Universidade Federal de Pernambuco

Centro de Informática

Graduação em Engenharia da Computação

Análise Comparativa do Uso de Controladores para a Escalabilidade Horizontal no Kubernetes

Zilde Souto Maior Neto

Trabalho de Graduação

Recife, Pernambuco

2025

Universidade Federal de Pernambuco
Centro de Informática

Zilde Souto Maior Neto

Análise Comparativa do Uso de Controladores para a Escalabilidade Horizontal no Kubernetes

Trabalho apresentado ao Curso de Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Área de concentração: Sistemas Distribuídos
Orientador: Nelson Souto Rosa

Recife, Pernambuco
2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Souto Maior Neto, Zilde.
Análise Comparativa do Uso de Controladores para a Escalabilidade Horizontal no Kubernetes / Zilde Souto Maior Neto. - Recife, 2025.
68 p. : il., tab.

Orientador(a): Nelson Souto Rosa
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2025.
Inclui referências, apêndices.

1. Escalabilidade Horizontal. 2. Teoria de Controle. 3. Kubernetes. 4. Microsserviços. I. Rosa, Nelson Souto. (Orientação). II. Título.

000 CDD (22.ed.)

ZILDE SOUTO MAIOR NETO

Análise Comparativa do Uso de Controladores para a Escalabilidade Horizontal no Kubernetes

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Engenharia da Computação.

Aprovado em: 28/03/2025

BANCA EXAMINADORA

_____

Prof. Dr. Nelson Souto Rosa (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. Jamilson Ramalho Dantas (Examinador Interno)

Universidade Federal de Pernambuco

Dedico este trabalho à Deus e minha família.

Agradecimentos

À minha mãe, Lusinete Figuerêdo, muito obrigado por toda dedicação, sacrifício e orações ao longo desses anos, desde o colégio até a faculdade. Nunca vou esquecer suas noites em claro, sempre com muito esforço e renúncia, para garantir que a única preocupação dos seus filhos fossem os estudos. Obrigado por ser o nosso porto seguro, sempre que perdíamos nossa direção. Obrigado por todos os conselhos, lições, discussões e adversidades; tudo isso me moldou para ser quem sou hoje. Obrigado por todo o seu amor incondicional, que sempre me deu energia para continuar lutando, pois sempre me fez sentir e saber que alguém estava lá por mim, independentemente do que acontecesse. Obrigado pelas incontáveis viagens, acompanhamentos, sagas e histórias. Obrigado por sempre estar comigo em Recife, principalmente nos dias em que eu mais me sentia só. Obrigado por compreender todos os meus momentos e, principalmente, por me apoiar nos mais difíceis. Obrigado por ser esse exemplo de mulher, mãe e filha. Obrigado por ser você, mãe. Eu amo a senhora do fundo do meu coração.

Para meu pai, Zilde Filho, muito obrigado por tudo. Apesar da distância, o senhor sempre fez questão de estar muito presente. Obrigado por todo o seu esforço para garantir que eu sempre tivesse tudo que precisava. Obrigado por todas as mensagens de incentivo, orações, ligações frequentes e por todas aquelas idas e vindas a Limoeiro, que eram a certeza das sextas-feiras e um dos dias que eu mais me divertia na semana. O senhor sempre se esforçou para que tivéssemos tudo aquilo que o senhor não teve, e espero ter honrado isso com todo o esforço até aqui. Obrigado por tudo, pai. Te amo!

À minha namorada e companheira, Ana Letícia, obrigado por tudo. Obrigado por ser a pessoa incrível que você é, um exemplo de filha, neta, irmã, companheira e, principalmente, amiga. Obrigado por compartilhar comigo todos os bons e maus momentos e, principalmente, por nos apoiarmos incondicionalmente em todos eles. Obrigado por ter sido minha família em Recife e por ter feito da sua família a minha também. Obrigado por ser não só a minha inspiração, mas também para muitas outras pessoas. Obrigado por me amar verdadeiramente e por me compreender. Obrigado por sempre me encher de orgulho e por me arrancar sorrisos, mesmo em dias difíceis. Obrigado por ser você, por sermos nós. Este trabalho também é seu. Te amo!

À minha irmã, Bruna Laís, obrigado por todas as conversas, conselhos e, principalmente, pela parceria de irmã. Compartilhamos boa parte dessa trajetória juntos e, com certeza, carrego todas aquelas idas e vindas para o colégio no meu coração. Você sempre me ajudou quando precisei, e espero que você também sempre possa contar comigo em qualquer momento. Ao meu cunhado, Mauro Florentino, obrigado por todos os conselhos, conversas, momentos e pelo apoio incondicional sempre que precisei. Você tem minha eterna gratidão por tornar todos esses momentos mais leves. E claro, não posso esquecer de agradecer ao meu cachorro Billy, meu parceiro de todas as horas. Obrigado por sempre estar ali, deitado do meu lado enquanto eu virava noites estudando e por ser especialista em arrancar sorrisos mesmo nos dias mais puxados.

Aos meus avós maternos, Maria Figuerêdo e Manoel Gomes de Moura, muito obrigado por

todo o acolhimento, incentivo e apoio. Vocês foram parte fundamental em todos os momentos da minha vida. Vovó, muito obrigado por compartilhar suas histórias, pelas "chamadas no cantinho"e por sempre me acolher como um filho. Vovô, obrigado por preencher, muitas vezes, o papel de pai. Dizem que os avós são pais duas vezes, e o senhor comprovou isso. Obrigado por todos os domingos em que chegava cedo na porta da minha casa para me levar para passear; era o dia que eu mais aguardava ansioso na semana e que sinto muitas saudades até hoje. Eu amo vocês incondicionalmente.

Dedico também este trabalho, *in memoriam*, aos meus queridos avós Zilde Souto Maior e Maria Ângela da Mota. Suas histórias, ensinamentos e amor continuam vivos em mim e foram fundamentais para que eu chegasse até aqui. Terei orgulho se conseguir chegar perto do que vocês representam para mim. Esta conquista também é de vocês.

Também agradeço a todos os meus colegas de curso que acompanharam toda essa jornada de desafios e emoções desde 2018. Em especial, gostaria de agradecer aos meus amigos Daniel Perazzo, Fernando Macedo, José Carlos, Lucas Ambrósio, Luís Eduardo, Matheus Teotônio, Pedro Nogueira, Riei Joaquim, Victor Hugo, Victor Miguel, Victor Ximenes, Washington Santos e Zênio Ângelo. Obrigado por terem compartilhado de perto toda essa trajetória, desde as famosas viradas nos laboratórios de graduação, as aventuras virtuais na época da pandemia e pelo apoio incondicional até os momentos finais da graduação. Carrego todos vocês no meu coração e espero que possamos continuar sempre juntos.

À Pedro Nogueira, agradeço por todo o companheirismo e por termos compartilhado tantas coisas juntos, desde os perrengues da faculdade até os desafios da Neurotech. Foram muitas histórias e guardarei todos os momentos no meu coração.

À Victor Hugo, meu nobre, muito obrigado por ter me acolhido e estar comigo em tantos momentos. Nunca vou me esquecer de todas as nossas aventuras, momentos de estudo, de brincadeiras e conversas mais sérias. Meu nobre, a gente aprendeu foi junto!

À Victor Miguel, só tenho a agradecer por toda a trajetória que tivemos. Você foi um dos meus maiores apoiadores na universidade e um amigo para toda a hora. Obrigado por ter compartilhado e me deixar compartilhar tanto, vou carregar todos os momentos para sempre no meu coração. Você me confortou em muitos momentos difíceis e serei sempre eternamente grato.

Agradeço também aos que vieram antes de mim, em especial José Victor, Lucas Santana, Marcos Oliveira, Gabriel Mendes e Mateus Valgueiro, muito obrigado por terem servido como inspiração e também pelo acolhimento durante toda essa trajetória. Nunca esquecerei das conversas, conselhos e ensinamentos que vocês me proporcionaram.

Agradeço também aos companheiros das *gameplays* de todas as noites, em especial ao grande primo Mateus Elias, por estarem sempre presentes depois de um longo dia, garantindo boas risadas e aquela dose de diversão necessária. Vocês tornaram muito mais leves os dias em que trabalho e faculdade colidiam — mesmo que, na maioria das vezes, eu tenha tido que carregar o time nas costas (o que não foi nada fácil, diga-se de passagem). Brincadeiras à parte, sou grato por cada partida, cada zoeira, cada briga e por toda a parceria ao longo dessa caminhada. Obrigado demais, pessoal!

Agradecimentos mais que especiais também aos amigos que fiz durante minha jornada no Maracatronics. O projeto Armorial e os Autônomos Coletivos sempre estarão marcados na minha memória, desde as competições virtuais que arrancaram tanto lágrimas de tristeza quanto de felicidade, até as viagens para outros estados, que permitiram estreitar laços com pessoas de outras universidades. A Pedro Jorge e Júlia Dias, em especial, por terem sido meus pais na robótica e por sempre estarem me dando apoio nessa trajetória, agradeço imensamente a vocês dois por me fazerem sentir parte de algo tão especial e único. Agradeço também a Adam

pela confiança no meu trabalho com o VSSS e a Lang, tanto por ter cedido espaço para que eu pudesse ajudar no SSL, quanto por estreitar o contato com a equipe Warthog, que inspirou e tracionou o projeto de robótica no Maracatronics.

Aos amigos do Voxar, em especial a João Marcelo, VT e Maria, que acreditaram em todo o meu potencial e sempre se fizeram presentes para dar apoio e suporte. Agradecimentos também a Pinho, Rafaela, Heitor, Gustavo e Will, pelas conversas e momentos no laboratório.

À Neurotech, por ter fornecido todo o suporte possível para a realização deste trabalho. Em especial a Bruno Silva, pelas conversas, desde pessoais até técnicas, e por ser minha referência na empresa; a Petrônio Braga, pelas excelentes discussões e acompanhamento durante o desenvolvimento deste trabalho; a Danilo Simei, pelo apoio incondicional, conversas, conselhos, inspiração e pelo esforço para tornar este trabalho possível; e a Douglas Vasconcelos, por todos os momentos e desafios que compartilhamos juntos. Obrigado por ser minha inspiração, Douguinho!

Por fim, mas não menos importante, agradecimentos especiais ao professor Nelson Souto Rosa, por toda paciência e empenho em me orientar na produção deste trabalho. Professor, com toda certeza este trabalho não seria possível sem o senhor. Obrigado por tudo!

“Se eu vi mais longe, foi por estar sobre ombros de gigantes"

— Isaac Newton

Resumo

A escalabilidade horizontal é essencial para aplicações baseadas em microsserviços (ABMs) implantadas em contêineres. No Kubernetes, a solução mais utilizada para esse propósito é o *Horizontal Pod Autoscaler* (HPA), que ajusta dinamicamente o número de réplicas de acordo com metas predefinidas de utilização de recursos computacionais, como CPU e memória. Contudo, por se tratar de uma solução genérica, o HPA nem sempre consegue se adaptar eficientemente aos dinâmicos padrões de carga, gerando desde sobre-provisionamento de recursos até instabilidade no serviço ao qual está acoplado.

Este trabalho propõe a utilização de teoria de controle para a construção de controladores que ajustem dinamicamente o número de réplicas da aplicação, avaliando o impacto dessas abordagens por meio de métricas quantitativas como tempo de resposta e precisão na utilização de recursos. Para isso, foi desenvolvido um ambiente experimental onde os controladores foram testados em diferentes cenários, permitindo analisar sua eficácia na adaptação da infraestrutura de microsserviços às variações de demanda. Os resultados obtidos demonstram o potencial dessas abordagens na melhoria da eficiência operacional e na estabilidade das aplicações em ambientes dinâmicos.

Palavras-chave: Escalabilidade Horizontal; Teoria de Controle; Kubernetes; Microsserviços;

Abstract

Horizontal scalability is essential for microservices-based applications deployed in containers. In Kubernetes, the most commonly used solution for this purpose is the Horizontal Pod Autoscaler (HPA), which dynamically adjusts the number of replicas based on predefined targets for computational resource usage, such as CPU and memory. However, as a generic solution, HPA may not always efficiently adapt to dynamic application load patterns, causing issues ranging from resource over-provisioning to instability in the associated service.

This work proposes applying control theory to design controllers capable of dynamically adjusting the application's number of replicas. The impact of these control-based approaches is evaluated through quantitative metrics such as response time and resource usage accuracy. To achieve this, an experimental environment was developed where the proposed controllers were tested under various scenarios, allowing an analysis of their effectiveness in adapting the microservices infrastructure to demand fluctuations. The obtained results demonstrate the potential of these approaches to improve operational efficiency and service stability in dynamic environments.

Keywords: Horizontal Scalability; Control Theory; Kubernetes; Microservices;

Lista de Tabelas

3.1 Recursos alocados para cada microsserviço da aplicação *Online Boutique* 23
3.2 Controladores e seus parâmetros 28
3.3 Situações e Ações do Controlador AsTAR Modificado 30

4.1 Parâmetros do sistema (P_s) e de carga (P_c) 34
4.2 Fatores utilizados no experimento 35
4.3 Comparação dos controladores sob diferentes *setpoints* para 10 usuários 37
4.4 Comparação dos controladores sob diferentes *setpoints* para 30 usuários 40
4.5 Comparação dos controladores sob diferentes *setpoints* para 50 usuários 44
4.6 Resultado dos Controladores para *Setpoint* Variável 47

Lista de Figuras

2.1 Arquitetura da virtualização com máquinas virtuais utilizando o *hypervisor* . . . 5
2.2 Arquitetura da virtualização através de contêineres 6
2.3 Arquitetura do Kubernetes e seus componentes 9
2.4 Na escalabilidade vertical, aumenta-se os recursos alocados à aplicação 11
2.5 Na escalabilidade horizontal, aumenta-se as réplicas da aplicação 12
2.6 Sistema de controle no domínio do tempo com realimentação (malha fechada) . 13
2.7 Sistema de controle no domínio da frequência com realimentação (malha fechada) 14

3.1 Ilustração do processo iterativo de metodologia executado 20
3.2 Arquitetura da aplicação *Online Boutique* 21
3.3 Página inicial da aplicação *Online Boutique* 22
3.4 Arquitetura do cluster Kubernetes implantado na AWS 23
3.5 Arquitetura do Istio no Cluster Kubernetes 24
3.6 Fluxo de operações do Custom Pod Autoscaler 25
3.7 Ilustração da planta para os controladores 26

4.1 Ilustração da métrica RTT na interação entre Cliente e Servidor 33
4.2 Acompanhamento do uso de CPU para o *setpoint* de 30% e 10 usuários 38
4.3 Réplicas alocadas no decorrer do tempo para o *setpoint* de 30% e 10 usuários . . 39
4.4 Acompanhamento do uso de CPU para o *setpoint* de 50% e 30 usuários 42
4.5 Réplicas alocadas no decorrer do tempo para o *setpoint* de 50% e 30 usuários . . 43
4.6 Acompanhamento do uso de CPU para o *setpoint* de 50% e 50 usuários 45
4.7 Réplicas alocadas no decorrer do tempo para o *setpoint* de 50% e 50 usuários . . 46
4.8 Resultado dos controladores para o cenário de *setpoint* variável 48
4.9 Réplicas alocadas no decorrer do tempo para o cenário de *setpoint* variável . . . 49

Sumário

1 Introdução **1**

1.1 Motivação 1

1.2 Objetivos 2

1.3 Contribuições 3

1.4 Estrutura do Documento 3

2 Conceitos Básicos **4**

2.1 Virtualização de Ambientes 4

2.1.1 Máquinas Virtuais 5

2.1.2 Contêineres 6

2.2 Orquestração de contêineres 7

2.3 Kubernetes 8

2.3.1 Arquitetura 8

2.3.2 Criação e Gerenciamento de Recursos 10

2.3.3 Escalabilidade 11

2.4 Teoria de Controle 12

2.4.1 Controlador PID 14

2.4.2 Controlador AsTAR 16

2.5 Trabalhos Relacionados 17

3 Construção e Desenvolvimento dos Controladores **20**

3.1 Construção do Ambiente 21

3.1.1 Aplicação 21

3.1.2 Cluster Kubernetes 22

3.1.3 *Custom Pod Autoscaler* 24

3.2 Desenvolvimento dos Controladores 25
3.2.1 Definição da Planta 26
3.2.2 Construção do Modelo 26
3.2.3 Controlador PID 27
3.2.4 Controlador AsTAR 28

4 Avaliação Experimental **31**
4.1 Definição do Sistema 31
4.2 Métricas Avaliadas 32
4.3 Parâmetros e Fatores do Sistema 33
4.4 Simulações de Carga 35
4.5 Resultados da Avaliação 36
4.5.1 Testes com 10 usuários 36
4.5.2 Testes com 30 usuários 40
4.5.3 Testes com 50 usuários 42
4.5.4 Testes com *setpoint* variável 47

5 Conclusão e trabalhos futuros **50**
5.1 Contribuições 51
5.2 Limitações 51
5.3 Trabalhos futuros 51

Referências Bibliográficas **53**

Capítulo 1

Introdução

Este capítulo apresenta a motivação para o desenvolvimento deste trabalho, destacando a importância e relevância do tema tanto do ponto de vista acadêmico quanto industrial. Em seguida, são definidos os objetivos principais da pesquisa, detalhando os pontos que serão abordados ao longo do estudo. Também são apresentadas as contribuições esperadas, ressaltando como os resultados obtidos poderão auxiliar futuros estudos e aplicações práticas relacionadas à escalabilidade dinâmica de aplicações baseadas em microsserviços.

Por fim, é descrita brevemente a estrutura do restante deste documento, fornecendo ao leitor uma visão clara e organizada dos conteúdos abordados em cada um dos capítulos subsequentes.

1.1 Motivação

Com o crescimento acelerado da computação em nuvem, empresas e desenvolvedores têm buscado sistemas e serviços cada vez mais flexíveis e capazes de se adaptar rapidamente às mudanças de demanda [1]. Antigamente, as aplicações eram predominantemente grandes sistemas monolíticos, difíceis de atualizar, implantar e escalar. Cada novo ciclo de implantação exigia grandes esforços manuais das equipes de desenvolvimento e operações, especialmente em caso de falhas no hardware. Atualmente, esses grandes sistemas monolíticos estão sendo gradualmente substituídos por arquiteturas baseadas em microsserviços, que dividem a aplicação em componentes menores, independentes e mais facilmente gerenciáveis [2].

Nesse contexto, a escalabilidade horizontal se torna uma abordagem essencial, permitindo ajustar dinamicamente o número de réplicas desses componentes de acordo com métricas observadas em tempo real, garantindo eficiência no uso dos recursos e mantendo a estabilidade

dos serviços. O Kubernetes, um *framework open source* para gerenciamento de contêineres, tem desempenhado um papel central nesse cenário, fornecendo mecanismos automáticos como o HPA[1], que escala aplicações com base em métricas de desempenho, se adaptando à carga [3].

Por ser uma abordagem genérica, o HPA trás diversos benefícios pela sua simplicidade e eficiência na maioria dos cenários. Entretanto, ele possui limitações para se adaptar a cenários mais dinâmicos de carga, o que levou a criação de mecanismos complementares que levassem em consideração outras métricas ou até mesmo a utilização de novas técnicas como aprendizagem de máquina para solucionar esses problemas [4]. Este trabalho propõe uma abordagem baseada na teoria de controle, objetivando fornecer ajustes mais precisos e que se adaptem melhor a variação de carga, melhorando o provisionamento dos recursos e garantindo a estabilidade operacional das aplicações.

1.2 Objetivos

O objetivo geral deste trabalho é propor, implementar e avaliar controladores para a escalabilidade horizontal de ABMs no Kubernetes, com foco na eficiência da alocação de recursos computacionais. Busca-se, ainda, comparar quantitativamente essas abordagens com o HPA tradicional, utilizando um ambiente experimental e o framework CPA (*Custom Pod Autoscaler*) como base para o desenvolvimento dos controladores.

Especificamente, pretende-se:

- Desenvolver uma abordagem alternativa para escalabilidade horizontal baseada na teoria de controle;
- Utilizar o framework CPA para desenvolver controladores de escalabilidade horizontal;
- Criar um ambiente experimental capaz de reproduzir diferentes cenários de carga para ABMs;
- Avaliar quantitativamente a eficiência dos controladores propostos por meio de métricas como tempo de resposta, consumo de recursos e precisão no provisionamento das réplicas;
- Comparar os resultados obtidos pelos controladores propostos com os resultados obtidos pelo HPA tradicional, identificando os cenários em que a nova abordagem demonstra vantagens significativas;

[1] `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`

- Prover recomendações para utilização prática da abordagem proposta, destacando suas vantagens e limitações em contextos reais.

1.3 Contribuições

É esperado que os resultados deste trabalho possam trazer importantes contribuições para o campo do avanço teórico e prático do cenário de escalabilidade horizontal de ABMs, destacando-se pelos seguintes aspectos:

- Proposição de uma metodologia que pode servir de base para futuras pesquisas e implementações de abordagens mais dinâmicas e eficientes;
- Estabelecimento de uma base sólida para futuras pesquisas sobre escalabilidade horizontal que explorem ainda mais o uso de teoria de controle e outras técnicas avançadas nesse contexto.
- Enriquecimento do conjunto de soluções disponíveis de técnicas para escalabilidade horizontal de ABMs utilizando técnicas de teoria de controle;
- Fornecimento de resultados empíricos obtidos a partir de um ambiente experimental detalhado, contribuindo para a validação prática e análise crítica das abordagens propostas;

1.4 Estrutura do Documento

Este trabalho está estruturado da seguinte forma: no Capítulo 2 é apresentada uma revisão teórica que abrange conceitos essenciais, como virtualização de ambientes, orquestração de contêineres, fundamentos da teoria de controle e também uma análise dos trabalhos relacionados na área. No Capítulo 3 será apresentada a metodologia adotada, detalhando as propostas de controladores para escalabilidade horizontal, a criação do ambiente utilizado para o experimento, a aplicação que será utilizada para comparação dos controladores e o processo de obtenção dos parâmetros destes. Em seguida, o Capítulo 4 expõe os resultados obtidos por meio da simulação de carga na aplicação, onde serão analisadas as métricas de desempenho dos controladores nos diferentes cenários. Por fim, o Capítulo 5 apresenta as conclusões derivadas dos resultados dos experimentos, juntamente com sugestões de melhorias e futuros direcionamentos para o aprimoramento do trabalho.

Capítulo 2

Conceitos Básicos

Este capítulo tem como objetivo apresentar os principais conceitos que fundamentam este trabalho, servindo como base para a compreensão das abordagens adotadas. Inicialmente, são introduzidos os conceitos de virtualização de ambientes, com uma breve distinção entre máquinas virtuais e contêineres. Em seguida, aborda-se a orquestração de contêineres, ressaltando sua importância para o gerenciamento de aplicações distribuídas.

Também são discutidos os elementos essenciais do Kubernetes, incluindo sua arquitetura, os mecanismos de criação e gerenciamento de recursos e suas funcionalidades relacionadas à escalabilidade. Por fim, são apresentados conceitos da teoria de controle, com destaque para controladores que serão utilizados neste trabalho e são revisados trabalhos relacionados que oferecem suporte e contexto para as propostas deste estudo.

2.1 Virtualização de Ambientes

A virtualização é uma tecnologia que permite a execução de múltiplas instâncias de sistemas operacionais ou aplicações em um ambiente isolado, criado a partir de uma máquina física única. Essa abordagem permite a maximização do uso do *hardware* e proporciona flexibilidade e escalabilidade, se tornando cada vez mais comum principalmente com o crescimento dos ambientes de computação em nuvem [5], já que facilita o gerenciamento de recursos e isolamento de processos.

Existem diferentes formas de implementar a virtualização, sendo as mais comuns as máquinas virtuais, que fornecem o que é chamado de virtualização completa, e os contêineres, que fornecem uma abordagem mais leve e eficiente, como apresentado por *Potdar et al.* [6] que

realizam uma avaliação das máquinas virtuais e contêineres Docker[1] sob uma perspectiva de métricas quantitativas em diferentes cenários de utilização.

2.1.1 Máquinas Virtuais

As máquinas virtuais representam a forma mais tradicional de virtualização, emulando completamente um sistema físico. Cada máquina virtual contém seu próprio sistema operacional, chamado de *Guest OS*, permitindo que ela possa executar de forma totalmente independente como se estivesse em um computador físico separado do que a comporta. Isso é possível graças ao *hypervisor*, que é o software responsável por gerenciar as máquinas virtuais e realizar a abstração entre o hardware e os sistemas operacionais virtualizados [6]. A Figura 2.1 ilustra a arquitetura deste tipo de virtualização.

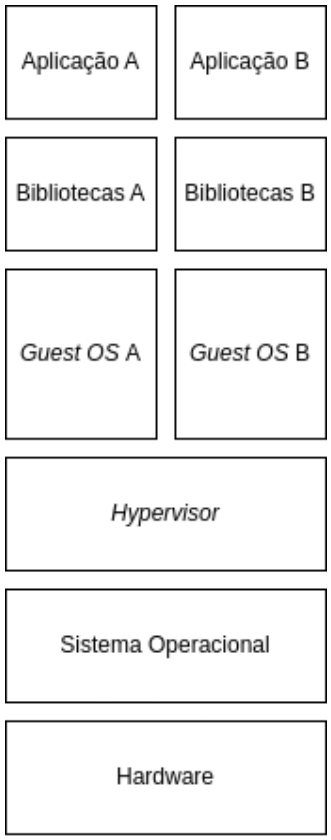


Figura 2.1: Arquitetura da virtualização com máquinas virtuais utilizando o *hypervisor*

Apesar da principal vantagem desta forma de virtualização ser, de fato, o isolamento completo de recursos (que vão desde o sistema de arquivos até drivers), a desvantagem mais explícita é que há um maior consumo de recurso acompanhado de um maior tempo de inicialização, o

[1] `https://www.docker.com/`

que pode afetar diretamente a escalabilidade e a eficiência na gestão de recursos, já que haverá um maior gasto computacional para manter toda a estrutura [7].

2.1.2 Contêineres

Os contêineres são uma tecnologia de virtualização mais leve, projetada para permitir que aplicações sejam executadas de maneira isolada, mas sem a sobrecarga de um sistema operacional completo, como ocorre nas máquinas virtuais [6, 7, 8]. Para isso, os contêineres compartilham o *kernel* do sistema operacional que os comporta, mas isolam as aplicações e seus ambientes de execução. Para fins práticos, isso significa que todos os contêineres em uma máquina se utilizam do mesmo sistema operacional, mas cada um tem suas próprias bibliotecas, dependências e configurações. Entretanto, o compartilhamento do *kernel* também é responsável pela maior desvantagem desse meio de virtualização: um erro no *kernel* pode afetar todos os contêineres de uma máquina simultaneamente. A Figura 2.2 ilustra a arquitetura deste tipo de virtualização.

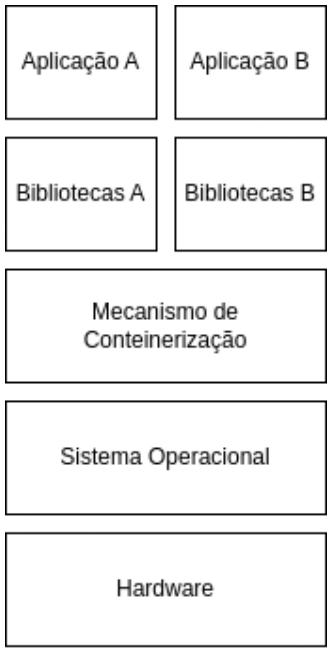


Figura 2.2: Arquitetura da virtualização através de contêineres

Uma das formas mais comuns de conteinerização é através da ferramenta Docker, que permite o empacotamento da aplicação em conjunto com todas as suas dependências em uma imagem, que pode ser executada em qualquer ambiente que suporte o Docker.

2.2 Orquestração de contêineres

À medida que a tecnologia de contêineres ganhou popularidade, surgiu também a necessidade de ferramentas que pudessem organizar, controlar e otimizar a execução de múltiplos contêineres distribuídos, especialmente em grandes escalas. A orquestração de contêineres é uma solução que automatiza esse processo, garantindo que os contêineres sejam implantados, escalados, monitorados e recuperados de maneira eficiente, sem intervenção manual.

É esperado também que estes orquestradores ofereçam funcionalidades tais como controle do uso de recursos, agendamento de contêineres, balanceamento de carga, *health check*, tolerância a falhas e escalonamento automático [9]. *Casalicchio* [9] define cada uma destas funcionalidades como a seguir:

- **Controle do uso de recursos:** Restringe a quantidade de CPU e memória que cada contêiner pode usar, evitando interferências entre eles.
- **Agendamento de contêineres:** Define onde e como os contêineres são distribuídos nos nós, com base em recursos e afinidade de nós.
- **Balanceamento de carga:** Distribui o tráfego de rede entre múltiplas instâncias de contêineres para garantir desempenho equilibrado.
- ***Health Check*:** Monitora se os contêineres estão funcionando corretamente, verificando a capacidade de responder a solicitações.
- **Tolerância à falhas:** Mantém a disponibilidade do serviço, criando réplicas de contêineres e substituindo contêineres falhos.
- **Escalonamento automático:** Adiciona ou remove contêineres automaticamente com base em métricas como uso de CPU ou memória para ajustar a capacidade à demanda.

No campo da orquestração de contêineres, as soluções podem ser divididas entre opções *on-premise* e gerenciadas. Entre as principais soluções *on-premise*, destacam-se o Docker Swarm, que é o orquestrador nativo do Docker, e o Kubernetes, que vem sendo adotado como padrão da indústria. Já entre as soluções gerenciadas, destacam-se o *Google Kubernetes Engine*[2], o *Amazon Elastic Kubernetes* Service[3] e o *Microsoft Azure Kubernetes* Service[4], que fornecem

[2] `https://cloud.google.com/kubernetes-engine`
[3] `https://aws.amazon.com/pt/eks/`
[4] `https://azure.microsoft.com/en-us/products/kubernetes-service/`

orquestração de contêineres como serviço, simplificando a gestão e escalabilidade em ambientes em nuvem [9].

2.3 Kubernetes

O Kubernetes é um *framework* de código aberto projetado para automatizar o processo de implantação, escalabilidade e gerenciamento de aplicações conteinerizadas. Ele oferece uma plataforma portátil e extensível que possibilita a implantação e manutenção eficientes das aplicações [2].

Ele foi desenvolvido inicialmente pelo Google e lançado em 2015, incorporando mecanismos avançados para gerenciar o ciclo de vida das aplicações conteinerizadas, facilitando a implementação, administração e escalabilidade dos contêineres. Além disso, a abertura do Kubernetes como um projeto de código aberto pela *Cloud Native Computing Foundation* (CNCF)[5] contribuiu para sua rápida adoção em todo o setor, permitindo que empresas de todos os portes utilizassem a plataforma de maneira flexível e escalável.

2.3.1 Arquitetura

A arquitetura do Kubernetes é composta por dois componentes principais: o *master node*, que controla o estado do *cluster*, e os *worker nodes*, responsáveis por executar os contêineres [10]. Cada contêiner é empacotado em uma unidade chamada *pod*, que é a menor unidade de implantação no Kubernetes, podendo conter um ou mais contêineres que compartilham a mesma rede e volumes de armazenamento. A Figura 2.3 ilustra a arquitetura que é executada pelo Kubernetes.

Master Node

No Kubernetes, o *Master Node* é responsável pela gestão e orquestração do *cluster*, centralizando funções de controle como o agendamento de *workloads*, monitoramento da saúde dos nós e a exposição da API para a interação com clientes [2]. Ele é composto por quatro componentes principais: o *API Server*, que comunica as interações entre os componentes do *cluster*; o *Scheduler*, que aloca os Pods nos nós de trabalho conforme requisitos de recursos;

[5] `https://www.cncf.io/`

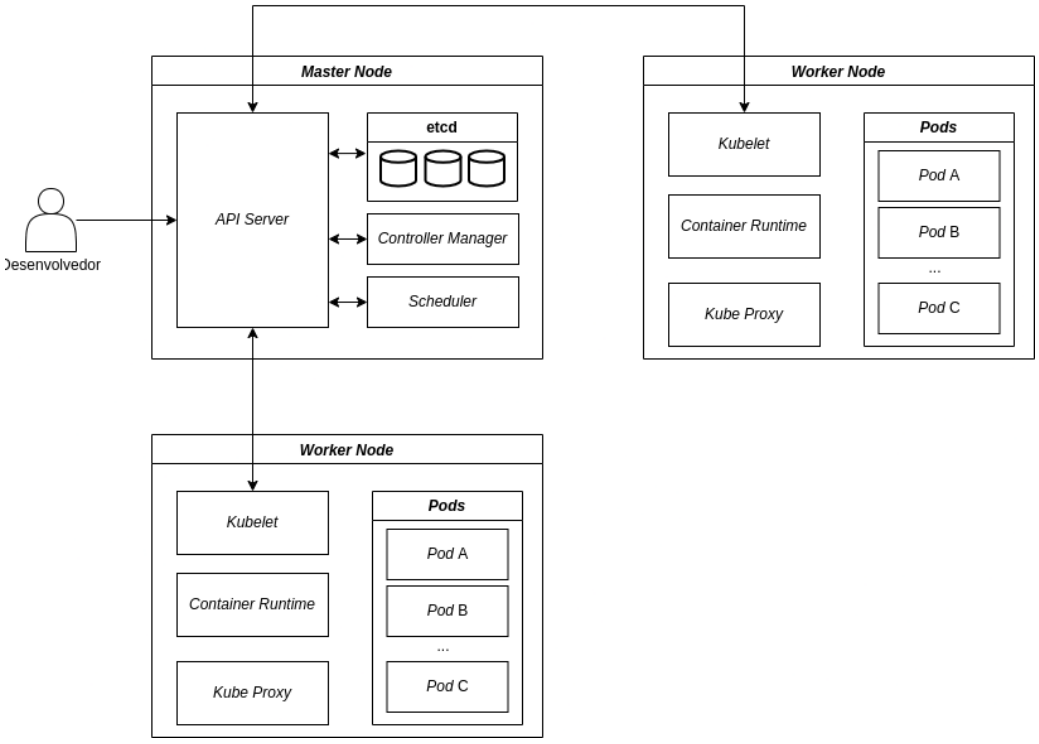


Figura 2.3: Arquitetura do Kubernetes e seus componentes

o *Controller Manager*, que mantém o estado desejado do *cluster* e gerencia falhas de nós; e o *etcd*, um repositório distribuído que armazena a configuração e o estado do *cluster*.

Worker Node

O *Worker Node* é a unidade de execução dos Pods, podendo ser uma máquina virtual ou física, sendo responsável por fornecer os recursos computacionais, de armazenamento e rede necessários para rodar as aplicações [2]. Cada *Worker Node* contém três componentes essenciais: o *Kubelet*, que gerencia os contêineres e comunica-se com o *API Server*; o *Kube Proxy*, que lida com o balanceamento de carga e roteamento de tráfego para os serviços; e o *Container Runtime*, responsável por executar os contêineres.

Pod

O *Pod* é a unidade mínima de implantação no Kubernetes, geralmente composta por um ou mais containers que compartilham a mesma rede e armazenamento, sendo altamente escalável e de natureza efêmera [10]. Isso significa que um Pod não é projetado para ser persistente: ele pode ser criado, destruído e recriado a qualquer momento pelo Kubernetes, seja devido a falhas, atualizações ou ajustes de escalabilidade. Cada Pod é atribuído a um único nó do

cluster e pode conter contêineres que, embora executem diferentes aplicações ou partes de uma aplicação, precisam compartilhar recursos, como volumes de armazenamento ou endereços IP.

2.3.2 Criação e Gerenciamento de Recursos

No Kubernetes, arquivos YAML e JSON são usados para descrever objetos dentro do cluster, chamados também de manifestos [10]. Esses arquivos são estruturados para fornecer uma interface bastante declarativa e clara sobre os objetos a que se referem. No Código 2.1 é possível ver um exemplo destes arquivos, que descreve um *Pod*:

```
1  apiVersion: apps/v1
2  kind: Pod
3  metadata:
4    name: nginx
5  spec:
6    containers:
7      - name: nginx
8        image: nginx:latest
9        resources:
10         requests:
11           memory: "256Mi"
12           cpu: "250m"
13         limits:
14           memory: "512Mi"
15           cpu: "500m"
```

Código 2.1: Exemplo de manifesto YAML no Kubernetes

Também é possível notar no manifesto o gerenciamento de recursos no objeto criado. No Kubernetes, esse gerenciamento é realizado por meio de limites e requisições de CPU e memória. Esses parâmetros definem o uso mínimo e máximo de recursos para contêineres presentes nos Pods, garantindo que as aplicações não consumam mais recursos do que o necessário, além de permitir o agendamento eficiente e a alocação de recursos dentro do *cluster*.

Neste cenário, o Pod é configurado para requerer *256Mi* de memória e *250m* de CPU. Aqui, *Mi* representa a unidade *Mebibyte* e *m* significa *millicore*, que equivale a 1/1000 de um núcleo de CPU. Essa definição granular permite que múltiplos Pods compartilhem um mesmo núcleo, proporcionando uma melhor distribuição dos recursos computacionais.

2.3.3 Escalabilidade

Já no que se refere à escalabilidade, no Kubernetes ela pode ser realizada de duas formas: vertical e horizontal [2]. A escalabilidade vertical envolve o ajuste dos recursos (como CPU e memória) alocados a um único Pod, aumentando ou diminuindo sua capacidade interna para lidar com variações de carga. Já a escalabilidade horizontal diz respeito à replicação de Pods, ou seja, ao aumento ou diminuição do número de instâncias de uma aplicação, distribuindo a carga de forma mais eficiente entre múltiplos contêineres.

Escalabilidade Vertical

O Kubernetes oferece o Vertical Pod Autoscaler (VPA)[6] para auxiliar no gerenciamento da escalabilidade vertical de forma dinâmica [2]. O VPA monitora continuamente o uso de recursos dos Pods e, com base nesse monitoramento, realiza ajustes para otimizar a alocação de recursos. Em cenários onde a aplicação enfrenta variações significativas na carga, o VPA pode sugerir aumentos ou reduções nos recursos, ajudando a evitar tanto o super provisionamento quanto a insuficiência de recursos.

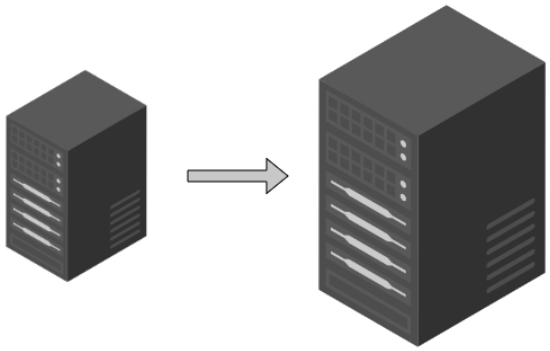


Figura 2.4: Na escalabilidade vertical, aumenta-se os recursos alocados à aplicação

Escalabilidade Horizontal

A escalabilidade horizontal no Kubernetes consiste na replicação dinâmica de Pods para distribuir a carga de trabalho e garantir a disponibilidade e o desempenho da aplicação [10]. Esse mecanismo permite que o sistema se ajuste automaticamente ao aumento ou à redução da demanda, evitando a sobrecarga da aplicação e otimizando o uso dos recursos do *cluster*.

O componente central para essa abordagem é o HPA, que monitora continuamente métricas como uso de CPU e memória. Com base nos limiares estabelecidos, o HPA ajusta o número de

[6] `https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler`

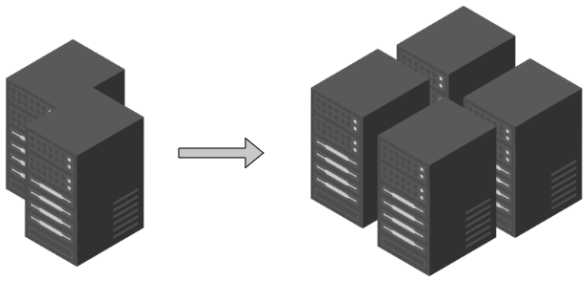


Figura 2.5: Na escalabilidade horizontal, aumenta-se as réplicas da aplicação

réplicas dos Pods, escalando conforme a carga à qual a aplicação está sendo submetida. O HPA calcula o número de réplicas desejadas conforme a Equação 2.1, com base no número atual de réplicas, multiplicado pela razão entre a utilização atual e a desejada da métrica-alvo.

$$\text{Réplicas Desejadas} = \left(\frac{\text{Valor Atual da Métrica}}{\text{Valor Alvo da Métrica}}\right) \times \text{Réplicas Atuais} \tag{2.1}$$

2.4 Teoria de Controle

A Teoria de Controle é um campo da engenharia e da matemática que se concentra no comportamento de sistemas dinâmicos. Seu objetivo principal é projetar e analisar sistemas que manipulem ou regulem seu comportamento de modo a atingir resultados desejados. Essa abordagem tem sido amplamente utilizada em várias áreas, como engenharia, física, biologia, economia e, mais recentemente, no desenvolvimento de sistemas adaptativos de software.

Em sistemas de controle, a ideia central é o uso de controladores para ajustar dinamicamente o comportamento de sistemas. Segundo *Franklin et al.* [11], um controlador é o componente responsável por determinar o sinal de controle necessário para levar a saída de um sistema até o valor desejado, também chamado de *setpoint*. Esse processo é realizado utilizando-se da diferença entre a saída medida do sistema e a entrada de referência, por onde se obtém o erro da medida.

Dentre os componentes básicos de um sistema de controle, podemos destacar a planta, que é o elemento a ser controlado descrito por um modelo matemático obtido formal ou experimentalmente; o controlador, que é o algoritmo responsável por gerar a entrada de controle baseando-se no erro calculado entre a variável medida e o *setpoint*; a entrada de controle, que é o sinal ou comando aplicado ao sistema, gerada pelo controlador, para influenciar seu comportamento; a variável medida, que é a característica observada ou medida; o setpoint, que representa o valor desejado para a variável medida; e, por fim, o ganho de controle, que representa a intensidade

da ação a ser tomada pelo controlador em resposta ao erro computado [12].

Os controladores podem ser classificados em malha aberta e malha fechada [12], dependendo da forma como controlam o sistema. Em um sistema de controle em malha aberta, a saída não é monitorada nem utilizada para ajustar a entrada; o controle é aplicado com base em um modelo pré-determinado, tornando-o simples, porém suscetível a erros causados por variações externas e incertezas no sistema. Já no sistema de controle em malha fechada, também chamado de sistema com realimentação (*feedback*), a saída é constantemente medida e comparada com o *setpoint*, e o controlador ajusta a entrada para minimizar o erro. A Figura 2.6 ilustra um exemplo de sistema de controle no domínio do tempo que opera em malha fechada, onde é possível notar a atuação do *feedback* realimentando a entrada do sistema.

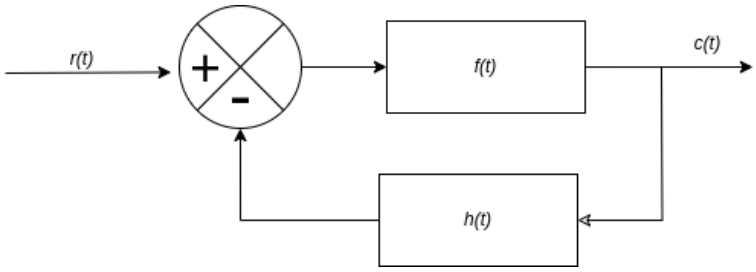


Figura 2.6: Sistema de controle no domínio do tempo com realimentação (malha fechada)

Qualquer saída de um sistema de controle com realimentação negativa que seja estável pode ser representada por uma equação diferencial linear invariante no tempo que relaciona a entrada $r(t)$ com a saída $c(t)$ [12], como demonstra a Equação 2.2. Porém, comumente estes tipos de sistema são convertidos utilizando transformadas de Laplace, já que no domínio da frequência derivadas e integrais são transformadas em simples multiplicações e divisões, respectivamente.

$$a_n \frac{d^n c(t)}{dt^n} + a_{n-1} \frac{d^{n-1} c(t)}{dt^{n-1}} + \cdots + a_0 c(t) = b_m \frac{d^m r(t)}{dt^m} + b_{m-1} \frac{d^{m-1} r(t)}{dt^{m-1}} + \cdots + b_0 r(t) \qquad (2.2)$$

Aplicando a transformada de Laplace na Equação 2.2, podemos obter a função de transferência que descreve o sistema no domínio da frequência, sendo a planta do sistema. O processo de transformação está descrito na Equação 2.3, onde $G(s)$ representa a planta do sistema de controle no domínio da frequência:

$$a_n s^n C(s) + a_{n-1} s^{n-1} C(s) + \cdots + a_0 C(s) = b_m s^m R(s) + b_{m-1} s^{m-1} R(s) + \cdots + b_0 R(s) \quad (2.3)$$

$$(a_n s^n + a_{n-1} s^{n-1} + \cdots + a_0) C(s) = (b_m s^m + b_{m-1} s^{m-1} + \cdots + b_0) R(s) \quad (2.4)$$

$$\frac{C(s)}{R(s)} = G(s) = \frac{a_n s^n + a_{n-1} s^{n-1} + \cdots + a_0}{b_m s^m + b_{m-1} s^{m-1} + \cdots + b_0} \quad (2.5)$$

Com isso em mãos, podemos adaptar o sistema visto na Figura 2.6 para o domínio da frequência. A função de transferência para este tipo de sistema em malha fechada pode ser vista na Equação 2.6 e o sistema modificado na Figura 2.7. Essa remodelagem facilita tanto a análise de propriedades do controlador quanto a modelagem destes, como no controlador PID (Proporcional-Integrativo-Derivativo).

$$\frac{C(s)}{R(s)} = G_e(s) = \frac{G(s)}{1 + G(s)H(s)} \quad (2.6)$$

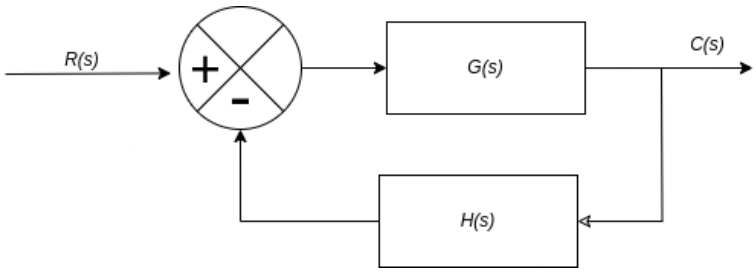


Figura 2.7: Sistema de controle no domínio da frequência com realimentação (malha fechada)

2.4.1 Controlador PID

Os controladores PID são amplamente utilizados devido à sua simplicidade e eficácia em uma variedade de sistemas de controle. Eles ajustam a entrada de controle com base em três parâmetros: o parâmetro proporcional K_p, que responde diretamente ao erro atual; o parâmetro integral K_i, que acumula os erros passados para eliminar o erro estacionário; e o parâmetro derivativo K_d, que prevê a tendência futura do erro, proporcionando uma resposta mais rápida [12]. A Equação 2.7 descreve a fórmula utilizada neste tipo de controlador, onde $u(t)$ descreve o sinal de controle no instante t e $e(t)$ fornece o erro no instante t.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de(t)}{dt} \quad (2.7)$$

O ajuste dos parâmetros K_p, K_i e K_d é fundamental para garantir o desempenho ideal do controlador PID. Para sistemas lineares e simples, os parâmetros podem ser ajustados manualmente por tentativa e erro, mas, em sistemas mais complexos, técnicas baseadas em otimização podem ser aplicadas para um ajuste mais preciso. O ajuste adequado desses parâmetros é crucial, pois um valor excessivo para qualquer um dos termos pode levar a uma resposta excessivamente rápida ou instável, enquanto valores muito baixos podem resultar em uma resposta lenta ou ineficaz. Este controlador pode ter também mais algumas variações onde se remove a componente integrativa, derivativa ou ambas.

Controlador P

O Controlador P (Proporcional) é amplamente utilizado em sistemas simples devido à sua resposta rápida às variações do erro. Esse tipo de controle ajusta a saída do sistema proporcionalmente ao erro medido, sendo definido pelo ganho K_p. No entanto, ele possui uma limitação significativa, que é a presença de um erro residual em regime permanente [12]. A Equação 2.8 descreve a fórmula deste tipo de controlador.

$$u(t) = K_p e(t) \tag{2.8}$$

Controlador PI

O Controlador PI (Proporcional-Integral) é utilizado quando se deseja eliminar o erro residual em regime permanente presente no controlador proporcional. O termo integral acumula o erro ao longo do tempo e ajusta continuamente a saída do sistema até que o erro seja reduzido a zero, sendo ideal em sistemas que exigem precisão. No entanto, a adição da ação integrativa pode aumentar o tempo de estabilização e levar a oscilações indesejadas, exigindo um ajuste criterioso dos ganhos K_p e K_i [12]. A Equação 2.9 descreve a fórmula deste tipo de controlador.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau \tag{2.9}$$

Controlador PD

O Controlador PD (Proporcional-Derivativo) combina a ação proporcional com um termo derivativo que reage à taxa de variação do erro, atuando de forma benéfica no regime transitório. Esse comportamento permite que o controlador antecipe mudanças no sistema, reduzindo

oscilações e melhorando a estabilidade [12]. Este controlador é frequentemente utilizado em sistemas onde uma resposta rápida e estável é necessária, como em controle de movimento e posicionamento. No entanto, por não atuar no regime permanente, não corrige o erro residual. A Equação 2.10 descreve a fórmula deste tipo de controlador.

$$u(t) = K_p e(t) + K_d \frac{de(t)}{dt} \tag{2.10}$$

2.4.2 Controlador AsTAR

O controlador AsTAR proposto por *Yang et al.* [13] trás uma abordagem voltada para a gestão adaptativa de tarefas em dispositivos de *Internet of Things* (IoT), visando garantir a operação sustentável em ambientes dinâmicos ao ajustar automaticamente as taxas de execução das tarefas conforme a energia disponível. A abordagem deste controlador é baseada no conceito do *Additive Increase Multiplicative Decrease* (AIMD), permitindo uma adaptação eficiente às variações na disponibilidade de energia. O algoritmo 1 demonstra o pseudocódigo para este controlador.

Algoritmo 1: Pseudocódigo do controlador AsTAR

Dados: Tensão do Capacitor (V), Tensão de Desligamento (SV), Tensão Ótima (OV)

Resultado: Taxa de execução adaptada das tarefas

```
se V < SV então
    Entrar no estado de desligamento;
    Desativar todas as tarefas;
senão se V < OV então
    se a tensão estiver diminuindo então
        Reduzir a taxa de execução de forma multiplicativa;
    senão
        Aumentar a taxa de execução de forma aditiva;
senão se V > OV então
    se a tensão estiver diminuindo então
        Diminuir a taxa de execução de forma aditiva;
    senão
        Aumentar a taxa de execução de forma multiplicativa;
senão
    Manter a taxa de execução no estado de voltagem ótima;
```

Desta forma, o AsTAR tenta trazer uma adaptação eficiente à disponibilidade de energia, aumentando gradualmente a taxa de execução das tarefas quando há recursos suficientes e reduzindo de forma agressiva a taxa quando a energia se torna limitada. Essa estratégia assegura que o sistema mantenha um equilíbrio entre o consumo de energia e a execução das tarefas, garantindo a operação sustentável mesmo em cenários de variação energética.

2.5 Trabalhos Relacionados

A importância da escalabilidade tem se consolidado como um fator crucial para o desempenho eficiente de aplicações baseadas em microsserviços. Diversos estudos têm realizado diferentes tipos de propostas para solucionar estes cenários, abordando tanto escalabilidade horizontal quanto vertical. Como o foco deste trabalho se dá especificamente em cenários de escalabilidade horizontal de microsserviços no Kubernetes, os trabalhos a seguir foram selecionados para

permear esta premissa.

Rosa e Cavalcanti [14] propõem o uso de teoria de controle para ajustar dinamicamente parâmetros de brokers em sistemas de middleware orientados a mensagens (MOMs), visando evitar a sobrecarga de consumidores. Para isso, foram implementados e avaliados dezoito controladores, incluindo PID, variantes, controladores adaptativos e não convencionais como HPA e AsTAR. A eficácia dos controladores foi analisada em diferentes cenários sob métricas quantitativas, demonstrando que alguns deles conseguem melhorar significativamente o desempenho do broker ao adaptar-se às variações do ambiente.

Nguyen et al. [15] investigam o uso do HPA no Kubernetes, realizando uma série de experimentos para avaliar o comportamento operacional do HPA comparando o uso das métricas padrão de recursos (CPU e memória) com as métricas personalizadas fornecidas pelo Prometheus[7]. É valido citar que uma das principais contribuições do estudo é a avaliação detalhada do HPA sob diferentes métricas e configurações de intervalo de análises, onde é também discutido como a escolha destes parâmetros pode influenciar o desempenho do controlador.

Khaleq e Ra [16] apresentam uma abordagem agnóstica baseada no HPA, observando o consumo de recursos da aplicação no Kubernetes para realizar adaptações dinâmicas nos parâmetros do HPA, como quantidade mínima e máxima de réplicas e *thresholds* das métricas observadas. Os testes realizados demonstram uma melhoria em até 20% no tempo de resposta dos microsserviços em comparação com o escalonamento padrão do HPA baseado em CPU, demonstrando a importância da customização dos parâmetros do HPA para se extrair maior eficiência do escalador.

Serracanta et al. [17] aplicam teoria de controle para modelar o HPA como um sistema de controle, apresentando a modelagem de um sistema de microsserviços interconectados. O estudo demonstra a construção e modelagem formal do sistema em questão e também conta com simulações e experimentos em ambiente real, demonstrando a possibilidade de se prever o comportamento do HPA e se antecipar a alocação de recursos de forma coordenada, antecipando a escala do sistema antes que ocorram sobrecargas ou degradação no desempenho.

Baarzi e *Kesidis* [18] descrevem o SHOWAR, um framework de escalonamento vertical e horizontal que visa otimizar a alocação de recursos e melhorar o desempenho das aplicações. Para o escalonamento horizontal, um controlador PID é proposto para ajustar o número de réplicas com base em métricas de desempenho do sistema, demonstrando uma expressiva melhoria na

[7] `https://prometheus.io/`

alocação de recursos.

Joshi et al. [19] demonstram uma abordagem para escalabilidade utilizando o algoritmo ARIMA (Auto-Regressive Integrated Moving Average) e teoria de controle por meio de um controlador PID. Essa abordagem busca antecipar a demanda futura da aplicação por recursos e aplica estas demandas como entrada para o controlador, buscando atingir estabilidade e melhor utilização dos recursos combinados com redução da latência.

Estes estudos demonstram a diversidade de abordagens para o escalonamento de microsserviços, com uma grande ênfase em técnicas preditivas para antecipar a demanda de recursos, que vem ganhando cada vez mais destaque principalmente para aplicações submetidas a padrões de carga imprevisíveis, mas que ainda assim precisam garantir estabilidade [20].

Capítulo 3

Construção e Desenvolvimento dos Controladores

Neste capítulo será descrita a metodologia empregada para o desenvolvimento do trabalho. Inicialmente, foi necessário definir a aplicação utilizada para o comparativo dos controladores, seguido da construção do ambiente que será utilizado para os testes. Por fim, será descrito o processo de desenvolvimento dos controladores, descrevendo as etapas realizadas para obtenção dos parâmetros e modificações realizadas para tornar os controladores compatíveis com o cenário de escalabilidade horizontal no Kubernetes. A Figura 3.1 ilustra o processo iterativo adotado para a execução desta etapa.

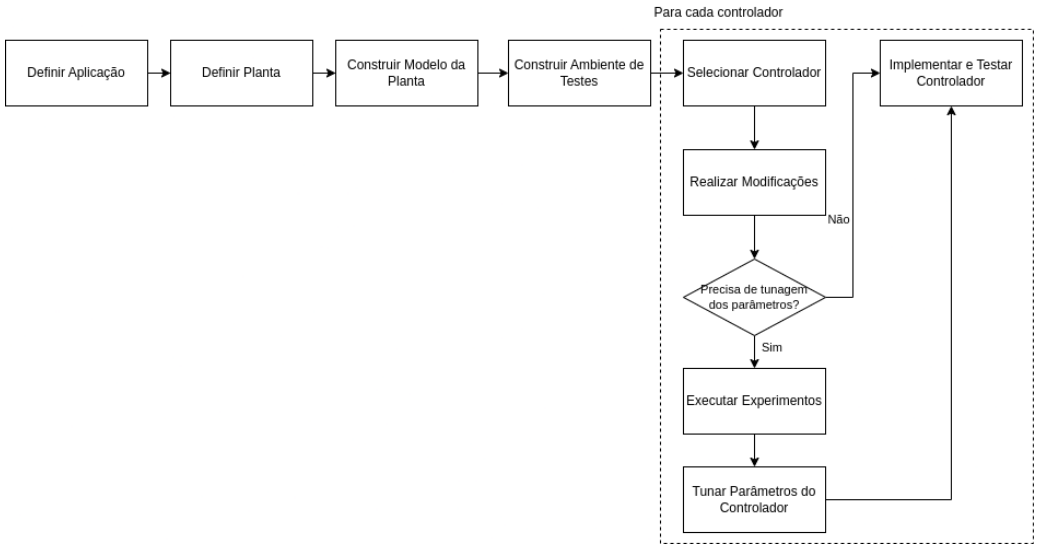


Figura 3.1: Ilustração do processo iterativo de metodologia executado

3.1 Construção do Ambiente

Como parte essencial do processo de preparação para a condução dos experimentos, foi necessário fazer uma etapa anterior ao desenvolvimento dos controladores, a construção do ambiente. Nesta seção serão descritos os processos adotados para a construção do cluster Kubernetes que comportará os experimentos, a descrição do framework utilizado para a construção de escaladores personalizados e também a aplicação definida para os testes.

3.1.1 Aplicação

Durante a revisão de trabalhos relacionados, foi possível observar que algumas aplicações são comumente utilizadas como *benchmark* no cenário de escalabilidade horizontal. Normalmente, estas aplicações contam com um vasto número de microsserviços interagindo entre si, sendo o cenário caótico onde se apresentam a mais vasta quantidade de desafios para os controladores. Dentre essas aplicações, se destaca a *Google Online Boutique*[1], composta por 11 microsserviços nos quais os usuários podem comprar itens online por meio de um carrinho de compras e realizar pagamentos. Esse ambiente simula uma aplicação de *e-commerce* ideal para avaliar o comportamento de escalabilidade e desempenho de microsserviços, sendo, portanto, utilizado neste trabalho como a aplicação de teste no cluster Kubernetes. A Figura 3.2 ilustra todos os microsserviços e como eles interagem entre si.

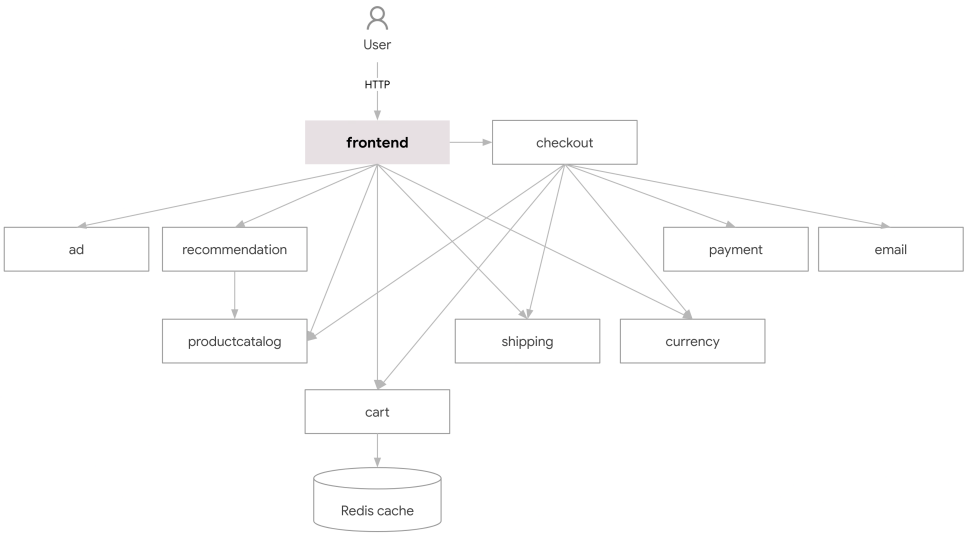


Figura 3.2: Arquitetura da aplicação *Online Boutique*

Nesta aplicação, cada microsserviço conta com uma função específica: o *Frontend* fornece

[1] `https://github.com/GoogleCloudPlatform/microservices-demo`

a interface do usuário para navegação e gerenciamento do carrinho de compras; o *Checkout* gerencia o processo de finalização da compra; o *Payment* processa as transações financeiras; o *Email* envia notificações aos usuários, como confirmações de pedidos; o *Currency* converte valores para diferentes moedas com base nas taxas de câmbio; o *Shipping* gerencia o cálculo e a logística de envio dos pedidos; o *Cart* manipula a adição e remoção de itens no carrinho; o *ProductCatalog* gerencia a lista de produtos disponíveis na loja; o *Recommendation* sugere produtos com base no comportamento do usuário; o *Ad* gerencia a exibição de anúncios dentro da aplicação; e o *Redis Cache* armazena e acelera a recuperação dos dados do carrinho. A Figura 3.3 ilustra a página inicial da aplicação.

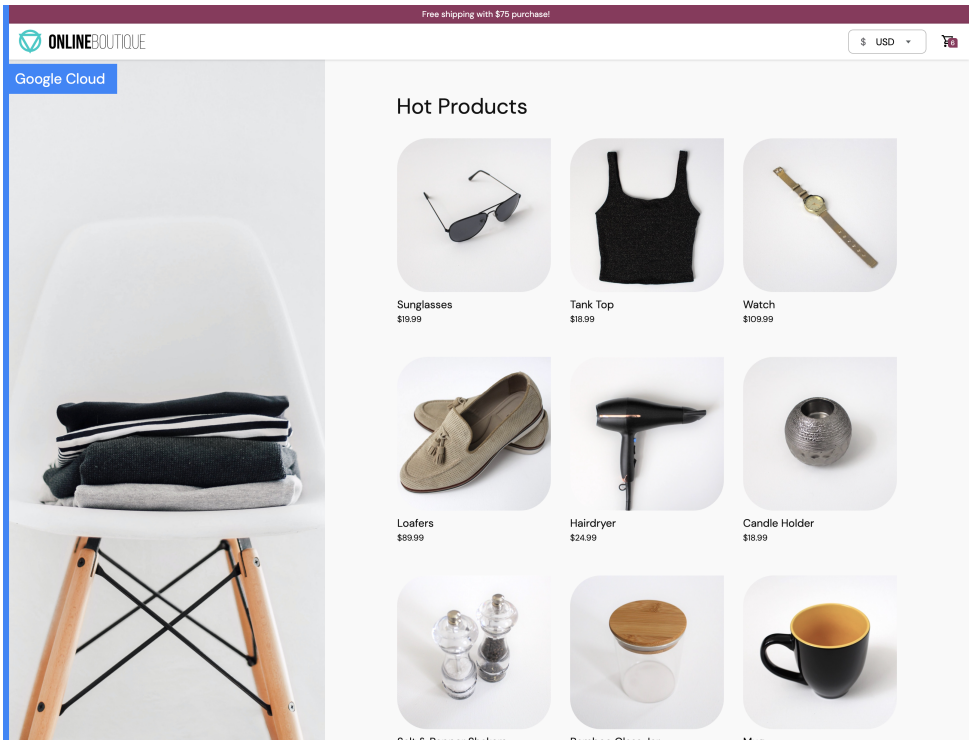


Figura 3.3: Página inicial da aplicação *Online Boutique*

Além disso, o repositório também trás algumas pré-definições dos recursos de CPU e memória que devem ser alocados para cada um dos microsserviços. Essas definições podem ser encontradas na Tabela 3.1.

3.1.2 Cluster Kubernetes

Para a construção do ambiente experimental deste trabalho, foi utilizado o serviço Amazon EKS para a criação de um cluster Kubernetes, possibilitando a construção de um ambiente de orquestração de contêineres. Neste cluster também foi realizada a configuração de componentes

Tabela 3.1: Recursos alocados para cada microsserviço da aplicação *Online Boutique*

Microsserviço	**CPU**		**Memória**	
	Requests	**Limits**	**Requests**	**Limits**
adservice	200m	300m	180Mi	300Mi
cartservice	200m	300m	64Mi	128Mi
checkoutservice	100m	200m	64Mi	128Mi
currencyservice	100m	200m	64Mi	128Mi
emailservice	100m	200m	64Mi	128Mi
frontend	100m	200m	64Mi	128Mi
paymentservice	100m	200m	64Mi	128Mi
productcatalogservice	100m	200m	64Mi	128Mi
recommendationservice	100m	200m	220Mi	450Mi
shippingservice	100m	200m	64Mi	128Mi
redis-cart	70m	125m	200Mi	256Mi

como Prometheus, Grafana[2] e Istio[3]. Esses componentes serão parte fundamental para as etapas de monitoramento e observabilidade que serão discutidas nas próximas seções. A Figura 3.4 ilustra a arquitetura com os componentes da AWS utilizados para a construção do cluster.

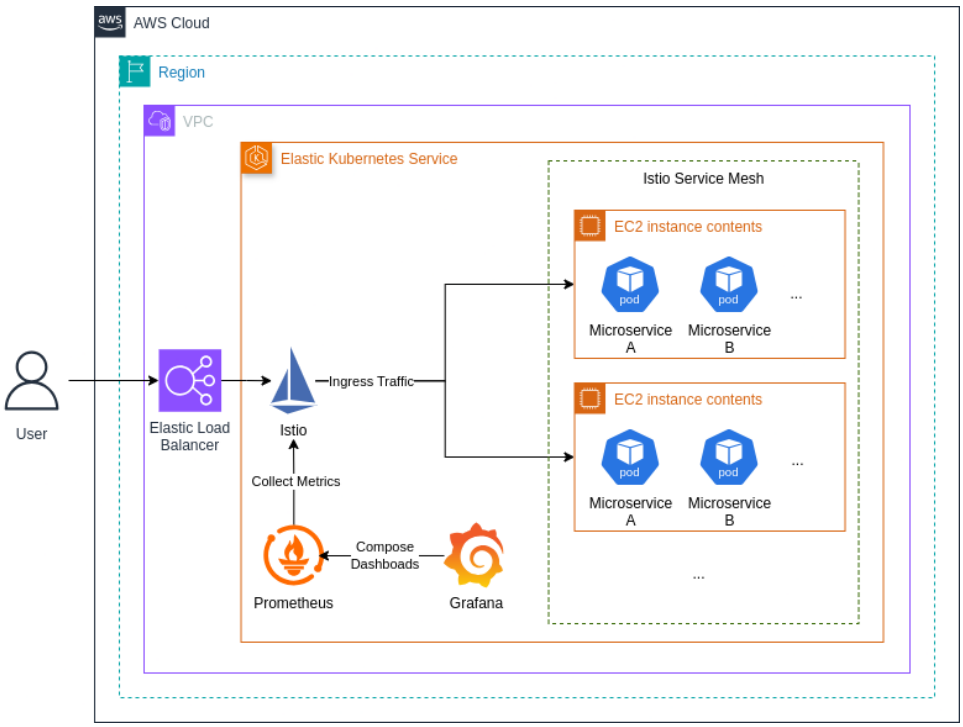


Figura 3.4: Arquitetura do cluster Kubernetes implantado na AWS

A escolha do componente Istio também é essencial por permitir um gerenciamento mais simplificado do balanceamento de carga e também gera um ponto único de monitoramento

[2] `https://grafana.com/`
[3] `https://istio.io/`

sem a necessidade de se realizar alterações nas aplicações. Para isto, o Istio divide a sua malha de serviço em dois pontos, o *Data Plane*, gerenciando um conjunto de *proxies* que interceptam todo o tráfego para a aplicação e também sendo os responsáveis por aplicar regras para balanceamento de carga e geração de métricas, e o *Control Plane*, responsável por fazer a descoberta dos *proxies* e o gerenciamento e configuração destes para lidar com o roteamento de tráfego através do componente Istiod. Essa estruturação é ilustrada na Figura 3.5, onde é possível observar a divisão da malha de serviço do Istio e como os serviços se integram a ela.

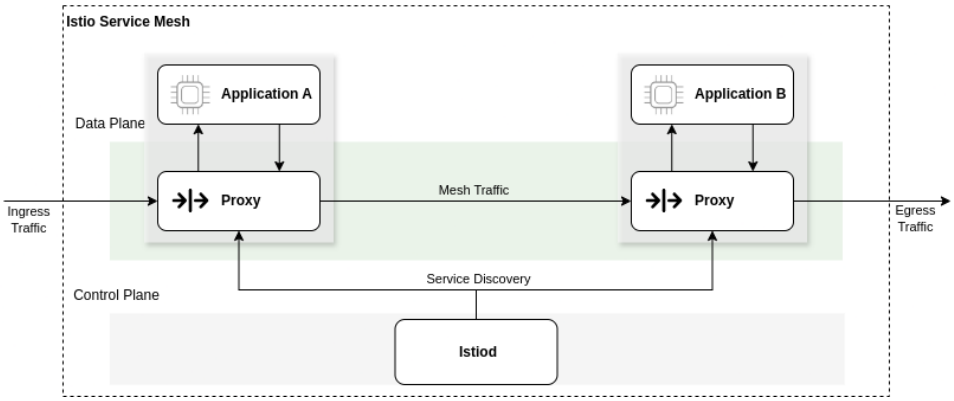


Figura 3.5: Arquitetura do Istio no Cluster Kubernetes

Para compor os *worker nodes*, foi configurado no console AWS o recurso *Node Pool*, onde foram disponibilizadas instâncias do tipo *c7g.xlarge*[4]. A escolha desse tipo de instância se deu principalmente pelas características observadas da aplicação de teste, permitindo um dimensionamento efetivo da máquina para os testes, combinando eficiência e custo.

3.1.3 *Custom Pod Autoscaler*

Um desafio inerente ao processo de compor e desenvolver novos controladores no Kubernetes é que não existe uma forma direta e fácil de implementá-los na arquitetura do cluster. Normalmente, são criadas aplicações na linguagem Go que se comunicam diretamente com o *API Server*, como demonstrado no livro *Programming Kubernetes*[21]. Essas aplicações fazem todo o papel de realizar o monitoramento das aplicações e implementar também a lógica do controlador, se comunicando com o *API Server* para poder realizar as operações de escalonamento das réplicas da aplicação.

Para abstrair todo esse processo de desenvolvimento e interações com o Kubernetes, o *Custom Pod Autoscaler Framework*[5] (CPA) foi desenvolvido. O CPA é um *framework* de esca-

[4] `https://aws.amazon.com/pt/ec2/instance-types/c7g/`
[5] `https://custom-pod-autoscaler.readthedocs.io/en/latest/`

labilidade horizontal no Kubernetes que permite a criação de controladores personalizados para ajustar o número de réplicas de pods, possibilitando a definição de controladores personalizados conforme as necessidades específicas da aplicação ou do ambiente.

O funcionamento do CPA pode ser dividido em 3 etapas: a obtenção de métricas, onde é feita uma integração com o *API Server* do Kubernetes para coletar dados sobre a aplicação como quantidade réplicas e uso de CPU e memória, a etapa de avaliação, avalia as métricas coletadas no controlador implementado e o escalador, que recebe o número de métricas a serem escaladas pela etapa de avaliação e realiza a comunicação com o *API Server* para que o Kubernetes escale a aplicação. Este fluxo é ilustrado na Figura 3.6.

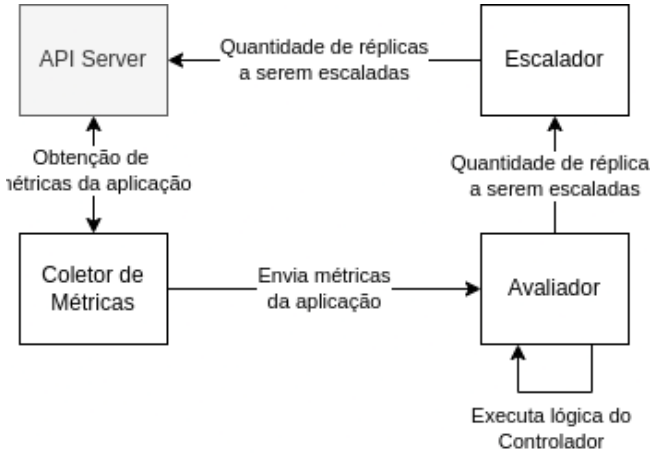


Figura 3.6: Fluxo de operações do Custom Pod Autoscaler

Uma das grandes vantagens deste *framework* e também o motivo pelo qual ele foi escolhido neste trabalho é devido a simplificação dos processos, onde a criação de novos controladores abstrai etapas como as integrações ao *API Server* do Kubernetes para obtenção de métricas das aplicações e também as operações de escalonamento. É possível executar códigos em qualquer linguagem em cada uma das etapas, além de também ser possível estendê-las para realizar qualquer outro tipo de operação, como coletar as métricas de outra fonte ou consultar uma API externa para executar o controlador, por exemplo.

3.2 Desenvolvimento dos Controladores

Dando sequência ao processo de desenvolvimento do trabalho, temos a etapa de construção dos controladores no ambiente do Kubernetes, utilizando o CPA como *framework*. A seguir serão descritos o processo de desenvolvimento para cada um dos controladores PID e AsTAR na estrutura proposta, apresentando os processos para análise e obtenção dos parâmetros em

conjunto com as adaptações feitas para trazer os controladores para o contexto da escalabilidade horizontal de contêineres.

3.2.1 Definição da Planta

A aplicação de conceitos da teoria de controle de um sistema se inicia com a identificação da planta, que expressa o elemento a ser controlado. No contexto do Kubernetes, podemos expressar a planta em função das interações do controlador com o *API Server* do Kubernetes, responsável por controlar o número de réplicas da aplicação, conforme ilustrado na Figura 3.7.

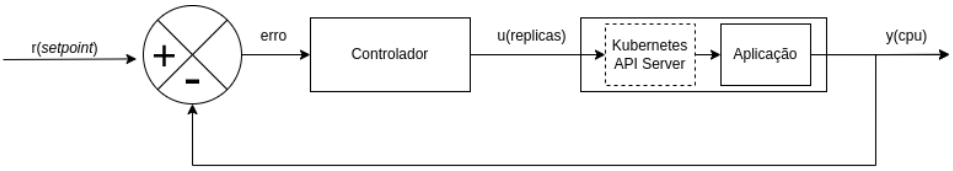


Figura 3.7: Ilustração da planta para os controladores

O *setpoint* do sistema para este trabalho foi estabelecido como o nível de uso de CPU desejado para a aplicação. Dessa forma, o objetivo dos controladores desenvolvidos é utilizar a diferença entre o *setpoint* e o uso atual de CPU para determinar a quantidade de réplicas que devem ser alocadas. Por fim, isso se traduzirá no monitoramento do uso de CPU após a alocação das réplicas, realimentando o sistema em malha fechada. A obtenção do número de réplicas e da utilização de CPU foi feita através do *Kubernetes Metrics Server*[6], que disponibiliza métricas para fluxos de escalabilidade, monitorando os recursos consumidos pelos *pods* da aplicação.

3.2.2 Construção do Modelo

Após a identificação da planta, alguns dos meios de obtenção dos parâmetros do controlador PID exigem a construção de um modelo matemático que expresse o comportamento da planta. Para isso, são conduzidos experimentos com a utilização de técnicas de modelagem estatística para construir um modelo que descreva apropriadamente a relação entre a entrada e saída do sistema.

Dada a própria natureza das ABMs, é necessário um estímulo na aplicação para que sejam consumidos recursos computacionais. No caso da *Online Boutique*, este estímulo é dado por meio de uma simulação de carga, onde usuários realizam requisições ao sistema. Deste modo,

[6] `https://github.com/kubernetes-sigs/metrics-server`

foi possível mapear a relação entre a utilização de CPU e a quantidade de réplicas requisitadas para diferentes usos de recurso. O microsserviço monitorado para esta etapa foi o *frontend*, cuja escolha será mais bem justificada no Capítulo 4 onde há também a definição do sistema avaliado.

Um desafio notado neste processo é a dificuldade de se estabelecer uma relação direta entre a utilização de CPU e a quantidade de réplicas solicitadas pelo controlador, pois a depender do cenário atual de quantidade de réplicas, serão necessárias cada vez mais para atender à demanda, crescendo de forma proporcional. Essa afirmação também fundamenta a escolha da proporção entre uso atual e alvo presentes no algoritmo do HPA, como visto na Equação 2.1.

Para superar este desafio, o mapeamento foi realizado entre o uso de CPU e a proporção em que as réplicas deveriam ser acrescidas. Isso permitiu um mapeamento mais direto e, por consequência, mais genérico aos diferentes cenários de utilização e escalabilidade em que a aplicação poderia se encontrar. Desse modo, a saída do controlador passa a ser uma proporção que, quando multiplicada ao número de réplicas, representará a quantidade de réplicas que devem ser acrescidas ou removidas da aplicação.

Os dados coletados nesta simuçação foram utilizados para a obtenção de um modelo da planta e sua respectiva função de transferência, que descreve a resposta do sistema a diferentes frequências de entrada, permitindo prever seu comportamento, estabilidade e desempenho. O método dos mínimos quadrados [22] foi utilizado para encontrar a função de transferência descrita na Equação 3.1, que será utilizada para a construção e tunagem de parâmetros do controlador PID.

$$TF(s) = \frac{2.092}{0.4364s + 1} \tag{3.1}$$

3.2.3 Controlador PID

Com a função de transferência definida, podemos partir para a tunagem dos parâmetros e implementação do controlador PID. Neste controlador, optamos por explorar as abordagens utilizando os controladores P, PI e PID. O controlador PD foi descartado, pois ele não elimina o erro em regime permanente, o que pode levar a problemas nos ajustes da quantidade de réplicas.

Para a tunagem dos parâmetros destes controladores foi utilizada a ferramenta MATLAB[7],

[7] `https://matlab.mathworks.com/`

por meio da qual instanciamos a planta vista na Figura 3.7 em conjunto com a função de transferência da Equação 3.1. A função *PID Tuner* foi então utilizada para gerar os parâmetros de cada um dos controladores, como visto na Tabela 3.2. Os parâmetros de tunagem utilizados foram os padrões da própria ferramenta, mas ela também disponibiliza diferentes configurações para propósitos como velocidade para alcançar o regime permanente e também robustez do controlador, onde cada uma gera um conjunto de parâmetros apropriados para a seleção em questão.

Tabela 3.2: Controladores e seus parâmetros

Controlador	K_p	K_i	K_d
P	1.0212757	-	-
PI	0.8905078	2.4297620	-
PID	0.8472042	2.7918027	0.0205105

3.2.4 Controlador AsTAR

Conforme visto no Capítulo 2, o controlador AsTAR foi originalmente desenvolvido para o contexto de dispositivos IoT, buscando fornecer uma abordagem adaptativa para a execução de tarefas conforme a energia disponível no sistema.

Para o contexto da escalabilidade, podemos fazer um paralelo: se tomarmos o algoritmo 1 como referência, podemos realizar adaptações nele para remodelá-lo como um controlador para a escalabilidade horizontal. Em vez de monitorar a tensão de um capacitor, podemos usar uma métrica de desempenho do sistema, tal como o uso de CPU ou memória, como sendo a voltagem observada. Nesta linha, a baixa voltagem pode ser interpretada como uma situação onde o consumo de recursos está abaixo de um *setpoint* predefinido, sugerindo que o sistema pode operar com menos réplicas. Já a alta voltagem indicaria uma situação onde o consumo de recursos está acima deste setpoint, sinalizando a necessidade de adicionar réplicas para lidar com a demanda.

O algoritmo 2 ilustra a adaptação do AsTAR para este contexto. Junto à adaptação para o contexto de escalabilidade, também foi adicionado um pequeno ajuste adicionando uma faixa de histerese, auxiliando consideravelmente a tomada de decisão do controlador quanto o uso de CPU estiver próximo do *setpoint*. Essa estratégia foi utilizada devido ao dinamismo e parâmetros que normalmente influenciam o sistema dinâmico das ABMs.

Algoritmo 2: Controlador AsTAR Modificado para o Contexto de Escalabilidade Horizontal

Dados: Uso de CPU (CPU), Setpoint (S), Réplicas (R), Faixa de Histerese (H)

Resultado: Quantidade de réplicas a serem alocadas

```
1  se CPU < (S - H) então
2      se o consumo de CPU estiver aumentando então
3          R ← R + 1;
4      senão
5          R ← R/2;
6  senão se CPU > (S + H) então
7      se o consumo de CPU estiver diminuindo então
8          R ← R − 1;
9      senão
10         R ← R × 2;
11 senão
12     Manter a quantidade de réplicas;
```

Deste modo, o que se tomava por frequência das tarefas no AsTAR pode ser refletido no número de réplicas de contêineres. Quando o sistema está sobrecarregado (alta voltagem), o número de réplicas pode ser aumentado, e quando a carga diminui (baixa voltagem), o número de réplicas pode ser reduzido. De forma análoga, quando o sistema está em uma margem aceitável (garantida pela faixa de histerese), o número de réplicas será mantido.

Tabela 3.3: Situações e Ações do Controlador AsTAR Modificado

Situação	**Ação**
CPU abaixo do limite inferior e consumo de CPU aumentando	Incrementa réplicas em 1
CPU abaixo do limite inferior e consumo de CPU não está aumentando	Reduz réplicas pela metade
CPU acima do limite superior e consumo de CPU diminuindo	Decrementa réplicas em 1
CPU acima do limite superior e consumo de CPU não está diminuindo	Dobra as réplicas
CPU dentro da faixa de histerese	Mantém as réplicas

Capítulo 4

Avaliação Experimental

Neste capítulo, inspirado pelo trabalho de *Jain* [23] na análise da performance de sistemas computacionais, serão relatadas as etapas seguidas para realizar a avaliação dos controladores descritos no Capítulo 3. Essas etapas incluem a definição do sistema, a escolha das métricas que conduzirão o processo de comparação de performance dos controladores, a descrição dos parâmetros e fatores do sistema que podem impactar a performance, a definição da carga ao qual o sistema será submetido e, por fim, a apresentação dos resultados da avaliação nos diferentes cenários descritos.

4.1 Definição do Sistema

A definição do sistema é um passo essencial na avaliação experimental, pois estabelece os limites, os componentes e as interações que serão analisadas, assegurando que a análise seja conduzida dentro de um escopo bem delimitado. Para o contexto deste trabalho, o alvo da nossa análise é a aplicação *Online Boutique*, ilustrada na Figura 3.2.

Para simplificação do estudo, o foco da análise será exclusivamente no microsserviço *frontend*, uma vez que ele atua como ponto central de entrada para as requisições dos usuários e desempenha um papel crítico na experiência do usuário. A escolha desse microsserviço justifica-se pelo fato de que sua demanda reflete diretamente a carga do sistema, tornando-o um componente estratégico para avaliar o impacto das estratégias de escalabilidade horizontal.

Essa abordagem também simplifica o experimento, proporcionando um maior controle sobre os fatores analisados e viabilizando uma comparação mais precisa entre os controladores. Isso ocorre porque, caso múltiplos microsserviços fossem considerados, cada um exigiria a construção

de um modelo específico, conforme descrito na Subseção 3.2.2, uma vez que suas respostas a um mesmo estímulo de carga poderiam variar significativamente.

4.2 Métricas Avaliadas

Como parte da avaliação de desempenho, é necessário entender o quão performático os controladores foram sob os diferentes cenários de teste. Para o caso deste trabalho, é necessário analisar como cada um dos controladores gerencia o consumo de CPU nos ambientes, onde ele precisa manter o uso deste recurso em um *setpoint* definido, ajustando dinamicamente o número de réplicas da aplicação. Como visto na Figura 3.4, a aplicação Prometheus será responsável por coletar os dados da aplicação, que serão posteriormente transformados nas métricas de avaliação.

Para a avaliação dos controladores, *Janert* [24] propõe a utilização das métricas RMSE (*Root Mean Squared Error*) e NRMSE (*Normalised Root Mean Squared Error*), que irão quantificar a precisão do ajuste do número de réplicas em relação ao *setpoint* definido, sendo que menores valores indicam uma melhor aderência do controlador ao comportamento esperado.

Assumindo n como o número total de amostras do uso de CPU, y_i como o uso de CPU na i-ésima observação, $\hat{y}$ como o *setpoint* e $\bar{y}$ como a média dos *setpoints*, é possível descrever a métrica RMSE e NRMSE como as Equação 4.1 e Equação 4.2, respectivamente.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y})^2}{n}} \tag{4.1}$$

$$NRMSE = \frac{RMSE}{\bar{y}} \tag{4.2}$$

No contexto da análise de performance dos controladores, *Rosa e Cavalcanti* [14] também consideram a métrica GR (*Goal Range*), que representa a porcentagem de medições que permaneceram dentro dos limites aceitáveis de variação, refletindo a estabilidade do controlador; valores mais elevados são desejáveis, pois indicam que o controlador conseguiu manter o uso de recursos dentro da faixa estipulada. Para este experimento, esta faixa foi considerada como sendo dentro do intervalo de 80% a 120% do *setpoint*. A Equação 4.3 ilustra essa métrica, onde N é a quantidade de amostragens do uso de CPU y_i tal que $y_i \in [0.8 \cdot \hat{y}, 1.2 \cdot \hat{y}]$.

$$GR = \frac{N}{n} \times 100 \tag{4.3}$$

No contexto da aplicação, *Nguyen et al.*[15] e *Khaleq e Ra* [16] apresentam em seus trabalhos as análises de métricas como RTT e quantidade média de réplicas alocadas. O RTT (ilustrado na Figura 4.1) é a métrica que mede o tempo total para uma requisição percorrer o sistema incluindo atrasos de rede e tempo de processamento, logo sendo desejável que essa métrica permaneça reduzida para minimizar a latência percebida pelos usuários. Já a quantidade média de réplicas alocadas pode trazer também um indicativo do desempenho do controlador, já que menos réplicas podem significar menor alocação de recursos para executar a mesma carga contanto que a utilização destes fique em uma margem aceitável.

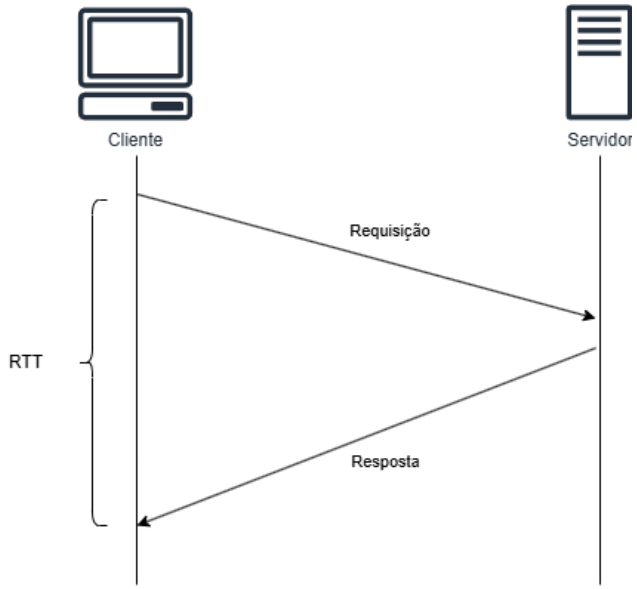


Figura 4.1: Ilustração da métrica RTT na interação entre Cliente e Servidor

4.3 Parâmetros e Fatores do Sistema

Identificar os parâmetros e fatores de um sistema é essencial na análise de desempenho, pois permite compreender quais variáveis influenciam diretamente o comportamento do sistema e como elas interagem entre si. Parâmetros são valores fixos que estabelecem as condições de um sistema, já os fatores são variáveis que podem ser ajustadas ou modificadas para entender seu impacto no desempenho.

Os parâmetros podem ser divididos também em duas categorias: parâmetros do sistema (P_s) e parâmetros de carga (P_c). Os parâmetros do sistema se referem a condições de um sistema,

como alocação de recursos. Já os parâmetros de carga são constantes configuradas na geração de carga, como o tempo entre chamadas ao sistema.

Nos parâmetros do sistema é possível citar o tipo de máquina utilizada como sendo instâncias do tipo *c7g.xlarge* da AWS. As configurações de recursos para os *pods* da aplicação alvo também foram mantidos de acordo com a definição original da aplicação. O *Istio* foi utilizado como intermediador das chamadas internas do serviço, se tornando também um parâmetro que pode influenciar o desempenho já que todo o tráfego de entrada e saída é interceptado por ele, além também de ocupar recurso em cada *pod* já que é utilizado como um *sidecar container*. O intervalo entre avaliações dos controladores foi configurado para 15 segundos, que define a frequência com que a métrica de CPU será avaliada sob o *setpoint*. Por sua vez, a janela de estabilização dos controladores, que mantém a maior quantidade de réplicas definida em um intervalo de tempo, foi fixada também em 15 segundos.

Para os parâmetros de carga foi fixado o intervalo entre requisições como sendo de um segundo, a duração dos testes configurada como trinta minutos, com um novo usuário sendo criado a cada 5 segundos. Essa configuração foi tomada objetivando garantir que a carga gerada durante um teste fosse aplicada de maneira gradual e controlada, evitando impactos abruptos no sistema. Isso também possibilita a observação da resposta do sistema à medida que a carga aumenta progressivamente, permitindo a avaliação da eficácia dos controladores para se adaptar à nova demanda. A Tabela 4.1 sintetiza os parâmetros de carga e do sistema.

Tabela 4.1: Parâmetros do sistema (P_s) e de carga (P_c)

Categoria	**Parâmetro**	**Descrição**
P_s	Tipo de máquina	Instâncias AWS do tipo *c7g.xlarge*
P_s	Configuração dos Pods	Mantida conforme definição original da aplicação (Tabela 3.1)
P_s	Istio	Utilizado como *sidecar container*, interceptando tráfego interno e consumindo recursos
P_s	Intervalo de Avaliação	15 segundos
P_s	Janela de Estabilização	15 segundos
P_c	Intervalo entre requisições	1 segundo
P_c	Duração dos testes	30 minutos
P_c	Criação de novos usuários	1 novo usuário a cada 5 segundos

Os fatores que guiam o experimento foram definidos de forma a possibilitar a avaliação dos controladores em diferentes cenários de *setpoints* e cargas, permitindo analisar o comportamento de cada controlador frente a distintos níveis de utilização de recursos. Serão avaliados os controladores HPA, AsTAR, P, PI e PID, considerando *setpoints* configurados para 30%, 50% e 70% de utilização da CPU, com cargas compostas por 10, 30 e 50 usuários em cada um desses cenários. A escolha dessa faixa de *setpoints* visa contemplar uma gama representativa de condições operacionais, reconhecendo que a definição adequada de *setpoints* para aplicações é, por si só, uma tarefa complexa e desafiadora [25]. A Tabela 4.2 sintetiza os fatores e os valores utilizados neste experimento, que totalizam 45 cenários a serem avaliados, com duração de 30 minutos cada.

Tabela 4.2: Fatores utilizados no experimento

Fator	**Valores**
Controladores	HPA, AsTAR, P, PI, PID
Setpoints	30%, 50%, 70% de utilização da CPU
Número de usuários	10, 30, 50 usuários

4.4 Simulações de Carga

Como parte do processo de avaliação dos controladores, são necessários testes sistemáticos que simulem cenários realistas de carga. Para isso, é essencial utilizar aplicações representativas e ferramentas robustas que permitam a geração de tráfego.

Com isso em mente, as simulações de carga foram realizadas utilizado o *Locust*[1], uma ferramenta de teste de carga que permite simular múltiplos usuários concorrentes e avaliar o desempenho de sistemas sob diferentes níveis de tráfego. Por meio dele é possível também configurar cenários personalizados como padrões de acesso, número de requisições por segundo e distribuição da carga entre diferentes rotas da aplicação, além de também disponibilizar a métrica do RTT que irá compor parte da avaliação dos controladores.

Para os fins deste trabalho, foi definido um cenário de consumo onde o usuário acessa apenas a página inicial da aplicação. Esse comportamento foi escolhido dentre os demais pois ele demonstrou ser o que mais estressava a aplicação *frontend*, já que a chamada a página inicial

[1] `https://locust.io/`

estimula a interação com outros microsserviços como o *currency*, *recommendation* e *ad* para o carregamento dos componentes da página.

4.5 Resultados da Avaliação

Após a definição dos parâmetros do sistema, dos fatores de análise e das ferramentas empregadas, os testes foram realizados na aplicação com o objetivo de coletar as métricas necessárias para a etapa de avaliação. Os resultados obtidos foram organizados por cenários de carga, sendo apresentada, para cada um deles, uma tabela contendo os valores das métricas estabelecidas na Seção 4.2 para cada controlador e seus respectivos *setpoints*. Em seguida, para cada cenário, os resultados são discutidos com o apoio de representações gráficas que ilustram o comportamento dos controladores ao longo do tempo, evidenciando suas tomadas de decisão e a utilização de recursos.

4.5.1 Testes com 10 usuários

Neste cenário, a aplicação foi submetida a uma carga constante de 10 usuários simultâneos. O objetivo é avaliar o comportamento dos controladores sob uma demanda leve, observando sua capacidade de manter a estabilidade e eficiência do sistema. A Tabela 4.3 apresenta os resultados obtidos para cada controlador, acompanhada de discussões sobre cada um dos cenários.

Para o *setpoint* de 30%, observa-se que os controladores PI, PID e HPA apresentaram um bom desempenho, evidenciado pelos baixos valores de erro e elevados GR, mantendo, em média, cerca de duas réplicas alocadas. Além disso, esses controladores conseguiram manter o RTT em níveis satisfatórios, com destaque para o HPA e PI, que apresentaram os menores tempos. O controlador AsTAR demonstrou um comportamento mais agressivo ao aumentar o número médio de réplicas alocadas, o que resultou em um incremento no erro e uma redução no GR; contudo, esse comportamento também refletiu em um RTT mais baixo, o menor entre os controladores. Por outro lado, o controlador P mostrou-se incapaz de se adaptar ao cenário, mantendo apenas uma réplica alocada, o que resultou no maior erro registrado, no menor GR e em um RTT superior aos demais.

Para o *setpoint* de 50%, com exceção do AsTAR, os controladores mantiveram-se alocados, em média, com uma réplica. Neste cenário, mesmo com um número próximo de réplicas, existem

Tabela 4.3: Comparação dos controladores sob diferentes *setpoints* para 10 usuários

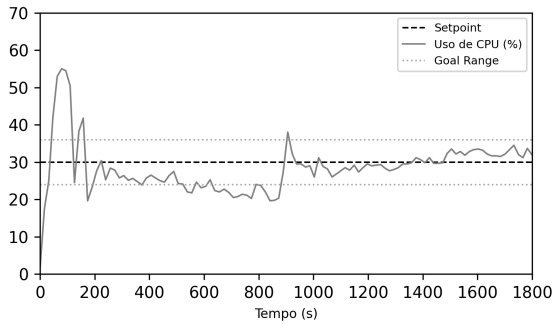
Setpoint	Controlador	RMSE	NRMSE	GR (%)	RTT (ms)	Média de Réplicas
30%	P	11.127	0.371	23.077	33.225	1.000
	PI	6.796	0.227	81.197	28.289	1.949
	PID	6.701	0.223	85.345	32.651	1.948
	AsTAR	7.083	0.236	73.043	27.634	3.191
	HPA	5.435	0.181	81.579	29.431	1.965
50%	P	10.531	0.211	61.667	37.000	1.000
	PI	9.880	0.198	70.339	40.345	1.000
	PID	8.577	0.172	76.667	36.952	1.000
	AsTAR	14.945	0.299	54.237	27.498	1.831
	HPA	10.065	0.201	64.103	39.304	1.026
70%	P	28.035	0.401	0.000	29.621	1.000
	PI	29.866	0.427	0.000	29.124	1.000
	PID	26.008	0.372	1.681	26.212	1.000
	AsTAR	44.202	0.631	0.000	27.952	2.879
	HPA	28.050	0.401	1.709	29.442	1.000

flutuações nos resultados obtidos, o que elucida os parâmetros do sistema que podem interferir no processo de avaliação. O controlador HPA em determinado momento também aumentou seu número de réplicas, o que contribuiu para a sua distanciação dos controladores P, PI e PID. É possível notar também que neste cenário a agressividade de alocação do AsTAR foi punida, pois foi o controlador que mais se distanciou nas métricas observadas, apesar desse comportamento mais uma vez ter sido refletido em um RTT mais baixo.

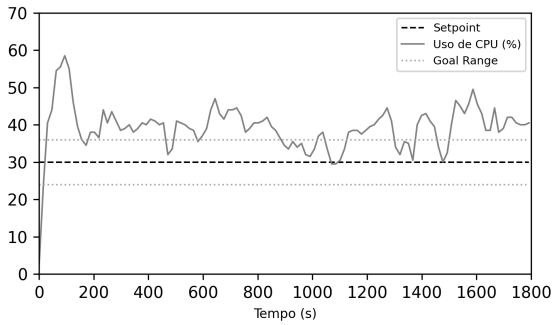
Para o *setpoint* de 70% é possível notar que todos os controladores não conseguiram performar. Este cenário é particularmente desafiador porque com uma carga leve, assim como possível visualizar no cenário de 50% de *setpoint*, a aplicação já estava com dificuldades para se manter próximo ao *setpoint* mesmo com apenas uma réplica. Neste cenário, o AsTAR continou tentando realizar ajustes em suas réplicas, mas sem sucesso, indicando um ponto de melhoria que pode ser considerado no algoritmo do controlador. Já os outros controladores se mantiveram estáveis em uma réplica, já que não precisaram escalar por estarem distantes do *setpoint*, como evidenciado pelos erros obtidos.

De modo geral, os controladores PID, PI e HPA apresentaram desempenho consistente ao longo dos diferentes *setpoints*, com baixos erros, altos GR e número similar de réplicas. O

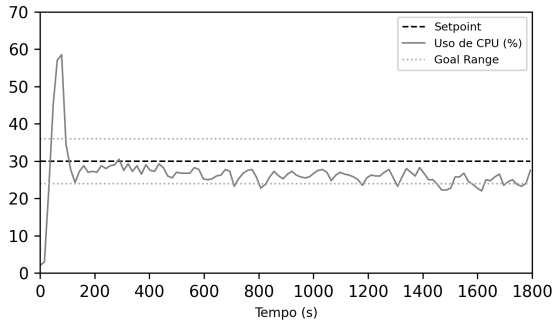
HPA destacou-se pela eficiência na adaptação à demanda, mantendo estabilidade nas réplicas e RTTs satisfatórios. O AsTAR, por sua vez, adotou uma estratégia mais agressiva, alocando mais réplicas e obtendo os menores RTTs, mas com maior erro e menor GR, especialmente em cargas elevadas. Já o controlador P apresentou desempenho inferior, incapaz de ajustar o número de réplicas, resultando nos maiores erros, menores GRs e RTTs superiores. A Figura 4.2 ilustra o cenário de 30% de *setpoint*, onde houve um maior esforço dos controladores para se adaptar à carga.



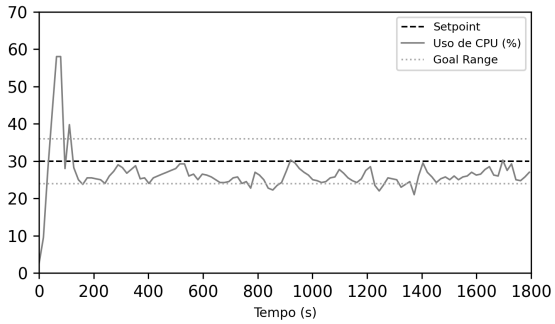
(a) Controlador AsTAR



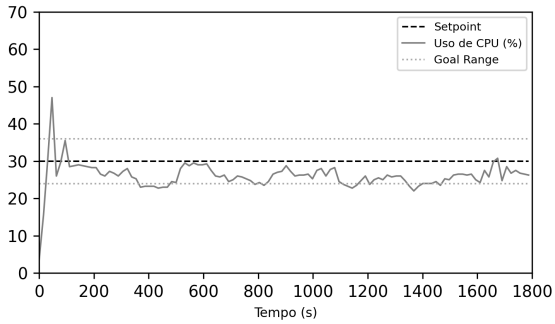
(b) Controlador P



(c) Controlador PI



(d) Controlador PID

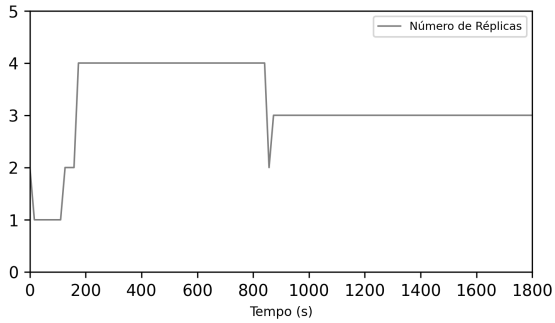


(e) Controlador HPA

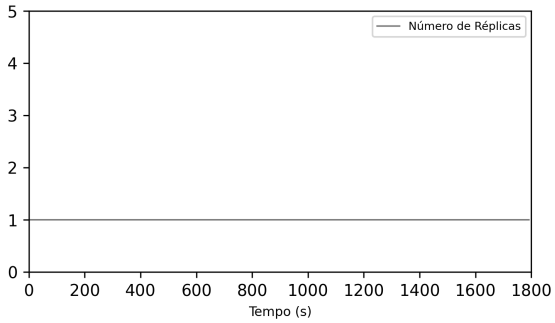
Figura 4.2: Acompanhamento do uso de CPU para o *setpoint* de 30% e 10 usuários

É possível notar a forma com que os controladores se adaptaram e também o seu tempo de resposta à carga. O controlador HPA reagiu de forma mais antecipada do que os controladores

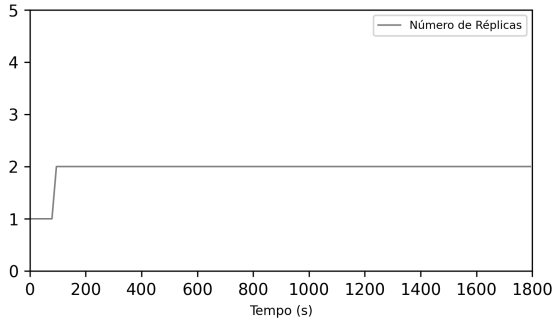
PI e PID, se adaptando preemptivamente e não deixando a utilização de CPU passar dos 50%. O controlador AsTAR flutuou bastante até alcançar um ponto de equilíbrio, por volta da metade da duração do experimento. O controlador P, como visto nos resultados, não conseguiu se adaptar e manteve sua única réplica, que ficou boa parte do tempo do experimento fora da zona do *Goal Range*. A Figura 4.3 ilustra a variação do número de réplicas dos controladores, onde é possível confirmar a instabilidade do controlador AsTAR, estabilidade dos controladores HPA, PI e PID e incapacidade de escalabilidade do controlador P.



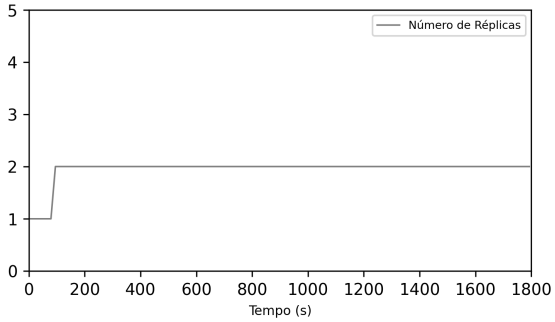
(a) Controlador AsTAR



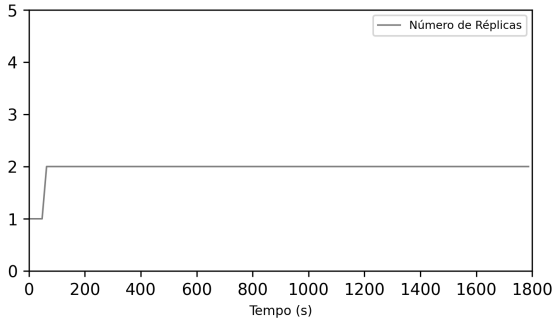
(b) Controlador P



(c) Controlador PI



(d) Controlador PID



(e) Controlador HPA

Figura 4.3: Réplicas alocadas no decorrer do tempo para o *setpoint* de 30% e 10 usuários

4.5.2 Testes com 30 usuários

Neste cenário, a aplicação foi submetida a uma carga constante de 30 usuários simultâneos. O objetivo é avaliar o comportamento dos controladores sob uma demanda média, observando sua capacidade de manter a estabilidade e eficiência do sistema. A Tabela 4.4 apresenta os resultados obtidos para cada controlador, acompanhada de discussões sobre cada um dos cenários.

Tabela 4.4: Comparação dos controladores sob diferentes *setpoints* para 30 usuários

Setpoint	Controlador	RMSE	NRMSE	GR (%)	RTT (ms)	Média de Réplicas
30%	P	15.505	0.517	74.528	30.895	6.406
	PI	12.422	0.414	79.439	29.026	5.710
	PID	12.715	0.424	76.190	29.105	6.143
	AsTAR	8.168	0.272	36.275	29.998	6.392
	HPA	10.680	0.356	86.792	30.779	6.462
50%	P	15.412	0.308	56.364	43.502	2.473
	PI	11.020	0.220	70.536	30.461	3.098
	PID	12.492	0.250	73.636	30.558	3.491
	AsTAR	11.990	0.240	71.429	48.350	3.366
	HPA	8.068	0.161	90.991	28.797	3.685
70%	P	23.918	0.342	67.826	32.076	2.270
	PI	15.702	0.224	76.106	26.615	2.442
	PID	14.139	0.202	92.105	37.918	1.912
	AsTAR	24.283	0.347	40.708	44.886	2.814
	HPA	14.863	0.212	85.965	31.817	2.728

Para o *setpoint* de 30%, diferente do visto no teste com 10 usuários, foi possível notar que o controlador P conseguiu se adaptar à carga. Os controladores P, PI, PID e HPA obtiveram resultados semelhantes, mas com destaque para o HPA que conseguiu os menores erros e também o maior GR. Apesar disso o controlador PI apresentou o segundo melhor resultado nestas métricas, mas com um número menor de réplicas utilizadas. Mais uma vez, o controlador AsTAR, apesar de um baixo erro, teve dificuldades em se adaptar à carga, demonstrada pelo baixo GR. Neste cenário, todos os controladores apresentaram a métrica RTT semelhante, com o controlador PI detendo o menor dos valores.

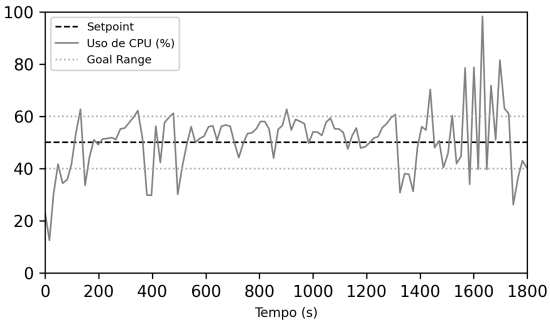
Para o *setpoint* de 50%, o controlador P teve dificuldades para se adaptar, evidenciado pelo maior dos erros e também o menor GR dentre os controladores. Os controladores PI,

PID e AsTAR obtiveram resultados semelhantes, com o PI tendo destaque pelo menor RTT e quantidade de réplicas dentre estes. O controlador HPA mais uma vez obteve excelentes resultados, com o maior GR e menor RTT dentre todos os controladores, mas para isso alocando um maior número médio de réplicas.

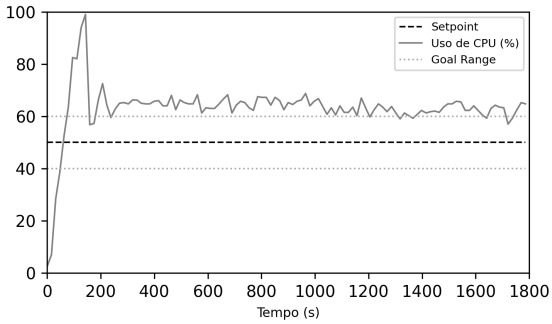
Para o *setpoint* de 70%, o controlador AsTAR novamente não conseguiu se adaptar bem, ficando com o maior dos erros e a menor GR dentre todos os controladores. O controlador PID se destacou neste cenário por ter apresentado os menores erros, maior GR e menor quantidade de réplicas dentre os controladores, mas apresentou o segundo maior RTT. O controlador P teve o segundo menor erro e GR, mas conseguiu melhores números que o controlador AsTAR, se adaptando um pouco melhor ao cenário. Por sua vez, o PI apresentou resultados medianos, ficando com o melhor RTT, o terceiro melhor dos erros, GR e quantidade média de réplicas.

De forma geral, mais uma vez os controladores PI, PID e HPA mantiveram um desempenho consistente no decorrer dos diferentes *setpoints* utilizados para este cenário de carga, apresentando baixos valores de erro, elevados GR e uma alocação semelhante de réplicas, com destaque para o HPA nas métricas de erro e GR e para os controladores PI e PID no número de réplicas alocadas. O controlador AsTAR, apesar de ter conseguido se adaptar no *setpoint* de 50%, mais uma vez apresentou um comportamento instável nos outros cenários. Já o controlador P, apesar das dificuldades apresentadas principalmente nos *setpoints* de 50% e 70%, conseguiu ser mais estável nos cenários do que o AsTAR. A Figura 4.4 ilustra a utilização de CPU da aplicação no cenário de 50% de *setpoint*, onde é possível expor os *trade-offs* entre eficiência, estabilidade e consumo de recursos dos controladores.

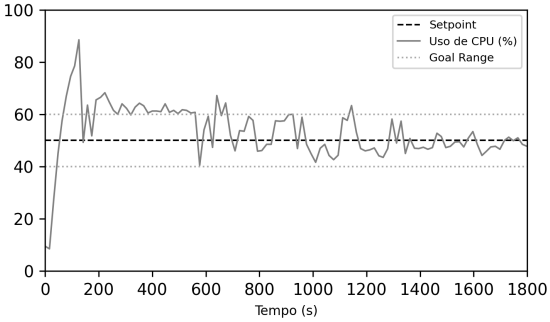
Neste *setpoint*, o controlador AsTAR apresentou um comportamento estabilizado, mas logo quando saiu da faixa de histerese definida apresentou variações e não voltou a se estabilizar no fim da duração do teste, evidenciando o quanto pode ser instável às variações de carga ou do sistema. O controlador P, apesar de ter escalado, não conseguiu se adaptar o suficiente, ficando com uma utilização de recursos acima do esperado. Os controladores PI e PID se adaptaram, mas com uma alta variação, apesar de dentro do *Goal Range*, da utilização de CPU. O HPA por sua vez apresentou a maior das estabilidades, se mantendo dentro da margem aceitável por quase toda a duração do teste. A Figura 4.5 ilustra a variação do número de réplicas dos controladores, evidenciando as flutuações dos controladores AsTAR, PI e PID, seguido da incapacidade de adaptação do controlador P e da estabilidade do controlador HPA, que teve apenas uma leve flutuação e depois voltou ao número de réplicas anterior.



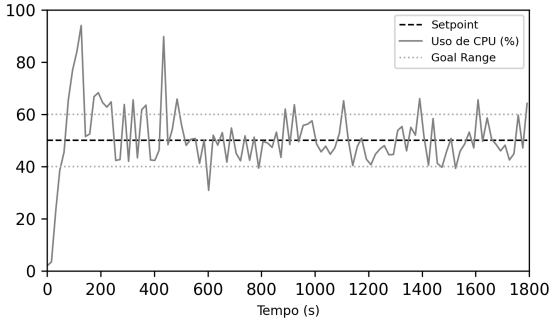
(a) Controlador AsTAR



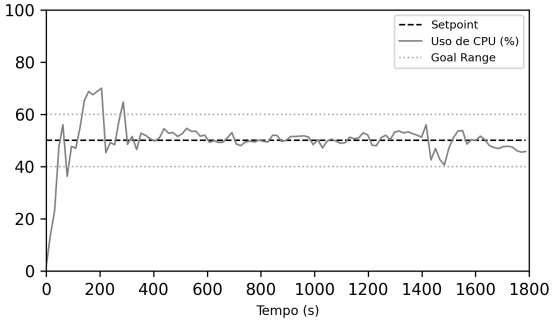
(b) Controlador P



(c) Controlador PI



(d) Controlador PID



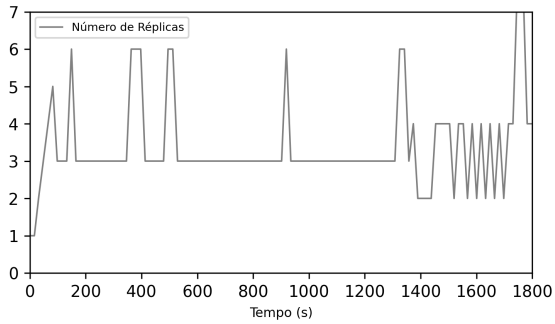
(e) Controlador HPA

Figura 4.4: Acompanhamento do uso de CPU para o *setpoint* de 50% e 30 usuários

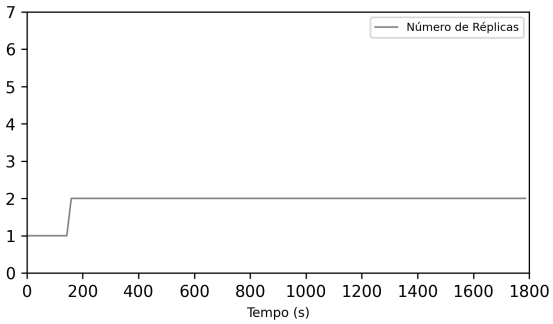
4.5.3 Testes com 50 usuários

Neste cenário, a aplicação foi submetida a uma carga constante de 50 usuários simultâneos. O objetivo é avaliar o comportamento dos controladores sob uma demanda alta, observando sua capacidade de manter a estabilidade e eficiência do sistema. A Tabela 4.5 apresenta os resultados obtidos para cada controlador, acompanhada de discussões sobre cada um dos cenários.

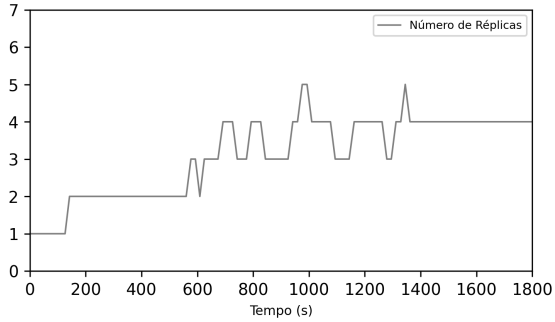
No *setpoint* de 30%, o controlador HPA obteve os menores valores de erro e também apresentou o maior GR, indicando um desempenho mais preciso na manutenção da métrica dentro dos limites desejados. O controlador AsTAR por sua vez apresentou um GR consideravelmente



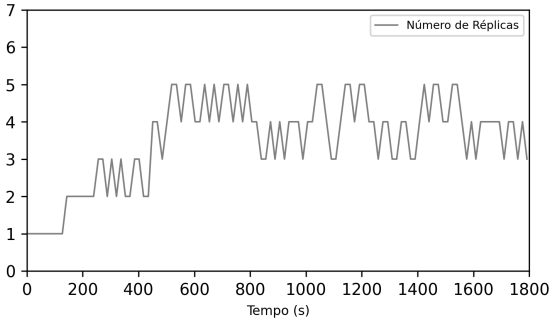
(a) Controlador AsTAR



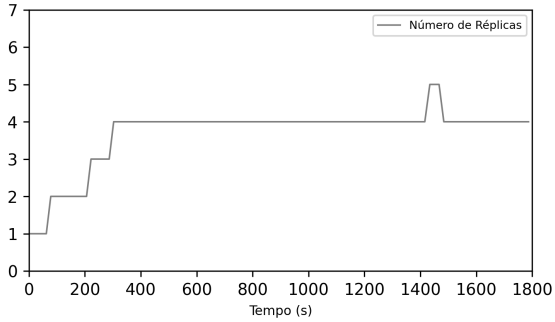
(b) Controlador P



(c) Controlador PI



(d) Controlador PID



(e) Controlador HPA

Figura 4.5: Réplicas alocadas no decorrer do tempo para o *setpoint* de 50% e 30 usuários

inferior, tendo dificuldades em manter a métrica dentro do intervalo esperado. Em termos da métrica RTT, o controlador PID apresentou o menor valor, enquanto o PI obteve o maior dos tempos, mas ainda próximo de controladores como o P e o AsTAR. O HPA obteve também o maior número na média de réplicas, sendo um indicativo de que necessitou de um maior número de réplicas para atingir sua estabilidade.

Para o *setpoint* de 50%, os controladores P, PI, PID e HPA demonstraram um GR superior a 85%, com destaque para o HPA. Novamente, o AsTAR apresentou um GR inferior, o que indica limitações no ajuste. Neste cenário, o menor RTT foi alcançado pelo controlador PI e o maior pelo controlador AsTAR, mas com os controladores P, PID e HPA relativamente

Tabela 4.5: Comparação dos controladores sob diferentes *setpoints* para 50 usuários

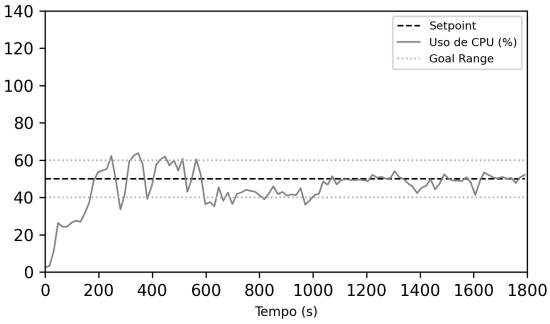
Setpoint	Controlador	RMSE	NRMSE	GR (%)	RTT (ms)	Média de Réplicas
30%	P	9.974	0.332	82.105	24.773	8.663
	PI	8.150	0.272	80.208	25.701	8.865
	PID	11.609	0.387	84.375	21.291	9.135
	AsTAR	8.762	0.292	17.347	25.544	7.959
	HPA	6.749	0.225	87.368	22.199	9.463
50%	P	13.492	0.270	85.981	24.823	4.561
	PI	15.633	0.313	86.916	22.361	4.551
	PID	12.932	0.259	85.047	23.952	4.692
	AsTAR	11.543	0.231	74.312	26.689	5.266
	HPA	9.736	0.195	86.111	24.058	5.083
70%	P	14.084	0.201	90.909	25.067	2.836
	PI	14.634	0.209	86.364	27.299	2.818
	PID	14.761	0.211	83.495	28.734	2.786
	AsTAR	22.924	0.327	43.396	24.386	4.689
	HPA	12.598	0.180	83.486	22.766	3.624

próximos. A média de réplicas manteve-se relativamente homogênea, sugerindo que todos os controladores conseguiram operar dentro de um intervalo de alocação semelhante dos recursos computacionais.

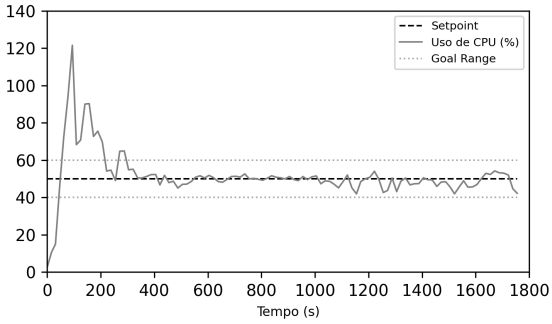
Já no *setpoint* de 70%, o controlador HPA seguiu com os menores erros, mas foi possível notar uma disparidade principalmente na métrica GR, com destaque para o controlador P, que conseguiu se manter acima de 90%. Ainda no GR, o controlador AsTAR teve novamente o menor dos resultados, consolidando sua dificuldade no ajuste para atender ao alvo de utilização. Em termos de RTT, o HPA teve o menor dos tempos, com o PID tendo o pior destes. É valido lembrar que nesse cenário com um alto valor de *setpoint* é esperado que haja um estresse na aplicação, levando a maiores tempos de latência. Já para a média de réplicas, os controladores P, PI e PID ficaram próximos, com o AsTAR tendo a maior alocação de réplicas, seguido do HPA.

No geral, os resultados indicam que neste cenário de carga o controlador HPA apresenta um desempenho consistente em termos de precisão, com um menor erro (RMSE e NRMSE) e maior GR, mas com um maior número de réplicas alocadas, o que pode implicar em um consumo adicional de recursos computacionais. O controlador AsTAR, por sua vez, apresentou

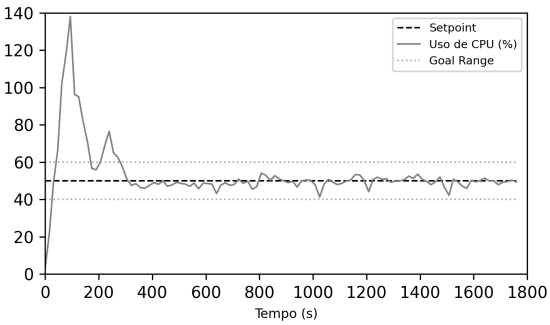
um desempenho inferior na métrica GR e nos erros, indicando dificuldades em lidar com a estabilização das réplicas da aplicação. Os controladores P, PI e PID apresentaram desempenhos intermediários, mas sempre com uma menor média de réplicas alocadas se comparados ao HPA e com eventuais destaques em alguns cenários. A Figura 4.6 ilustra o monitoramento do uso de CPU sob o tempo para cada um dos controladores no cenário onde o *setpoint* foi fixado em 50%.



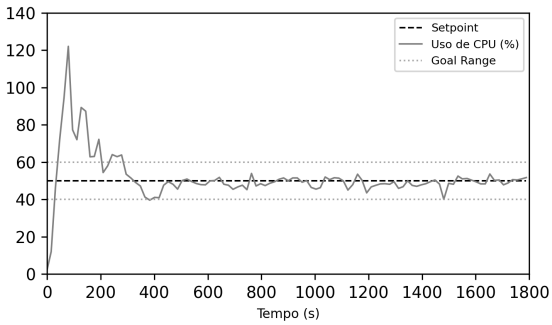
(a) Controlador AsTAR



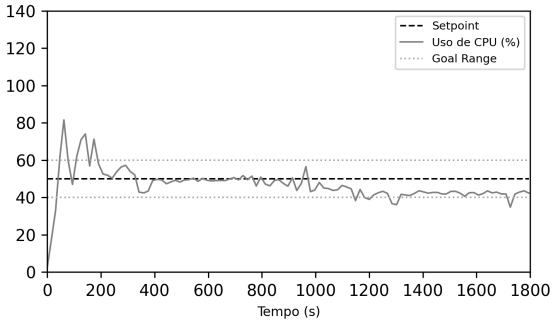
(b) Controlador P



(c) Controlador PI



(d) Controlador PID



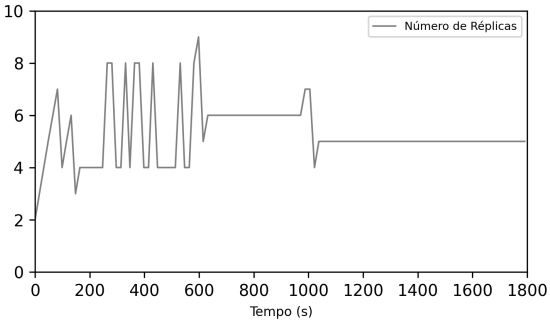
(e) Controlador HPA

Figura 4.6: Acompanhamento do uso de CPU para o *setpoint* de 50% e 50 usuários

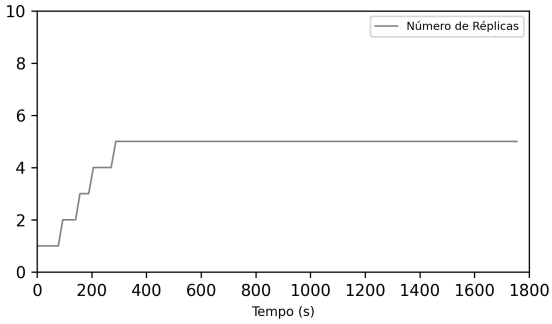
É possível notar que neste cenário, assim como mostrado nos resultados, os controladores se comportaram bem. Os controladores P, PI e PID apresentaram um *overshoot* nos primeiros segundos, mas se estabilizaram pouco tempo depois, ficando inclusive mais próximos ao setpoint

ao longo do tempo do que o controlador HPA. Este cenário foi positivo para o controlador AsTAR principalmente por ele ter conseguindo alcançar um número de réplicas em que ele se estabilizou, reforçando a importância da modificação apresentada na Subseção 3.2.4.

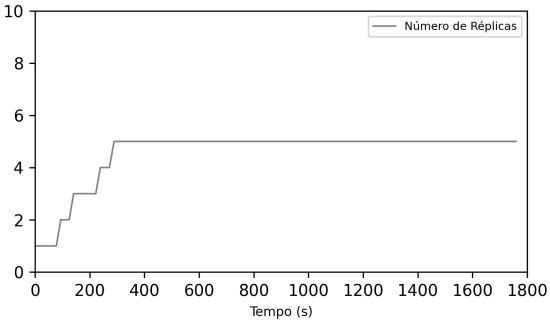
A Figura 4.7 ilustra o número de réplicas alocadas pelos controladores no decorrer do tempo. É possível notar a instabilidade do número de réplicas alocadas pelo controlador AsTAR nos primeiros segundos até atingir, conforme visto na Figura 4.6a, um ponto de estabilidade a partir dos 600 segundos de experimento. Outro ponto importante é que todos os controladores conseguiram se estabilizar nas 5 réplicas no decorrer do experimento, com exceção do controlador HPA, que se manteve com 6 réplicas.



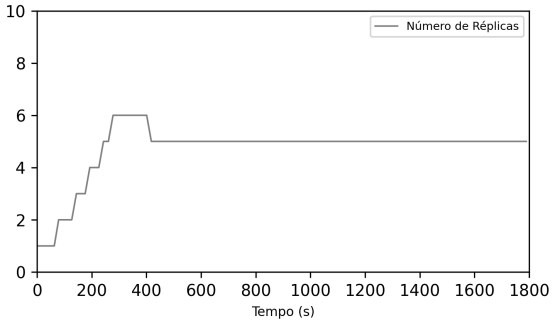
(a) Controlador AsTAR



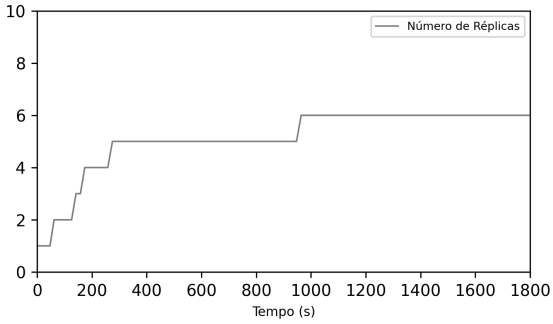
(b) Controlador P



(c) Controlador PI



(d) Controlador PID



(e) Controlador HPA

Figura 4.7: Réplicas alocadas no decorrer do tempo para o *setpoint* de 50% e 50 usuários

4.5.4 Testes com *setpoint* variável

Um controlador tem que ser capaz de se adaptar ao dinamismo do ambiente. No cenário de escalabilidade horizontal, esta adaptação é intrínseca ao *setpoint* que é configurado, e normalmente este valor é fixado e sofre poucas alterações com o tempo devido a dificuldade para definí-lo [25]. Apesar disso, existe uma tendência de novas propostas que buscam ajustar esse *setpoint* dinamicamente para as aplicações [25, 26], permitindo uma adaptação das aplicações de acordo com a sua evolução de utilização de carga.

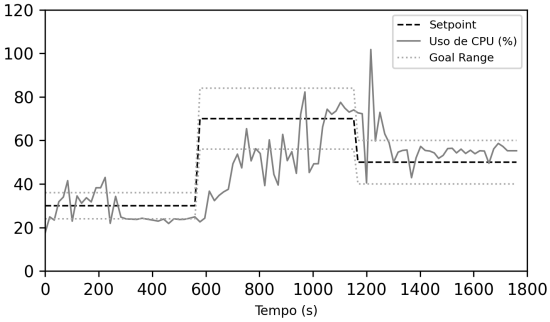
Com esse objetivo, uma segunda leva de testes foi realizada variando dinamicamente o *setpoint* durante a duração da sessão de teste com 50 usuários. Esse cenário de usuários foi escolhido porque foi o que mais demonstrou variação no número de réplicas, tornando as diferenças de cada *setpoint* mais evidentes. O teste foi inicializado com um *setpoint* de 30% nos primeiros 10 minutos, aumentado para 70% nos 10 minutos seguintes e, por fim, ajustado para 50% nos últimos 10 minutos. Essa configuração foi escolhida pois representa variações consideráveis no *setpoint*, que se refletirão em mudanças também consideráveis na quantidade de réplicas alocadas. A Tabela 4.6 apresenta os resultados das métricas para este cenário de teste, onde os melhores valores para cada uma das métricas foram destacados.

Tabela 4.6: Resultado dos Controladores para *Setpoint* Variável

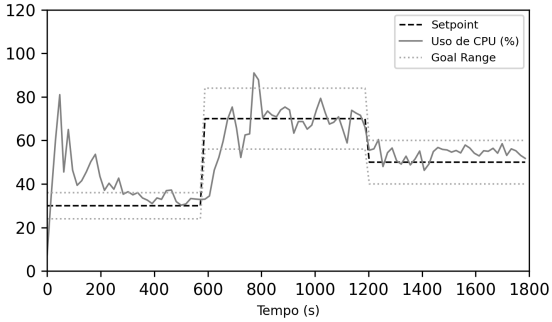
Controlador	RMSE	NRMSE	GR (%)	RTT (ms)	Média de Réplicas
P	11.596	0.229	74.528	55.298	**4.170**
PI	10.617	0.212	**80.000**	**47.106**	4.829
PID	13.274	0.264	78.505	48.918	4.308
AsTAR	15.625	0.308	56.863	48.124	5.578
HPA	**8.917**	**0.177**	79.808	56.250	5.010

Neste cenário, o HPA continuou se destacando com o menor erro e um GR próximo ao controlador PI. O controlador PI por sua vez apresentou a menor das latências, e, dentre os controladores propostos, o menor erro. Também é possível notar que o controlador AsTAR continuou com o maior erro e uma baixa GR, indicando que também teve problemas para se estabilizar neste cenário mais dinâmico. A Figura 4.8 ilustra o resultado dos controladores neste cenário.

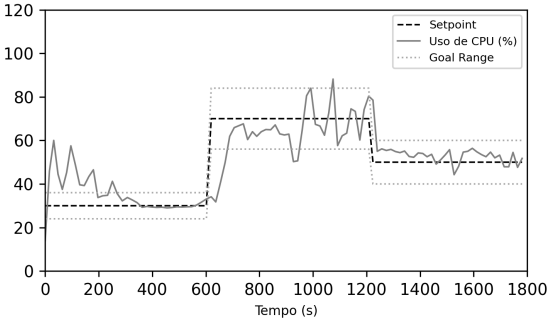
É possível comprovar as flutuações no controlador AsTAR, com grandes dificuldades para se adaptar às variações em quase todos os cenários, com melhores resultados aos 50% de *setpoint*. Os controladores P, PI e PID obtiveram resultados semelhantes, com o controlador PID



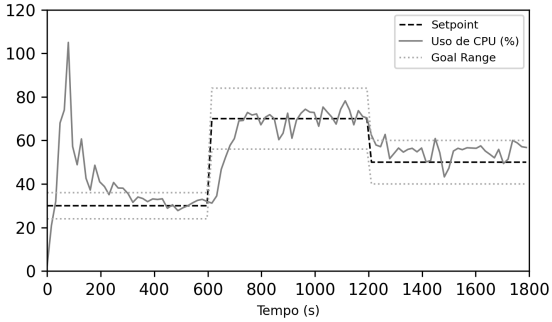
(a) Controlador AsTAR



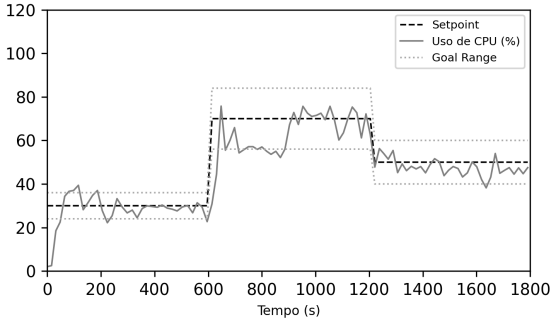
(b) Controlador P



(c) Controlador PI



(d) Controlador PID

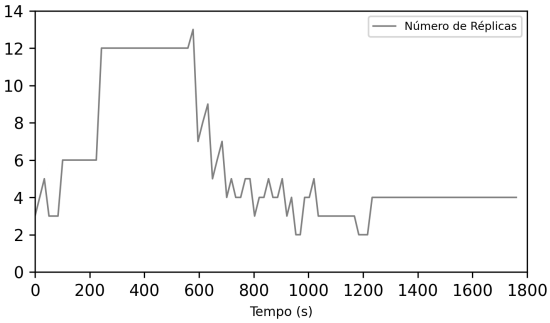


(e) Controlador HPA

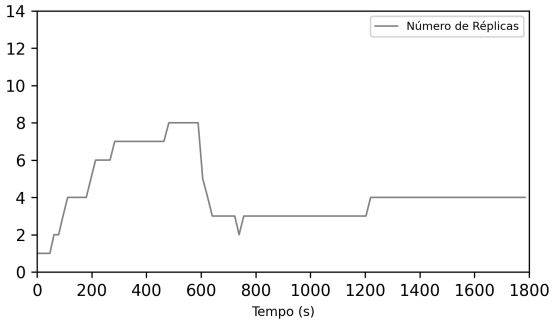
Figura 4.8: Resultado dos controladores para o cenário de *setpoint* variável

aparentando ser o mais estável dentre estes quando o *setpoint* foi configurado em 70%. Já o controlador HPA se mostrou estável na maior parte do tempo, tendo apenas uma pequena dificuldade quando o *setpoint* foi configurado em 70%.

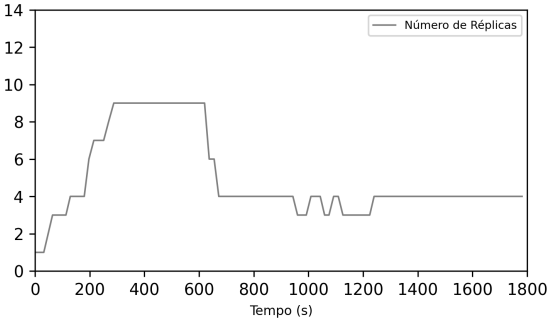
Por fim, a Figura 4.9 ilustra as réplicas alocadas no decorrer do tempo para este cenário. É possível notar que as flutuações vistas no controlador AsTAR se dão, de fato, pela incapacidade de se estabilizar as réplicas. Já para os controladores P e PI, a instabilidade notada nos 70% de setpoint pode ser explicada pelas flutuações no número de réplicas, que não acontecem no controlador PID. O controlador PID também aparentou ser mais estável e alocar uma menor quantidade de réplicas se comparado ao controlador HPA.



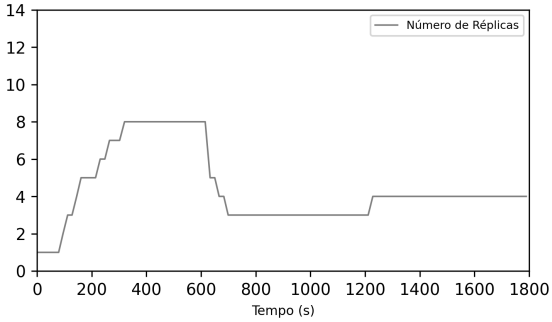
(a) Controlador AsTAR



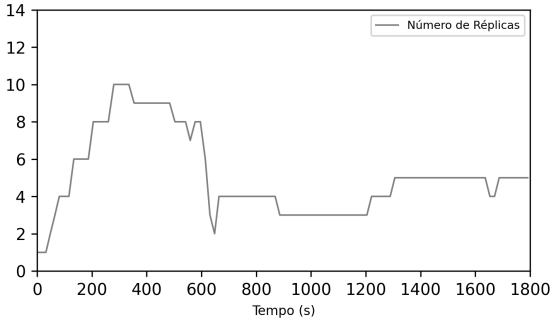
(b) Controlador P



(c) Controlador PI



(d) Controlador PID



(e) Controlador HPA

Figura 4.9: Réplicas alocadas no decorrer do tempo para o cenário de *setpoint* variável

Capítulo 5

Conclusão e trabalhos futuros

Neste trabalho foi descrito o processo de desenvolvimento, implantação e avaliação de controladores alternativos no contexto da escalabilidade horizontal no Kubernetes, realizando comparações com a tradicional abordagem do HPA sobre métricas como RMSE, NMRSE, *Goal Range*, RTT e número de réplicas alocadas para diferentes cenários de uso de recursos.

A partir dos experimentos conduzidos, foi possível comprovar a eficácia do HPA para os mais diversos cenários. Entretanto, ele continua sendo uma abordagem generalista e nem sempre pode ser a abordagem mais efetiva principalmente levando-se em consideração a alocação de recursos. O controlador PID, por exemplo, apesar de ainda distantes do HPA em algumas métricas analisadas, conseguiu reagir de forma estável aos cenários em que foi aplicado e manteve um uso inferior de réplicas alocadas, o que pôde colocá-lo de forma competitiva na comparação. Por sua vez, o controlador AsTAR, pela abordagem AIMD, apresentou consideráveis flutuações no número de réplicas, um comportamento não desejável apesar de em alguns cenários ter conseguido alcançar a estabilização devido à modificação com a adição da faixa de histerese.

Diante disso, é possível concluir que o desenvolvimento de estratégias de escalabilidade personalizadas fundamentadas na teoria de controle, especificamente utilizando-se do controlador PID podem representar uma alternativa viável para otimizar o consumo de recursos computacionais das ABMs, contanto que estas sejam flexíveis para os contrapontos vistos nas métricas avaliadas.

5.1 Contribuições

Este trabalho contribui para a área de orquestração de contêineres ao detalhar e realizar o processo de desenvolvimento, implantação e comparação de controladores não convencionais para escalabilidade horizontal no Kubernetes, demonstrando um aumento da eficiência de alocação de recursos computacionais ao realizar a comparação dos controladores propostos em comparação ao controlador HPA sobre uma análise quantitativa.

Além disso, o trabalho também trás a utilização do *framework* CPA, que auxilia na construção de novos controladores e pode servir de base para novas abordagens que sejam feitas no futuro por operadores de *clusters* Kubernetes. É esperado que os resultados obtidos possam servir como referência e inspiração para o desenvolvimento de controladores que objetivem a eficiência da escalabilidade das aplicações.

5.2 Limitações

Apesar das contribuições apresentadas, este estudo possui algumas limitações que devem ser consideradas. Primeiramente, a aplicação utilizada nos testes, a *Online Boutique*, apresenta um padrão de carga específico, o que pode limitar a generalização dos resultados para outras arquiteturas de microsserviços com diferentes perfis de utilização.

Além disso, é válido notar que controladores que precisam de uma etapa de tunagem de parâmetros, como os controladores P, PI e PID, foram treinados especificamente para o cenário de um único microsserviço da aplicação escolhida. Isso significa que cada microsserviço que vá utilizar esses controladores precisa passar previamente por essa etapa e, em caso de alterações que vão desde a novas funcionalidades da aplicação ou até redimensionamento do contêiner, é necessário que esta etapa seja repetida.

5.3 Trabalhos futuros

Diante das limitações identificadas, há diversas oportunidades para a continuidade desta pesquisa:

- **Testes com mais aplicações e microsserviços:** Já que os testes foram realizados em um escopo reduzido, considerando apenas um único microsserviço da aplicação *Online*

Boutique, é esperado que testes com mais aplicações e contemplando todos os microsserviços possam fornecer anaĺises mais completas sobre a performance da proposta nestes diferentes cenários. É também interessante que seja possível realizar a implementação destes controladores em um ambiente produtivo, utilizando aplicações reais para investigar e comprovar se os ganhos percebidos se refletem nestes ambientes.

- **Simplificar ou automatizar o processo de tunagem de parâmetros:** Uma das maiores dificuldades notadas no desenvolvimento do trabalho foi o processo para tunagem dos parâmetros da aplicação. É esperado que próximas contribuições simplifiquem esta operação de modo que ela seja o mais transparente possível, permitindo que o controlador possa se adaptar dinamicamente em caso de alterações da aplicação.

- **Avaliar aumento do intervalo de avaliação e janela de estabilização:** Em muitos dos cenários testados foi possível notar uma grande variação da quantidade de réplicas configuradas pelos controladores. Essa variação, apesar de ser uma característica da aplicação, pode ser minimizada através do aumento do intervalo de avaliação para evitar flutuações nas decisões tomadas ou até mesmo pelo aumento da janela de estabilização utilizada. Futuramente, esses podem virar fatores de futuros experimentos conduzidos, já que algumas aplicações, por terem um tempo maior de inicialização, exigem maiores intervalos entre as avaliações.

- **Testes com outras métricas:** Neste trabalho, foi considerado apenas a utilização de CPU. Apesar de ser uma métrica comumente utilizada, é possível e interessante expandir este trabalho para realizar testes com outras métricas da aplicação, como número de usuários e latência, que geralmente estão relacionados a *Service Level Agreement* (SLA) e *Service Level Objective* (SLO), que definem metas de desempenho e disponibilidade dos serviços, garantindo que a experiência do usuário e a qualidade do serviço atendam aos requisitos estabelecidos por meio de métricas específicas e limites aceitáveis.

Referências Bibliográficas

[1] L. Wang, G. Von Laszewski, A. Younge, X. He, M. Kunze, J. Tao, and C. Fu, "Cloud computing: a perspective study," *New generation computing*, vol. 28, pp. 137–146, 2010.

[2] M. Luksa, *Kubernetes in action*. Simon and Schuster, 2017.

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Lessons learned from three container-management systems over a decade," *Communications of ACM*, 2016.

[4] M.-N. Tran, D.-D. Vu, and Y. Kim, "A survey of autoscaling in kubernetes," in *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*, pp. 263–265, IEEE, 2022.

[5] V. G. da Silva, M. Kirikova, and G. Alksnis, "Containers for virtualization: An overview," *Applied Computer Systems*, vol. 23, no. 1, pp. 21–27, 2018.

[6] A. M. Potdar, D. Narayan, S. Kengond, and M. M. Mulla, "Performance evaluation of docker container and virtual machine," *Procedia Computer Science*, vol. 171, pp. 1419–1428, 2020.

[7] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *2015 IEEE International Conference on cloud engineering*, pp. 386–393, IEEE, 2015.

[8] Z. Kozhirbayev and R. O. Sinnott, "A performance comparison of container-based technologies for the cloud," *Future Generation Computer Systems*, vol. 68, pp. 175–182, 2017.

[9] E. Casalicchio, "Container orchestration: A survey," *Systems Modeling: Methodologies and Tools*, pp. 221–235, 2019.

[10] G. Sayfan, *Mastering Kubernetes: Master the art of container management by using the power of Kubernetes*. Packt Publishing Ltd, 2018.

[11] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback control of dynamic systems*. Upper Saddle River, NJ: Pearson, 6 ed., Sept. 2009.

[12] N. S. Nise, *Control systems engineering*. John Wiley & Sons, 2020.

[13] F. Yang, A. S. Thangarajan, G. S. Ramachandran, W. Joosen, and D. Hughes, "Astar: Sustainable energy harvesting for the internet of things through adaptive task scheduling," *ACM Transactions on Sensor Networks (TOSN)*, vol. 18, no. 1, pp. 1–34, 2021.

[14] N. S. Rosa and D. J. Cavalcanti, "Exploiting controllers to adapt message-oriented middleware," in *2024 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pp. 91–100, IEEE, 2024.

[15] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal pod autoscaling in kubernetes for elastic container orchestration," *Sensors*, vol. 20, no. 16, p. 4621, 2020.

[16] A. A. Khaleq and I. Ra, "Agnostic approach for microservices autoscaling in cloud applications," in *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 1411–1415, IEEE, 2019.

[17] B. Serracanta, A. Lukács, A. Rodriguez-Natal, A. Cabellos, and G. Rétvári, "On the stability of the kubernetes horizontal autoscaler control loop," *IEEE Access*, 2025.

[18] A. F. Baarzi and G. Kesidis, "Showar: Right-sizing and efficient scheduling of microservices," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 427–441, 2021.

[19] N. S. Joshi, R. Raghuwanshi, Y. M. Agarwal, B. Annappa, and D. Sachin, "Arima-pid: container auto scaling based on predictive analysis and control theory," *Multimedia Tools and Applications*, vol. 83, no. 9, pp. 26369–26386, 2024.

[20] J. Dogani, R. Namvar, and F. Khunjush, "Auto-scaling techniques in container-based cloud and edge/fog computing: Taxonomy and survey," *Computer Communications*, vol. 209, pp. 120–150, 2023.

[21] M. Hausenblas and S. Schimanski, *Programming Kubernetes: developing cloud-native applications*. O'Reilly Media, 2019.

[22] Å. Björck, "Least squares methods," *Handbook of numerical analysis*, vol. 1, pp. 465–652, 1990.

[23] R. Jain, *The art of computer systems performance analysis*. Nashville, TN: John Wiley & Sons, Apr. 1991.

[24] P. K. Janert, *Feedback control for computer systems: introducing control theory to enterprise programmers*. "O'Reilly Media, Inc.", 2013.

[25] O. Pozdniakova, D. Mažeika, and A. Cholomskis, "Sla-adaptive threshold adjustment for a kubernetes horizontal pod autoscaler," *Electronics*, vol. 13, no. 7, p. 1242, 2024.

[26] H. Ahmad, C. Treude, M. Wagner, and C. Szabo, "Smart hpa: A resource-efficient horizontal pod auto-scaler for microservice architectures," in *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, pp. 46–57, IEEE, 2024.