"}
```

Os valores retornados, *sessionId* e a resposta com o *user*, foram posteriormente utilizados para testar a finalização da sessão do usuário. A chamada HTTP de *logout* foi executada por meio de uma nova requisição `curl`, desta vez incluindo, no cabeçalho *Cookie*, os valores obtidos anteriormente durante o processo de autenticação:

```
curl -v --insecure \
  -H "Referer: https://endereco.xx.xx/" \
  -H "X-Requested-With: XMLHttpRequest" \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -H "Cookie: sessionid=19f1f5065f8ec5cf1429; name=test" \
  -d "username=nada&password=nada" \
  https://endereco.xx.xx/logoutPortal
```

A resposta foi recebida com sucesso, apresentando novamente o código *HTTP/1.1 200 OK* e o objeto JSON no corpo da resposta, confirmando a finalização da sessão do usuário.

```
{"logout": "test"}
```

Além disso, foram realizados testes complementares utilizando a ferramenta *Postman*. Inicialmente, as chamadas não funcionaram devido a algumas configurações padrão da ferramenta, como a omissão de certos cabeçalhos personalizados e a validação automática de certificados SSL. Após os devidos ajustes, foi possível validar as requisições com sucesso, confirmando que a autenticação e o encerramento da sessão também funcionaram corretamente por meio dessa interface.

Dessa forma, os testes confirmaram que o sistema manteve uma estrutura de autenticação baseada em sessões válidas por meio dos cabeçalhos HTTP, evidenciando que interações diretas com o servidor puderam ser realizadas por um agente externo — aspecto essencial para a validação técnica da camada de *middleware* implementada para integrar a nova interface *web* ao sistema existente.

4.3 ANÁLISE DA SOLUÇÃO ARQUITETURAL

O *middleware* de teste foi construído utilizando Java 19 e o *framework* Spring Boot, sendo responsável por receber as chamadas da interface *web*. O NGINX desempenhou o papel de *proxy* reverso e balanceador de carga, distribuindo as requisições do *frontend* entre as múltiplas instâncias do *middleware*. Por fim, o Docker foi empregado para empacotar todos os serviços, viabilizando sua execução de forma isolada e padronizada em qualquer ambiente com suporte a contêineres.

Para testar a comunicação entre a nova interface *web* e o *middleware*, foi projetada uma estrutura baseada em contêineres utilizando o Docker. Essa abordagem, além de atender ao propósito dos testes, priorizou a facilidade de manutenção, escalabilidade e portabilidade do sistema. Ao virtualizar os serviços por meio de contêineres, foi possível observar o comportamento da aplicação em um ambiente moderno, flexível e alinhado às práticas adotadas no mercado. Além disso, a estrutura desenvolvida contribuiu para aprimorar o desempenho, a organização dos ambientes e o isolamento dos serviços, garantindo que o sistema estivesse preparado para cenários de maior demanda, expansão de funcionalidades e implantação em ambientes de nuvem. Os testes foram compostos por três principais serviços executados nos contêineres: o *frontend*, o *middleware* e o NGINX, que atuou como *proxy* reverso e balanceador de carga.

4.3.1 Arquitetura do Middleware

Para o presente trabalho, o *middleware* foi projetado adotando uma estrutura em camadas, a fim de garantir separação de responsabilidades, organização, facilidade de manutenção e evolução da aplicação. Inicialmente sua principal função será como um redirecionador de chamadas vindas do *front-end* e repassando-as para um servidor interno por meio de chamadas HTTP. No entanto, com a arquitetura em camadas, o projeto poderá ser adaptado para atuar como servidor, visto que será possível realizar as consultas diretamente ao banco de dados, eliminando a necessidade de intermediar outras APIs. Essa divisão segue os princípios das arquiteturas limpas, como a arquitetura hexagonal (Ports and Adapters), possibilitando que a camada de *services* não dependam diretamente da forma como os dados serão buscados, ilustrado na Figura 15. O projeto foi organizado da seguinte forma:

- Controller: Responsável por expor os *endpoints* REST. As requisições realizadas pelo

frontend são tratadas e reencaminhadas para o servidor interno e retornadas em forma de respostas HTTP. Esta camada não possui regras de negócio, apenas manipula o fluxo de requisições.

- Service: Centralizador das regras de negócio e do processamento da aplicação. Esta camada é responsável por coordenar as ações, aplicando decisões, lógicas e validações antes de acionar a última camada, que poderá ser uma nova integração ou acesso ao banco de dados.
- Integration: Responsável pela comunicação externa por meio de chamadas HTTP para o servidor interno. Essa camada é isolada, pois encapsula todos os detalhes da comunicação externa, como URL, autenticação e tratamento das respostas.

No cenário desenvolvido, a estrutura do *middleware* realizou as chamadas REST para o servidor existente. A comunicação com esse servidor ficou sob responsabilidade da camada *integration*, garantindo que quaisquer mudanças na origem dos dados não impactassem a camada de negócios nem os *endpoints* expostos. Dessa forma, ao se tornar viável a integração direta com o banco de dados, a alteração será feita exclusivamente na camada *integration*, onde passará a implementar um repository. As demais camadas, *controller* e *service*, permaneceram intactas, mantendo o mesmo contrato de comunicação com o *frontend* e preservando as regras de negócio existentes.

Uma aplicação projetada dessa forma, permite que a evolução seja feita de maneira controlada e incremental, sem que mudanças internas impactem os consumidores externos, além de contribuir para manutenções futuras e manter boas práticas no desenvolvimento de sistemas.

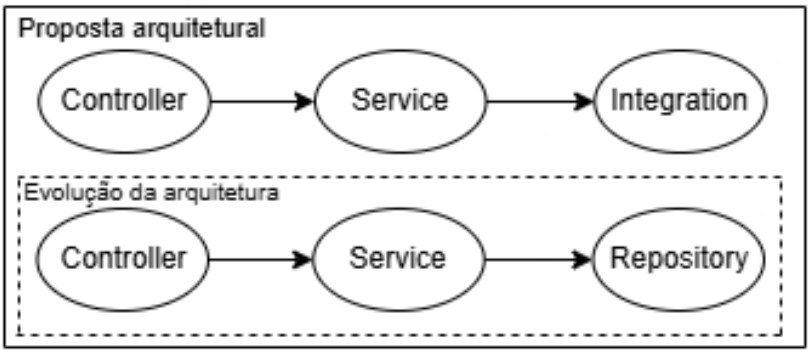


Figura 15 – Diagrama de camadas

Fonte: O Autor (2025)

4.3.2 Virtualização por Containers

A estrutura de contêineres Docker foi composta por quatro serviços principais: o *frontend*, duas instâncias do *middleware* e um contêiner do NGINX atuando como balanceador de

carga e *proxy* reverso. O *frontend* foi disponibilizado como uma aplicação estática na porta `3000`, enquanto o NGINX, exposto na porta `8088`, recebeu as chamadas da interface e as direcionou para uma das instâncias do *middleware*, que executavam a mesma aplicação Java. O balanceamento de carga foi realizado automaticamente por meio do mecanismo *round-robin*, definido nas configurações do NGINX, garantindo a distribuição uniforme das requisições entre os contêineres. Todos os serviços se comunicaram por meio de uma rede interna do Docker, assegurando integração controlada e isolamento entre os componentes da aplicação.

4.3.3 Distribuição e Gerenciamento de Requisições

A configuração do NGINX estabeleceu dois papéis principais na arquitetura: disponibilizar os arquivos estáticos do *frontend* e atuar como *proxy* reverso para as requisições de API. Como *proxy* reverso, o NGINX funcionou como ponto de entrada para as requisições HTTP do *frontend*, reencaminhando as chamadas para os serviços internos conforme as regras definidas.

O bloco `upstream` definiu um *cluster* de instâncias do *middleware*, agrupadas sob o nome `backend_cluster`. Todas as chamadas recebidas no caminho `/api/` foram direcionadas a esse *cluster* por meio da diretiva `proxy_pass`, utilizando o balanceamento de carga *round-robin* para distribuir igualmente as requisições entre as instâncias disponíveis. Essa abordagem aumentou a tolerância a falhas do sistema e melhorou a escalabilidade. Já as chamadas que não eram destinadas ao caminho `/api/` foram tratadas diretamente pelo NGINX, que serviu os arquivos estáticos da aplicação por meio da diretiva `root`, a partir do diretório padrão configurado.

Além disso, o NGINX adicionou cabeçalhos Cross-Origin Resource Sharing (CORS) às respostas, permitindo que a aplicação fosse acessada por diferentes origens. Essa configuração foi essencial para o funcionamento adequado da comunicação entre o *frontend* e os serviços internos por meio do navegador, sem que o cliente precisasse conhecer diretamente as localizações reais dos serviços.

4.3.4 Infraestrutura para Disponibilidade

O uso de contêineres Docker permitiu que a aplicação fosse implementada de maneira escalável e simplificada em diversos ambientes. Ademais, destacaram-se três plataformas de computação em nuvem amplamente utilizadas, que ofereceram suporte completo a esse tipo de estrutura:

- Amazon Web Services (AWS): A AWS disponibiliza diversos serviços de hospedagem para aplicações baseadas em contêineres. Por exemplo, a aplicação pode ser implantada no Amazon Elastic Compute Cloud (EC2), com instâncias virtuais ou soluções mais especializadas, como o Amazon Elastic Container Service (ECS) e o AWS Fargate, que oferecem um serviço de mais alto nível para gerenciar contêineres.

- Oracle Cloud: A Oracle Cloud Infrastructure (OCI), permite a criação de máquinas virtuais, no serviço Always Free e também disponibiliza o Container Engine for Kubernetes (OKE) e o Container Instances, estes permitem o deploy de aplicações de forma escalável, com integração nativa com outros serviços Oracle e com alta disponibilidade.
- Microsoft Azure: A Azure Container Instances (ACI), permite executar contêineres diretamente na nuvem, sem a necessidade de gerenciar servidores. Além disso, para cenários que demandam maior controle, é possível utilizar o Azure Virtual Machines, criando uma infraestrutura sob demanda, ou ainda Azure Kubernetes Service (AKS) para a escalabilidade e orquestração de múltiplos contêineres.

Portanto, a utilização de contêineres proporcionou portabilidade à aplicação, viabilizando sua execução em qualquer ambiente que oferecesse suporte adequado ao Docker, seja em serviços de nuvem ou em servidores locais.

Somado a isso, é importante ressaltar que, para fortalecer ainda mais a escalabilidade e a padronização dos ambientes, prevê-se a possibilidade de adoção do conceito de Infraestrutura como Código (IaC). Com essa abordagem, toda a infraestrutura poderia ser descrita por meio de arquivos de configuração no formato YAML, abrangendo redes, máquinas virtuais, *clusters* Kubernetes, balanceadores e serviços. Dessa forma, seria possível que os arquivos organizados no projeto permitissem a criação automatizada e reprodutível dos ambientes em qualquer servidor local ou nuvem, reduzindo erros manuais e garantindo consistência na infraestrutura. Vale destacar que o uso de IaC foi planejado como uma etapa futura, independente da automação de *deploy* de código (CI/CD), que é responsável por construir, testar e entregar as aplicações.

4.3.5 Gerenciamento e provisionamento da Infraestrutura

Uma maneira eficiente de facilitar a criação, manutenção e reprodução da infraestrutura planejada neste trabalho seria por meio do conceito de IaC. Essa abordagem possibilitaria descrever toda a infraestrutura, incluindo redes, contêineres, balanceadores de carga e máquinas virtuais, em arquivos de configuração no formato YAML. Dessa forma, seria viável automatizar o provisionamento e garantir a entrega de ambientes padronizados e reprodutíveis. Entre as ferramentas amplamente utilizadas, destaca-se o Terraform, por sua sintaxe declarativa e compatibilidade com diversos serviços de nuvem, o que facilitaria a integração com práticas de DevOps (MORRIS, 2020).

No contexto deste trabalho, o uso da infraestrutura como código foi considerado como uma etapa futura, visando permitir que a execução do sistema pudesse ser criada automaticamente em qualquer ambiente. Essa abordagem poderia contribuir para melhorar a escalabilidade, reduzir erros manuais e permitir a criação rápida de novos ambientes. Ademais, como mostra o estudo sistemático de Rahman et al. 2019, o IaC é um recurso importante para aumentar a confiabilidade de ambientes complexos e acelerar as entregas contínuas. Nesse sentido,

a utilização dessa ferramenta foi planejada como uma possibilidade de evolução do WebAlgo, preparando-o para integrações futuras com pipelines CI/CD e aprimorando o controle e a automação da arquitetura.

4.3.6 Observabilidade e Monitoramento

Para garantir a confiabilidade, o desempenho e a capacidade de diagnóstico da aplicação, foi essencial utilizar ferramentas de observabilidade, especialmente em arquiteturas distribuídas que utilizam contêineres. Nesse sentido, foram empregadas as ferramentas Grafana, para visualização de métricas e criação de *dashboards* em tempo real (SALITURO, 2023), e OpenTelemetry, para coleta de *logs* e métricas (BOTEN; MAJORS, 2022). Essas ferramentas permitiram o monitoramento contínuo dos serviços, facilitando a detecção de falhas e o acompanhamento do comportamento do sistema em tempo real. De acordo com Beyer et al. 2016, a observabilidade é uma das práticas essenciais em sistemas modernos, permitindo maior controle sobre a operação de aplicações críticas em produção.

No contexto do WebAlgo, a adoção de ferramentas de observabilidade como o Grafana e o OpenTelemetry mostrou-se fundamental, uma vez que a arquitetura envolveu diversas camadas e serviços interconectados. Com a instrumentação adequada, foi possível visualizar o comportamento do sistema em tempo real, detectar falhas em requisições, analisar a latência entre serviços e avaliar o uso de recursos de forma detalhada.

5 DESENVOLVIMENTO

Este capítulo detalha o desenvolvimento da solução proposta, descrevendo as decisões arquiteturais, a estrutura dos componentes e as tecnologias empregadas. Apresentam-se o *middleware* implementado, as integrações realizadas com o *front-end* já existente, bem como as estratégias de balanceamento de carga e o modelo de monitoramento adotado para garantir desempenho e disponibilidade.

De forma interativa e incremental o desenvolvimento foi conduzido priorizando a modularidade, a escalabilidade e a facilidade de manutenção. Para isso, adotou-se uma abordagem orientada a *containers*, com o uso do Docker, além da aplicação dos princípios da Arquitetura Hexagonal (*Ports and Adapters*) na camada do *middleware*, que atua como ponto central de comunicação entre o *front-end* e as APIs externas, acessível em: <https://github.com/GabrielMoscone/web-algo>.

5.1 VISÃO GERAL DA SOLUÇÃO

A solução desenvolvida tem como objetivo modernizar o funcionamento da aplicação existente WebAlgo, proporcionando uma arquitetura flexível, escalável e compatível com ambientes locais e em nuvem. Para isso, foi criado um *middleware* para receber e tratar as requisições provenientes do *front-end*, redirecionando-as para as APIs do servidor existente do WebAlgo. Assim, o *middleware* atua como uma camada intermediária de abstração entre o cliente e os serviços de *backend*, permitindo a substituição futura do servidor legado sem a necessidade de alterações no *front-end*.

O *middleware* foi implementado em Java 21, utilizando o Spring Boot como principal *framework* de desenvolvimento, possibilitando a criação de *endpoints* REST, a injeção de dependências e a configuração modular de forma simplificada. A escolha dessa tecnologia se deu pela maturidade do ecossistema Java, pela ampla documentação e pela compatibilidade com ferramentas modernas de monitoramento e *deploy* de *containers*.

A Figura 16 apresenta uma visão geral da arquitetura proposta, ilustrando a interação entre os principais componentes do sistema. Nela, observa-se o fluxo de comunicação entre o navegador do usuário, o NGINX, as instâncias do *middleware* e o servidor WebAlgo. Além disso, o diagrama evidencia que o NGINX é responsável tanto por servir os arquivos estáticos da interface *web* quanto por realizar o roteamento e o balanceamento das requisições.

Essa estrutura possibilita o desenvolvimento independente e o versionamento isolado de cada componente. Ademais, o uso do Docker como camada de empacotamento assegura que o ambiente de execução seja idêntico tanto em contextos locais quanto em ambientes de nuvem, como o Azure App Service e o Azure Kubernetes Service.

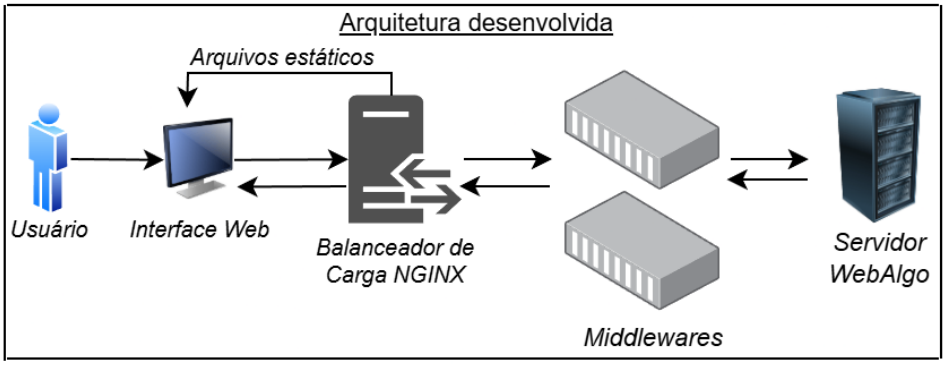


Figura 16 – Visão geral da arquitetura desenvolvida.

Fonte: O Autor (2025)

A arquitetura adotada no *middleware* segue o modelo Hexagonal, que tem como objetivo principal desacoplar o domínio central da aplicação das implementações externas. Com essa abordagem, as regras de negócio permanecem isoladas no centro da aplicação, enquanto os módulos externos (APIs, interfaces *web* e integrações futuras com banco de dados) são tratados como adaptadores conectados às *ports*. Essa estrutura garante que o *middleware* possa evoluir futuramente para um novo servidor, conectado diretamente ao banco de dados, sem a necessidade de grandes modificações.

Toda a solução foi projetada para trabalhar em um ecossistema baseado em *containers* Docker, facilitando a implementação e o gerenciamento padronizado dos serviços. O uso do Docker Compose permitiu a orquestração conjunta dos *containers*, incluindo o NGINX, as instâncias do *middleware*, e os componentes de monitoramento e observabilidade. A estrutura completa da solução está detalhada no Apêndice A, enquanto os arquivos de configuração do ambiente encontram-se reunidos no Apêndice D, permitindo uma compreensão mais aprofundada da implementação.

5.2 DESENVOLVIMENTO DO MIDDLEWARE

Responsável por realizar a intermediação entre o *front-end* e o servidor, o *middleware* constitui o núcleo da solução desenvolvida. Sua função principal é receber as requisições provenientes da interface *web*, processar e validar os dados recebidos, redirecionar as chamadas para as APIs correspondentes e, por fim, retornar as respostas padronizadas ao cliente. Essa camada assegura o isolamento entre a aplicação de interface e as regras de comunicação com o servidor, proporcionando maior flexibilidade e manutenção independente.

O *middleware* foi desenvolvido em Java 21, utilizando o *framework* Spring Boot, que oferece uma base consistente e simplificada para a criação de aplicações *web*. Embora a proposta de solução inicial previsse o uso do Java 19, optou-se pela versão 21 por se tratar de uma versão Long-Term Support (LTS), mais estável e com suporte estendido, garantindo lon-

gevidade ao projeto. O uso dessa tecnologia possibilitou a implementação de *endpoints* REST de forma estruturada, com suporte à injeção de dependências, ao controle de exceções e às configurações de ambiente.

5.2.1 Arquitetura Hexagonal

A criação do *middleware* foi fundamentada na Arquitetura Hexagonal (*Ports and Adapters*), com o objetivo de desacoplar o núcleo de negócio (domínio) dos componentes externos, como APIs, bancos de dados e interfaces. Os controladores REST expõem os *endpoints* como *adapters* de entrada; a camada de aplicação define as portas (*ports*) e implementa os casos de uso e serviços; o domínio concentra os modelos e as exceções; e a infraestrutura configura a integração com serviços externos, conforme ilustrado na Figura 17. Essa abordagem facilita a manutenção, os testes e a evolução independente das integrações.

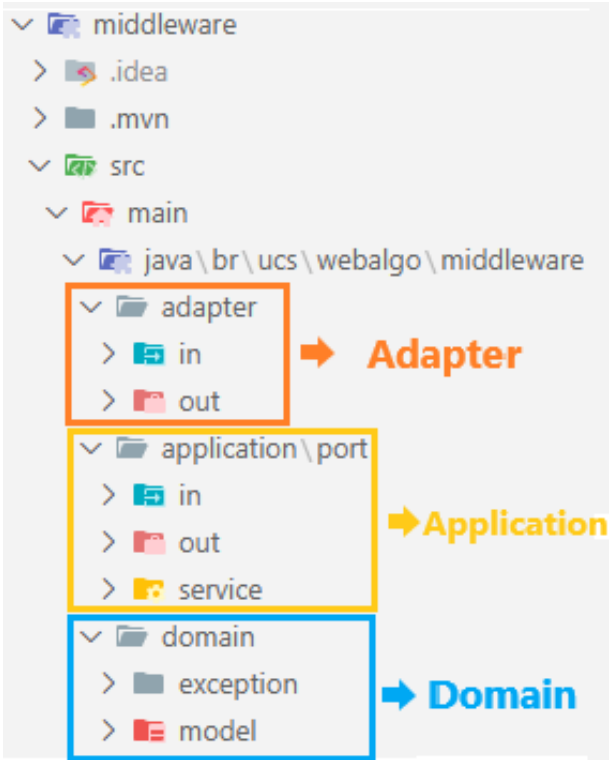


Figura 17 – Visão geral da arquitetura Hexagonal no *middleware*

Fonte: O Autor (2025)

Cada uma dessas camadas possui responsabilidades bem definidas dentro da arquitetura hexagonal. A camada *Adapter* representa os adaptadores de entrada e saída do sistema. Ela é responsável por expor os *endpoints* REST, receber as requisições externas e realizar a comunicação com serviços ou componentes externos, funcionando como a ponte entre o mundo externo e o núcleo da aplicação. A camada *Application* concentra a lógica de aplicação, sendo responsável pela orquestração dos casos de uso. Nela estão definidas as *ports* (interfaces) e suas respectivas implementações de serviço, que conectam o *middleware* às integrações externas e

coordenam o fluxo entre os adaptadores e o domínio. Por fim, a camada *Domain* reúne o núcleo conceitual da aplicação, contendo as entidades e exceções do sistema. No contexto deste projeto, contudo, essa camada não é utilizada de forma ativa, uma vez que o processamento das regras de negócio permanece no servidor legado do WebAlgo, cabendo ao *middleware* apenas intermediar a comunicação entre o novo *front-end* e o sistema existente.

5.2.2 Comunicação com o Front-end

O *front-end* existente foi preservado, mas precisou ser ampliado para se comunicar adequadamente com o *middleware* por meio de requisições HTTP REST. Embora já houvesse uma estrutura inicial da interface, ela não continha diversas funcionalidades necessárias para uso completo do sistema. Dessa forma, novos *endpoints* foram criados seguindo convenções RESTful, utilizando métodos como GET, POST, PUT e DELETE, de acordo com o tipo de operação executada. Cada *endpoint* recebe as requisições do *front-end*, realiza as validações necessárias e as encaminha para a API externa correspondente. Após o processamento, a resposta é convertida e devolvida ao *front-end* em formato JSON, facilitando a integração e o tratamento de dados na camada de interface.

Durante o desenvolvimento, foi necessário estender o *front-end* já existente, implementando funcionalidades que até então não haviam sido desenvolvidas e que eram indispensáveis para o funcionamento do sistema. Entre elas estão as telas de seleção de exercícios, *login*, recuperação de senha e cadastro de novos usuários. Além dessas telas, foram adicionados botões para salvar, criar e buscar problemas, permitindo que o usuário interaja dinamicamente com os recursos disponíveis. Sem essas implementações, o sistema não poderia ser utilizado de forma completa, pois a interface original não contemplava os fluxos necessários para comunicação com o *middleware* e para o uso real do WebAlgo na *web*.

A camada intermediária também desempenha um papel fundamental na segurança e padronização das respostas, impedindo que o *front-end* se comunique diretamente com os serviços externos. Dessa forma, eventuais alterações em *endpoints* internos ou em APIs externas podem ser absorvidas pelo *middleware* sem impactar a interface do usuário.

5.3 CONFIGURAÇÃO DO PROXY REVERSO

O NGINX foi empregado como *proxy* reverso, desempenhando múltiplas funções dentro da solução: disponibilizar os arquivos estáticos do *front-end*, redirecionar as requisições para o *middleware* e realizar o balanceamento de carga entre as instâncias em execução. A configuração foi definida no arquivo `nginx.conf`, localizado no diretório `nginx/` do projeto, contendo todas as diretivas necessárias para o roteamento das requisições, tratamento de sessões e exposição de métricas.

A configuração do NGINX foi estruturada de modo a combinar a entrega eficiente do

front-end com o *proxy* e o balanceamento de carga do *middleware*. O bloco `upstream` adota o algoritmo *round-robin* com conexões *keep-alive* e *failover* passivo, reduzindo o custo de conexão e garantindo a continuidade mesmo diante de falhas temporárias das instâncias. O bloco `/api` preserva os cabeçalhos de origem, aplica *timeouts* e *buffers* otimizados e habilita tentativas (*retries*) seletivas para erros transitórios. Paralelamente, o servidor fornece arquivos estáticos diretamente e expõe métricas via *stub_status*, integrando a camada de borda ao pipeline de observabilidade. Por fim, as regras de redirecionamento baseadas em *cookies* de sessão controlam o acesso ao conteúdo autenticado, mantendo a aplicação responsiva e segura.

O uso do NGINX traz diversos benefícios ao ambiente da aplicação. O balanceamento de carga entre múltiplas instâncias contribui para a redução da latência e evita a sobrecarga de um único *container*, melhorando o desempenho e a escalabilidade horizontal, visto que novas instâncias podem ser adicionadas ao *cluster* com facilidade. Além disso, o NGINX atua como camada intermediária de segurança e isolamento, impedindo o acesso direto do *front-end* aos serviços internos. A escolha dessa tecnologia deve-se à sua leveza, estabilidade e eficiência em ambientes baseados em *containers*. Além de atuar como servidor HTTP, permite definir regras de roteamento, controle de acesso e monitoramento.

A Figura 18 apresenta trechos representativos da configuração do NGINX, evidenciando os principais blocos que compõem a camada de borda da aplicação. O Lado A exibe o *proxy* reverso responsável pelo encaminhamento das requisições da API e pela preservação dos cabeçalhos de origem. O Lado B ilustra o balanceamento de carga entre as instâncias do *middleware*, configurado com *round-robin*, *failover* e conexões *keep-alive*. O Lado C apresenta o controle de sessão por meio de *cookies*, com redirecionamento automático entre a página de *login* e a principal. Por fim, o Lado D mostra o *endpoint* de métricas internas, utilizado para o monitoramento via Prometheus e Grafana.

A

```
# PROXY PARA API
location /api/ {
  proxy_pass http://backend_cluster;
  proxy_http_version 1.1;

  proxy_set_header Connection "";
  proxy_set_header Host $host;
  proxy_set_header X-Real-IP $remote_addr;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header X-Forwarded-Proto $scheme;

  proxy_connect_timeout 10s;
  proxy_send_timeout 60s;
  proxy_read_timeout 60s;

  proxy_buffering on;
  proxy_buffer_size 8k;
  proxy_buffers 32 8k;
  proxy_busy_buffers_size 16k;
}
```

B

```
# UPSTREAM COM ROUND-ROBIN
upstream backend_cluster {
  server middleware1:8080 max_fails=3 fail_timeout=30s weight=1;
  server middleware2:8080 max_fails=3 fail_timeout=30s weight=1;

  keepalive 128;
  keepalive_timeout 60s;
  keepalive_requests 1000;
}
```

C

```
# CONTROLE DE SESSÃO
map $http_cookie $has_session {
  default 0;
  "~*sessionid=" 1;
}
location = / {
  if ($has_session) {
    return 302 /index.html;
  }
  return 302 /login.html;
}
```

D

```
# MÉTRICAS DO NGINX
location /nginx_status {
  stub_status;
  access_log off;
  allow 127.0.0.1;
  allow 172.16.0.0/12;
  deny all;
}
```

Figura 18 – Principais configurações do NGINX.

Fonte: O Autor (2025)

A Figura 19 apresenta uma visão geral do funcionamento do *proxy* reverso, ilustrando

o fluxo das requisições entre o navegador do usuário, o NGINX e as instâncias do *middleware*.

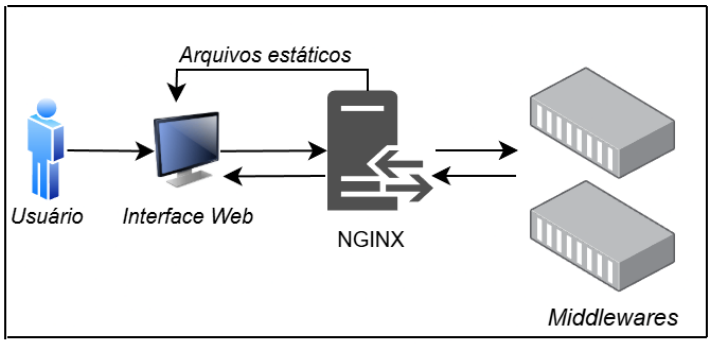


Figura 19 – Visão geral do funcionamento do *proxy* reverso NGINX.

Fonte: O Autor (2025)

A estrutura do NGINX e suas principais funções dentro da arquitetura proposta estão detalhadas no Apêndice B, complementando as informações apresentadas nesta seção.

5.4 ORQUESTRAÇÃO COM DOCKER COMPOSE

Como forma de execução simultânea de todos os componentes do sistema, foi configurado o Docker Compose, responsável pela orquestração dos *containers* que compõem a solução. Dessa forma, é possível automatizar o processo de inicialização, interconexão e monitoramento dos serviços, tornando o ambiente de desenvolvimento e execução mais padronizado, portátil e reprodutível. O arquivo `docker-compose.yml`, localizado na raiz do projeto, define de forma declarativa todos os serviços necessários para o funcionamento do sistema, incluindo as duas instâncias do *middleware*, o NGINX e os módulos de monitoramento implementados.

O uso do Docker Compose traz diversos benefícios ao processo de desenvolvimento e implantação. A padronização do ambiente assegura que o sistema seja executado de forma idêntica em diferentes contextos, eliminando discrepâncias entre os ambientes local e produtivo, simplificando o processo de *deploy* e facilitando a escalabilidade.

No arquivo, cada serviço possui sua própria definição de imagem, variáveis de ambiente, portas expostas, dependências e configurações de rede.

- nginx: atua como ponto de entrada da aplicação, servindo os arquivos estáticos do *front-end* e redirecionando as requisições para o *middleware*.
- middleware1 e middleware2: representam duas instâncias idênticas do *middleware* em execução simultânea. Ambas utilizam a mesma imagem gerada a partir do `Dockerfile`, localizado no diretório `middleware/`.
- Rede interna (*bridge*): todos os *containers* utilizam uma rede interna definida pelo Docker Compose, permitindo a comunicação direta entre os serviços por meio de seus nomes

lógicos. Essa configuração elimina a necessidade de expor portas externas para a comunicação entre *containers*, reforçando a segurança do ambiente.

Além dessas definições, o arquivo também especifica volumes, os quais permitem o compartilhamento de arquivos entre o *host* e os *containers*.

5.5 MONITORAMENTO E OBSERVABILIDADE

Durante o desenvolvimento da solução, foi construída uma arquitetura de monitoramento e observabilidade com o objetivo de acompanhar o comportamento dos serviços, analisar o consumo de recursos e identificar possíveis gargalos de desempenho. Além de apoiar os testes de desempenho realizados durante o desenvolvimento, essa estrutura permanece relevante para a operação contínua do sistema, possibilitando o acompanhamento em tempo real. A implementação foi realizada por meio de *containers* Docker, integrando ferramentas consolidadas ao ecossistema de monitoramento.

Embora a proposta inicial previsse o uso do OpenTelemetry, optou-se pela adoção do Prometheus para a coleta e agregação das métricas, devido à sua maturidade, facilidade de integração e ampla compatibilidade com as demais ferramentas utilizadas.

A arquitetura de observabilidade foi estruturada com base em três pilares fundamentais: métricas, *logs* e visualização. Para isso, foram empregadas as ferramentas *Loki*[1], *Promtail*[2], *Node Exporter*[3], *cAdvisor*[4], *Nginx Exporter*[5], *Prometheus*[6] e *Grafana*[7], compondo um ecossistema completo e de fácil integração.

5.5.1 Coleta de Métricas

O Prometheus foi empregado como principal mecanismo de coleta e armazenamento de métricas. Ele opera consultando periodicamente, por meio de *scrapes*[8], os *endpoints* dos serviços monitorados e armazenando as informações em uma base de dados temporal.

Cada *container* do sistema expõe um *endpoint* de métricas como */actuator/prometheus*, no caso do *middleware*, e */metrics* para os exportadores configurados. O Prometheus coleta dados referentes ao uso de memória, processamento (Central Processing Unit (CPU)), número de requisições, tempo de resposta e taxas de erro, possibilitando a análise quantitativa do desempenho dos componentes.

[1] <https://grafana.com/oss/loki/>
[2] <https://grafana.com/docs/loki/latest/send-data/promtail/>
[3] <https://prometheus.io/docs/guides/node-exporter/>
[4] <https://github.com/google/cadvisor>
[5] <https://github.com/nginxinc/nginx-prometheus-exporter>
[6] <https://prometheus.io/>
[7] <https://grafana.com/>
[8] No contexto do Prometheus, *scrape* é o processo de coleta periódica de métricas, no qual o servidor Prometheus realiza requisições HTTP aos *endpoints* configurados para obter os dados de monitoramento.

Além disso, a configuração foi ajustada para incluir o Nginx Exporter, responsável por capturar estatísticas do *endpoint* `/nginx_status`, fornecendo informações como o número de conexões ativas, requisições por segundo e tempo médio de resposta do servidor *proxy*.

5.5.2 Visualização e Dashboards

O Grafana foi utilizado para a visualização das informações coletadas, integrando-se ao Prometheus e permitindo a criação de *dashboards* personalizados e interativos. Esses painéis exibem, em tempo real, gráficos e indicadores sobre o funcionamento dos *containers* e o tráfego entre os serviços.

Durante o desenvolvimento, foram configurados *dashboards* específicos para o *middleware*, o NGINX e a infraestrutura Docker, possibilitando o acompanhamento de métricas como:

- utilização de memória e CPU das instâncias;
- número de requisições por segundo recebidas pelo *middleware*;
- taxa de erros de requisição;
- tempo médio de resposta por *endpoint*;
- *status* de saúde dos *containers*.

Esses painéis possibilitam validar o comportamento do sistema sob diferentes condições e servem como base para os testes de desempenho realizados.

5.5.3 Centralização de Logs

Para o tratamento dos *logs*, foram utilizados o Loki e o Promtail, ambos desenvolvidos pela Grafana Labs. O Promtail é executado como agente coletor dentro do ambiente Docker, sendo responsável por ler os arquivos de *log* gerados pelos *containers* e enviá-los ao Loki, que atua como servidor de armazenamento e indexação. Entre as informações armazenadas estão os *logs* de acesso e de erro do NGINX, os *logs* de requisições processadas pelo *middleware* e os eventos de inicialização e falhas nos *containers*.

Essa integração permite que todos os *logs* da aplicação sejam centralizados e visualizados diretamente no Grafana, possibilitando a correlação entre eventos e métricas e a identificação rápida da causa de falhas ou picos de consumo.

5.5.4 Métricas do Host e Containers

Com o objetivo de obter uma visão mais ampla da infraestrutura física e virtual do ambiente em execução, foram incluídos o Node Exporter e o cAdvisor. O Node Exporter coleta informações sobre memória, CPU, disco e rede em nível de sistema operacional. Já o cAdvisor monitora especificamente o desempenho dos *containers* Docker, apresentando estatísticas sobre os recursos utilizados, processos ativos e tempo de execução. Esses dados são enviados ao Prometheus, possibilitando a identificação de eventuais gargalos no ambiente de execução, como sobrecarga de CPU ou saturação de memória.

5.5.5 Integração Geral e Benefícios

Todas as ferramentas foram integradas por meio do Docker Compose, o que possibilitou a execução independente de cada serviço dentro de uma mesma rede. Dessa forma, o Prometheus pôde acessar diretamente os *endpoints* dos *containers*, enquanto o Grafana coletou as métricas e os *logs* gerados, assegurando uma integração eficiente entre monitoramento e visualização.

Ademais, a correlação entre métricas e eventos permitiu rastrear com precisão o impacto de erros específicos sobre o desempenho geral da aplicação. A solução também se destacou pela facilidade de escalabilidade, permitindo a adição de novos serviços ao monitoramento apenas por meio da configuração de novos *scrapes* no Prometheus.

Os dados coletados serviram de base para as análises de desempenho realizadas com o Grafana K6, detalhadas na seção seguinte. Essas informações possibilitaram uma avaliação precisa da estabilidade e do comportamento da aplicação sob diferentes níveis de carga.

5.6 TESTES DE DESEMPENHO

Com o objetivo de avaliar o comportamento da aplicação sob diferentes condições de uso, foram realizados testes de desempenho utilizando a ferramenta Grafana K6. Esses testes tiveram como propósito mensurar o tempo de resposta, a capacidade de recuperação, a estabilidade, a ocorrência de falhas, o suporte a múltiplos usuários e o desempenho sob situações de carga progressiva. A execução foi conduzida em ambiente local conteinerizado, por meio do Docker Compose. Todas as execuções foram integradas ao conjunto de monitoramento, composto por Prometheus, Grafana e Loki, possibilitando a visualização em tempo real das métricas de cada cenário.

Ademais, para assegurar maior confiabilidade aos resultados, cada tipo de teste foi executado três vezes. As médias obtidas representam o comportamento típico da aplicação sob as condições impostas, enquanto os desvios e variações foram considerados na interpretação dos resultados. Essa metodologia permitiu identificar tendências consistentes, comparar de forma

equilibrada o desempenho entre as execuções e minimizar oscilações pontuais.

Os testes foram organizados em sete categorias, cada uma voltada a um aspecto específico do sistema:

- Latência: medição do tempo total de resposta entre o cliente, o NGINX e o *middleware*;
- Balanceamento de carga: distribuição das requisições entre as instâncias do *middleware*;
- Tolerância a falhas: análise do comportamento do sistema diante da indisponibilidade de uma instância;
- Validação funcional: verificação do funcionamento geral da aplicação e dos fluxos de autenticação;
- Concorrência de usuários: avaliação da estabilidade sob múltiplos acessos simultâneos;
- Capacidade máxima: determinação do limite máximo suportado pela aplicação antes da degradação perceptível de desempenho;
- Carga progressiva: observação do comportamento do sistema sob aumento gradual de requisições.

Nas subseções seguintes, são apresentados os testes de cada categoria, contendo a descrição do cenário e a média dos resultados das três execuções, expressos em métricas como tempo médio de resposta, percentil 95, disponibilidade e taxa de sucesso.

5.6.1 Teste de Latência

O teste de latência de rede avaliou o tempo total de resposta nas diferentes camadas da aplicação, incluindo o *proxy* reverso e o *middleware*. Esse tipo de teste é essencial para mensurar a eficiência da comunicação e identificar possíveis gargalos de rede ou de processamento no servidor. Cada execução foi realizada com 10 usuários virtuais, durante 3 minutos, totalizando aproximadamente 1.680 requisições por teste. O procedimento foi repetido três vezes para garantir maior consistência estatística, contemplando as seguintes medições:

- *nginx_latency*: tempo gasto no *proxy* reverso, incluindo a conexão TCP e o repasse da requisição;
- *middleware_latency*: tempo de processamento da requisição no *middleware*, considerando o tratamento da lógica de negócio;
- *total_latency*: soma de todos os tempos desde o envio até o recebimento da resposta.

Conforme apresentado na Tabela 2, os resultados indicaram uma latência média total próxima de 70 ms e valores de percentil 95 inferiores a 300 ms, representando respostas rápidas e estáveis mesmo sob múltiplos acessos simultâneos. Ainda que o ambiente de teste apresente baixa latência por se tratar de uma execução local, a diferença entre a latência do NGINX e do *middleware* é perceptível: enquanto o *proxy* reverso apresentou média de apenas 0,22 ms, o *middleware* concentrou o tempo principal de processamento, com média de 70 ms, refletindo a execução das regras internas e o acesso às APIs externas.

Indicador	**Valor Médio Consolidado**
Total de requisições	≈5.040
Tempo médio total de resposta	70 ms
Percentil 95	265 ms
Latência média no *middleware*	70 ms
Latência média no NGINX	0,22 ms
Taxa de sucesso	≈99,99%

Tabela 2 – Métricas consolidadas dos testes de latência.

Fonte: O Autor (2025)

Nas três execuções avaliadas, os resultados mostraram que a variação máxima observada nas médias e nos percentis foi inferior a 10%. Com apenas um desvio isolado, um pico de latência próximo a 3 segundos na última execução, não impactando nas métricas agregadas nem no desempenho geral. Deste modo, o sistema apresentou latências muito baixas e estáveis, com tempo de resposta médio abaixo de 100 ms. A arquitetura baseada em *containers*, aliada ao NGINX, demonstrou boa eficiência, garantindo tempos de processamento uniformes.

5.6.2 Teste de Balanceamento de Carga

O teste de balanceamento de carga teve como objetivo verificar se o NGINX estava distribuindo de forma uniforme as requisições entre as duas instâncias *middleware1* e *middleware2*, configuradas em modo *round-robin*. Esse tipo de teste é essencial para validar a escalabilidade horizontal da aplicação, assegurando que o tráfego seja distribuído adequadamente, sem sobrecarregar uma única instância.

Cada execução teve duração de 5 minutos e envolveu 20 usuários virtuais realizando chamadas contínuas aos *endpoints* do *middleware*, simulando um fluxo real de uso simultâneo do sistema. Foram realizadas três execuções consecutivas, totalizando aproximadamente 2.900 requisições por teste. Em todas as execuções, a taxa de sucesso foi de 100%, sem registro de falhas ou quedas de conexão. O tempo médio de resposta foi de aproximadamente 74 ms, e o percentil 95 manteve-se inferior a 330 ms, confirmando que o balanceamento de carga não introduziu atrasos significativos na comunicação entre as instâncias. Os resultados consolidados estão apresentados na Tabela 3.

Indicador	Valor Médio Consolidado
Total de requisições	8.700
Tempo médio de resposta	≈74 ms
Percentil 95	330 ms
Taxa de sucesso	100%
Erros registrados	0

Tabela 3 – Métricas consolidadas dos testes de balanceamento de carga

Fonte: O Autor (2025)

O NGINX se mostrou eficaz como balanceador de carga, garantindo bom desempenho e distribuição homogênea de carga entre as instâncias do *middleware*, além da constância no consumo de CPU e memória durante todo o período de teste, mantendo 100% de sucesso nas requisições e tempos de resposta médios inferiores a 100 ms, validando a escalabilidade da arquitetura conteinerizada proposta. A distribuição das cargas pode ser observada na Figura 20, e os dados de CPU e memória na Figura 21.

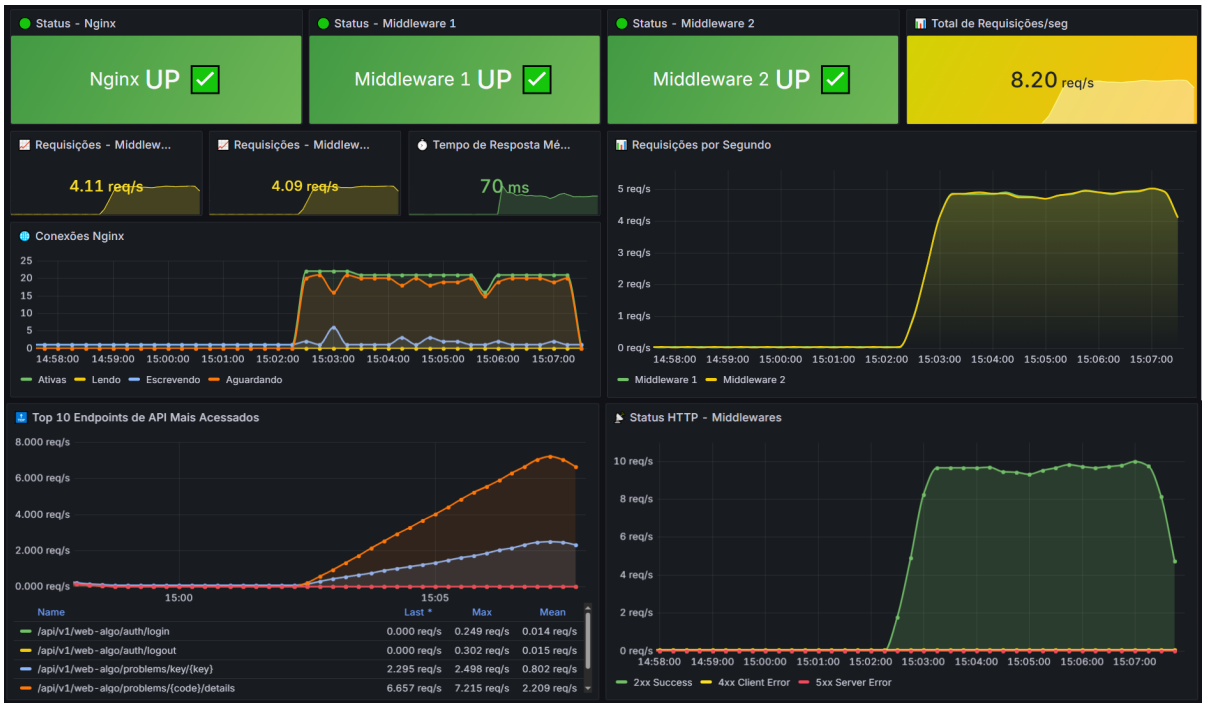


Figura 20 – Painel de monitoramento no Grafana durante o teste de balanceamento de carga.

Fonte: O Autor (2025)

Name	CPU (%)	Memory usage/limit	Memory (%)	Disk read/write
nginx	0.38%	34.38MB / 1GB	3.36%	0B / 12.3KB
middleware2	8.7%	249.1MB / 2GB	12.16%	14.2MB / 1.55MB
middleware1	5.57%	254.4MB / 2GB	12.42%	11.2MB / 1.79MB

Figura 21 – CPU e memória dos *containers* durante o teste de balanceamento de carga.

Fonte: O Autor (2025)

5.6.3 Teste de Failover

O teste de *failover* teve como objetivo avaliar o comportamento da aplicação durante a interrupção temporária de uma das instâncias do *middleware*, verificando a capacidade do balanceador de redirecionar as requisições e garantir a continuidade do serviço. O cenário foi dividido em três fases distintas:

- Normal (0–2 min): ambas as instâncias ativas;
- *Failover* (2–4 min): desligamento manual de uma das instâncias;
- Recuperação (4–6 min): religamento da instância interrompida.

Cada experimento foi executado com 20 usuários virtuais durante 6 minutos, totalizando aproximadamente 9.500 requisições por teste. Os resultados consolidados evidenciam a capacidade do sistema em tolerar falhas e manter boa disponibilidade. A média geral dos três testes está apresentada na Tabela 4.

Observou-se um comportamento consistente em todas as fases do experimento. Durante o período de *failover*, ocorreu apenas um erro isolado, enquanto nas fases normal e de *recuperação* o sistema manteve 100% de sucesso nas requisições. Mesmo com a desconexão de uma das instâncias, o NGINX foi capaz de redirecionar automaticamente as requisições para o *container* ativo, garantindo a continuidade das operações sem necessidade de intervenção manual. Os picos de latência próximos de 10 segundos foram pontuais e esperados, ocorrendo exatamente nos instantes de desligamento ou reinicialização da instância. Fora desses intervalos, o tempo de resposta manteve-se dentro de uma faixa estável, variando entre 50 e 200 ms, validando a eficiência do balanceamento e o comportamento da arquitetura sob condições de falha. Esse comportamento é ilustrado na Figura 22, que exibe o painel de monitoramento do Grafana, destacando o momento de indisponibilidade e a redistribuição automática das requisições, e na Figura 23, que apresenta os dados de CPU e memória dos *containers* durante o teste.

Portanto, o sistema demonstrou boa disponibilidade e resiliência durante as falhas controladas, apresentando recuperação automática e operação contínua do serviço. A arquitetura mostrou-se capaz de suportar a perda temporária de instâncias sem degradação perceptível para o usuário final.

Indicador	Valor Médio Consolidado
Total de requisições	≈28.500
Disponibilidade geral	99,99%
Taxa de falha	0,01%
Tempo médio de resposta	60 ms
Percentil 95	170 ms
Pico máximo de latência	≈10.000 ms

Tabela 4 – Métricas consolidadas dos testes de falha

Fonte: O Autor (2025)

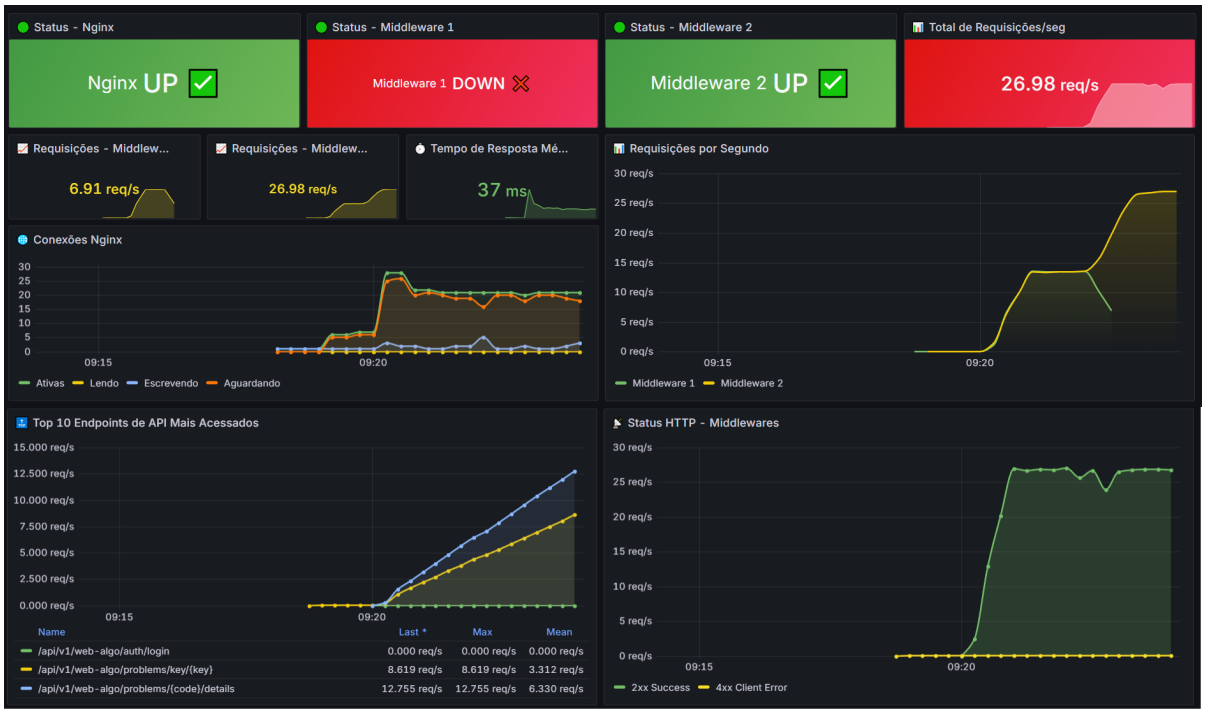


Figura 22 – Painel de monitoramento no Grafana durante o teste de falha.

Fonte: O Autor (2025)

Name	CPU (%)	Memory usage/limit	Memory (%)	Disk read/write
nginx	0.71%	35.82MB / 1GB	3.5%	7.78MB / 4.1KB
middleware2	5.16%	317.1MB / 2GB	15.48%	46.6MB / 1.04MB
middleware1	0%	0B / 0B	0%	0B / 0B

Figura 23 – CPU e memória dos *containers* durante o teste de falha.

Fonte: O Autor (2025)

5.6.4 Teste Funcional da Aplicação

O teste funcional desempenhou papel fundamental na validação do funcionamento completo da aplicação WebAlgo após a integração entre o *front-end*, o NGINX e o *middleware*. O objetivo foi verificar se os principais fluxos, desde o acesso às páginas estáticas até o login e as chamadas autenticadas, estavam operando corretamente. Cada execução foi realizada com apenas um usuário virtual, em um único ciclo de iteração, simulando o comportamento de um usuário real navegando na aplicação. As etapas a seguir descrevem os testes realizados:

1. Disponibilidade: checagem inicial do status da aplicação antes do início da execução;
2. Teste de páginas: validação das rotas principais, verificando o retorno de status 200 e o tempo de carregamento;
3. Fluxo de autenticação: execução de tentativas de login inválido e válido, testando o correto tratamento das respostas HTTP, bem como a criação e captura do *cookie* de sessão;
4. *Logout* e limpeza de sessão: confirmação do encerramento correto da sessão;
5. Testes de erro e exceções: envio de requisições inválidas e métodos incorretos para garantir o retorno adequado dos status 400, 404 e 405.

Todas as execuções tiveram duração aproximada de 3,6 s, e as 37 verificações foram aprovadas com sucesso, indicando que as rotas e fluxos da aplicação estão funcionais e estáveis. O fluxo de login operou corretamente: tentativas inválidas retornaram o status 401 *Unauthorized* e as válidas retornaram 200 *OK*, com o *cookie* `sessionid` devidamente gerado e armazenado no cliente. Os *endpoints* da API (`/problems`, `/solutions`, `/details`) apresentaram respostas coerentes conforme os cenários testados. O tratamento de erros também foi validado: as requisições inválidas retornaram os códigos esperados. Em síntese, as rotas estáticas, os *endpoints* REST e os fluxos autenticados operaram de forma previsível e integrada, obtendo êxito nos 37 casos verificados e tempo médio de resposta inferior a 35 ms.

5.6.5 Teste de Usuários Simultâneos

O teste de usuários simultâneos teve como objetivo avaliar o comportamento da aplicação diante de alta carga concorrente, simulando múltiplos acessos simultâneos aos principais fluxos da plataforma. Essa análise permitiu verificar a estabilidade do sistema, o tempo médio de resposta e a capacidade da arquitetura em sustentar múltiplas conexões ativas. Foram definidos dois cenários paralelos de teste:

1. Múltiplos logins: até 50 usuários virtuais executando o processo completo de autenticação e *logout*;

2. Busca de soluções: até 30 usuários virtuais realizando requisições contínuas aos *endpoints* de pesquisa e visualização de problemas.

No total, cada execução contou com 80 usuários ativos, durante 6 minutos em um ciclo de carga constante. A média consolidada das três execuções está apresentada na Tabela 5.

Durante todas as execuções, a aplicação obteve êxito em 100% das requisições, sem registro de falhas, apresentando tempo médio de resposta de aproximadamente 73 ms e percentil 95 inferior a 350 ms, mesmo sob carga contínua e com 80 usuários concorrendo por recursos.

A análise dos *logs* e das métricas no Grafana evidenciou consumo estável de CPU e memória nas instâncias do *middleware*, além de uma distribuição uniforme do tráfego entre os *containers*, conforme ilustrado na Figura 24. O *throughput* manteve-se consistente ao longo das três execuções, com média de 31 requisições por segundo e tempo médio de iteração de 5,3 s, confirmando a estabilidade do ciclo de autenticação, busca e *logout*. Já os *thresholds* definidos como tempo de resposta no percentil 95 inferior a 2 s e taxa de sucesso de login superior a 80% , foram atingidos com folga. Não ocorreram quedas de sessão, e os tempos de resposta permaneceram homogêneos, indicando bom gerenciamento das conexões simultâneas.

Portanto, o sistema demonstrou excelente estabilidade sob carga concorrente, sustentando 80 usuários simultâneos com tempo médio de resposta inferior a 100 ms e sucesso em todas as requisições. Esses resultados validam a escalabilidade da arquitetura, confirmando a eficácia do balanceamento de carga entre as instâncias do *middleware* mesmo sob alta demanda.

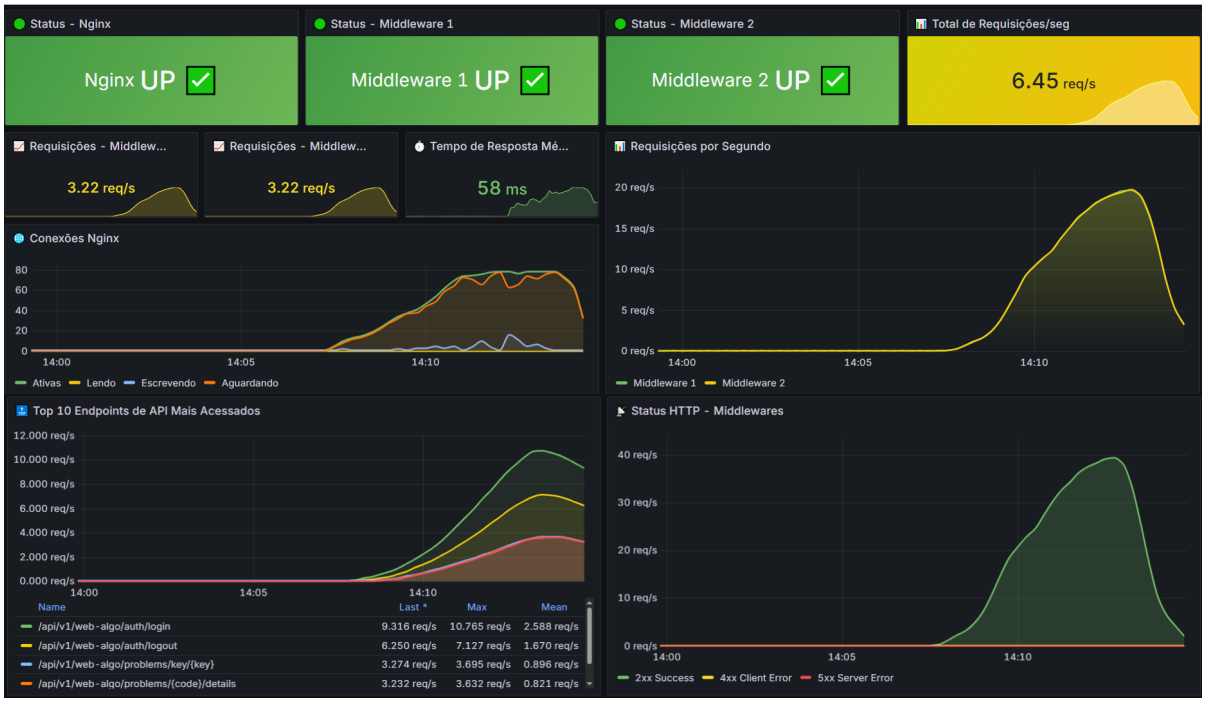


Figura 24 – Painel de monitoramento no Grafana durante o teste de usuários simultâneos.

Fonte: O Autor (2025)

Indicador	Valor Médio Consolidado
Usuários virtuais simultâneos (VUs)	80
Total de requisições	≈13.100
Tempo médio de resposta	73 ms
Percentil 95	342 ms
Requisições por segundo	≈31 req/s
Taxa de sucesso	100%
Taxa de login bem-sucedido	100%

Tabela 5 – Métricas consolidadas dos testes com usuários simultâneos.

Fonte: O Autor (2025)

5.6.6 Teste de Capacidade máxima

O teste de capacidade máxima teve como objetivo identificar o ponto de saturação da aplicação sob carga crescente, determinando o limite de estabilidade operacional da arquitetura e o comportamento do sistema diante da exaustão de seus recursos internos e dependências externas. O ensaio foi estruturado em seis estágios progressivos, variando de 50 a 400 requisições por segundo e utilizando até 500 usuários virtuais. Cada estágio manteve a carga por um intervalo entre 2 e 3 minutos, totalizando aproximadamente 16 minutos de execução.

Durante todo o processo, a aplicação foi monitorada em tempo real por meio do Grafana, utilizando métricas de desempenho coletadas pelo Prometheus e registros de eventos capturados pelo Loki. Para garantir consistência nos resultados, o experimento foi repetido três vezes, de modo a eliminar variações pontuais e consolidar médias representativas do desempenho observado.

Conforme apresentado na Tabela 6, os resultados médios dos três testes indicam uma duração total de aproximadamente 330 s, com 22.414 requisições processadas, das quais 20.890 foram bem-sucedidas e 1.524 resultaram em erro. A latência média foi de cerca de 160 ms, enquanto o percentil 95 apresentou tempo de aproximadamente 387 ms. O pico de usuários virtuais simultâneos atingiu 267 VUs, e a capacidade média sustentada foi estimada em cerca de 63 requisições por segundo.

No início, o sistema manteve 100% de disponibilidade, com tempos médios de resposta inferiores a 100 ms e sem falhas. A degradação iniciou-se a partir de aproximadamente 270 VUs, equivalente a cerca de 250 req/s, quando começaram a ocorrer falhas de conexão (*connection reset by peer*) e respostas HTTP 500 retornadas pelo servidor legado, indicando o surgimento do gargalo da aplicação. Os testes foram interrompidos logo após o aparecimento das primeiras falhas. A métrica de *throughput* médio considera tanto os períodos de estabilidade inicial quanto o início da degradação, resultando em um valor global menor, porém mais fiel à média real de processamento observada. A Figura 25 ilustra o comportamento registrado no painel do Grafana durante essa fase de degradação.

Métrica	Valor Médio Consolidado
Duração do Teste	330,66 s
Requisições Totais	22.414
Sucessos	20.890
Erros	1.524
Taxa de Erro	6,44%
Latência Média	160,75 ms
Percentil 95	387,38 ms
VUs Máximos	267
Capacidade Estimada	63,22 req/s

Tabela 6 – Métricas consolidadas dos três testes de capacidade máxima.

Fonte: O Autor (2025)

É importante destacar a diferença entre o limite de estabilidade e a capacidade média sustentada. O limite de estabilidade (≈250 req/s) representa o maior volume de requisições por segundo em que o sistema ainda opera de forma estável, configurando um pico de operação saudável observado antes do início da degradação. Já a capacidade média sustentada (≈63 req/s) corresponde à média global obtida ao longo de todo o teste, incluindo as fases de *ramp-up* e de degradação. Esse valor reflete o desempenho efetivo mantido ao longo do tempo, e não apenas o ponto de maior estabilidade.

A análise dos *logs* confirmou que as falhas não se originaram no *middleware* nem no NGINX, mas sim nas APIs externas do servidor legado, que passaram a recusar novas conexões TCP quando o volume ultrapassou 250 req/s. Mesmo durante o cenário de sobrecarga, os componentes internos permaneceram estáveis e responsivos, com o *middleware* mantendo tempo médio de processamento entre 90 e 100 ms, sem reinícios de *containers* ou travamentos. A Figura 26 apresenta o comportamento dos recursos de CPU e memória durante essa fase, evidenciando a estabilidade do NGINX e das instâncias do *middleware* sob carga elevada.

Em síntese, o valor de 63 req/s deve ser interpretado como a taxa média de requisições efetivamente processadas, enquanto o ponto de aproximadamente 250 req/s indica o limite máximo de estabilidade operacional da arquitetura antes da influência do gargalo externo. A Figura 27 ilustra esse comportamento, mostrando a falha de conexão observada em uma requisição direta à API externa no momento do colapso.

De forma geral, o sistema demonstrou consistência mesmo diante do aumento progressivo de carga, mantendo desempenho estável e disponibilidade contínua dos serviços. Esses resultados confirmam que a aplicação está apta a operar com eficiência em cenários de uso intensivo, apresentando boa resiliência e controle sob grandes volumes de requisições. Além disso, o comportamento observado durante os testes reforça a boa eficiência da arquitetura conteinerizada e do balanceamento de carga, que contribuíram para a estabilidade do ambiente mesmo em condições próximas ao limite de saturação do servidor existente.

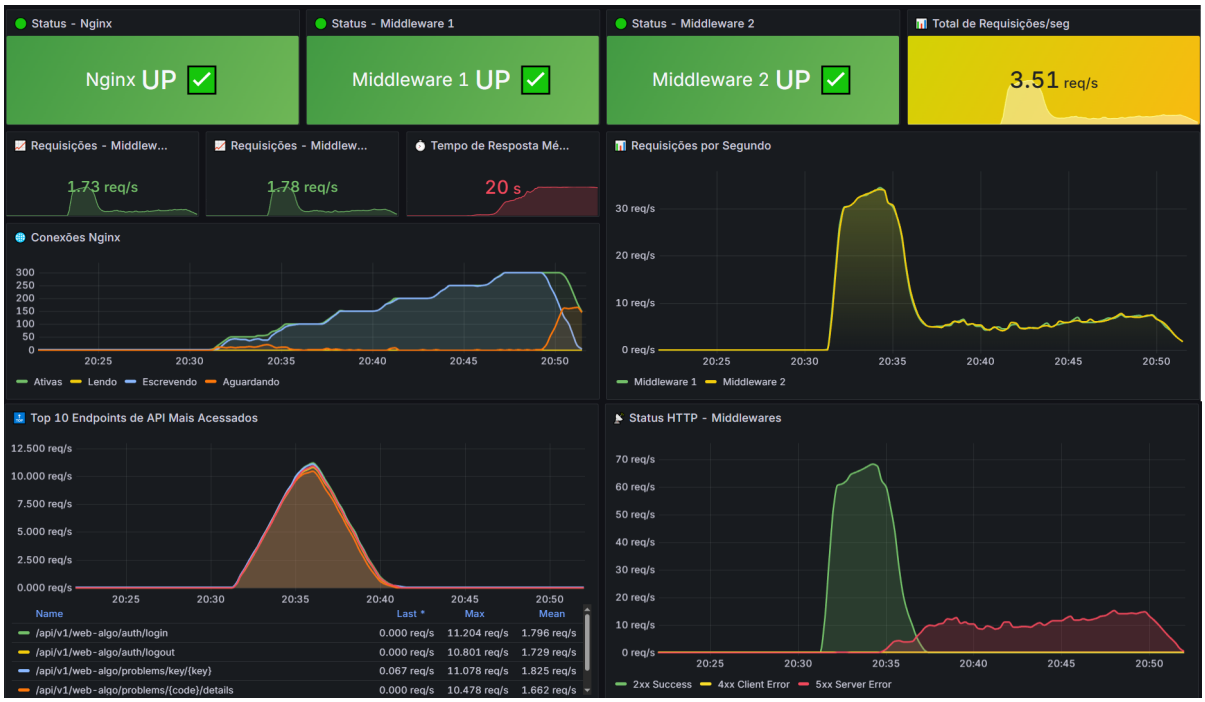


Figura 25 – Painel de monitoramento no Grafana durante o teste de capacidade máxima

Fonte: O Autor (2025)

Name	CPU (%)	Memory usage/limit	Memory (%)	Disk read/write
nginx	1%	36.73MB / 1GB	3.59%	7.79MB / 4.1KB
middleware2	4.18%	356MB / 2GB	17.38%	272MB / 17.5MB
middleware1	3.88%	370MB / 2GB	18.07%	226MB / 16.7MB

Figura 26 – CPU e memória dos *containers* durante o teste de capacidade máxima.

Fonte: O Autor (2025)

```
Microsoft Windows [versão 10.0.19045.6456]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\Guilherme>curl -v --insecure -H "Referer: https://[redacted].ucs.br/" -H "X-Requested-With: XMLHttpRequest" -H
"Content-Type: application/x-www-form-urlencoded" -d "username=gsimioni&password=[redacted]" https://[redacted].ucs.br/lo
gUsuario
* Host [redacted].ucs.br:443 was resolved.
* IPv6: (none)
* IPv4: 200.160.[redacted]
*   Trying 200.160.[redacted]:443...
* connect to 200.160.[redacted] port 443 from 0.0.0.0 port 64455 failed: Timed out
* Failed to connect to [redacted].ucs.br port 443 after 21069 ms: Could not connect to server
* closing connection #0
curl: (28) Failed to connect to [redacted].ucs.br port 443 after 21069 ms: Could not connect to server
```

Figura 27 – Tentativa de requisição direta à API externa, evidenciando falha de conexão.

Fonte: O Autor (2025)

5.6.7 Teste de Carga e Spike

O teste combinado de carga e *spike* teve como objetivo avaliar o comportamento da aplicação diante de dois cenários distintos de estresse. O primeiro, denominado *load_test*, simulou até 100 usuários virtuais executando operações de login, acesso a problemas e soluções, e *logout* de forma contínua por 24 minutos. O segundo, o *spike_test*, introduziu um pico súbito de 200 usuários virtuais adicionais durante 1 minuto e 20 segundos, totalizando 300 VUs simultâneos no momento de maior carga. Dessa forma, buscou-se identificar a capacidade média sustentada da aplicação, sua estabilidade durante longos períodos de operação e a resposta frente a variações abruptas no volume de acessos.

Conforme apresentado na Tabela 7, o teste obteve aproximadamente 98,5% de sucesso ao longo da execução, com latência média inferior a 500 ms e pico de P95 em 1,30 s, demonstrando boa capacidade de resposta mesmo sob condições de carga sustentada. O *throughput* médio estabilizou-se em torno de 47 requisições por segundo, representando um desempenho consistente considerando o tempo de iteração médio de aproximadamente 9 s por usuário virtual.

Métrica	**Valor Médio Consolidado**
Duração total do teste	24 min
Requisições processadas	≈69.183
Sucessos	≈67.980
Falhas	≈1.200
Taxa de falha	1,73%
Latência média	436 ms
Percentil 95	1,30 s
Taxa média de requisições	≈47 req/s
Usuários virtuais máximos	300
Taxa de sucesso no login	78,10%

Tabela 7 – Métricas consolidadas dos três testes combinados de carga e *spike*.

Fonte: O Autor (2025)

A fase de *spike* foi iniciada após 20 minutos de execução, elevando o número de usuários virtuais de 100 para 300. Nesse ponto, observou-se um aumento pontual na latência e o surgimento de falhas na autenticação, indicando um gargalo momentâneo na camada de login. Ainda assim, o sistema manteve-se operacional e recuperou-se rapidamente após o pico. Durante os testes, o *middleware* e o NGINX permaneceram estáveis e responsivos, com *logs* evidenciando que as falhas não se originaram nesses componentes, mas sim nas APIs externas integradas ao *middleware*, conforme ilustrado na Figura 30. Quando o volume de conexões ultrapassou o limite do serviço externo, ocorreram recusas de novas conexões TCP, resultando em respostas HTTP 500.

Nos primeiros minutos, o sistema manteve 100% de disponibilidade, com tempos de

resposta inferiores a 100 ms e sem erros. No decorrer da fase de aumento contínuo de usuários, o sistema permaneceu estável até cerca de 250 req/s, ponto em que começaram a surgir falhas esporádicas de autenticação. A Figura 29 apresenta o comportamento de CPU e memória dos *containers*, enquanto a Figura 28 exibe o painel de monitoramento do Grafana, evidenciando a degradação gradual do *throughput* e o aumento da latência durante o pico do teste.

Os resultados demonstram que a aplicação manteve desempenho satisfatório e comportamento resiliente sob carga prolongada e durante variações de acesso. Mesmo diante de picos de 300 usuários simultâneos, a infraestrutura foi capaz de processar aproximadamente 69 mil requisições, com taxa de falhas inferior a 2% e latência média bem abaixo de 1 segundo, evidenciando a eficiência da arquitetura distribuída e a boa escalabilidade horizontal do sistema.

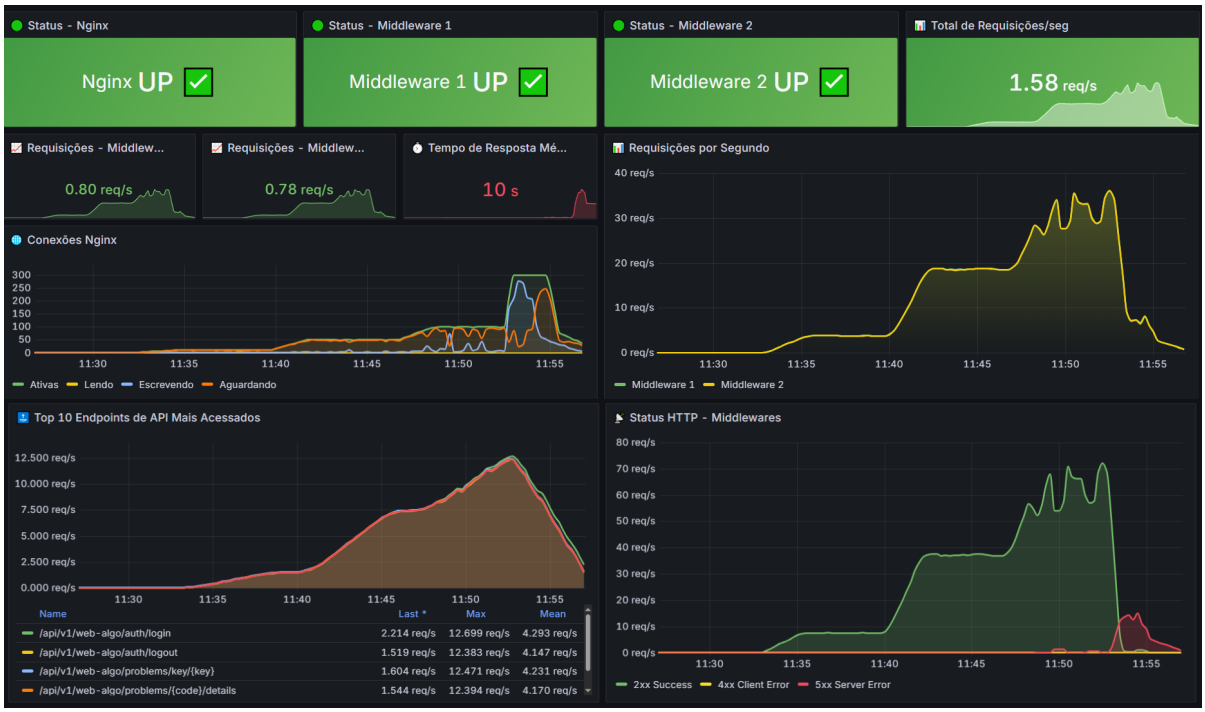


Figura 28 – Painel de monitoramento no Grafana ao final do teste combinado de carga e *spike*.

Fonte: O Autor (2025)

Name	CPU (%)	Memory usage/limit	Memory (%)	Disk read/write	Network I/O
nginx	3.39%	38.25MB / 1GB	3.74%	2.26MB / 12.3KB	368MB / 598MB
middleware2	3.87%	380.3MB / 2GB	18.57%	74.9MB / 6.27MB	146MB / 217MB
middleware1	4.09%	386.1MB / 2GB	18.85%	54.9MB / 6.37MB	145MB / 215MB

Figura 29 – CPU e memória dos *containers* durante o teste de carga e *spike*.

Fonte: O Autor (2025)

Figura 30 – Tentativa de requisição direta à API externa, evidenciando falha de conexão durante os testes de carga.

Fonte: O Autor (2025)

5.6.8 Considerações Gerais dos Testes

Os testes de desempenho realizados com o Grafana K6 permitiram avaliar de forma abrangente a escalabilidade, estabilidade e resiliência da arquitetura proposta. A análise conjunta dos resultados evidencia que o comportamento da aplicação foi consistente nos diferentes cenários simulados, apresentando alta disponibilidade, baixos tempos de resposta e boa eficiência na utilização dos recursos.

Nos testes iniciais, como latência, balanceamento de carga, *failover* e usuários simultâneos, o sistema manteve elevadas taxas de sucesso e tempos médios de resposta inferiores a 100 ms, validando a eficiência do balanceamento realizado pelo NGINX e a integração correta entre os *containers* do *middleware*. Além disso, a arquitetura demonstrou resiliência diante de falhas controladas, recuperando-se automaticamente de interrupções sem impacto perceptível ao usuário final.

Por outro lado, os testes de carga sustentada, *spike* e capacidade máxima evidenciaram os limites operacionais da solução quando submetida a volumes intensos de requisições. Observou-se que, até aproximadamente 250 requisições por segundo, o sistema manteve desempenho estável e latência reduzida. A partir desse ponto, surgiram falhas de conexão e respostas HTTP 500, provocadas pela saturação da API externa integrada ao *middleware*, e não por gargalos internos do sistema. Essa hipótese foi confirmada por meio de testes complementares realizados fora do ambiente conteinerizado, incluindo requisições diretas via `curl` à API externa e o uso da aplicação legada em versão *desktop*, ambos apresentando erros simultâneos durante o período de indisponibilidade das APIs. Mesmo diante dessas condições, o *middleware* e o NGINX permaneceram estáveis, com uso controlado de CPU e memória, confirmando que o colapso observado foi causado pela limitação do serviço externo.

Dessa forma, o valor médio de aproximadamente 63 req/s deve ser interpretado como a capacidade sustentada média da aplicação ao longo de todo o ensaio, enquanto o ponto de

cerca de 250 req/s representa o limite de estabilidade máxima antes do impacto da dependência externa. Esses resultados reforçam que a arquitetura desenvolvida é eficiente e escalável dentro dos limites impostos pelas integrações externas, mantendo comportamento previsível e controlado sob diferentes níveis de carga.

Em síntese, os testes confirmam que a aplicação apresenta alto desempenho, estabilidade e resiliência, validando a viabilidade da arquitetura conteinerizada e o balanceamento horizontal entre instâncias. As falhas observadas em cargas extremas refletem restrições externas ao sistema principal, não comprometendo a robustez e a confiabilidade da solução proposta.

5.7 DEPLOY NA NUVEM

A arquitetura do projeto foi concebida para permitir sua implementação tanto em ambientes locais quanto em plataformas de nuvem, sem a necessidade de alterações estruturais. Todos os componentes foram conteinerizados por meio do Docker, e as configurações são expostas por variáveis de ambiente. Os serviços seguem o princípio *stateless*, o que simplifica processos de escalonamento, atualização e recuperação.

Com essa abordagem, a aplicação pôde ser publicada em plataformas de nuvem para fins de validação prática em sala de aula, possibilitando a execução dos testes funcionais e de desempenho diretamente sobre infraestruturas reais. Essa implantação também permitiu a coleta de métricas de uso e consumo de recursos, fundamentais para estimar a configuração de máquina mais adequada à aplicação e observar o comportamento da solução em diferentes ambientes de nuvem.

Foram adotadas duas estratégias complementares de implantação: o App Service para Linux, com suporte a Docker Compose, e o Azure Kubernetes Service (AKS), destinado à execução orquestrada de *containers* em um *cluster* Kubernetes. A primeira estratégia oferece um caminho rápido e gerenciado para *workloads* multicontêiner, enquanto a segunda fornece orquestração completa, alta disponibilidade e escalabilidade horizontal de *pods* e nós.

5.7.1 Azure App Service

No App Service foi utilizado o modo *multi-container* via Docker Compose, que permite declarar os serviços e suas dependências em um único manifesto, sendo o próprio Azure responsável por provisionar a infraestrutura subjacente, a rede interna entre os *containers*, a coleta básica de *logs* e o reinício automático em caso de falhas.

A solução foi publicada em duas camadas principais: o *front-end* estático, executado pelo NGINX, que serve a interface *web* e atua como *proxy* reverso encaminhando as chamadas das APIs para o *backend*; e o *middleware*, responsável pelo processamento das regras de negócio da aplicação. As imagens dos *containers* estavam previamente versionadas em um Registro de Contêiner do Azure (ACR), e o App Service as consome por meio de *pull* seguro utilizando

credenciais gerenciadas, garantindo integridade e rastreabilidade das versões. As informações sensíveis, como URLs internas, portas e chaves, foram mantidas em Application Settings, evitando sua inclusão no processo de *build* e permitindo variações por ambiente sem necessidade de recompilação das imagens. Essa arquitetura proporciona separação de responsabilidades, implantações imutáveis e maior segurança operacional, além de facilitar *rollbacks* e a gestão de configurações entre ambientes.

Para esse ambiente foi adotado um App Service Plan Básico B1 (Linux), com 1 vCPU e 1,75 GB de RAM. O custo aproximado é de 0,02 USD por hora (cerca de 14,60 USD por mês) por instância, sendo suficiente para os testes práticos do projeto, com a vantagem de permitir a escalabilidade vertical do plano quando necessário, sem alterações na configuração dos *containers*. Esse arranjo atendeu ao objetivo de disponibilizar a aplicação rapidamente, com baixa complexidade operacional e custo previsível.

5.7.2 Azure Kubernetes Service

O Azure Kubernetes Service (AKS) foi utilizado como segunda estratégia de implantação, com o objetivo de demonstrar o funcionamento da aplicação em um ambiente de orquestração completo, dotado de alta disponibilidade, escalabilidade automática e monitoramento nativo. Essa abordagem garante maior controle sobre o ciclo de vida dos *containers*, a distribuição equilibrada de carga entre instâncias e a resiliência da aplicação diante de falhas ou picos de demanda.

O *cluster* foi criado em um sistema operacional Ubuntu Linux com Kubernetes v1.32.7 e utiliza um *node pool* baseado em máquinas virtuais do tipo *Standard_D2s_v6*, contendo 2 vCPUs e 8 GiB de memória RAM por nó. O dimensionamento automático do *pool* foi habilitado, permitindo variação dinâmica entre dois nós ativos, de acordo com a utilização real dos recursos computacionais. Esse recurso, conhecido como Cluster Autoscaler, assegura que o ambiente aloque capacidade adicional sob alta carga e reduza custos automaticamente durante períodos de ociosidade.

Assim como no App Service, as imagens dos *containers* foram hospedadas no ACR e recuperadas pelo AKS, o que aprimora a segurança e o controle de versionamento. Foram implantados dois *deployments* principais:

- *Middleware* (Spring Boot): executando réplicas redundantes, com *probes* de disponibilidade e leitura para garantir reinicialização automática em caso de falhas;
- NGINX: responsável por servir a interface *web* e atuar como *proxy* reverso para o *middleware*.

A comunicação entre os serviços foi realizada por meio de *ClusterIP Services*, assegurando isolamento de rede, enquanto as configurações de ambiente e credenciais foram mantidas

por meio de *ConfigMaps* e *Secrets*, em conformidade com os princípios da Twelve-Factor App.

Além da orquestração de *containers*, o AKS oferece um ecossistema nativo de observabilidade gerenciada, integrando a coleta e a visualização de métricas por meio do Azure Monitor Managed Prometheus e do Azure Managed Grafana. O Prometheus gerenciado elimina a necessidade de manutenção manual de instâncias, realizando a coleta automática de métricas do *cluster* e dos *pods* por meio de *exporters* e *PodMonitors* definidos no Kubernetes. Essas métricas incluem consumo de CPU, uso de memória, número de requisições HTTP e tempo médio de resposta por contêiner. O Managed Grafana, por sua vez, foi configurado como painel de visualização conectado ao *workspace* de métricas do Azure Monitor. Ele possibilita a criação de painéis em tempo real para análise de desempenho da aplicação e do *cluster*, sem necessidade de instalação local. Métricas provenientes do Prometheus e dos *exporters* do NGINX, *middleware* e sistema são integradas automaticamente ao Grafana, fornecendo um panorama completo da saúde operacional do ambiente.

Em síntese, a implantação no AKS consolidou o ambiente da aplicação em uma infraestrutura moderna, escalável e monitorável, combinando os benefícios de um *cluster* Kubernetes gerenciado com as soluções nativas do Azure voltadas à automação, segurança e visibilidade operacional.

5.7.3 Análise de Custos na Nuvem

A Figura 31 apresenta o custo acumulado da assinatura Azure referente ao mês de outubro de 2025, contemplando todos os grupos de recursos utilizados no projeto. Observa-se um custo total de aproximadamente R$156,82, distribuído principalmente entre os grupos *mc_webalgo-rg_webalgo*, *webalgo-rg*, associado ao ambiente do AKS, e *rg-weba*, correspondente ao ambiente do App Service.

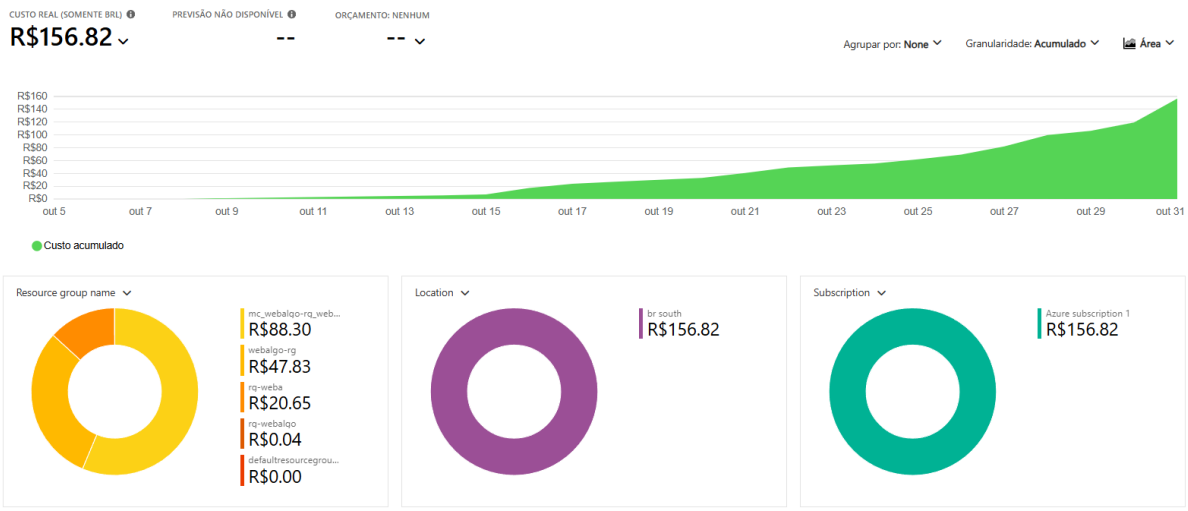


Figura 31 – Custo acumulado da assinatura Azure durante o mês de outubro de 2025.

Fonte: O Autor (2025).

Durante o período de monitoramento, o ambiente do App Service foi criado em 8 de outubro de 2025, enquanto o ambiente do AKS foi implantado em 15 de outubro de 2025, permanecendo ambos em execução até o final do mês. É importante destacar que nenhum dos ambientes permaneceu ativo continuamente: tanto o App Service quanto o AKS foram ativados apenas durante os testes e sessões práticas, sendo desligados logo após o uso. Essa estratégia de operação sob demanda reduziu significativamente o custo final da assinatura.

A Figura 32 apresenta os recursos vinculados ao grupo *rg-weba*, responsável pelo ambiente do App Service. Nele estão incluídos o plano de serviço Linux B1, o ambiente *multi-container* com os aplicativos *frontend-app* e *middleware-app*, o registro de *containers acrweba* e o *workspace* de *logs*. O custo total desse grupo no período foi de aproximadamente R$20,65, evidenciando um ambiente de baixo custo e alta previsibilidade mensal. A Figura 33 apresenta os valores mencionados.

Nome ↑	Tipo	Localização
acrweba	Container registry	Brazil South
env-weba	Ambiente de Aplicativos de Contêiner	Brazil South
frontend-app	Aplicativo de contêiner	Brazil South
middleware-app	Aplicativo de contêiner	Brazil South
plan-weba-linux	Plano do Serviço de Aplicativo	Brazil South
weba-multi	Serviço de Aplicativo	Brazil South
workspace-rgwebaW1Eh	Workspace do Log Analytics	Brazil South

Figura 32 – Recursos associados ao grupo de recursos do App Service.

Fonte: O Autor (2025).

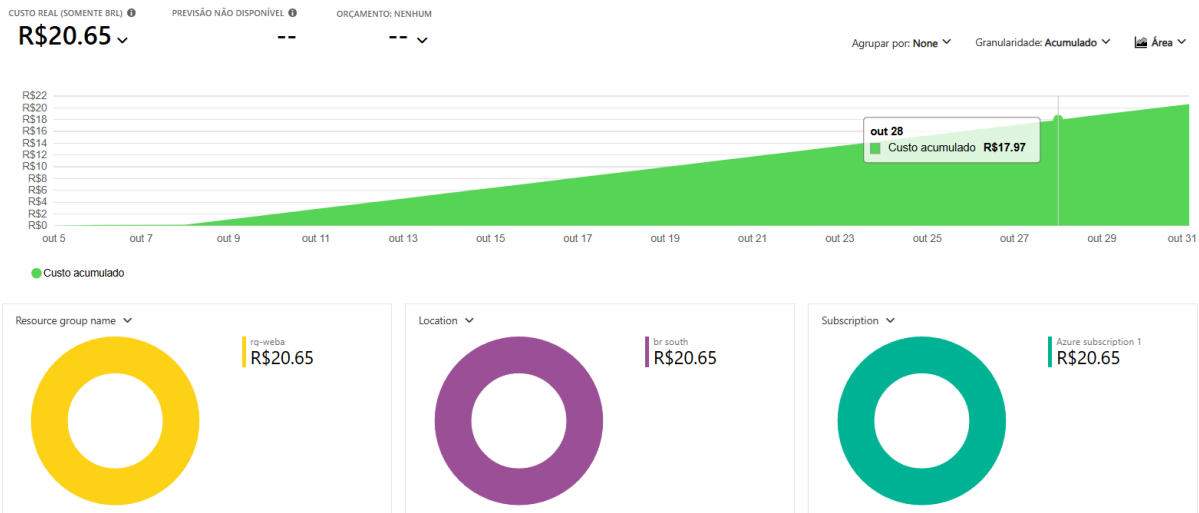


Figura 33 – Custo acumulado dos recursos App Service.

Fonte: O Autor (2025).

Já a Figura 34 apresenta os recursos do grupo *webalgo-rg*, responsável por hospedar o *cluster* AKS. Além do próprio serviço de Kubernetes, o grupo inclui o espaço de trabalho do Azure Monitor, o Grafana gerenciado e múltiplos conjuntos de regras e *exporters* do Prometheus, responsáveis pela coleta e agregação de métricas. O custo consolidado do ambiente AKS no período analisado foi de aproximadamente R$136,14, representando o total acumulado de todos os serviços associados ao *cluster*, incluindo monitoramento, armazenamento e componentes auxiliares de telemetria. A Figura 35 apresenta os valores mencionados.

Nome ↑	Tipo	Localização
amw-webalgo-aks	Espaço de trabalho do Azure Monitor	Brazil South
grafana-webalgo-aks	Espaço Gerenciado do Azure para Grafana	Brazil South
KubernetesRecordingRulesRuleGroup-webalgo-aks	Grupo de regras do Prometheus	Brazil South
MSCI-brazilsouth-webalgo-aks	Regra de coleta de dados	Brazil South
MSProm-brazilsouth-webalgo-aks	Ponto de extremidade da coleta de dados	Brazil South
MSProm-brazilsouth-webalgo-aks	Regra de coleta de dados	Brazil South
NodeAndKubernetesRecordingRulesRuleGroup-Win-webalgo-aks	Grupo de regras do Prometheus	Brazil South
NodeRecordingRulesRuleGroup-webalgo-aks	Grupo de regras do Prometheus	Brazil South
NodeRecordingRulesRuleGroup-Win-webalgo-aks	Grupo de regras do Prometheus	Brazil South
UXRecordingRulesRuleGroup - -webalgo-aks	Grupo de regras do Prometheus	Brazil South
UXRecordingRulesRuleGroup-Win - -webalgo-aks	Grupo de regras do Prometheus	Brazil South
webalgo-aks	Serviço do Kubernetes	Brazil South
webalgoacr	Container registry	Brazil South

Figura 34 – Recursos associados ao grupo de recursos do AKS.

Fonte: O Autor (2025).

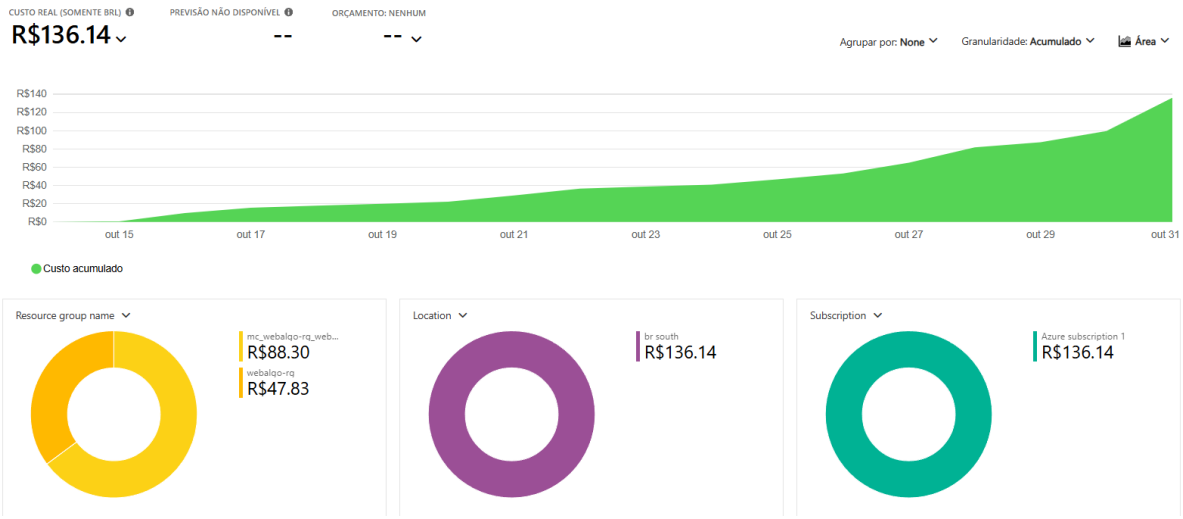


Figura 35 – Custo acumulado dos recursos AKS.

Fonte: O Autor (2025).

5.7.3.1 Custos diários

Os custos diários do App Service apresentaram comportamento estável ao longo do período analisado, conforme ilustrado na Figura 36. Após os primeiros dias de operação, que apresentaram pequenas variações iniciais associadas à ativação do serviço, o custo manteve-se em torno de R$0,83 por dia, refletindo o baixo consumo de recursos do ambiente e a natureza leve da aplicação hospedada no plano básico do Azure.

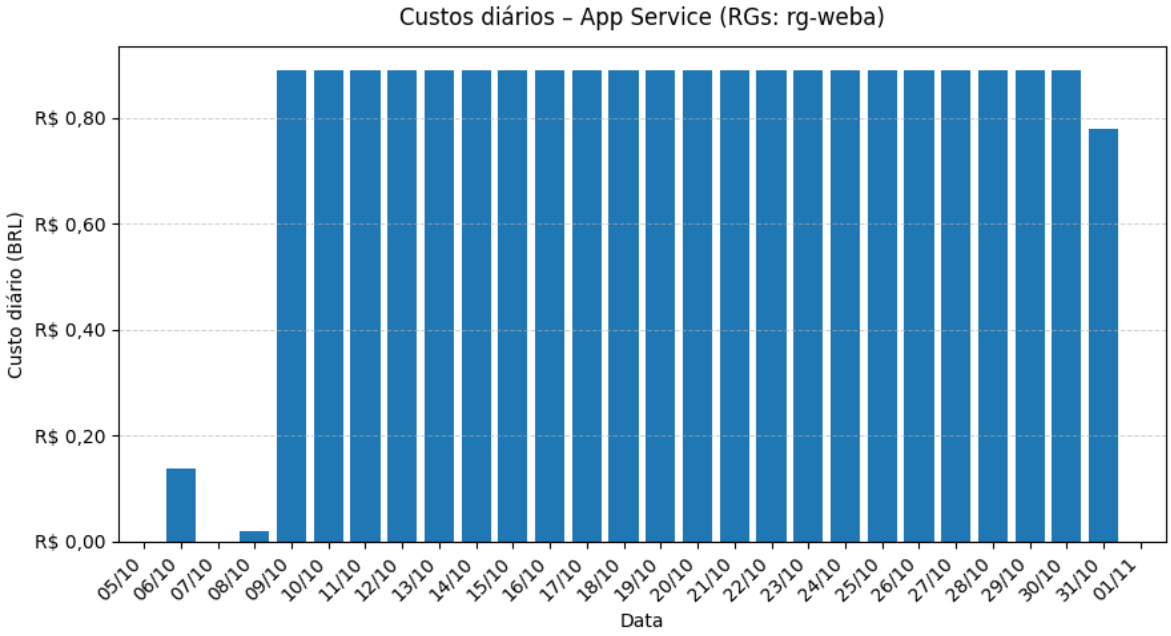


Figura 36 – Custos diários do serviço App Service.

Fonte: O Autor (2025).

Em contraste, o AKS apresentou variação mais expressiva nos custos diários, conforme ilustrado na Figura 37, com valores oscilando entre aproximadamente R$ 2,00 e R$16,00. Essa flutuação evidencia a relação direta entre a carga computacional e o custo variável do *cluster* Kubernetes.

No período total considerado, o custo consolidado foi de aproximadamente R$20,00 para o App Service (16,5%) e de R$104,00 para o AKS (83,5%), totalizando cerca de R$124,00. Destaca-se que o valor referente ao serviço de monitoramento Grafana gerenciado não foi contabilizado nos gráficos diários, uma vez que sua cobrança ocorre de forma mensal, o que explica a diferença em relação ao total de aproximadamente R$156, observado na assinatura completa.

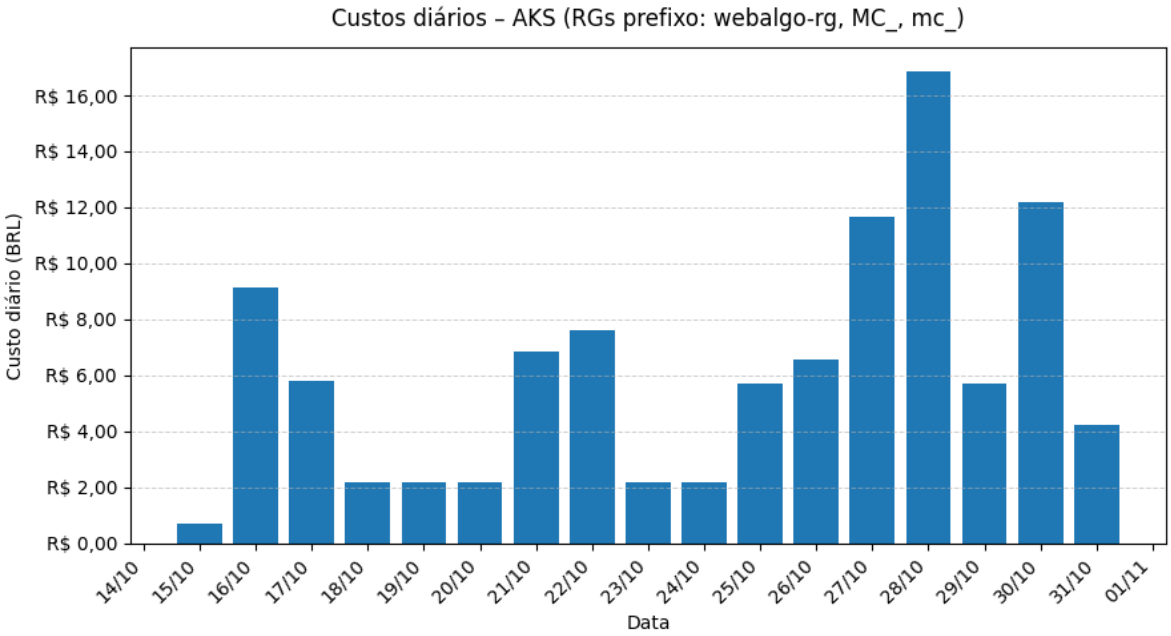


Figura 37 – Custos diários do serviço AKS.

Fonte: O Autor (2025).

5.7.4 Testes com usuários reais

Com o objetivo de validar o comportamento da aplicação em situações reais de uso, foram realizados testes práticos em sala de aula com diferentes turmas, simulando acessos simultâneos e operações típicas do sistema. Os participantes utilizaram a aplicação hospedada tanto no ambiente de *containers* orquestrados pelo AKS quanto no Azure App Service, a fim de comparar o desempenho e a estabilidade entre as duas estratégias de implantação.

1. Cenário 1: 22 alunos utilizando o ambiente AKS;
2. Cenário 2: 9 alunos utilizando o ambiente AKS;
3. Cenário 3: 26 alunos utilizando o AKS e o App Service;
4. Cenário 4: 21 alunos utilizando o AKS e o App Service;
5. Cenário 5: 22 alunos utilizando o App Service.

Os detalhes completos de cada teste, incluindo as métricas coletadas e as imagens dos painéis de monitoramento, encontram-se descritos no Apêndice C.

Durante os testes, os alunos realizaram operações de cadastro, autenticação, busca e criação de problemas, representando o fluxo típico de utilização do sistema. As métricas de uso foram monitoradas em tempo real por meio do Grafana, conectado ao Prometheus gerenciado, e também pelos painéis de gerenciamento nativos do Azure, permitindo observar o comportamento da infraestrutura sob diferentes níveis de carga.

De modo geral, os testes evidenciaram diferenças significativas na utilização de recursos entre os ambientes avaliados. O Azure App Service apresentou maior uso de CPU e memória quando comparado ao AKS, o que reforça a distinção em termos de capacidade e gerenciamento de recursos entre as duas plataformas. Enquanto o AKS manteve consumo reduzido e comportamento estável mesmo sob carga elevada, o App Service demonstrou maior sensibilidade a variações, refletindo sua natureza de serviço mais básico e menos escalável em relação ao ambiente orquestrado por *containers*.

5.7.4.1 Teste 1

O primeiro teste, conduzido com 22 alunos no ambiente AKS, apresentou desempenho estável e eficiente, com baixo consumo de CPU e memória e boa distribuição de carga entre os nós do *cluster*. As métricas demonstraram que o sistema manteve-se responsivo mesmo sob múltiplas requisições simultâneas, sem sinais de sobrecarga ou degradação perceptível.

5.7.4.2 Teste 2

O segundo teste, realizado com 9 alunos, evidenciou comportamento previsível e baixo consumo de recursos, refletindo a leve carga de trabalho aplicada. O ambiente AKS manteve estabilidade de CPU e memória, além de latência média reduzida e tráfego de rede moderado, demonstrando eficiência do balanceamento interno e da alocação de recursos.

5.7.4.3 Teste 3

O terceiro teste, com 26 alunos, comparou diretamente o desempenho do AKS e do App Service. O AKS manteve consumo reduzido e baixa latência, enquanto o App Service apresentou maior uso de CPU e memória sob carga. Apesar de ambos os ambientes responderem adequadamente, o AKS demonstrou desempenho mais equilibrado e consistente, evidenciando maior controle de recursos.

5.7.4.4 Teste 4

No quarto teste, com 21 alunos utilizando o AKS e o App Service, observou-se novamente superioridade do ambiente em *containers*. O AKS manteve consumo reduzido e latência baixa, enquanto o App Service apresentou maior utilização de CPU e tráfego de rede, embora sem instabilidades. O resultado reforça que o AKS entrega melhor eficiência sob cargas controladas, mantendo desempenho constante.

5.7.4.5 Teste 5

O quinto teste, conduzido somente no App Service com 22 alunos, apresentou comportamento estável ao longo da execução, ainda que um pico de 100% de uso de CPU tenha exigido a reinicialização temporária do ambiente. Após a recuperação, o serviço manteve respostas consistentes e desempenho regular, demonstrando boa capacidade operacional. Apesar de sua limitação em escalabilidade e maior consumo de recursos, o App Service mostrou-se adequado para cenários de teste e uso controlado do sistema.

6 CONSIDERAÇÕES FINAIS

Este trabalho propõe, desenvolve e valida uma arquitetura para o sistema educacional WebAlgo, tornando-o acessível pela web e compatível com tecnologias atuais. A proposta contemplou a criação de um *middleware* desenvolvido em Java, responsável por mediar a comunicação entre o novo *front-end* e o servidor legado, além da utilização do NGINX como *proxy* reverso e balanceador de carga, todos executados em *containers* Docker.

A implementação seguiu princípios da Arquitetura Hexagonal, favorecendo a modularidade e a independência entre as camadas da aplicação, permitindo futuras evoluções e mitigando o impacto no sistema. Durante a criação da aplicação, foram aplicadas boas práticas de desenvolvimento e observabilidade, com o uso de ferramentas como Prometheus, Grafana e Loki, garantindo o monitoramento contínuo de desempenho e confiabilidade do ambiente.

Os resultados obtidos nos testes de desempenho demonstraram que a solução é capaz de suportar $\approx$70 requisições por segundo sem comprometer o sistema, mantendo tempos de resposta reduzidos (latência média $\approx$160 ms; P95 $\approx$387 ms). Além disso, os testes com usuários reais mostraram que o App Service apresenta-se adequado para ambientes de ensino com uso de aproximadamente 20 usuários simultâneos, enquanto o AKS se mostrou mais adequado por oferecer um ambiente mais completo, com maior capacidade de gerenciamento e suporte a múltiplos serviços integrados, configurando-se como alternativa para cenários de maior uso.

6.1 TRABALHOS FUTUROS

Como trabalhos futuros, sugere-se a integração direta do *middleware* com o banco de dados institucional, eliminando a necessidade de sistemas intermediários e permitindo maior eficiência nas consultas e atualizações de dados. Além disso, recomenda-se a implementação de pipelines de integração e entrega contínua (*CI/CD*), a fim de automatizar o processo de atualização, execução de testes e implantação de novas versões do sistema. Por fim, a adoção de práticas de *Infraestrutura como Código* (*IaC*), possibilitando o provisionamento automatizado e reprodutível dos ambientes, facilitando a manutenção e o escalonamento da solução em diferentes contextos de implantação.

REFERÊNCIAS

ABDELLATIF, M. *et al.* A taxonomy of service identification approaches for legacy software systems modernization. **Journal of Systems and Software**, v. 173, p. 110868, 2021. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121220302582>.

ADAM, B. M.; BESARI, A. R. A.; BACHTIAR, M. M. Backend server system design based on rest api for cashless payment system on retail community. In: **2019 International Electronics Symposium (IES)**. [S.l.: s.n.], 2019. p. 208–213.

AMARAL, E. *et al.* Algo+ uma ferramenta para o apoio ao ensino de algoritmos e programação para alunos iniciantes. In: . [S.l.: s.n.], 2017. p. 1677.

BEYER, B. *et al.* **Site reliability engineering: how Google runs production systems**. [S.l.]: "O'Reilly Media, Inc.", 2016.

BOTEN, A.; MAJORS, C. **Cloud-Native Observability with OpenTelemetry: Learn to gain visibility into systems by combining tracing, metrics, and logging with OpenTelemetry**. [S.l.]: Packt Publishing Ltd, 2022.

CAVALCANTE, D.; SILVA, A. T. da; VITORINO, A. Lsgames: Plataforma de jogos educacionais para o ensino de matemática para surdos através da libras. In: **Anais da XX Escola Regional de Computação Bahia, Alagoas e Sergipe**. Porto Alegre, RS, Brasil: SBC, 2020. p. 203–208. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/erbase/article/view/15478>.

CAZZOLA, W.; FAVALLI, L. Software modernization powered by dynamic language product lines. **Journal of Systems and Software**, v. 218, p. 112188, 2024. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121224002322>.

CONNOLLY, R.; HOAR, R. **Fundamentals of Web Development, Global Edition**. 1. ed. [S.l.]: Pearson Education Limited, 2015. 1024 p.

CRUZ, A. K. B. S. d. e. a. Utilização da plataforma beecrowd de maratona de programação como estratégia para o ensino de algoritmos. In: SBC. **Anais Estendidos do XXI Simpósio Brasileiro de Jogos e Entretenimento Digital**. [S.l.], 2022. p. 754–764.

DALMIA, A.; CHOWDARY, A. R. The new era of full stack development. **International Journal of Engineering Research & Technology (IJERT)**, v. 09, n. 04, April 2020. Volume 09, Issue 04 (April 2020).

DORNELES, R. V.; JR, D. P.; ADAMI, A. G. Algoweb: a web-based environment for learning introductory programming. In: **2010 10th IEEE International Conference on Advanced Learning Technologies**. [S.l.]: IEEE, 2010. p. 83–85.

EMMERICH, W. Software engineering and middleware: a roadmap. In: **Proceedings of the Conference on the Future of Software Engineering**. [S.l.: s.n.], 2000. p. 117–129.

GAZIS, A.; KATSIRI, E. Middleware 101: What to know now and for the future. **Queue**, Association for Computing Machinery, New York, NY, USA, v. 20, n. 1, p. 10–23, mar. 2022. ISSN 1542-7730. Disponível em: <https://doi.org/10.1145/3526211>.

IBGE. **Pesquisa Nacional por Amostra de Domicílios Contínua - 2023**. Instituto Brasileiro de Geografia e Estatística, 2023. Acesso em: 20 nov. 2024. Disponível em: <https://www.ibge.gov.br/estatisticas/sociais/trabalho/9171-pnad-continua.html>.

JARAMILLO-ALCAZAR, A. *et al.* Towards an accessible mobile serious game for electronic engineering students with hearing impairments. In: **2018 IEEE World Engineering Education Conference (EDUNINE)**. [S.l.: s.n.], 2018. p. 1–5.

KIRILOV, N.; DUGAS, M. Evaluation of technical approaches for real-time data transfer from electronic health record systems. **Computer Methods and Programs in Biomedicine**, v. 255, p. 108347, 2024. ISSN 0169-2607. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0169260724003407>.

KOLB, D. A. **Experiential Learning: Experience as the Source of Learning and Development**. 2. ed. Upper Saddle River: FT Press, 2014. ISBN 9780133892406.

KOVACS, A. M. Comparative analysis of traditional and modern proxy solutions in cyber security. **International Journal of Communication Networks and Information Security (IJCNIS)**, v. 16, n. 2, Aug. 2024. Disponível em: <https://ijcnis.org/index.php/ijcnis/article/view/6689>.

KUTSCHER, V. *et al.* Upgrading of legacy systems to cyber-physical systems. **Proceedings of TMCE 2020**, 2020.

LEI, Z. *et al.* Cost-effective server-side re-deployment for web-based online laboratories using nginx reverse proxy. **IFAC-PapersOnLine**, v. 53, n. 2, p. 17204–17209, 2020. ISSN 2405-8963. 21st IFAC World Congress. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2405896320323545>.

LIMA, P. D. da C. *et al.* Libras tech: Software educacional para o ensino gamificado da informática adaptado a libras. **Anais do Computer on the Beach**, v. 15, p. 051–056, 2024.

MARGARIN, A. **Evolução da Ferramenta de Gerenciamento do Portal de Algoritmos da UCS**. Trabalho de Conclusão de Curso (Bacharelado) — Universidade de Caxias do Sul, Caxias do Sul, RS, 2018.

MIOTTO, F. **Desenvolvimento de um compilador e uma máquina virtual de Python para o ambiente Webalgo**. 2019. Disponível em: <https://repositorio.ucs.br/xmlui/handle/11338/6308>. Disponível em: <https://repositorio.ucs.br/xmlui/handle/11338/6308>.

MIRANDA, A. M. L. **Integração de Sistemas Legados para Aprendizado a Distância: Estudo de Caso em Planejamento de Sistemas Móveis Celulares**. Dissertação (Dissertação de Mestrado) — Universidade Federal do Pará, 2003. Programa de Pós-Graduação em Engenharia Elétrica.

MORRIS, K. **Infrastructure as code**. [S.l.]: O'Reilly Media, 2020.

OLIVEIRA, J. B. de *et al.* Integração de sistemas legados e atuais em instituição de registro público utilizando um gateway para comunicação de diferentes plataformas: Using a gateway for integration of different platforms systems in public registry institution. **Brazilian Journal of Development**, v. 8, n. 7, p. 54376–54381, 2022.

OLUWATOSIN, H. S. Client-server model. **IOSR Journal of Computer Engineering**, v. 16, n. 1, p. 67–71, 2014.

PRATES, R. T. C. **Libras Game: trabalhando o ensino da matemática com alunos surdos dos anos iniciais através do uso de aplicativo educacional**. Dissertação (B.S. thesis) — Universidade Tecnológica Federal do Paraná, 2018.

QIAN, Y.; LEHMAN, J. Students' misconceptions and other difficulties in introductory programming: A literature review. **ACM Transactions on Computing Education (TOCE)**, ACM New York, NY, USA, v. 18, n. 1, p. 1–24, 2017.

RAHMAN, A.; MAHDAVI-HEZAVEH, R.; WILLIAMS, L. A systematic mapping study of infrastructure as code research. **Information and Software Technology**, v. 108, p. 65–77, 2019. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584918302507>.

REIS, T. *et al.* Ensino-aprendizagem de programação: uma revisão sistemática da literatura. **Revista Brasileira de Informática na Educação**, v. 23, p. 182, 04 2015.

ROSENBERG, D.; STEPHENS, M. **Use Case Driven Object Modeling with UML: Theory and Practice**. 1. ed. [S.l.]: Apress, 2007. 440 p.

SALITURO, E. **Learn Grafana 10. x: A beginner's guide to practical data analytics, interactive dashboards, and observability**. [S.l.]: Packt Publishing Ltd, 2023.

SOMMERLAD, P. Reverse proxy patterns. In: **European Conference on Pattern Languages of Programs**. [s.n.], 2003. p. 431–458. Disponível em: <https://api.semanticscholar.org/CorpusID:44532344>.

STEEN, M. van; TANENBAUM, A. S. **Distributed Systems: Principles and Paradigms**. 4th. ed. [S.l.]: Maarten van Steen, 2023. [S.l.]. ISBN 9081540637, 9789081540636.

SUSIN, G. L. **WEBALGO: Desenvolvimento de um Compilador da Linguagem C e uma Máquina Virtual Baseada em Registradores Utilizando C3E na Web**. Caxias do Sul: [s.n.], 2024. [s.n.]. Trabalho de Conclusão de Curso. Orientador: Prof. Dr. Ricardo Vargas Dorneles.

WIEGERS, K. E.; BEATTY, J. **Software Requirements**. 3. ed. [S.l.]: Microsoft Press, 2013. 672 p.

ZHANG, Z. *et al.* A new remote web-based mdsplus data visualization system for east. **Fusion Engineering and Design**, v. 186, p. 113337, 2023. ISSN 0920-3796. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0920379622003283>.

APÊNDICE A – ESTRUTURA DO PROJETO

Visando à clareza na divisão de responsabilidades e à facilidade de manutenção, a aplicação foi projetada em uma estrutura modular. O diretório principal, denominado `web-algo`, reúne todos os componentes necessários para a execução completa da solução, apresentados na Figura 38, tais como o *middleware*, o NGINX, o *front-end*, os serviços de monitoramento e os *scripts* de testes de desempenho.

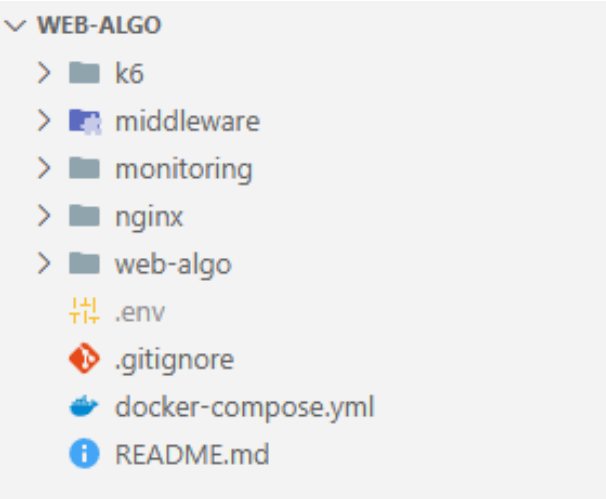


Figura 38 – Diretório web-algo

Fonte: O Autor (2025)

A seguir, estão listadas as principais pastas e arquivos que compõem o projeto:

- *middleware*: diretório que contém o código-fonte do *middleware*, desenvolvido em Java 21 com o *framework* Spring Boot. Essa camada é responsável por receber as requisições do *front-end*, processar as informações e redirecioná-las para as APIs do servidor interno da UCS.
- `nginx`: diretório responsável pela configuração do *proxy* reverso, atuando como ponto de entrada das requisições HTTP. O NGINX serve os arquivos estáticos do *front-end* e realiza o balanceamento de carga entre as instâncias do *middleware*.
- `web-algo`: contém o código do *front-end* existente, composto por páginas HTML, CSS e JavaScript. Durante o desenvolvimento, foram implementadas novas funcionalidades, como as telas de *login*, cadastro de usuários, recuperação de senha e seleção de exercícios, além da integração com as chamadas REST do *middleware*.
- *monitoring*: agrupa os arquivos de configuração de monitoramento e observabilidade, utilizando ferramentas como Prometheus, Grafana, Loki, Promtail, Node Exporter e cAd-

visor. Esses serviços permitem a coleta de métricas e *logs* dos *containers*, bem como a visualização em tempo real por meio de painéis personalizados.

- `k6`: diretório que reúne os *scripts* de testes de desempenho e carga desenvolvidos com o Grafana K6, empregados para avaliar o comportamento do sistema sob diferentes níveis de requisições simultâneas, validando a escalabilidade e os tempos de resposta da aplicação.
- `docker-compose.yml`: arquivo principal de configuração dos *containers*, que define todos os serviços que compõem a aplicação, as redes internas, dependências e volumes compartilhados. Esse arquivo possibilita a inicialização automatizada de todo o ecossistema por meio de um único comando.

APÊNDICE B – ESTRUTURA NGINX E FUNÇÕES PRINCIPAIS

A configuração do NGINX está dividida em seções que controlam o comportamento do servidor. As mais relevantes são:

- Mapeamento de sessão: o NGINX reconhece a existência de uma sessão válida por meio do *cookie* `sessionid`. Assim, o usuário é direcionado automaticamente para a página correta, garantindo uma navegação fluida e segura.
- Controle de acesso: foram definidas regras específicas para proteger as rotas da aplicação. Caso o usuário tente acessar diretamente a tela principal (`index.html`) sem possuir uma sessão ativa, o NGINX realiza o redirecionamento automático para a tela de *login* (`login.html`). Por outro lado, ao acessar a raiz do site (`/`), o servidor verifica a presença de um *cookie* de sessão e encaminha o usuário para a página correspondente.
- Serviço de arquivos estáticos: responsável por entregar ao cliente os arquivos do *front-end* (HTML, CSS e JavaScript). A raiz do diretório foi configurada para apontar ao local em que os arquivos estáticos são armazenados dentro do *container*. Dessa forma, o *front-end* é servido diretamente pelo NGINX, sem depender do *middleware* para o carregamento das páginas.
- Balanceamento de carga: o bloco `upstream` define o agrupamento de duas instâncias idênticas do *middleware*, configuradas em modo *round-robin*. As requisições são distribuídas de forma equilibrada entre as instâncias, o que proporciona maior disponibilidade e mantém a aplicação responsiva mesmo sob alta demanda.
- Redirecionamento para a API: todas as requisições com o prefixo `/api` são interceptadas e encaminhadas ao *cluster* do *middleware*, garantindo a comunicação correta entre o *front-end* e as APIs. Esse roteamento preserva os cabeçalhos originais de requisição, como `Host`, `X-Real-IP` e `X-Forwarded-For`, permitindo rastreabilidade e consistência nos *logs*.
- Métricas e monitoramento: o *endpoint* `/nginx_status` foi configurado para expor métricas internas do servidor, como o número de requisições processadas e de conexões ativas. Essas informações são coletadas periodicamente pelo Nginx Exporter e integradas ao sistema de monitoramento composto por Prometheus e Grafana.

APÊNDICE C – TESTES EM SALA DE AULA

Este apêndice apresenta os resultados dos testes realizados em sala de aula, conduzidos com turmas reais durante o uso do sistema. O objetivo é demonstrar o comportamento da aplicação em um ambiente de ensino, analisando sua estabilidade e desempenho. A seguir, são descritos os testes efetuados e suas principais observações.

C.1 TESTE 1

O primeiro teste, realizado com 22 alunos no ambiente AKS, evidenciou estabilidade operacional consistente e uso eficiente dos recursos do *cluster*. Durante toda a execução, a utilização de CPU manteve-se em níveis reduzidos, com média de aproximadamente 1,25% e picos pontuais apenas nos momentos de maior atividade. Esse comportamento confirma a capacidade do sistema de lidar com múltiplas requisições simultâneas sem sobrecarga perceptível, assegurando boa distribuição de carga entre os nós do ambiente.

Em relação à memória, observou-se um padrão linear e controlado, com valores variando entre 190 MB e 250 MB por contêiner. A ausência de oscilações abruptas ou de crescimento progressivo no consumo indica uma gestão consistente dos recursos e ausência de vazamentos, fatores que reforçam a confiabilidade da aplicação sob condições normais de uso.

O tráfego de requisições HTTP por *pod* apresentou comportamento estável, atingindo até 0,7 requisições por segundo nos picos de interação dos alunos, com predominância de respostas bem-sucedidas. Esses resultados demonstram eficiência no balanceamento e boa responsividade da aplicação, com tempos de resposta adequados e sem degradação perceptível. As métricas correspondentes, apresentadas na Figura 39, evidenciam visualmente o comportamento estável do ambiente AKS durante a execução do teste.

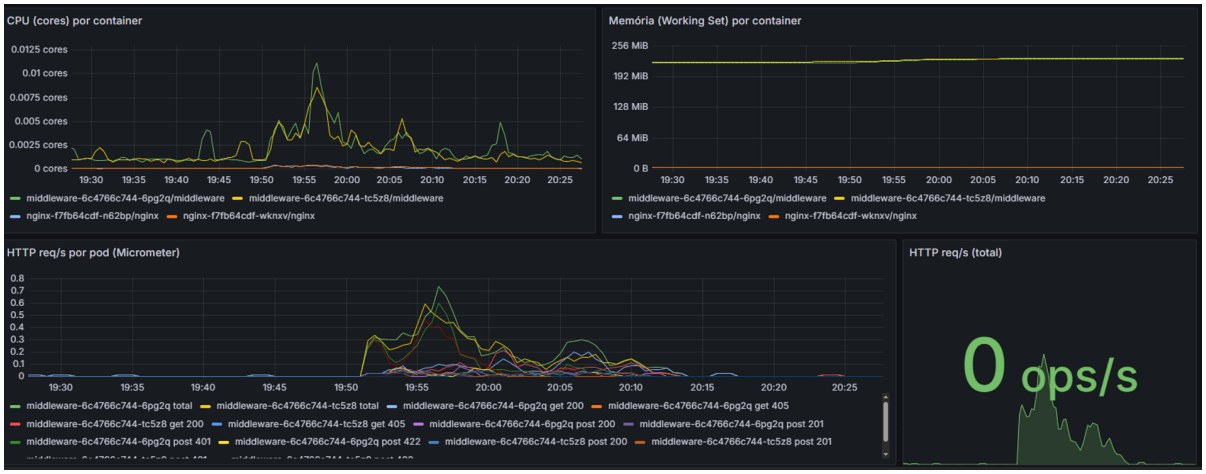


Figura 39 – AKS - Painel no Grafana durante o primeiro teste

Fonte: O Autor (2025)

C.2 TESTE 2

O segundo teste apresentou desempenho estável e baixo consumo de recursos, refletindo a menor carga de usuários em comparação ao primeiro experimento. A utilização média de CPU do *pool* de nós manteve-se em torno de 11%, com pequenas variações nos períodos de maior atividade, o que demonstra o bom balanceamento interno do *cluster* e a eficiência do Kubernetes na distribuição de tarefas entre os nós. A memória total permaneceu estável, com aproximadamente 35% de uso, reforçando a capacidade do ambiente em sustentar múltiplas operações simultâneas sem sinais de saturação. O tráfego de rede, por sua vez, apresentou crescimento gradual ao longo da execução, atingindo picos de 83 kB de entrada e 97 kB de saída, correspondentes aos momentos de maior interação dos alunos. Esse comportamento progressivo indica que o sistema responde bem a aumentos leves de demanda, mantendo estabilidade sem impacto perceptível no desempenho.

No nível dos contêineres, o consumo de CPU não ultrapassou 0,6%, enquanto a memória manteve-se estabilizada em cerca de 192 MB por *pod*, evidenciando consistência no uso dos recursos e ausência de variações abruptas de consumo. O volume total de requisições HTTP chegou a aproximadamente 20 por minuto, com latência P95 média de 369 ms, valores que indicam excelente responsividade, sobretudo considerando o ambiente distribuído e o número reduzido de usuários simultâneos. Essa estabilidade reforça que o *middleware* e o NGINX operaram dentro dos parâmetros esperados, sem quedas de desempenho ou atrasos significativos.

De modo geral, os resultados confirmam que a arquitetura em *containers* se manteve eficiente e previsível mesmo sob carga leve, demonstrando maturidade da infraestrutura e boa relação entre consumo de recursos e desempenho entregue. As métricas observadas estão ilustradas nas Figuras 41, 42 e 40, que apresentam a visualização detalhada do comportamento do sistema durante o experimento, evidenciando a estabilidade operacional e o comportamento controlado dos recursos ao longo de toda a execução.

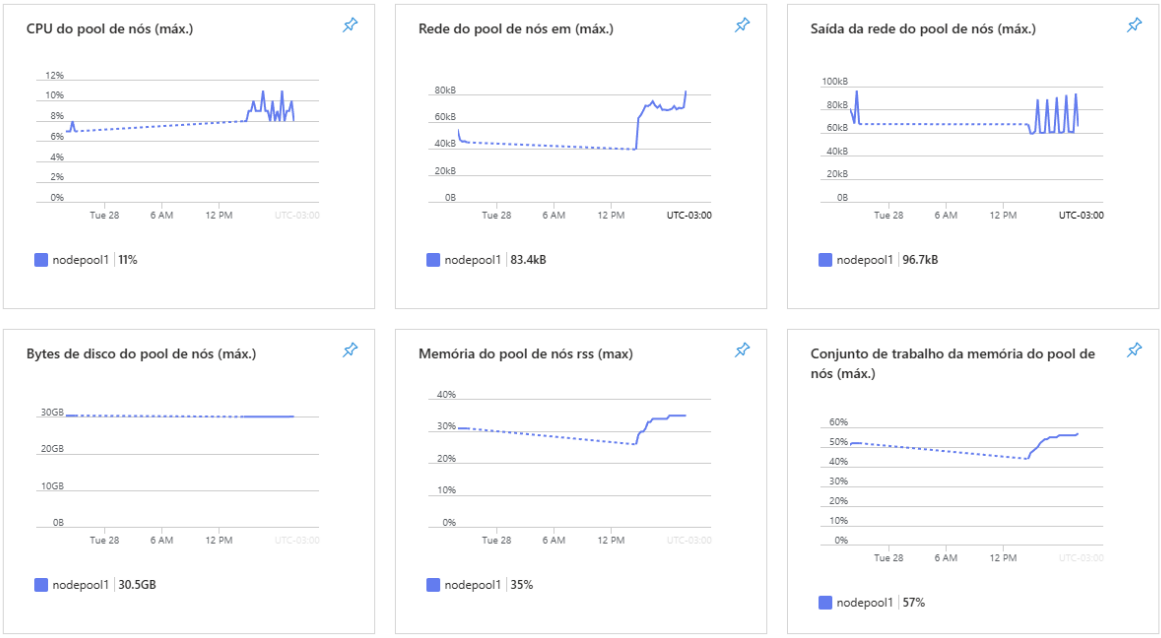


Figura 40 – AKS - Painel no Azure durante o segundo teste

Fonte: O Autor (2025)

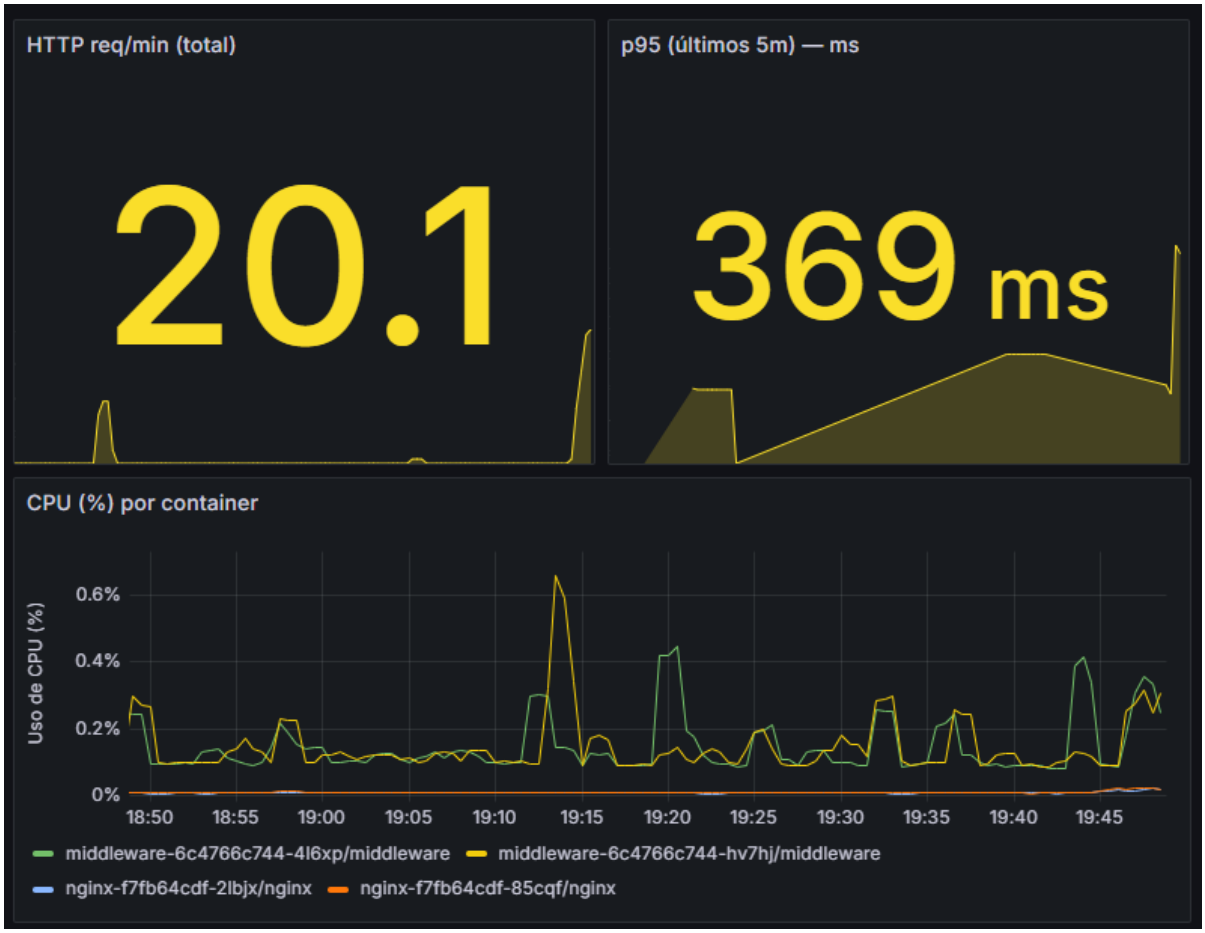


Figura 41 – AKS - Parte 1: Painel no Grafana durante o segundo teste.

Fonte: O Autor (2025)

Figura 42 – AKS - Parte 2: Painel no Grafana durante o segundo teste.

Fonte: O Autor (2025)

C.3 TESTE 3

No ambiente AKS, observou-se utilização eficiente dos recursos, com a CPU do *cluster* mantendo-se em torno de 8% e a memória total em aproximadamente 28%, sem qualquer indício de saturação. O tráfego de rede apresentou picos de 92 kB de entrada e 80 kB de saída, valores compatíveis com o volume de requisições simultâneas. Nos contêineres, o uso de CPU permaneceu estável em torno de 1%, enquanto a memória média foi de 190 MB por *pod*, confirmando a estabilidade e o gerenciamento eficiente dos recursos.

Por sua vez, no Azure App Service, verificou-se maior variabilidade de desempenho sob carga. A utilização média de CPU foi de 35,6%, enquanto a memória atingiu cerca de 83% de uso, evidenciando maior sensibilidade às requisições simultâneas. O tráfego de rede foi significativamente superior, com 2,31 MB de entrada e 13,15 MB de saída, reflexo de um processamento mais intenso de dados. A latência média foi de 190 ms, em contraste com o P95 de 106 ms observado no AKS.

De modo geral, ambos os ambientes mantiveram boa estabilidade e alta taxa de sucesso nas requisições. Entretanto, o AKS demonstrou desempenho mais eficiente, com menor latência, consumo reduzido e melhor balanceamento de recursos, enquanto o App Service apresentou resposta satisfatória, porém com maior variabilidade, especialmente no uso de CPU sob carga elevada.

As Figuras 43 a 46 ilustram as métricas de desempenho coletadas durante os experimentos, comparando o comportamento da aplicação nos ambientes AKS e App Service.

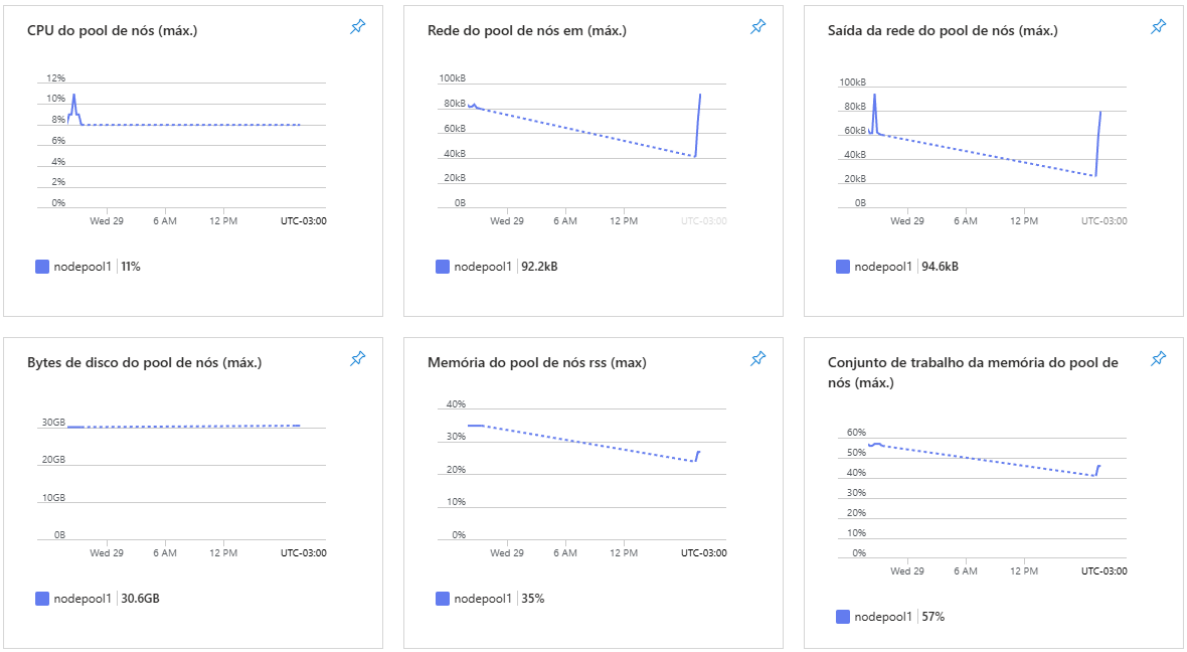


Figura 43 – AKS - Painel no Azure durante o terceiro teste

Fonte: O Autor (2025)

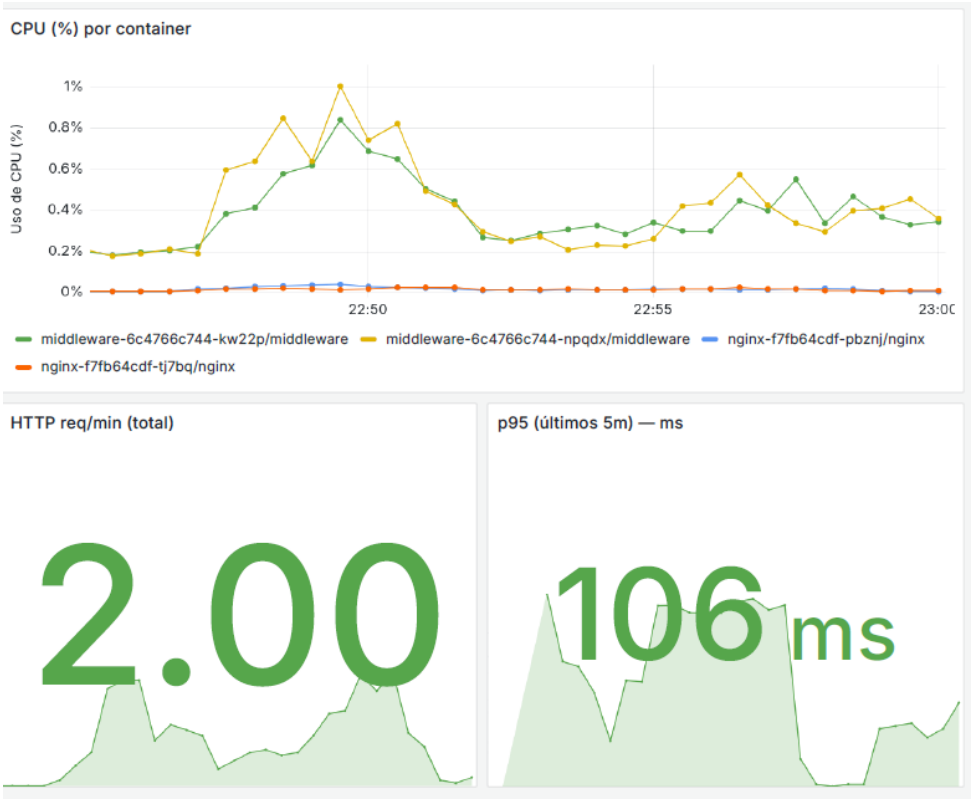


Figura 44 – AKS - Parte 1: Painel no Grafana durante o terceiro teste.

Fonte: O Autor (2025)

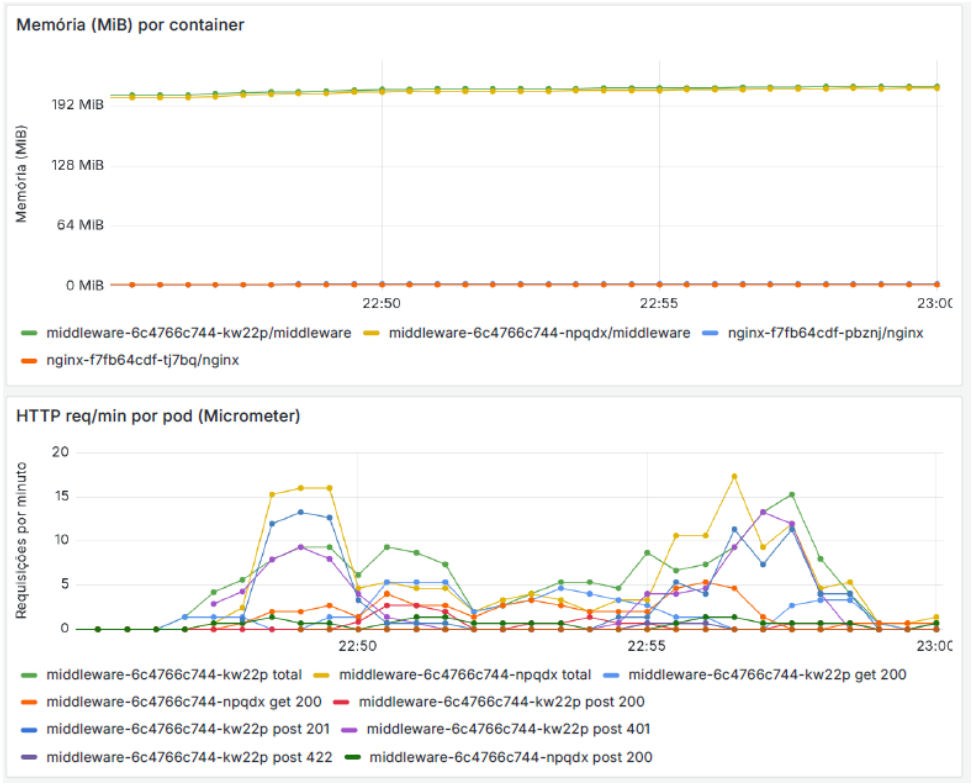


Figura 45 – AKS - Parte 2: Painel no Grafana durante o terceiro teste.

Fonte: O Autor (2025)

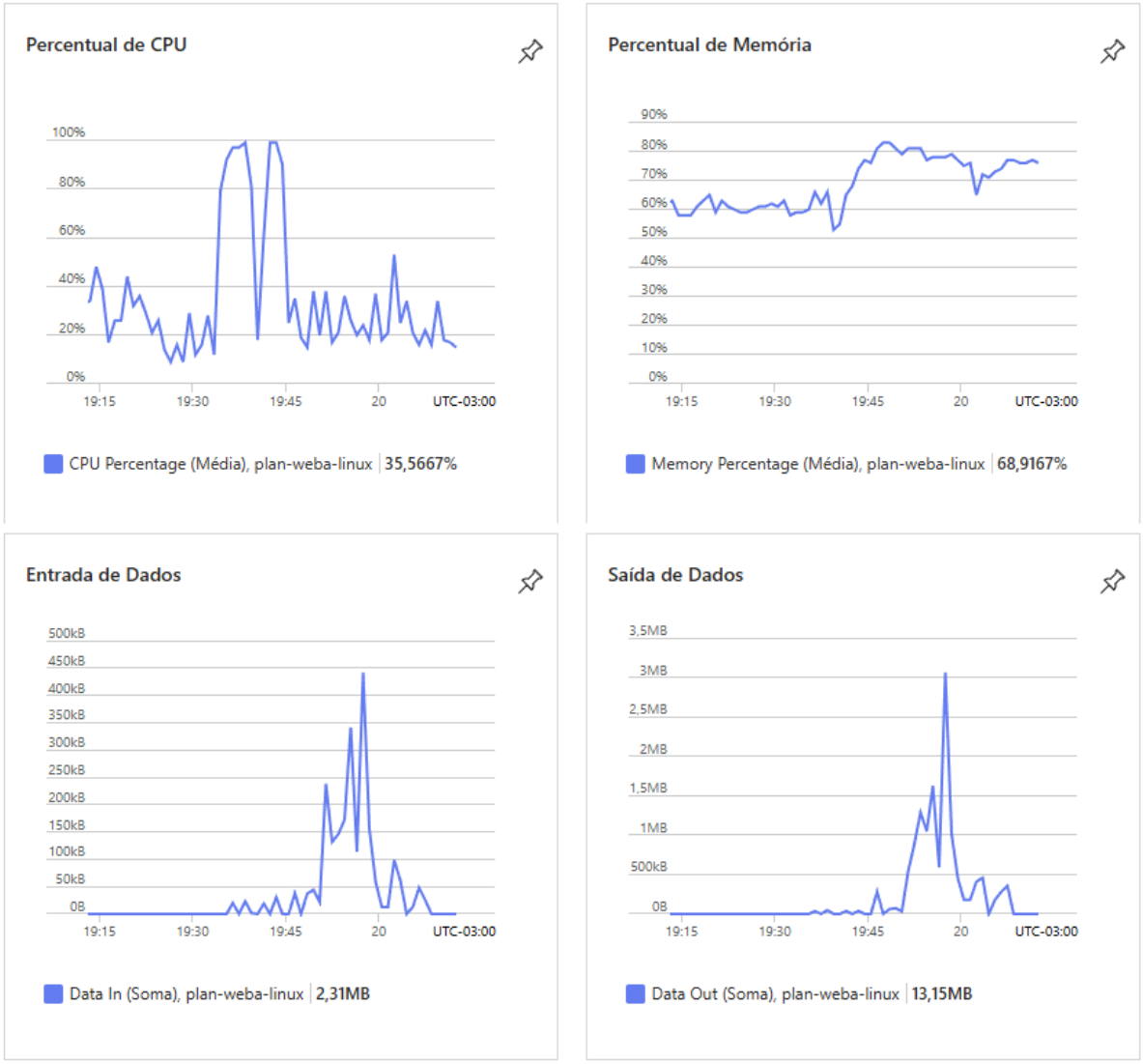


Figura 46 – App Service - Painel no Azure durante o terceiro teste

Fonte: O Autor (2025)

C.4 TESTE 4

No ambiente AKS, o uso de CPU do *pool* de nós foi reduzido, em torno de 10%, enquanto a memória total manteve-se próxima de 35%, sem indícios de saturação, conforme ilustrado na Figura 47. O tráfego de rede permaneceu estável, com aproximadamente 79 kB de entrada e 94 kB de saída, valores coerentes com o volume de requisições processadas. Nas métricas registradas pelo Grafana, a utilização de CPU por contêiner variou entre 0,2% e 0,8%, a memória manteve-se em torno de 190 MB e o P95 foi de 199 ms, indicando baixa latência e excelente desempenho sob carga controlada.

Por outro lado, no Azure App Service, observou-se maior utilização de recursos, com CPU média de 22,4% e uso de memória em torno de 71,4%, conforme apresentado na Figura 48. Apesar disso, o serviço manteve-se estável, sem sinais de gargalo. O tráfego de rede foi mais intenso, com aproximadamente 642 kB de entrada e 4,68 MB de saída, refletindo o processamento de um volume maior de dados. As métricas de aplicação registraram 34 requisições, com tempo médio de resposta de 119 ms e um pico isolado de 16 s.

Em síntese, o AKS demonstrou maior eficiência e previsibilidade, utilizando menos recursos para alcançar desempenho equivalente. O App Service, por sua vez, mostrou-se mais suscetível a oscilações, sobretudo no uso de CPU e memória, além de apresentar maior volume de tráfego de rede. Assim, o AKS se destaca pela estabilidade e otimização sob carga controlada, enquanto o App Service mantém desempenho satisfatório, porém com consumo mais elevado e leve variabilidade sob picos de demanda.

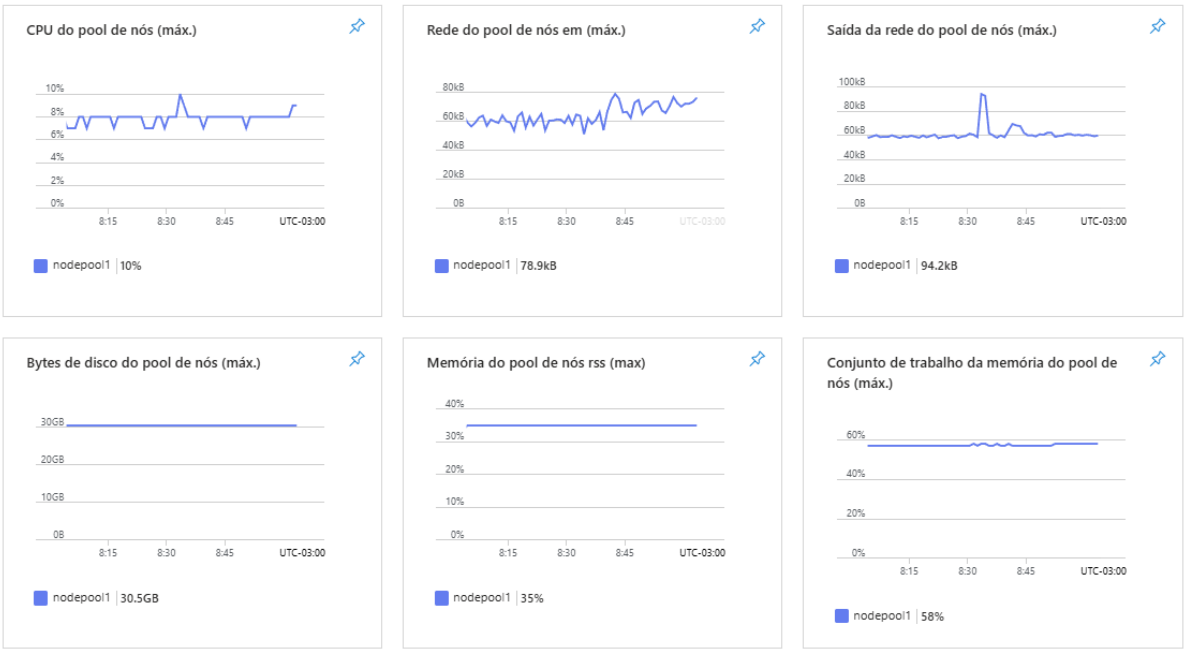


Figura 47 – AKS - Parte 1: Painel no Azure durante o quarto teste.

Fonte: O Autor (2025)

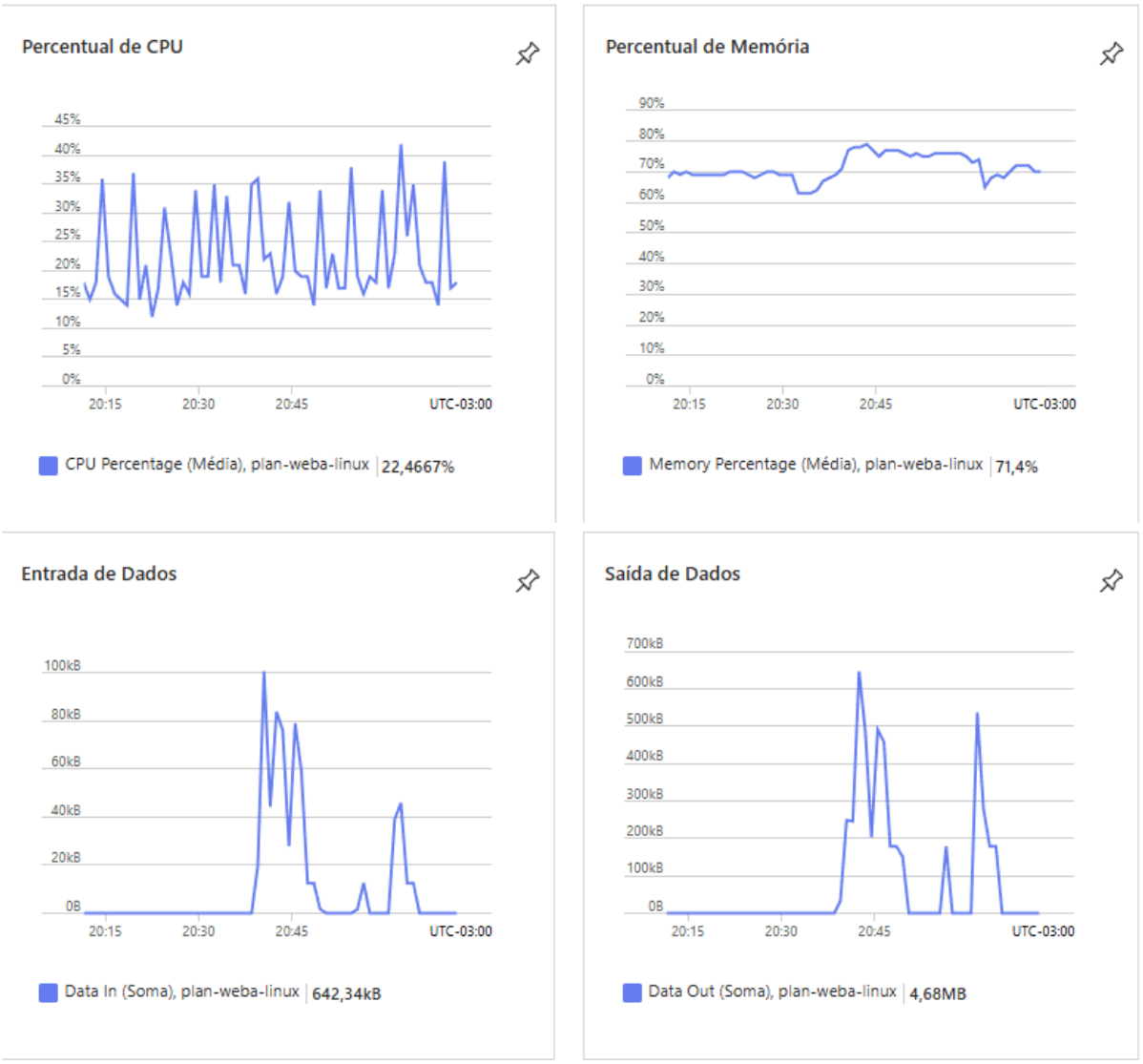


Figura 48 – App Service - Parte 2: Painel no Azure durante o quarto teste.

Fonte: O Autor (2025)

C.5 TESTE 5

O quinto teste, conduzido no Azure App Service com 22 alunos, demonstrou comportamento sólido ao longo de toda a execução, confirmando a estabilidade do ambiente mesmo sob atividade contínua. O volume total de requisições variou entre 74 e 87, das quais a maioria resultou em respostas HTTP bem-sucedidas, enquanto 37 apresentaram erros, característicos de falhas de cliente e não de instabilidade da aplicação. A latência média manteve-se baixa, em torno de 35 ms.

Em relação ao consumo de recursos, observou-se que, no início do teste, a CPU do App Service atingiu 100% de utilização, ocasionando brevemente a indisponibilidade da aplicação e exigindo sua reinicialização. Esse comportamento está ilustrado na Figura 51, que exibe o pico repentino de uso de CPU. Após o reinício, o ambiente estabilizou-se, mantendo média de

aproximadamente 23,5% de uso de CPU, com variações regulares durante os períodos de maior demanda, evidenciando boa capacidade de processamento e resposta consistente. A memória manteve-se próxima de 74%, sem indícios de vazamentos ou crescimento anormal, indicando gestão eficiente de alocação durante o uso prolongado.

Quanto às métricas de rede, observou-se entrada de dados de aproximadamente 3,6 MB e saída superior a 28 MB, valores compatíveis com o uso intensivo durante o período de teste. Esse volume expressivo de tráfego confirma o comportamento dinâmico da aplicação e o funcionamento adequado do serviço sob carga real.

De modo geral, o App Service manteve estabilidade, responsividade e consistência no processamento, demonstrando boa resiliência operacional durante o teste contínuo. As métricas completas e as visualizações gráficas estão apresentadas nas Figuras 49 e 50, que ilustram o comportamento dos recursos, das requisições HTTP e dos tempos de resposta ao longo da execução.

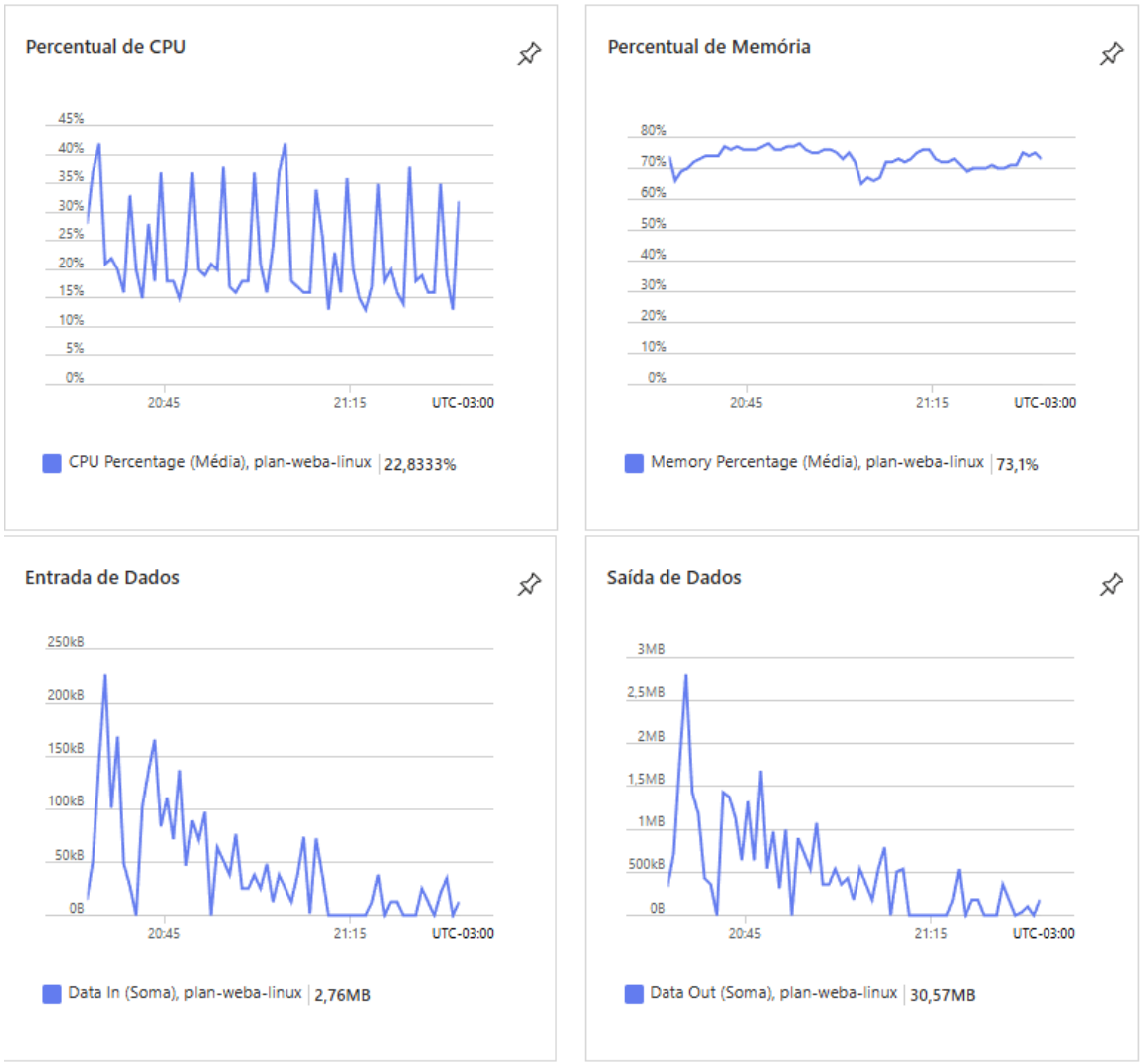


Figura 49 – App Service - Parte 1: Painel no Azure durante o quinto teste.

Fonte: O Autor (2025)

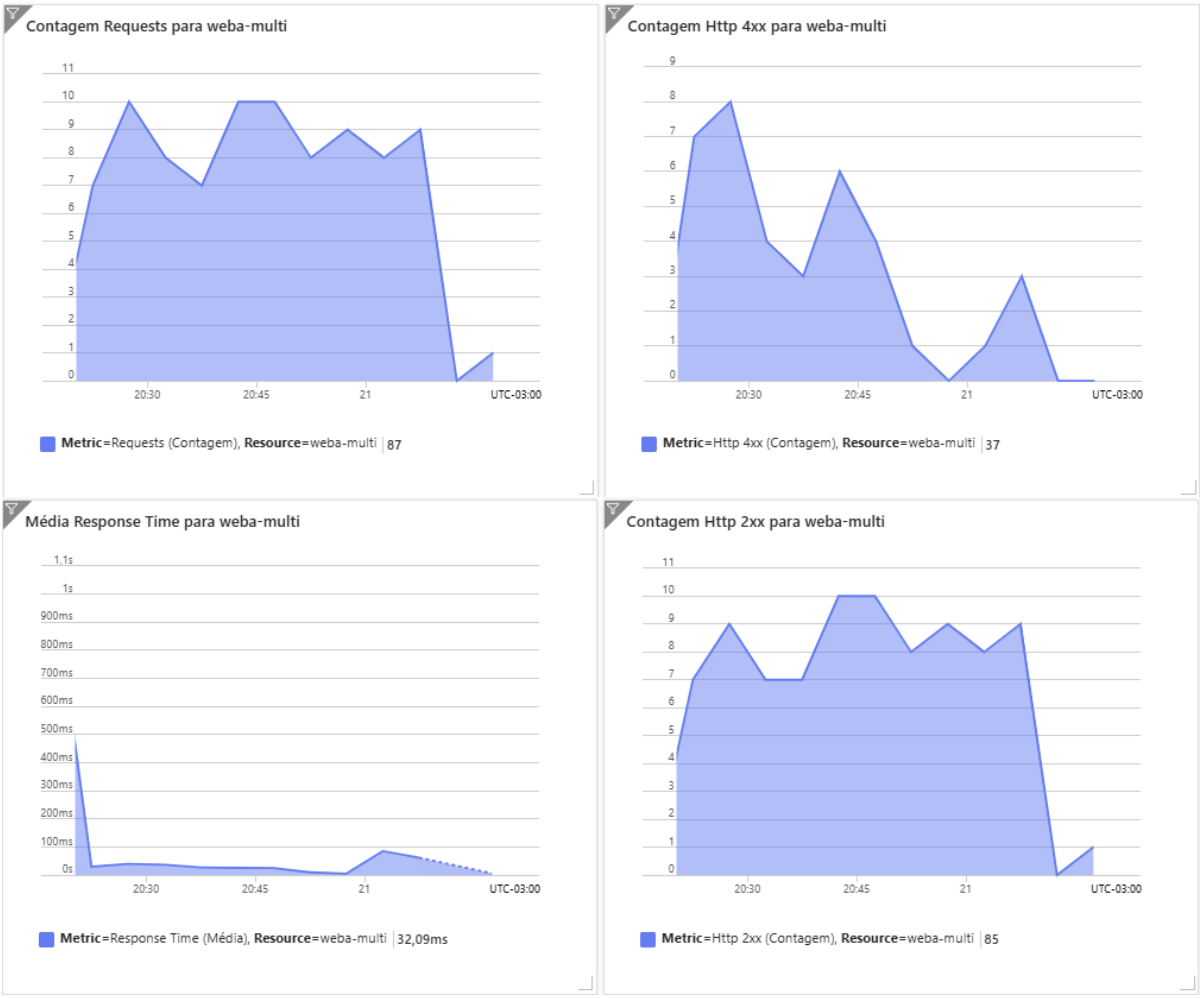


Figura 50 – App Service - Parte 2: Painel no Azure durante o quinto teste.

Fonte: O Autor (2025)

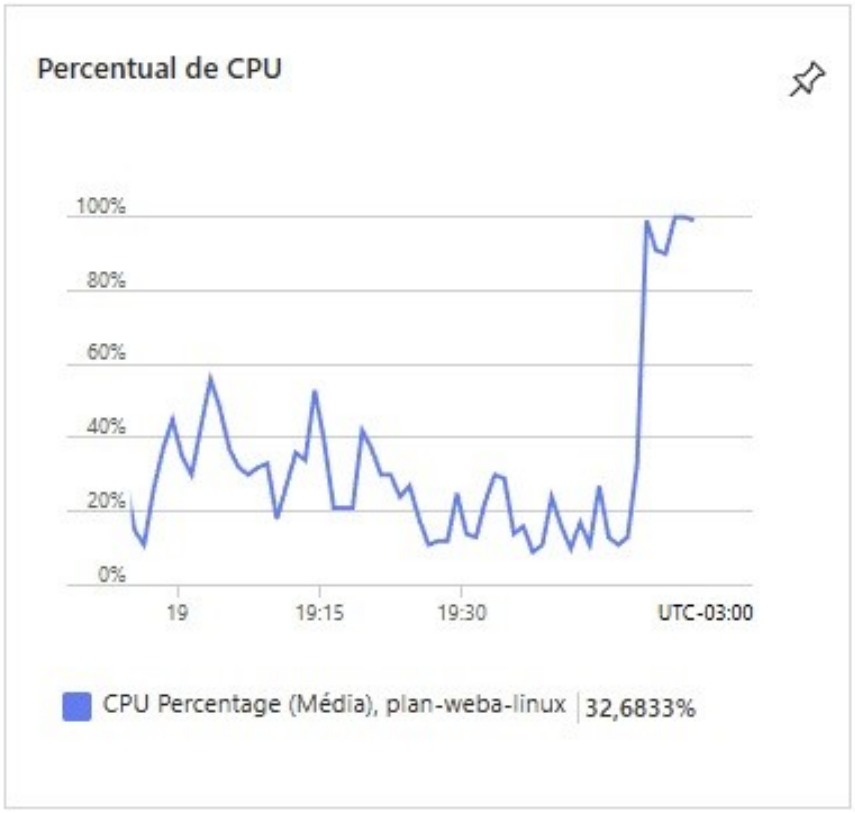


Figura 51 – Pico repentino de CPU do App Service.

Fonte: O Autor (2025)

APÊNDICE D – ARQUIVOS DE CONFIGURAÇÃO

Este apêndice organiza e descreve individualmente os principais arquivos de configuração utilizados no projeto. O Arquivo 1 reúne a definição dos serviços, redes e volumes da solução, permitindo a orquestração integrada de todos os componentes através do Docker Compose. A configuração do servidor NGINX, responsável pelo balanceamento de carga e roteamento das requisições, é apresentada no Arquivo 2. Na sequência, incluem-se os Dockerfiles utilizados no processo de construção das imagens: Arquivo 3 e o Arquivo 4. Por fim, são exibidos os três arquivos relacionados ao monitoramento da aplicação: a configuração do Loki no Arquivo 5, o Arquivo 6 de definição do Prometheus e o Arquivo 7 do Promtail, que juntos compõem o conjunto responsável pela coleta, armazenamento e consulta de métricas e *logs*. As versões completas e atualizadas de todos os arquivos estão disponíveis no repositório do projeto, acessível em: <https://github.com/GabrielMoscone/web-algo>

Arquivo 1 – Arquivo docker-compose.yml com a definição do serviço NGINX, contendo dependências dos middlewares, mapeamento das portas, volumes e controle dos recursos.

```
services:
  # ===== APLICACAO =====
  nginx:
    image: nginx:alpine
    container_name: nginx
    depends_on:
      - middleware1
      - middleware2
    ports: ["8088:80"]
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
      - ./web-algo/index.html:/usr/share/nginx/html/index.html:ro
      - ./web-algo/login.html:/usr/share/nginx/html/login.html:ro
      - ./web-algo/cadastro.html:/usr/share/nginx/html/cadastro.html:ro
      - ./web-algo/recuperar-senha.html:/usr/share/nginx/html/recuperar-sen
ha.html:ro
      - ./web-algo/includes:/usr/share/nginx/html/includes:ro
      - ./web-algo/media:/usr/share/nginx/html/media:ro
    restart: unless-stopped
    deploy:
      resources:
        limits:
          cpus: '2.0'
          memory: 1G
        reservations:
          cpus: '0.5'
```

```
27            memory: 256M
28      ulimits:
29        nofile:
30          soft: 65535
31          hard: 65535
32      networks:
33        - default
34
35    middleware1:
36      build:
37        context: ./middleware
38        dockerfile: Dockerfile
39      container_name: middleware1
40      deploy:
41        resources:
42          limits:
43            cpus: '2.0'
44            memory: 2G
45          reservations:
46            cpus: '1.0'
47            memory: 1G
48      environment:
49        SPRING_PROFILES_ACTIVE: prod
50        JAVA_OPTS: "-Xms512m -Xmx1536m -XX:+UseG1GC -XX:MaxGCPauseMillis=200
51 -XX:ParallelGCThreads=2 -XX:ConcGCThreads=1"
52        WEB_ALGO_URL: ${WEB_ALGO_URL}
53      expose:
54        - "8080"
55        - "8081"
56      ulimits:
57        nofile:
58          soft: 65535
59          hard: 65535
60      restart: unless-stopped
61      networks:
62        - default
63
64    middleware2:
65      build:
66        context: ./middleware
67        dockerfile: Dockerfile
68      container_name: middleware2
69      deploy:
70        resources:
71          limits:
72            cpus: '2.0'
73            memory: 2G
```

```
74          reservations:
75            cpus: '1.0'
76            memory: 1G
77      environment:
78        SPRING_PROFILES_ACTIVE: prod
79        JAVA_OPTS: "-Xms512m -Xmx1536m -XX:+UseG1GC -XX:MaxGCPauseMillis=200
80  -XX:ParallelGCThreads=2 -XX:ConcGCThreads=1"
81        WEB_ALGO_URL: ${WEB_ALGO_URL}
82      expose:
83        - "8080"
84        - "8081"
85      ulimits:
86        nofile:
87          soft: 65535
88          hard: 65535
89      restart: unless-stopped
90      networks:
91        - default
92
93    # ===== MONITORAMENTO =====
94    prometheus:
95      image: prom/prometheus:latest
96      container_name: prometheus
97      volumes:
98        - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml:ro
99      command:
100       - "--config.file=/etc/prometheus/prometheus.yml"
101       - "--storage.tsdb.retention.time=15d"
102       - "--enable-feature=remote-write-receiver"
103       - "--web.enable-remote-write-receiver"
104       - "--enable-feature=native-histograms"
105     ports: ["9090:9090"]
106     depends_on: [nginx]
107     networks:
108       - default
109
110   grafana:
111     image: grafana/grafana:latest
112     container_name: grafana
113     ports: ["3000:3000"]
114     environment:
115       - GF_SECURITY_ADMIN_PASSWORD=admin
116       - GF_INSTALL_PLUGINS=grafana-piechart-panel
117     depends_on: [prometheus, loki]
118     networks:
119       - default
120
```

```
121   cadvisor:
122     image: gcr.io/cadvisor/cadvisor:v0.49.1
123     container_name: cadvisor
124     privileged: true
125     ports:
126       - "8082:8080"
127     volumes:
128       - /:/rootfs:ro
129       - /var/run:/var/run:ro
130       - /sys:/sys:ro
131       - /var/lib/docker/:/var/lib/docker:ro
132       - /dev/disk/:/dev/disk:ro
133     devices:
134       - /dev/kmsg
135     command:
136       - '--docker_only=true'
137       - '--housekeeping_interval=10s'
138       - '--max_housekeeping_interval=15s'
139       - '--event_storage_event_limit=default=0'
140       - '--event_storage_age_limit=default=0'
141       - '--store_container_labels=true'
142       - '--whitelisted_container_labels=com.docker.compose.project,com.dock
143 er.compose.service,com.docker.compose.container-number'
144     restart: unless-stopped
145     networks:
146       - default
147
148   blackbox:
149     image: prom/blackbox-exporter:latest
150     container_name: blackbox
151     ports: ["9115:9115"]
152     networks:
153       - default
154
155   nginx-exporter:
156     image: ghcr.io/nginxinc/nginx-prometheus-exporter:0.11.0
157     container_name: nginx-exporter
158     command: ["-nginx.scrape-uri=http://nginx/nginx_status"]
159     depends_on: [nginx]
160     ports: ["9113:9113"]
161     networks:
162       - default
163
164   # ===== LOGS =====
165   loki:
166     image: grafana/loki:2.9.6
167     command: -config.file=/etc/loki/config/loki-config.yaml
```

```
168       ports: ["3100:3100"]
169       volumes:
170         - ./monitoring/loki-config.yaml:/etc/loki/config/loki-config.yaml:ro
171         - loki-data:/loki
172       networks:
173         - default
174
175     promtail:
176       image: grafana/promtail:2.9.6
177       command: -config.file=/etc/promtail/promtail-config.yml
178       volumes:
179         - ./monitoring/promtail-config.yml:/etc/promtail/promtail-config.yml
180 :ro
181         - /var/run/docker.sock:/var/run/docker.sock:ro
182         - /var/lib/docker/containers:/var/lib/docker/containers:ro
183       depends_on: [loki]
184       networks:
185         - default
186
187     k6:
188       image: grafana/k6
189       container_name: k6
190       volumes:
191         - ./k6/testes:/scripts:ro
192       environment:
193         - K6_PROMETHEUS_RW_SERVER_URL=http://prometheus:9090/api/v1/write
194       entrypoint: ["sleep", "infinity"]
195       depends_on: [prometheus]
196       networks:
197         - default
198
199 volumes:
200     loki-data: {}
201
202 networks:
203     default:
204       driver: bridge
```

Fonte: O Autor (2021)

Arquivo 2 – Configurações do servidor NGINX no arquivo nginx.conf, incluindo número de processos, conexões simultâneas, política de logs e diretivas de performance.

```
1 user nginx;
2
3 worker_processes auto;
4 worker_rlimit_nofile 65535;
5
```

```
6  error_log /var/log/nginx/error.log warn;
7  pid /var/run/nginx.pid;
8
9  events {
10   worker_connections 4096;
11   use epoll;
12
13   multi_accept on;
14 }
15
16 http {
17   include /etc/nginx/mime.types;
18   default_type application/octet-stream;
19
20   # ===== OTIMIZACOES DE PERFORMANCE =====
21   sendfile on;
22   tcp_nopush on;
23   tcp_nodelay on;
24
25   # Timeouts otimizados
26   keepalive_timeout 65;
27   keepalive_requests 100;
28   client_body_timeout 12;
29   client_header_timeout 12;
30   send_timeout 10;
31
32   # Buffers otimizados
33   client_body_buffer_size 128k;
34   client_max_body_size 10m;
35   client_header_buffer_size 1k;
36   large_client_header_buffers 4 8k;
37
38   # Compressao
39   gzip on;
40   gzip_vary on;
41   gzip_proxied any;
42   gzip_comp_level 6;
43   gzip_types text/plain text/css text/xml text/javascript application/json
44 application/javascript application/xml+rss;
45
46   # Detecta sessao pelo cookie "sessionid"
47   map $http_cookie $has_session {
48     default 0;
49     "~*sessionid=" 1;
50   }
51
52   # ===== UPSTREAM COM ROUND-ROBIN =====
```

```
53  upstream backend_cluster {
54    # Health check e failover
55    server middleware1:8080 max_fails=3 fail_timeout=30s weight=1;
56    server middleware2:8080 max_fails=3 fail_timeout=30s weight=1;
57
58    # Keepalive connections
59    keepalive 128;
60    keepalive_timeout 60s;
61    keepalive_requests 1000;
62  }
63
64  server {
65    listen 80;
66    server_name _;
67    root /usr/share/nginx/html;
68    index login.html index.html;
69
70    absolute_redirect off;
71
72    location = / {
73      if ($has_session) { return 302 /index.html; }
74      return 302 /login.html;
75    }
76
77    location = /index.html {
78      if ($has_session = 0) { return 302 /login.html; }
79      add_header Cache-Control "no-store";
80      try_files $uri =404;
81    }
82    location / {
83      add_header Cache-Control "no-store";
84      try_files $uri $uri/ =404;
85    }
86
87    # ===== PROXY PARA API =====
88    location /api/ {
89      proxy_pass http://backend_cluster;
90      proxy_http_version 1.1;
91
92      proxy_set_header Connection "";
93      proxy_set_header Host $host;
94      proxy_set_header X-Real-IP $remote_addr;
95      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
96      proxy_set_header X-Forwarded-Proto $scheme;
97
98      # Timeouts
99      proxy_connect_timeout 10s;
```

```
100         proxy_send_timeout 60s;
101         proxy_read_timeout 60s;
102 
103         # Buffers
104         proxy_buffering on;
105         proxy_buffer_size 8k;
106         proxy_buffers 32 8k;
107         proxy_busy_buffers_size 16k;
108 
109         # Retry em caso de erro
110         proxy_next_upstream error timeout invalid_header http_500 http_502
111 http_503 http_504;
112         proxy_next_upstream_tries 2;
113         proxy_next_upstream_timeout 5s;
114 
115         proxy_request_buffering off;
116     }
117 
118     # METRICAS DO NGINX
119     location /nginx_status {
120       stub_status;
121       access_log off;
122       allow 127.0.0.1;
123       allow 172.16.0.0/12;  # rede docker default
124       deny all;
125     }
126   }
127 
128   log_format simple '$remote_addr - $remote_user [$time_local] "$request"
129 $status $body_bytes_sent "$http_referer" "$http_user_agent" $request_time';
130   access_log /dev/stdout simple;
131   error_log  /dev/stderr warn;
132 }
```

Fonte: O Autor (2021)

Arquivo 3 – Estrutura do arquivo Dockerfile do middleware, apresentando as etapas de build com Maven, geração do artefato .jar e configuração do contêiner para execução da aplicação Spring Boot.

```
1 # Multi-stage build para reduzir tamanho da imagem
2 # Stage 1: Build
3 FROM maven:3.9-eclipse-temurin-21 AS build
4 WORKDIR /app
5 COPY pom.xml .
6 RUN --mount=type=cache,target=/root/.m2 mvn -q -DskipTests dependency:go-of
7 fline
8 
```

```
COPY src ./src
RUN --mount=type=cache,target=/root/.m2 mvn -q -DskipTests package

# Stage 2: Runtime
FROM eclipse-temurin:21-jre-alpine
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
RUN addgroup -S spring && adduser -S spring -G spring
USER spring:spring

# Healthcheck
HEALTHCHECK --interval=30s --timeout=3s --start-period=60s CMD wget --no-ve
rbose --tries=1 --spider http://localhost:8081/actuator/health || exit 1

# Expor portas
EXPOSE 8080 8081

ENTRYPOINT ["java", \
     "-XX:+UseContainerSupport", \
     "-XX:MaxRAMPercentage=75.0", \
     "-Djava.security.egd=file:/dev/./urandom", \
     "-jar", "app.jar"]
```

Fonte: O Autor (2021)

Arquivo 4 – Estrutura do arquivo Dockerfile do front-end baseada na imagem nginx:alpine, contendo a definição dos arquivos HTML, diretórios de recursos e exposição da porta 80.

```
FROM nginx:alpine

# Copiar arquivos HTML (contexto sera a raiz do projeto)
COPY index.html /usr/share/nginx/html/
COPY login.html /usr/share/nginx/html/
COPY cadastro.html /usr/share/nginx/html/
COPY recuperar-senha.html /usr/share/nginx/html/
COPY includes/ /usr/share/nginx/html/includes/
COPY assets/ /usr/share/nginx/html/assets/

# Expor porta
EXPOSE 80
```

Fonte: O Autor (2021)

Arquivo 5 – Arquivo loki-config.yaml, contendo parâmetros de rede, armazenamento local, esquema de indexação e controle de replicação.

```
auth_enabled: false
server:
```

```
3     http_listen_port: 3100
4
5   common:
6     path_prefix: /loki
7     storage:
8       filesystem:
9         chunks_directory: /loki/chunks
10        rules_directory: /loki/rules
11    replication_factor: 1
12    ring:
13      instance_addr: 127.0.0.1
14      kvstore:
15        store: inmemory
16
17  schema_config:
18    configs:
19      - from: 2020-10-24
20        store: boltdb-shipper
21        object_store: filesystem
22        schema: v13
23        index:
24          prefix: index_
25          period: 24h
26
27  ruler:
28    storage:
29      type: local
30      local:
31        directory: /loki/rules
32
33  table_manager:
34    retention_deletes_enabled: true
35    retention_period: 7d
36
37  chunk_store_config:
38    max_look_back_period: 0s
```

Fonte: O Autor (2021)

Arquivo 6 – Arquivo prometheus.yml, incluindo parâmetros globais de coleta e os serviços monitorados: middleware, NGINX Exporter, cAdvisor e Blackbox.

```
1   global:
2     scrape_interval: 15s
3     evaluation_interval: 15s
4
5   scrape_configs:
6     - job_name: 'prometheus'
```

```
7       static_configs: [{ targets: ['prometheus:9090'] }]
8
9     - job_name: "middleware"
10      metrics_path: /actuator/prometheus
11      static_configs:
12        - targets:
13            - "middleware1:8081"
14            - "middleware2:8081"
15
16    - job_name: 'nginx-exporter'
17      static_configs:
18        - targets: ['nginx-exporter:9113']
19
20    - job_name: 'cadvisor'
21      static_configs:
22        - targets: ['cadvisor:8080']
23
24    - job_name: 'blackbox'
25      metrics_path: /probe
26      params:
27        module: [http_2xx]
28      static_configs:
29        - targets:
30            - http://nginx:80
31            - http://nginx:80/login.html
32            - http://nginx:80/api/v1/web-algo/problems/key/S
33      relabel_configs:
34        - source_labels: [__address__]
35          target_label: __param_target
36        - source_labels: [__param_target]
37          target_label: instance
38        - target_label: __address__
39          replacement: blackbox:9115
```

Fonte: O Autor (2021)

Arquivo 7 – Arquivo promtail-config.yml, incluindo os parâmetros de comunicação com o Loki e a descoberta automática de contêineres Docker.

```
1  server:
2    http_listen_port: 9080
3    grpc_listen_port: 0
4
5  clients:
6    - url: http://loki:3100/loki/api/v1/push
7
8  positions:
9    filename: /tmp/positions.yaml
```

```
10
11 scrape_configs:
12   - job_name: docker
13     docker_sd_configs:
14       - host: unix:///var/run/docker.sock
15         refresh_interval: 5s
16     relabel_configs:
17       - source_labels: ['__meta_docker_container_name']
18         target_label: 'container_name'
19       - source_labels: ['__meta_docker_container_log_stream']
20         target_label: 'stream'
```

Fonte: O Autor (2021)