

Universidade de Brasília
Faculdade de Ciências e Tecnologias em
Engenharia

Projeto Final: Kubernetes

PROGRAMAÇÃO PARA SISTEMAS PARALELOS E DISTRIBUÍDOS / T01
ENGENHARIA DE SOFTWARE

Brasília
2025

Universidade de Brasília
Faculdade de Ciências e Tecnologias em
Engenharia

Projeto Final: Kubernetes

Heitor Marques Simões Barbosa - 202016462
José Luís Ramos Teixeira - 190057858
Pablo Christianno Silva Guedes - 200042416
Philippe de Sousa Barros - 170154319
Victor de Souza Cabral - 190038900

Professor: Fernando William Cruz

Brasília
2025

Sumário

1	Introdução	4
2	Metodologia	5
2.1	Organização e Planejamento	5
2.2	Roteiro dos Encontros	6
3	Kubernetes	7
3.1	Montagem do Cluster	7
3.2	Arquitetura do Cluster e Diagrama de Rede	8
3.3	Configurações e Arquivos Utilizados	8
3.4	Passos para Configuração	10
3.5	Testes de Performance e Tolerância a Falhas	12
3.6	Resultados Obtidos	13
3.7	Conclusões sobre os Testes	14
4	Kubernetes com Spark	16
4.1	Modificações na Aplicação	16
4.2	Configuração e Instalação	17
4.3	Testes de Performance e Tolerância a Falhas	18
4.4	Cenários de Teste Planejados	18
4.4.1	Cálculo de Pi com Monte Carlo	18
4.5	Resultados Obtidos	20
5	Kubernetes com MPI/OMP	22
5.1	Modificações na Aplicação	22
5.2	Configuração e Instalação	22
5.3	Testes de Performance e Tolerância a Falhas	22
6	Aspectos Teóricos sobre Cloud Native	24
6.1	Definição e Elementos do Cloud Native	24
6.2	Impactos do Kubernetes no Desenvolvimento e DevOps	25
6.3	Comparação entre Microsserviços e Monolitos	26
6.3.1	Monolitos	26
6.3.2	Microsserviços	26
6.3.3	A Abordagem Cloud Native	27
6.4	Monitoramento e Observabilidade	28
6.4.1	Pilares da Observabilidade	28
6.4.2	Monitoramento Tradicional vs. Observabilidade	29

7	Minikube (Extra)	31
7.1	Configuração e Instalação do Minikube	31
7.1.1	Instalar o Docker	31
7.1.2	Instalar o Minikube	31
7.1.3	Iniciar o Minikube	32
7.1.4	Verificar os Nós do Cluster	32
7.1.5	Abrir o Dashboard do Minikube	32
7.1.6	Instalar o Metrics Server	33
7.1.7	Removendo o Minikube (Opcional)	33
7.2	Justificativa para a não utilização	33
8	Conclusão e Autoavaliação	34
8.1	Conclusão Geral	34
8.2	Depoimentos dos Alunos Participantes	34
8.2.1	Heitor Marques Simões Barbosa	34
8.2.2	José Luís Ramos Teixeira	35
8.2.3	Pablo Christianno Silva Guedes	35
8.2.4	Philippe de Sousa Barros	36
8.2.5	Victor de Souza Cabral	36
8.3	Nota de Autoavaliação	36
	Referências	37

1 Introdução

Este projeto final, desenvolvido para a disciplina de Programação para Sistemas Paralelos e Distribuídos, tem como objetivo avaliar o uso do Kubernetes em diferentes cenários de computação distribuída e paralela. Através dessa iniciativa, será possível estudar conceitos de orquestração de contêineres, escalabilidade, tolerância a falhas e integração com outras tecnologias. O trabalho inclui uma análise detalhada sobre o desempenho do Kubernetes, tanto de forma isolada quanto em combinação com outras ferramentas (Spark e MPI/OMP).

O projeto é iniciado com a definição da metodologia, onde será descrita a organização do grupo e o planejamento das atividades realizadas. Em seguida, o foco recai sobre o Kubernetes, com a documentação da montagem de um cluster funcional. Serão apresentados o diagrama de rede, os arquivos de configuração utilizados, os procedimentos adotados para a instalação e, finalmente, os testes de desempenho e resiliência.

Posteriormente, exploramos a integração do Kubernetes com o Spark, um framework de processamento distribuído. Essa etapa envolve ajustes na aplicação, execução de testes sob diferentes condições e uma análise dos resultados obtidos. O trabalho também abrange a utilização de bibliotecas paralelas MPI e OMP no Kubernetes, observando o impacto dessa combinação sobre o desempenho da aplicação e a resistência do sistema a falhas.

Além das análises práticas, há a possibilidade de uma discussão teórica sobre o conceito de Cloud Native. Nesse contexto, serão apresentados elementos fundamentais desse paradigma, como microsserviços, DevOps e monitoramento de aplicações distribuídas.

O relatório é finalizado com uma seção de conclusão, onde se realiza uma análise comparativa dos resultados alcançados em cada etapa. Também são incluídos depoimentos dos alunos participantes, com reflexões sobre os aprendizados e uma avaliação do nível de envolvimento individual.

A tabela a seguir apresenta os links para o repositório do projeto e para a apresentação em vídeo no YouTube.

Tabela 1.1 – Links Importantes

Descrição	Link
Repositório do Projeto no GitHub	https://github.com/joseluis-rt/PSPD_ProjetoFinal
Vídeo de Apresentação no YouTube	https://youtu.be/1-cmqJN8cwA

Fonte: Elaboração própria.

2 Metodologia

A realização deste projeto exigiu uma organização adequada, além do uso de ferramentas específicas para experimentos e monitoramento do Kubernetes. O planejamento abrangeu tanto encontros presenciais quanto reuniões online, com o objetivo de otimizar o tempo e garantir a colaboração do grupo em todas as etapas do trabalho.

2.1 Organização e Planejamento

Para o desenvolvimento do projeto, utilizamos ferramentas amplamente reconhecidas na área de computação em nuvem e sistemas distribuídos. A principal ferramenta foi o Kubernetes, uma plataforma de orquestração de contêineres, que possibilita a escalabilidade e alta disponibilidade de aplicações ([Kubernetes 2025](#)).

Além disso, empregamos o Apache Spark para processamento paralelo de grandes volumes de dados ([Spark 2025](#)). Para a integração com bibliotecas de programação paralela, utilizamos MPI (Message Passing Interface) e OMP (OpenMP), tecnologias projetadas para execução eficiente de tarefas em múltiplos núcleos ([OpenMP 2025](#)).

Para a organização do trabalho e o controle de versão dos arquivos, utilizamos o GitHub ([GitHub 2025](#)). O repositório foi estruturado para armazenar o código-fonte, arquivos de configuração e documentação do projeto. Essa abordagem garantiu que todos os integrantes do grupo pudessem colaborar de forma eficiente e centralizada, mantendo o histórico de alterações e facilitando o acesso às informações.

Os encontros do grupo foram organizados de forma a atender diferentes atividades necessárias para o projeto, como a montagem do cluster Kubernetes, a execução de testes e a documentação dos resultados.

Utilizamos um **switch** e **cabos Ethernet** (Figura 2.1) para a interligação das máquinas Ubuntu, garantindo a comunicação eficiente na rede.



Figura 2.1 – Switch de 8 portas Fast Ethernet SF 800 utilizado no projeto.

2.2 Roteiro dos Encontros

A Tabela 2.1 apresenta o cronograma de atividades realizadas durante o desenvolvimento do projeto. Esse planejamento foi elaborado para garantir a realização de todas as etapas, respeitando o prazo de entrega final no dia 22 de fevereiro de 2025.

Tabela 2.1 – Cronograma de Atividades

Data	Modo	Atividade
10/02/2025	Online	Definição do escopo do projeto e organização das tarefas.
12/02/2025	Presencial	Configuração inicial do cluster Kubernetes e instalação das ferramentas.
14/02/2025	Online	Revisão e ajustes nas configurações do cluster.
16/02/2025	Presencial	Testes de performance no Kubernetes isolado.
18/02/2025	Online	Integração do Spark e execução de testes de carga.
19/02/2025	Online	Integração com MPI/OMP e validação dos resultados.
20/02/2025	Presencial	Testes e resultados, preparação do relatório e gravação do vídeo.
21/02/2025	Presencial	Revisão e testes finais e entrega do projeto.

Fonte: Elaboração própria.

3 Kubernetes

3.1 Montagem do Cluster

A montagem do cluster isolado seguiu uma abordagem bare-metal, utilizando **kube-
adm** para a instalação e configuração. Inicialmente, os pacotes do sistema foram atualizados, e o Docker foi instalado, sendo essencial para a execução dos containers Kubernetes. Em seguida, os pacotes kubeadm, kubelet e kubectl foram instalados e configurados. Durante o processo, enfrentamos problemas com a versão 1.32.2 do kubeadm e kubectl, o que nos levou a adotar a versão 1.30.10, que se mostrou mais estável para o nosso ambiente.



Figura 3.1 – Montagem do cluster isolado inicial



Figura 3.2 – Configuração dos nós do cluster



Figura 3.3 – Teste de falha em worker 1

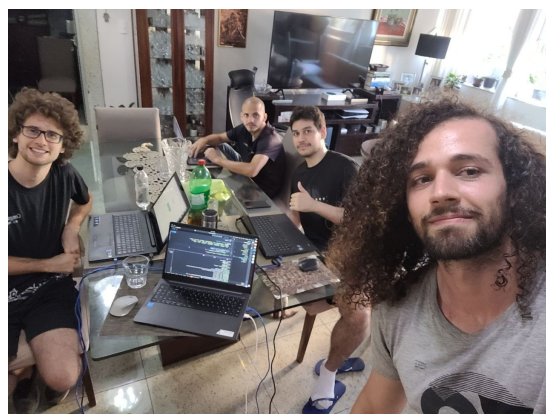
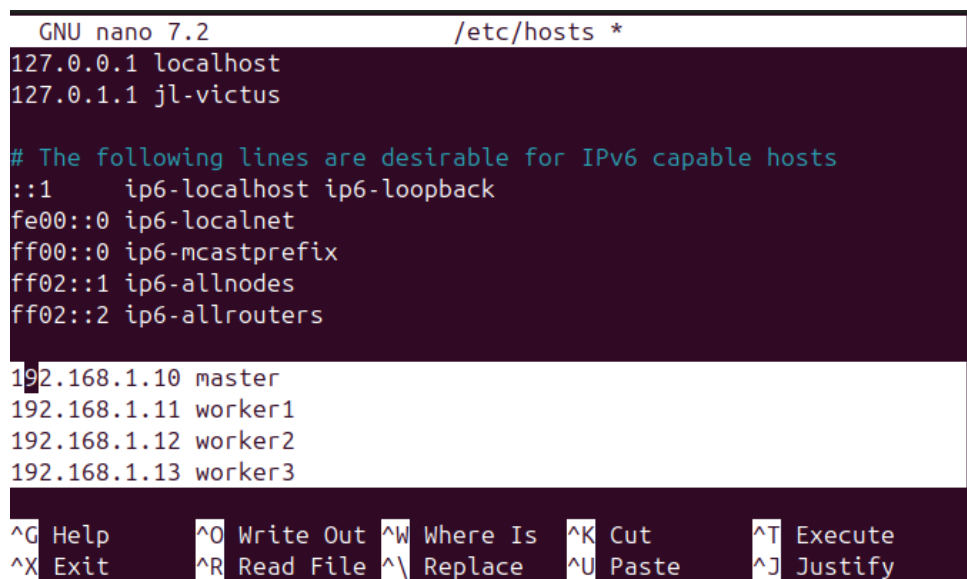


Figura 3.4 – Resultado final da contagem de palavras

3.2 Arquitetura do Cluster e Diagrama de Rede

O cluster foi projetado com uma estrutura contendo um nó mestre e múltiplos nós workers. A comunicação entre os nós é feita através de uma rede privada configurada manualmente em `/etc/hosts` em todas as máquina (master e workers). O diagrama de rede segue a seguinte estrutura:

- **Master Node** (192.168.1.10)
- **Worker 1** (192.168.1.11)
- **Worker 2** (192.168.1.12)
- **Worker 3** (192.168.1.13)

A screenshot of the GNU nano 7.2 text editor editing the /etc/hosts file. The editor's title bar shows 'GNU nano 7.2' and the file path '/etc/hosts *'. The content of the file is as follows:

```
127.0.0.1 localhost
127.0.1.1 jl-victus

# The following lines are desirable for IPv6 capable hosts
::1      ip6-localhost ip6-loopback
fe00::0  ip6-localnet
ff00::0  ip6-mcastprefix
ff02::1  ip6-allnodes
ff02::2  ip6-allrouters

192.168.1.10 master
192.168.1.11 worker1
192.168.1.12 worker2
192.168.1.13 worker3
```

The bottom status bar of the nano editor displays various keyboard shortcuts: ^G Help, ^O Write Out, ^W Where Is, ^K Cut, ^T Execute, ^X Exit, ^R Read File, ^\ Replace, ^U Paste, and ^J Justify.

Figura 3.5 – nano /etc/hosts

3.3 Configurações e Arquivos Utilizados

Além da configuração inicial do cluster, foi necessário criar uma imagem Docker específica para executar uma aplicação Python de contagem de palavras (**word-counter**). A imagem foi construída e publicada no Docker Hub para ser utilizada nos nós do cluster, permitindo escalabilidade e portabilidade para diferentes ambientes. Para garantir a execução adequada da aplicação, dois arquivos YAML foram criados e configurados para o nó master:

- **Deployment YAML:** Este arquivo é responsável por baixar a imagem do Docker Hub e iniciar a aplicação. Ele define o número de réplicas desejadas, as labels para identificação dos pods e as configurações de container.
- **Service YAML:** Configura o serviço para expor a aplicação na rede, tornando-a acessível externamente. Ele mapeia a porta do container para a porta do serviço, permitindo que as requisições sejam direcionadas para o pod adequado.

A seguir, apresentamos os arquivos YAML utilizados:

3.3.0.1 Deployment YAML

Código 3.1 – Deployment do Word Counter no Kubernetes

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: word-counter
5 spec:
6   replicas: 3
7   selector:
8     matchLabels:
9       app: word-counter
10  template:
11    metadata:
12      labels:
13        app: word-counter
14    spec:
15      containers:
16      - name: word-counter
17        image: pcsg1/word-counter:1
18        ports:
19      - containerPort: 80
```

3.3.0.2 Service YAML

Código 3.2 – Service do Word Counter no Kubernetes

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: word-counter-service
5 spec:
6   selector:
7     app: word-counter
8   ports:
9     - protocol: TCP
10     port: 80
11     targetPort: 80
12   type: NodePort
```

O código-fonte da aplicação, juntamente com a configuração do Kubernetes, pode ser acessado no repositório GitHub. A seguir, um link para o repositório, onde você pode encontrar todos os detalhes do projeto, incluindo o código da aplicação e os arquivos YAML configurados para o ambiente isolado:

[Repositório GitHub: PSPD Projeto Final](#)

3.4 Passos para Configuração

Este passo configura os pré-requisitos para o Kubernetes, incluindo módulos do kernel, pacotes necessários e o container runtime (containerd). Essas configurações devem ser feitas **em todas as máquinas** (master e workers).

3.4.0.1 Atualizar pacotes

```
1 sudo apt-get update && sudo apt-get upgrade -y
```

3.4.0.2 Configurar módulos do kernel

```
1 sudo modprobe overlay
2 sudo modprobe br_netfilter
```

3.4.0.3 Instalar dependências básicas

```
1 sudo apt-get install curl ca-certificates gnupg
```

3.4.0.4 Adicionar repositório do Docker

```
1 curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --
    dearmor -o /etc/apt/keyrings/docker.gpg
```

3.4.0.5 Configurar containerd

```
1 sudo apt-get install -y containerd.io
2 sudo containerd config default | sudo tee /etc/containerd/config.toml
3 sudo systemctl restart containerd
```

3.4.0.6 Adicionar repositório do Kubernetes

```
1 curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.32/deb/Release.key | sudo
    gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
2 sudo apt-get install -y kubelet=1.32.2-1.1 kubeadm=1.32.2-1.1 kubectl
    =1.32.2-1.1
```

3.4.0.7 Configuração do Master

```
1 sudo kubeadm init --apiserver-advertise-address=192.168.1.10 --pod-network-cidr=10.244.0.0/16 --kubernetes-version=1.32.2
2 mkdir -p $HOME/.kube
3 sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
4 sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

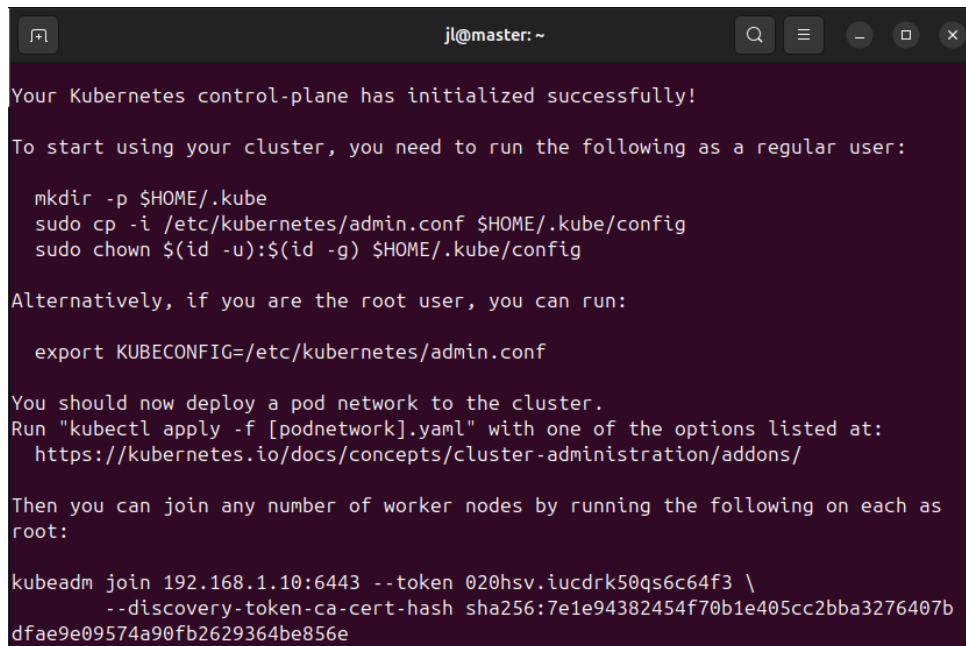


Figura 3.6 – Kubernetes initialized

3.4.0.8 Configuração dos Workers

Os workers devem se conectar ao master usando o comando gerado na inicialização do master:

```
1 kubeadm join 192.168.1.10:6443 --token --discovery-token-ca-cert-hash
```

Depois da configuração do cluster, a verificação dos pods e nós é essencial para garantir que todos os componentes estejam funcionando corretamente. As imagens 4.3 e 3.8 mostram o terminal durante o processo de montagem e configuração do cluster. Na primeira imagem (4.3), o terminal está exibindo a inicialização dos pods, enquanto na segunda (3.8), os nós estão sendo configurados e verificados para garantir que o cluster esteja operacional.

Essas imagens representam o processo de configuração do cluster isolado, no qual os pods e nós são configurados e validados antes de serem considerados prontos para o uso.

```

jl@master:~$ kubectl get pods -n kube-system
NAME                                READY   STATUS    RESTARTS   AGE
coredns-55cb58b774-84n6d           1/1     Running   0           66s
coredns-55cb58b774-k2g7g           1/1     Running   0           66s
etcd-master                         1/1     Running   3           82s
kube-apiserver-master               1/1     Running   4           82s
kube-controller-manager-master      1/1     Running   4           82s
kube-proxy-mpb97                    1/1     Running   0           66s
kube-scheduler-master               1/1     Running   4           82s
jl@master:~$ kubectl get nodes
NAME      STATUS   ROLES    AGE   VERSION
master    Ready    control-plane  88s   v1.30.10
jl@master:~$ kubectl get nodes
NAME      STATUS   ROLES    AGE   VERSION
master    Ready    control-plane  7m21s  v1.30.10
worker1    Ready    <none>     10s   v1.30.10
jl@master:~$

```

Figura 3.7 – kubectl nodes master iniciando

```

jl@master:~$ kubectl get nodes
NAME      STATUS   ROLES    AGE   VERSION
master    Ready    control-plane  88s   v1.30.10
jl@master:~$ kubectl get nodes
NAME      STATUS   ROLES    AGE   VERSION
master    Ready    control-plane  7m21s  v1.30.10
worker1    Ready    <none>     10s   v1.30.10
jl@master:~$ kubectl get nodes
NAME      STATUS   ROLES    AGE   VERSION
master    Ready    control-plane  15m   v1.30.10
worker1    Ready    <none>     8m1s  v1.30.10
jl@master:~$ kubectl get nodes
NAME      STATUS   ROLES    AGE   VERSION
master    Ready    control-plane  15m   v1.30.10
worker1    Ready    <none>     8m5s  v1.30.10
worker2    Ready    <none>     2s    v1.30.10
jl@master:~$

```

Figura 3.8 – kubectl nodes workers entrando na master

3.5 Testes de Performance e Tolerância a Falhas

Neste teste, utilizamos uma aplicação simples em Python para contar a ocorrência de palavras em um arquivo de texto. A aplicação foi implantada em um cluster Kubernetes isolado para avaliar tanto a performance quanto a tolerância a falhas. A aplicação foi testada em um ambiente com um único nó mestre e múltiplos workers, e durante os cálculos, simulamos falhas em um dos workers para verificar a continuidade do processamento.

Durante a execução do teste, o arquivo `palavras.txt` foi processado para contar a ocorrência de palavras. Para simular a falha, desconectamos um worker enquanto o cálculo estava em andamento. O objetivo era verificar se o cluster continuaria processando o arquivo sem interrupção significativa. A tolerância a falhas foi observada enquanto o Kubernetes redistribuía automaticamente a carga para os workers restantes.

A aplicação foi executada e testada para garantir que o Kubernetes lidasse adequadamente com a falha de um worker e que os resultados de contagem de palavras fossem gerados

corretamente. O teste também envolveu a medição do tempo de execução para avaliar a performance da aplicação em diferentes configurações de nós e workers.

```
j1l@master:~$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
master	Ready	control-plane	95m	v1.30.10
worker1	NotReady	<none>	94m	v1.30.10
worker2	Ready	<none>	94m	v1.30.10

Figura 3.9 – Kubectl Nodes

3.6 Resultados Obtidos

Durante os testes, observamos que a aplicação de contagem de palavras foi capaz de continuar a execução após a falha de um dos pods, demonstrando a resiliência do sistema Kubernetes. Para monitorar o estado dos pods responsáveis pelo processamento, utilizamos o comando:

```
1 kubectl get pods
```

A saída desse comando é mostrada na Figura 3.10, onde podemos observar que o pod `word-counter-67956c6d7b-mnfct` foi o primeiro a concluir sua execução, apresentando o status `Completed`, enquanto outros pods ainda estavam em execução.

```
j1l@master:~$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
kayvan-release-spark-master-0	1/1	Running	0	73m
kayvan-release-spark-worker-0	1/1	Running	0	112m
kayvan-release-spark-worker-1	1/1	Running	0	112m
kayvan-release-spark-worker-2	1/1	Running	0	111m
word-counter-67956c6d7b-f45wf	1/1	Running	1 (13s ago)	3m10s
word-counter-67956c6d7b-mnfct	0/1	Completed	1 (95s ago)	3m10s
word-counter-67956c6d7b-p6jxj	1/1	Running	0	3m10s

Figura 3.10 – Estado dos pods no Kubernetes após a execução da contagem de palavras.

Após a finalização do pod `word-counter-67956c6d7b-mnfct`, utilizamos o comando:

```
1 kubectl logs word-counter-67956c6d7b-mnfct
```

para visualizar a contagem de palavras processada pelo sistema. A Figura 3.11 apresenta a saída desse comando, exibindo as palavras únicas identificadas no arquivo de entrada juntamente com suas respectivas frequências.

```
jl@master:~$ kubectl logs word-counter-67956c6d7b-mnfct
Contagem de palavras:
afeio: 4762223
ambiente: 4758321
astro: 4763255
astrorei: 4758122
bemestar: 4761711
claridade: 4764211
companheirismo: 4763252
confiana: 4762828
contentamento: 4764990
correnteza: 4768551
existncia: 4758387
garoa: 4761257
jbilo: 4762658
lar: 4760732
ocupao: 4764396
parentesco: 4760839
pspd: 4764082
ptala: 4758022
riso: 4759937
serenidade: 4761960
serra: 4760266
```

Figura 3.11 – Saída do log do pod finalizado, exibindo a contagem de palavras.

Com isso, verificamos que a execução do processamento ocorreu conforme esperado, e o Kubernetes garantiu a conclusão da tarefa mesmo com a finalização de pods durante a execução. O sistema foi capaz de manter a integridade da contagem de palavras sem interrupções, aproveitando a natureza distribuída da plataforma.

O tempo total necessário para que um dos pods atingisse o estado Completed e seus logs pudessem ser analisados foi de **95 segundos**. Esse valor representa uma execução sem falhas e confirma que o Kubernetes gerenciou corretamente a carga de trabalho entre os pods disponíveis, assegurando a conclusão da contagem de palavras de forma eficiente.

A tolerância a falhas foi validada quando a desconexão de um worker não afetou a contagem das palavras, e o tempo de execução variou ligeiramente, mas sem grandes impactos na integridade do processamento.

3.7 Conclusões sobre os Testes

Os testes de performance e tolerância a falhas realizados no Kubernetes com a aplicação de contagem de palavras mostraram que o sistema é altamente resiliente a falhas de workers. O Kubernetes foi capaz de redistribuir as tarefas de maneira eficiente, garantindo que o processamento continuasse sem interrupções significativas.

Em termos de performance, o sistema demonstrou um tempo de execução aceitável para o tamanho do arquivo de entrada e a quantidade de workers. A aplicação foi capaz de lidar com o processamento em paralelo de maneira eficiente, e o tempo de execução não sofreu impactos significativos após a falha de um worker.

Em resumo, os testes confirmaram que a infraestrutura Kubernetes oferece alta disponibilidade e tolerância a falhas, permitindo que o cluster continue a funcionar mesmo diante de falhas de componentes individuais. Esses resultados indicam que o Kubernetes é

uma escolha sólida para a execução de tarefas distribuídas, como a contagem de palavras, em ambientes com alta demanda e necessidade de resiliência.

4 Kubernetes com Spark

4.1 Modificações na Aplicação

Nesta seção, detalhamos as modificações necessárias para executar uma aplicação Spark no Kubernetes utilizando o Spark Operator. O exemplo a seguir mostra a definição da aplicação `spark-pi`, que executa um cálculo de Pi utilizando o Apache Spark. A configuração da aplicação é definida no arquivo `spark-pi.yaml`.

Abaixo está o conteúdo do arquivo de configuração da aplicação:

Código 4.1 – Configuração da aplicação Spark-Pi

```
1 apiVersion: sparkoperator.k8s.io/v1beta2
2 kind: SparkApplication
3 metadata:
4   name: spark-pi
5   namespace: default
6 spec:
7   type: Scala
8   mode: cluster
9   image: spark:3.5.3
10  imagePullPolicy: IfNotPresent
11  mainClass: org.apache.spark.examples.SparkPi
12  mainApplicationFile: local:///opt/spark/examples/jars/spark-examples.jar
13  arguments:
14    - "5000"
15  sparkVersion: 3.5.3
16  driver:
17    labels:
18      version: 3.5.3
19    cores: 1
20    memory: 512m
21    serviceAccount: spark-operator-spark
22  executor:
23    labels:
24      version: 3.5.3
25    instances: 2
26    cores: 1
27    memory: 512m
```

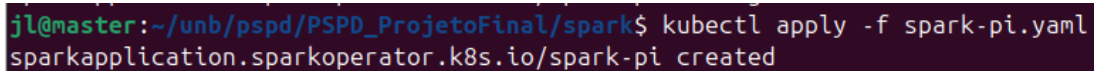
Nesta configuração:

- A aplicação Spark roda no modo cluster, utilizando a imagem `spark:3.5.3`.
- O código-fonte da aplicação está no JAR `/opt/spark/examples/jars/spark-examples.jar`.
- O cálculo de Pi é executado com 5000 iterações.
- O **driver** é configurado com 1 núcleo de CPU e 512MB de memória.

- Cada **executor** possui 1 núcleo de CPU e 512MB de memória, com um total de 2 instâncias.
- A aplicação usa a conta de serviço `spark-operator-spark` para permissões no cluster.

Após salvar esse arquivo, a aplicação pode ser enviada para o cluster com o comando:

```
kubectl apply -f spark-pi.yaml
```



```
jl@master:~/unb/pspd/PSPD_ProjetoFinal/spark$ kubectl apply -f spark-pi.yaml
sparkapplication.sparkoperator.k8s.io/spark-pi created
```

Figura 4.1 – Spark apply

Para verificar o status da aplicação:

```
kubectl get sparkapplications -n default
```

Caso seja necessário remover a aplicação:

```
kubectl delete -f spark-pi.yaml
```

4.2 Configuração e Instalação

Aqui, apresentamos os passos necessários para configurar e gerenciar o Spark Operator no Kubernetes utilizando o Helm. O Spark Operator facilita a execução de aplicações Apache Spark dentro do cluster, gerenciando automaticamente recursos como drivers e executores.

Primeiramente, adicionamos o repositório do Spark Operator ao Helm e atualizamos a lista de charts disponíveis:

```
1 helm repo add spark-operator https://kubeflow.github.io/spark-operator
2 helm repo update
```

Caso o operador já esteja instalado e seja necessário removê-lo antes de uma nova instalação ou atualização, utilize o seguinte comando:

```
1 helm uninstall spark-operator -n spark-operator
```

Para instalar o Spark Operator no cluster Kubernetes, execute:

```
1 helm install spark-operator spark-operator/spark-operator \
2   --namespace spark-operator --create-namespace \
3   --set webhook.enable=false
```

Se for necessário excluir uma aplicação Spark específica do cluster, utilize o comando:

```
1 kubectl delete sparkapplication spark-pi -n spark-operator
```

Por exemplo, para remover uma aplicação chamada `spark-pi` no namespace `default`, utilize:

```
1 kubectl delete sparkapplication spark-pi -n default
```

4.3 Testes de Performance e Tolerância a Falhas

Avaliamos a performance e a tolerância a falhas do cluster Kubernetes com Spark. O objetivo foi garantir que o sistema seja capaz de continuar operando mesmo após a falha de um dos seus componentes.

Durante o teste, a aplicação Spark foi configurada para calcular o valor de pi utilizando o método Monte Carlo. Enquanto o cálculo estava em execução, um dos workers do cluster foi desconectado de forma intencional.

O comportamento observado foi que, mesmo com a falha de um worker, a aplicação continuou a execução do cálculo sem interrupções significativas. O Kubernetes, através do Spark Operator, gerenciou a falha de forma eficiente, redistribuindo as tarefas do worker desconectado para os outros workers ativos, garantindo a continuidade do processamento.

Este teste demonstra a resiliência do sistema e sua capacidade de lidar com falhas em um ambiente distribuído, mantendo a integridade da aplicação e a execução dos cálculos de forma contínua.

4.4 Cenários de Teste Planejados

Para validar a configuração do Spark no Kubernetes, utilizaremos um cenário de teste baseado no cálculo de Pi, como já mencionado anteriormente, pelo método de Monte Carlo. Esse teste é amplamente utilizado para avaliar a execução distribuída no Apache Spark, pois envolve um grande número de operações independentes que podem ser paralelizadas de forma eficiente.

4.4.1 Cálculo de Pi com Monte Carlo

O método de Monte Carlo estima o valor de Pi gerando pontos aleatórios dentro de um quadrado e verificando quantos deles caem dentro de um círculo inscrito. A razão entre os pontos dentro do círculo e o total de pontos gerados aproxima o valor de Pi.

A aplicação Spark que implementa esse cálculo já está definida no arquivo `spark-pi.yaml`, conforme descrito na Seção 4.1. Para executar o teste, aplicamos a seguinte configuração no Kubernetes:

```
1 kubectl apply -f spark-pi.yaml
```

Após o envio da aplicação, podemos monitorar seu status com o comando:

```
1 kubectl get sparkapplications -n default
```

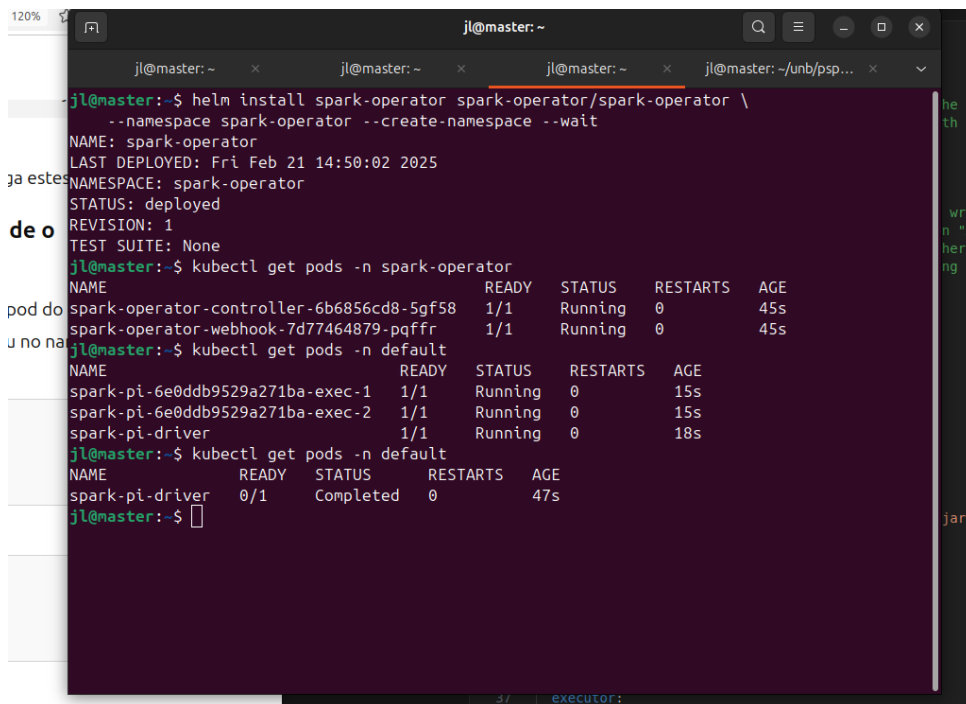
```
jl@master:~$ kubectl get sparkapplications -n default
NAME      STATUS    ATTEMPTS  START              FINISH             AGE
spark-pi  RUNNING   1         2025-02-21T16:31:39Z  <no value>         6s
jl@master:~$ kubectl get pods -n spark-operator
NAME                                            READY   STATUS    RESTARTS   AGE
spark-operator-controller-6b6856cd8-k76wc     1/1     Running   0          2m45s
spark-operator-webhook-7d77464879-jt8pg      1/1     Running   0          2m45s
```

Figura 4.2 – Get Sparkapplications

Para verificar os logs do driver e acompanhar a execução do cálculo:

```
1 kubectl logs -f spark-pi-driver -n default
```

Após a execução bem-sucedida, os logs da aplicação exibirão a estimativa do valor de Pi.



```
jl@master:~$ helm install spark-operator spark-operator/spark-operator \
--namespace spark-operator --create-namespace --wait
NAME: spark-operator
LAST DEPLOYED: Fri Feb 21 14:50:02 2025
NAMESPACE: spark-operator
STATUS: deployed
REVISION: 1
TEST SUITE: None
jl@master:~$ kubectl get pods -n spark-operator
NAME                                            READY   STATUS    RESTARTS   AGE
spark-operator-controller-6b6856cd8-5gf58     1/1     Running   0          45s
spark-operator-webhook-7d77464879-pqffr      1/1     Running   0          45s
jl@master:~$ kubectl get pods -n default
NAME                                            READY   STATUS    RESTARTS   AGE
spark-pi-6e0ddb9529a271ba-exec-1             1/1     Running   0          15s
spark-pi-6e0ddb9529a271ba-exec-2             1/1     Running   0          15s
spark-pi-driver                               1/1     Running   0          18s
jl@master:~$ kubectl logs -f spark-pi-driver -n default
NAME                                            READY   STATUS    RESTARTS   AGE
spark-pi-driver                               0/1     Completed 0          47s
jl@master:~$
```

Figura 4.3 – Spark Operator running

Esse cenário de teste nos permite avaliar a performance do Spark Operator no Kubernetes e garantir que os recursos do cluster estão sendo corretamente alocados para a execução distribuída.

4.5 Resultados Obtidos

Nesta seção, apresentamos os resultados obtidos durante os testes de desempenho da execução do Spark no Kubernetes. Os experimentos foram realizados variando o número de instâncias, núcleos de CPU e memória alocada para os executores, além do número de iterações do cálculo de π .

Tabela 4.1 – Resultados dos testes de desempenho no Spark Kubernetes

Instâncias	Cores	Memória	Iterações	Tempo (s)	Resultado de Pi
2	1	512 MB	5000	25.73	3.141646926283294
2	2	512 MB	5000	38.10	3.141648558283297
3	4	512 MB	5000	35.58	3.141558958283118
4	2	1024 MB	5000	21.04	3.1416031742832065
4	2	1024 MB	5000	28.36	3.1416454782832908
4	3	1024 MB	10000	30.76	3.1415991822831986

Fonte: Elaboração própria.

Valor de pi com 15 casas decimais aproximado para referência: 3.141592653589793

A Figura 6.1 exibe a interface do Spark Master no Kayvan, onde é possível visualizar o estado do cluster. No exemplo, observa-se que pelo menos um worker está ativo e pronto para executar as aplicações enviadas. Essa interface permite monitorar a alocação de recursos, como uso de CPU e memória, além do status das execuções em andamento no ambiente Kubernetes.

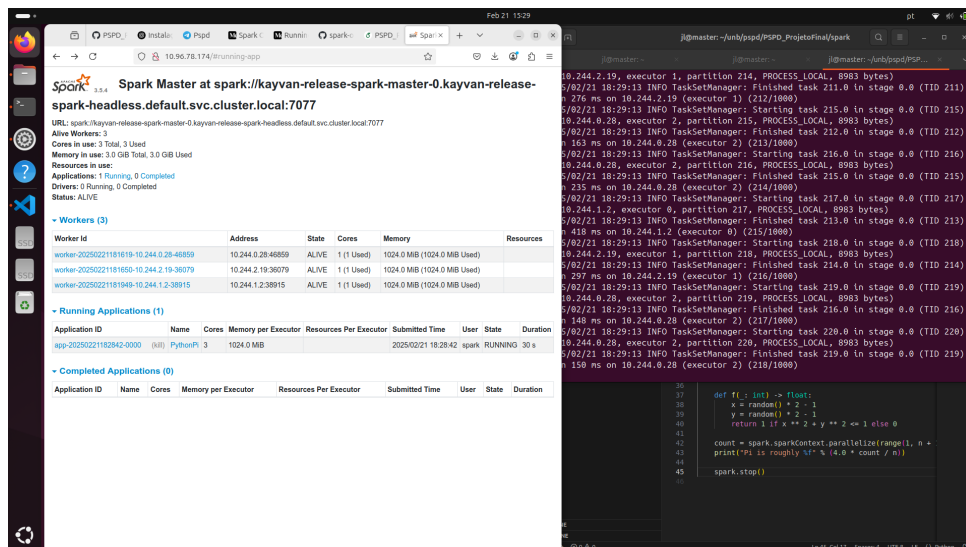


Figura 4.4 – Spark Master

Os testes indicaram que aumentar o número de instâncias e alocar mais recursos de CPU e memória para os executores melhora significativamente o tempo de execução do

Spark. O impacto é mais perceptível quando há mais iterações na execução do cálculo, demonstrando a capacidade do Spark de escalar horizontalmente e paralelizar as tarefas com eficiência.

Os gráficos a seguir ilustram a variação do tempo de execução em função do número de instâncias e configurações de recursos.

5 Kubernetes com MPI/OMP

5.1 Modificações na Aplicação

Nesta seção, abordaremos as modificações realizadas na aplicação para a utilização do Kubernetes com OpenMP. A principal mudança consiste na implementação de paralelismo e na configuração da infraestrutura para suportar o processamento em múltiplos núcleos, aproveitando os recursos do Kubernetes e da tecnologia OpenMP para distribuir as tarefas entre os workers de forma eficiente. A implementação do código foi adaptada para rodar em múltiplos nós e aproveitar a capacidade de processamento paralelo oferecida pelo Kubernetes com OpenMP.

5.2 Configuração e Instalação

Para configurar e instalar o ambiente necessário para a execução do MPI com Kubernetes e OpenMP, seguimos as etapas descritas a seguir. Esta configuração envolve a instalação de pacotes essenciais, a configuração do ambiente de execução de contêineres e a preparação do cluster Kubernetes para rodar aplicações paralelizadas com OpenMP.

Primeiro, é necessário instalar e configurar os módulos do Kubernetes, conforme descrito na seção anterior, com a adição dos repositórios necessários e a instalação do ‘mpich’ ou outro gerenciador MPI, bem como a configuração do ambiente OpenMP.

Após a instalação do MPI e do OpenMP, o Kubernetes será configurado para permitir a execução de jobs paralelos, aproveitando os recursos distribuídos de todos os nós no cluster. Para isso, tivemos que criar outra imagem docker com a configuração para executar uma aplicação MPI/openMP usamos como base a imagem do ubuntu 24.04, anteriormente foi testado com a versão 22.04 porém não foi obtido um resultado esperado do docker run. Então subimos a imagem para o docker hub, e criamos um arquivo .yaml para o deploy do job MPI e openMP no cluster

5.3 Testes de Performance e Tolerância a Falhas

Nesta etapa, realizamos testes de performance e de tolerância a falhas em um ambiente Kubernetes configurado com MPI e OpenMP. Durante os testes de performance, medimos a eficiência da aplicação ao utilizar múltiplos núcleos de CPU e o tempo de execução em diferentes configurações de cluster.

Além disso, simulamos falhas no cluster, como a desconexão de um worker durante a execução de uma tarefa paralelizada, para avaliar como a aplicação lida com essas situações. O objetivo foi garantir que a aplicação seja capaz de se recuperar e continuar sua execução sem perdas significativas de dados ou performance, demonstrando a robustez do sistema em um ambiente de produção.

6 Aspectos Teóricos sobre Cloud Native

6.1 Definição e Elementos do Cloud Native

A abordagem Cloud Native se refere não só de executar as aplicações na nuvem, mas sim de um método de software desenvolvido para operar em um ambiente de computação em nuvem. Elas levam em conta aspectos como ser dimensionável, altamente disponível e fácil de gerenciar. Três aspectos são imprescindíveis em qualquer arquitetura Cloud Native:

- **Ser containerizada:** Aplicações e processos são empacotados em contêineres, garantindo reprodutibilidade, transparência e isolamento de recursos.
- **Gerenciada dinamicamente:** Contêineres são orquestrados ativamente, o que otimiza o uso de recursos.
- **Orientada a microsserviços:** Segmentação das aplicações em microsserviços, havendo aumento da agilidade e facilidade de manutenção.

(GitLab s.d.).

Ao contrário das aplicações tradicionais, que muitas vezes dependem de infraestruturas rígidas e difíceis de escalar, as aplicações Cloud Native são projetadas para aproveitar ao máximo os recursos da nuvem, sendo uma abordagem moderna para o desenvolvimento de software. Elas utilizam tecnologias e práticas que garantem maior disponibilidade, automação e capacidade de adaptação às necessidades dinâmicas do mercado. Outros benefícios importantes de se mencionar quanto a uma aplicação Cloud Native são:

- **Independência:** Permite desenvolver, gerenciar e implementar cada componente separadamente.
- **Resiliência:** Permanece online caso haja falhas na infraestrutura.
- **Baseado em Padrões:** Para garantir interoperabilidade e portabilidade de carga de trabalho, se baseiam na tecnologias de código aberto e baseada em padrões.
- **Agilidade Comercial:** Ao serem menores e com mais flexíveis, facilitam o desenvolvimento, implementação e iteração.
- **Automação:** Permitem entregas contínuas e implementações sem impacto para os usuários ao integrar com DevOps,.
- **Sem Tempo de Inatividade:** Uso de orquestradores como Kubernetes possibilita atualizações sem interrupções.

(Oracle s.d.).

6.2 Impactos do Kubernetes no Desenvolvimento e DevOps

O DevOps é bastante impactado pelo uso de contêineres e seus orquestradores, assim como pelas tecnologias de computação em nuvem e clusters. A automação se mostra como um ponto importante dessa relação, já que permite uma alocação eficiente e rápida de infraestrutura em nuvem. Além disso, a utilização de contêineres se tornou essencial no paradigma de computação em nuvem, pois garante portabilidade, escalabilidade e isolamento de aplicações, assim como elimina as dependências de infraestrutura física, o que traz uma facilitação tanto no gerenciamento de recursos quanto na adaptação das demandas de carga variáveis.

O Kubernetes entra então como uma das principais ferramentas de orquestração de contêineres, na qual permite uma implantação automatizada, escalonamento e operação de aplicações em ambientes distribuídos. E já que soluções Cloud Native geralmente utilizam containers extensivamente, em situações que é exigido uma alta disponibilidade e resiliência deles o Kubernetes aparece como uma solução padrão na orquestração e gerenciamento em contextos Cloud Native ([Medium 2023](#)).

Se alinhando com as práticas de DevOps e Cloud Native, ele:

- Aprimora tanto o desenvolvimento quanto a manutenção de aplicações.
- Possui uma abordagem baseada em microserviços, permitindo atualizações e implantações independentes, reduzindo riscos sistêmicos e facilitando a escalabilidade.
- Os períodos de indisponibilidade são reduzidos, já que há um aumento da resiliência graças a automação e capacidade de autorrecuperação.
- Integrado a pipelines CI/CD, possibilita implantações contínuas e seguras, promovendo um workflow colaborativo entre desenvolvimento e operações.
- Exige um gerenciamento estruturado do ciclo de vida das aplicações, devido à maior complexidade imposta por essas tecnologias.

([GitLab s.d.](#)).

Além da eficiência e agilidade no desenvolvimento, o uso de Kubernetes e tecnologias Cloud Native proporciona benefícios estratégicos às organizações:

- As novas funcionalidades acabam tendo uma redução no seu tempo de entrega
- Recursos computacionais tem uma otimização e custos operacionais uma minimização.
- Simplificação da manutenção e escalabilidade de aplicações mais complexas.

([Oracle s.d.](#)).

Por fim, a implementação de uma abordagem Cloud Native fortalece as práticas de DevOps ao incentivar:

- O uso de entrega contínua (CD) e automação avançada.
- Desenvolvimento de aplicações containerizadas prontas para implantação, indepen-

dentemente da infraestrutura subjacente.

- Melhoria na qualidade e confiabilidade do software.
- Acaba sendo possível que a aplicação tenha múltiplas atualizações diárias sem afetar sua disponibilidade
- Maior flexibilidade e eficiência para empresas que buscam inovação constante e resposta ágil às demandas do mercado.

([Amazon s.d.](#)).

6.3 Comparação entre Microserviços e Monolitos

Para comparar microserviços e monolitos, especialmente no contexto de aplicações Cloud Native, é necessário entender as principais características de cada abordagem e como elas se aplicam a ambientes de nuvem, escalabilidade e manutenção.

6.3.1 Monolitos

Uma aplicação que contém toda a lógica dentro de si mesma é chamada de monolítica, geralmente consistindo em uma única base de código e implantada como uma unidade isolada, frequentemente executada dentro de um único processo ou contêiner ([Microsoft 2021](#); [Kanjilal 2020](#)). No entanto, ser monolítica não significa que uma aplicação não possa ser construída internamente a partir de diferentes componentes, como, por exemplo, ter camadas separadas para a interface de usuário e lógica de negócios. Aliás, nem todos esses componentes precisam fazer parte da aplicação em si; como qualquer outro modelo de aplicação, as aplicações monolíticas ainda podem utilizar bibliotecas externas.

Quando uma solução é pequena o suficiente ou não se espera que receba atualizações significativas após a implementação inicial, uma abordagem monolítica pode ser benéfica; afinal, projetar e desenvolver uma aplicação monolítica é mais rápido e fácil do que algumas das soluções mais complicadas ([Kanjilal 2020](#)). O gerenciamento de informações da aplicação, como configurações e autenticação, torna-se mais simples quando os dados relevantes estão prontamente disponíveis para todos os componentes dentro da base de código compartilhada. Devido à solução ser executada como uma única entidade, seus componentes geralmente compartilham memória; assim, não há necessidade de carregar ou buscar dados compartilhados em vários lugares, levando a um melhor desempenho geral nesse aspecto, ao contrário de soluções onde cada componente seria executado separadamente.

6.3.2 Microserviços

Embora a ideia central dos microserviços já exista há bastante tempo, o termo foi formalmente introduzido em 2005 por Dr. Peter Rodgers, que o chamou de "Micro-Web-

Services", durante uma conferência sobre computação em nuvem (Foote 2021). O termo "Microservice", mais direto e simplificado, foi proposto em 2011 por participantes de um workshop voltado para arquitetos de software, e seu uso foi consolidado no ano seguinte, quando os participantes sentiram que o termo descrevia de maneira mais precisa esse estilo arquitetural emergente.

O conceito de microserviço não foi criado ou definido por uma única organização ou grupo específico. Em vez disso, ele evoluiu ao longo do tempo, com diferentes interpretações surgindo de acordo com as necessidades e experiências de diversas equipes e especialistas. Isso torna a definição precisa de um microserviço mais desafiadora, já que muitos indivíduos e organizações têm suas próprias versões dessa definição.

Apesar das variações nas definições, há muitos elementos e conceitos comuns entre elas, com algumas dando maior ênfase a determinados aspectos do que outras. Por isso, pode-se entender o microserviço como a soma dessas diferentes definições, formando um panorama mais complexo quando todas são combinadas. Isso é feito ao reunir e integrar os aspectos chave presentes nas definições existentes, criando um resumo mais abrangente do que constitui um microserviço.

6.3.3 A Abordagem Cloud Native

Tanto a arquitetura monolítica quanto a arquitetura de microsserviços podem ser adotadas dentro do paradigma Cloud Native, porém, os microsserviços tendem a ser mais compatíveis com esse modelo, devido à sua natureza distribuída, escalável e focada em modularidade. O conceito de Cloud Native enfatiza a elasticidade, a automação e a agilidade, aspectos que estão fortemente alinhados com a arquitetura de microsserviços.

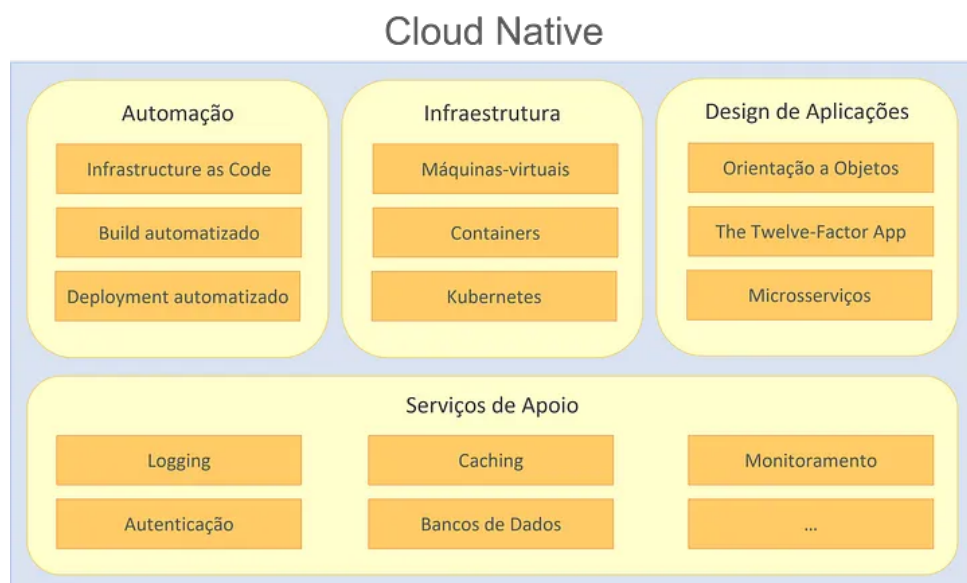


Figura 6.1 – Pilares em um projeto cloud native

Segundo ([Pikkumäki 2020](#)), em um contexto de desenvolvimento baseado em micro-serviços, cada serviço é autônomo e voltado para um propósito específico, o que facilita sua escalabilidade e flexibilidade, características essenciais em ambientes de nuvem. Isso se traduz em uma arquitetura altamente distribuída, onde os serviços podem ser escalados de maneira independente para atender à demanda de cargas de trabalho específicas. Essa independência facilita a utilização de containers e orquestração com ferramentas como o Kubernetes, além de permitir uma implementação mais ágil de pipelines de CI/CD (Integração Contínua e Entrega Contínua), otimizando o ciclo de vida do desenvolvimento e implantação de software.

Além disso, ao contrário da abordagem monolítica, que centraliza a lógica do sistema em uma única aplicação e requer esforços consideráveis para escalar e distribuir a carga, os microsserviços podem ser facilmente desacoplados e distribuídos entre diferentes nós, aproveitando a infraestrutura de nuvem para proporcionar escalabilidade dinâmica ([Pikkumäki 2020](#)). Isso se alinha perfeitamente com a abordagem Cloud Native, onde a automação da infraestrutura e a rápida adaptação às mudanças são fundamentais para garantir a agilidade e a resiliência das aplicações.

Por outro lado, a arquitetura monolítica, é menos flexível nesse contexto, já que as mudanças em um sistema monolítico geralmente exigem modificações em toda a aplicação, o que dificulta a implementação de práticas de automação e entrega contínua. Mesmo que seja possível adaptar uma aplicação monolítica ao paradigma Cloud Native, a flexibilidade, a escalabilidade e a independência dos microsserviços fazem dessa abordagem uma escolha mais natural e eficiente para ambientes de nuvem.

6.4 Monitoramento e Observabilidade

Observabilidade é a capacidade de entender o estado interno de um sistema a partir dos dados que ele gera, como logs, métricas e traces ([Livens 2021](#)). O conceito tem origem na Teoria de Controle, introduzida por Rudolf E. Kálmán em 1960, onde foi descrito como a capacidade de medir um sistema apenas por suas saídas. Embora o termo tenha se popularizado recentemente no DevOps, sua aplicação é essencial para que equipes de operações atinjam seus objetivos de serviço, reduzam o tempo de reparo e aumentem o intervalo médio entre falhas. Para desenvolvedores, a observabilidade é uma ferramenta crucial para depuração e aprimoramento da robustez dos serviços, fornecendo uma visão abrangente do sistema ([Crowdstrike 2021](#)).

6.4.1 Pilares da Observabilidade

Com o avanço da tecnologia e a necessidade de diferenciar os tipos de dados analisados, a observabilidade passou a ser dividida em três pilares principais ([Han 2019](#)):

6.4.1.1 Métricas

Representam medições numéricas de características do sistema em um período de tempo. Por exemplo, uso médio de CPU por minuto ou quantidade de requisições com erro por dia. As métricas podem ser coletadas de infraestrutura, balanceadores de carga e aplicações, permitindo identificar padrões de desempenho e anomalias.

6.4.1.2 Logs

São registros que indicam em que ponto do código uma requisição foi processada e se ocorreram falhas ou comportamentos inesperados. Além disso, logs podem armazenar informações de tentativas de acesso (como em logs de autenticação) e podem ser gerados tanto por aplicações quanto pelo sistema operacional (exemplo: syslog ou Windows Event Log).

6.4.1.3 Traces

Conhecidos como rastreamento distribuído, os traces registram o percurso das requisições dentro de um sistema distribuído, como arquiteturas baseadas em microsserviços ou serverless. Eles permitem diagnosticar gargalos de desempenho e identificar onde uma requisição está sendo processada de forma mais lenta. Tracing é essencial para depuração de sistemas distribuídos, pois facilita a identificação de problemas não determinísticos e difíceis de reproduzir localmente. Um trace é composto por múltiplos spans, sendo que o span raiz representa a requisição principal e os spans subsequentes detalham as operações internas envolvidas no processo.

Além dos três pilares principais, eventos também podem ser utilizados para complementar a observabilidade. Eventos registram ações específicas do sistema, como a execução de uma função administrativa, a atualização de um banco de dados ou a ocorrência de uma exceção no código. Eles são dados imutáveis, que podem ser analisados ao longo do tempo para identificar padrões. Eventos podem ser registrados em formatos plaintext, estruturado ou binário e podem, inclusive, ser considerados uma forma de log estruturado (O'Reilly n.d.).

6.4.2 Monitoramento Tradicional vs. Observabilidade

Com o crescimento das arquiteturas baseadas em nuvem e a complexidade dos sistemas modernos, as empresas perceberam a necessidade de ir além do monitoramento tradicional. O monitoramento, historicamente, tem sido uma solução baseada em métricas e logs predefinidos, permitindo que equipes configurem dashboards e alertas para detectar anomalias no sistema. Embora essa abordagem funcione, ela parte da premissa de que as equipes já sabem antecipadamente quais métricas e logs são relevantes para monitoramento (Livens 2021).

A observabilidade, por outro lado, permite uma análise mais aprofundada do sistema, indo além das métricas conhecidas. Enquanto o monitoramento responde à pergunta “o sistema está funcionando?”, a observabilidade permite questionar “por que o sistema não está funcionando?”. Isso significa que, em vez de depender apenas de métricas predefinidas, equipes podem explorar propriedades e padrões inesperados no comportamento do sistema, facilitando a detecção de falhas e a resolução proativa de problemas ([Google n.d.](#)).

Em resumo, a observabilidade é uma evolução do monitoramento, proporcionando uma visão mais ampla e detalhada do sistema. Ao combinar métricas, logs, traces e eventos, ela permite que as equipes compreendam melhor seus sistemas, otimizem seu desempenho e resolvam problemas de forma mais eficaz.

7 Minikube (Extra)

Levando em conta que um cluster do Kubernetes pode ser implementado tanto em máquinas físicas quanto virtuais, temos como uma das alternativas para rodar localmente o Minikube. Ele é uma implementação leve do Kubernetes que cria uma VM na máquina local e implanta um cluster simples contendo apenas um nó. Sendo uma ótima maneira de se ambientar com o desenvolvimento de aplicações para o Kubernetes, consegue fornecer operações básicas de inicialização para trabalhar com seu cluster, incluindo iniciar, parar, status e excluir (Kubernetes 2023).

Os pré-requisitos da máquina local para o Minikube funcionar corretamente são:

- 1 2 CPUs or more
- 2 2GB of free memory
- 3 20GB of free disk space
- 4 A good Internet connection
- 5 Container or virtual machine manager, such as [Docker](#), [Hyperkit](#), [Hyper-V](#), [KVM](#), [Parallels](#), [Podman](#), [VirtualBox](#), or [VMWare](#). Ensure you install any of the tools before you start with Minikube installation.

Figura 7.1 – Requisitos para o Minikube

7.1 Configuração e Instalação do Minikube

O exemplo abaixo leva em consideração a utilização do Ubuntu e Docker. Além disso, foi utilizado como base um guia para a instalação do Minikube: <https://devopscube.com/kubernetes-minikube-tutorial/> (Devopscube 2024).

7.1.1 Instalar o Docker

O Docker é necessário para rodar o Minikube. Siga a documentação oficial para instalá-lo no sistema operacional utilizado: <https://docs.docker.com/engine/install/ubuntu/>

7.1.2 Instalar o Minikube

```
1 curl -LO https://github.com/kubernetes/minikube/releases/latest/download/minikube-linux-amd64
2 sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd64
```


7.1.3 Iniciar o Minikube

```
1 minikube start --nodes 3 --driver=docker
```

```
victordsc@worker3:~/Documentos/pspd_final/mpi$ minikube start --nodes 3 --driver=docker
minikube v1.35.0 on Ubuntu 22.04
Using the docker driver based on user configuration
Using Docker driver with root privileges
Starting "minikube" primary control-plane node in "minikube" cluster
Pulling base image v0.0.46 ...
minikube was unable to download gcr.io/k8s-minikube/kicbase:v0.0.46, but successfully downloaded docker.io/kicbase/stable:v0.0.46@sha256:fd2d445ddcc33ebc5c6b68a17e6219ea207ce63c00
5095ea1525296da2d1a279 as a fallback image
Creating docker container (CPUs=2, Memory=2200MB) ...
Preparing Kubernetes v1.32.0 on Docker 27.4.1 ...
  Generating certificates and keys ...
  Booting up control plane ...
  Configuring RBAC rules ...
Configuring CNI (Container Networking Interface) ...
Verifying Kubernetes components...
  Using image gcr.io/k8s-minikube/storage-provisioner:v5
Enabled addons: storage-provisioner, default-storageclass
Starting "minikube-m02" worker node in "minikube" cluster
Pulling base image v0.0.46 ...
Creating docker container (CPUs=2, Memory=2200MB) ...
Found network options:
  NO_PROXY=192.168.49.2
Preparing Kubernetes v1.32.0 on Docker 27.4.1 ...
  env NO_PROXY=192.168.49.2
Verifying Kubernetes components...
Starting "minikube-m03" worker node in "minikube" cluster
Pulling base image v0.0.46 ...
Creating docker container (CPUs=2, Memory=2200MB) ...
Found network options:
  NO_PROXY=192.168.49.2,192.168.49.3
Preparing Kubernetes v1.32.0 on Docker 27.4.1 ...
  env NO_PROXY=192.168.49.2
  env NO_PROXY=192.168.49.2,192.168.49.3
Verifying Kubernetes components...
kubectl not found. If you need it, try: 'minikube kubectl -- get pods -A'
Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

Figura 7.2 – Minikube iniciado

7.1.4 Verificar os Nós do Cluster

Opção Temporária (Sem alterar configurações permanentes):

```
1 minikube kubectl -- get nodes
```

Opção Persistente (Criando um alias para facilitar o uso do kubectl):

```
1 echo 'alias kubectl="minikube kubectl --"' >> ~/.bashrc
2 source ~/.bashrc
3
4 kubectl get nodes
```

```
victordsc@worker3:~/Documentos/pspd_final/mpi$ kubectl get nodes
NAME           STATUS    ROLES           AGE      VERSION
minikube       Ready     control-plane   6m34s    v1.32.0
minikube-m02   Ready     <none>          6m7s     v1.32.0
minikube-m03   Ready     <none>          5m45s    v1.32.0
```

Figura 7.3 – Nodes Ready

7.1.5 Abrir o Dashboard do Minikube

```
1 minikube dashboard
```

7.1.6 Instalar o Metrics Server

O Metrics Server permite monitorar o uso de CPU e memória dos pods. Para instalá-lo, execute:

```
1 kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases
  /latest/download/components.yaml
```

Para verificar se os pods do Metrics Server estão rodando:

```
1 kubectl get pods -n kube-system
```

7.1.7 Removendo o Minikube (Opcional)

Caso queira remover qualquer cluster existente e reiniciar a configuração do Minikube, execute os seguintes comandos:

```
1 minikube delete
2 minikube start --nodes 3 --driver=docker
```

7.2 Justificativa para a não utilização

O grupo optou por não utilizar o Minikube, seguindo pelo método tradicional de configurar manualmente o master e workers em máquinas físicas interligadas. A escolha da abordagem escolhida acabou levando a mais percalços e desafios, porém trouxe a uma experiência mais enriquecedora para todos os membros, na qual acompanharam e debateram entre si durante todo o processo. Muitas das etapas são abstraídas ao utilizar o Minikube, e ao configurarmos manualmente foi possível ter uma visão mais específica de certas partes.

8 Conclusão e Autoavaliação

8.1 Conclusão Geral

Neste trabalho, realizamos a configuração e execução de diferentes aplicações em um ambiente Kubernetes isolado, explorando a integração com Spark, OpenMP e MPI. Através de experimentos práticos, testamos a resiliência do cluster, a eficiência de processamento distribuído e a tolerância a falhas em diferentes cenários.

Inicialmente, implementamos a execução distribuída do cálculo de π utilizando Spark, onde validamos que, mesmo com a falha de um worker durante a execução, o processamento pôde continuar sem interrupções, demonstrando a robustez do modelo de execução distribuída. Em seguida, analisamos a performance e a escalabilidade de uma aplicação de contagem de palavras em um cluster Kubernetes, garantindo que o ambiente pudesse lidar eficientemente com cargas variáveis de trabalho.

Além disso, realizamos a integração com OpenMP e MPI, visando explorar modelos híbridos de paralelismo. A configuração e instalação desses frameworks foram detalhadas, permitindo a replicação dos testes e a análise comparativa entre as abordagens.

Os resultados obtidos evidenciam que Kubernetes, aliado a tecnologias de computação distribuída, oferece um ambiente flexível e resiliente para processamento paralelo e execução de workloads intensivos. As simulações realizadas demonstraram como a arquitetura distribuída pode garantir alta disponibilidade e tolerância a falhas, reduzindo o impacto de eventos inesperados no cluster.

Dessa forma, concluímos que a utilização de Kubernetes, Spark e MPI, combinada com técnicas de paralelismo como OpenMP, apresenta uma abordagem eficiente para processamento distribuído. O ambiente configurado permite escalar aplicações de maneira dinâmica e garantir a continuidade da execução mesmo diante de falhas, sendo uma solução viável para aplicações que exigem alto desempenho e confiabilidade.

8.2 Depoimentos dos Alunos Participantes

8.2.1 Heitor Marques Simões Barbosa

Minha principal contribuição no trabalho foi na configuração e execução do Apache Spark dentro do cluster Kubernetes. Fiquei responsável pelo worker 2, garantindo que estivesse configurado corretamente e funcional. Achei desafiador integrar o Spark ao ambiente distribuído, mas foi uma experiência valiosa. Também participei ativamente das reuniões

presenciais e virtuais, discutindo soluções e ajudando a resolver problemas ao longo do desenvolvimento.

Um dos maiores desafios foi lidar com o firewall da faculdade, que bloqueava algumas comunicações necessárias para o Spark funcionar corretamente. Isso nos obrigou a buscar soluções alternativas e testar diferentes abordagens para contornar a limitação. Além disso, a necessidade de reconfigurar o cluster toda vez que mudávamos de sala dificultava a continuidade do trabalho e exigia muito tempo extra.

8.2.2 José Luís Ramos Teixeira

Durante o trabalho, fiquei encarregado de organizar a documentação e estruturar os guias de instalação, algo que considero fundamental para a reprodução do nosso experimento. Organizei o repositório no GitHub, mantive as informações centralizadas no Overleaf e cuidei da configuração do nó master do cluster. Essa parte foi desafiadora, pois exigiu um bom planejamento e ajustes constantes. Participei das reuniões tanto presencialmente quanto por vídeo para garantir alinhamento com o grupo.

Inicialmente, enfrentamos um grande problema com a versão 1.32.2 do Kubernetes, pois a rede caía constantemente e tornava a comunicação entre os nós instável. Após testar diferentes versões, conseguimos mais estabilidade ao utilizar a versão 1.30.10, o que melhorou significativamente o desempenho. No entanto, até chegarmos a essa solução, perdemos muito tempo lidando com quedas frequentes, o que impactou o andamento do projeto. Além disso, a necessidade de conciliar os horários de todos para reuniões presenciais foi um desafio extra, especialmente quando precisávamos testar configurações específicas.

8.2.3 Pablo Christianto Silva Guedes

Meu foco foi no desenvolvimento e integração do kubernetes isolado, além de soluções de paralelismo utilizando OpenMP e MPI dentro do cluster Kubernetes. Trabalhei bastante de forma independente para testar a viabilidade dessas abordagens e garantir que pudessem rodar de forma eficiente no ambiente distribuído. Foi um desafio interessante, pois exigiu ajustes finos na configuração e execução dos testes. Apesar de ter me concentrado mais nessa parte isolada, estive presente em discussões e reuniões para compartilhar os avanços.

O maior obstáculo foi adaptar a implementação para rodar no cluster Kubernetes, já que o comportamento de MPI e OpenMP não é tão direto em um ambiente distribuído como esperávamos. Além disso, a necessidade de remontar a infraestrutura a cada troca de sala nos forçou a repensar a forma de automatizar essa configuração, para evitar desperdício de tempo.

8.2.4 Philipe de Sousa Barros

Minha contribuição principal foi na documentação do projeto, tanto no Overleaf quanto na parte de organização dos desafios que enfrentamos. Também ajudei a estruturar a seção sobre Cloud Native, levantando informações importantes sobre a temática. Trabalhar sem um cabo de rede foi um obstáculo, mas consegui contornar isso e participar ativamente das discussões presenciais e por vídeo, garantindo que minha parte estivesse bem alinhada com as necessidades do grupo.

Algo que tornou o trabalho mais complicado foi a falta de um ambiente fixo para trabalharmos na faculdade. Cada vez que precisávamos nos reunir, era necessário encontrar um novo local e remontar tudo. Também percebi que a documentação exigia um esforço contínuo para ser mantida atualizada, já que o projeto evoluía rapidamente e algumas partes precisavam ser revisadas frequentemente.

8.2.5 Victor de Souza Cabral

Atuei principalmente na documentação adicional sobre Cloud Native e elaborei um material complementar sobre Minikube para aprendizado do grupo. Também participei da configuração do worker 3, garantindo que estivesse operacional. As reuniões presenciais e as discussões por vídeo foram essenciais para trocarmos ideias e garantir que tudo estivesse bem documentado e testado.

Um problema recorrente foi a necessidade de reiniciar o cluster frequentemente devido a mudanças de sala. Isso tornou o processo um pouco cansativo, pois precisávamos sempre configurar tudo do zero. Também tivemos que lidar com algumas dificuldades de integração entre os diferentes componentes do Kubernetes, o que exigiu paciência e testes constantes para identificar os melhores ajustes.

8.3 Nota de Autoavaliação

A tabela a seguir apresenta a autoavaliação dos alunos participantes do projeto.

Tabela 8.1 – Autoavaliação dos Alunos Participantes

Aluno	Autoavaliação	Nota
Heitor Marques Simões Barbosa	Excelente	10
José Luís Ramos Teixeira	Excelente	10
Pablo Christiano Silva Guedes	Excelente	10
Philipe de Sousa Barros	Excelente	10
Victor de Souza Cabral	Excelente	10

Fonte: Elaboração própria.

Referências

- AMAZON, A. **O que é suporte nativo de nuvem?** s.d. Acesso em: 18 fev. 2025. Disponível em: <https://aws.amazon.com/pt/what-is/cloud-native/>. Citado na p. 26.
- CROWDSTRIKE. **Observability Redefined.** 2021. Retrieved from <https://www.humio.com/blog/observabilityredefined/>. Acesso em: 20 fev. 2025. Citado na p. 28.
- DEVOPSCUBE. **How To Install Minikube: Comprehensive Tutorial for Beginners.** 2024. Acesso em: 18 fev. 2025. Disponível em: <https://devopscube.com/kubernetes-minikube-tutorial/>. Citado na p. 31.
- FOOTE, K. D. **A Brief History of Microservices.** 2021. <https://www.dataversity.net/a-brief-history-of-microservices>. Retrieved from Dataversity. Citado na p. 27.
- GITHUB. **Documentação do GitHub.** 2025. Acesso em: 10 fev. 2025. Disponível em: <https://docs.github.com/>. Citado na p. 5.
- GITLAB. **O que é uma abordagem de nuvem nativa?** s.d. Acesso em: 18 fev. 2025. Disponível em: <https://about.gitlab.com/pt-br/topics/cloud-native/>. Citado nas pp. 24 e 25.
- GOOGLE. **Google Cloud.** n.d. Retrieved from <https://cloud.google.com>. Acesso em: 20 fev. 2025. Citado na p. 30.
- HAN, C. **3 Pillars of Observability.** 2019. Retrieved from <https://medium.com/hepsiburadatech/3-pillars-of-observability-d458c765dd26>. Acesso em: 20 fev. 2025. Citado na p. 28.
- KANJILAL, J. **Pros and cons of monolithic vs. microservices architecture.** 2020. <https://www.techtarget.com/searchapparchitecture/tip/Pros-and-cons-of-monolithic-vs-microservices-architecture>. Acesso em: 20 fev. 2025. Citado na p. 26.
- KUBERNETES. **Usando Minikube para criar um cluster.** 2023. Acesso em: 18 fev. 2025. Disponível em: <https://kubernetes.io/pt-br/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/>. Citado na p. 31.
- KUBERNETES. **Documentação do Kubernetes.** 2025. Acesso em: 10 fev. 2025. Disponível em: <https://kubernetes.io/docs/>. Citado na p. 5.
- LIVENS, J. **What is Observability?** 2021. Retrieved from <https://www.dynatrace.com/news/blog/what-is-observability-2/>. Acesso em: 20 fev. 2025. Citado nas pp. 28 e 29.
- MEDIUM. **O que é Cloud Native? E o que não é... | Arquitetura em Nuvem.** 2023. Acesso em: 18 fev. 2025. Disponível em: <https://renatogroffe.medium.com/o-que-%C3%A9-cloud-native-e-o-que-n%C3%A3o-%C3%A9-arquitetura-em-nuvem-8ff9e26a9d5c>. Citado na p. 25.
- MICROSOFT. **Monolithic applications.** 2021. <https://docs.microsoft.com/en-us/dotnet/architecture/containerized-lifecycle/designdevelop-containerized-apps/>

[monolithic-applications](#). Acesso em: 20 fev. 2025. Citado na p. 26.

OPENMP. **Especificação do OpenMP**. 2025. Acesso em: 10 fev. 2025. Disponível em: <https://www.openmp.org/specifications/>. Citado na p. 5.

ORACLE. **O que é Nativo em Nuvem?** s.d. Acesso em: 18 fev. 2025. Disponível em: <https://www.oracle.com/br/cloud/cloud-native/what-is-cloud-native/>. Citado nas pp. 24 e 25.

O'REILLY. **Distributed Systems Observability - Chapter 4**. n.d. Retrieved from <https://www.oreilly.com/library/view/distributedsystems-observability/9781492033431/ch04.html>. Acesso em: 20 fev. 2025. Citado na p. 29.

PIKKUMÄKI, T. **Comparison of Monolithic, Micro-service, and Cloud development**. 2020. Retrieved from <https://www.theseus.fi/handle/10024/795180>. Acesso em: 20 fev. 2025. Citado na p. 28.

SPARK, A. **Documentação do Apache Spark**. 2025. Acesso em: 10 fev. 2025. Disponível em: <https://spark.apache.org/docs/latest/>. Citado na p. 5.



UnB