Containers

Do jeito certo

:)

Guto Carvalho

Platform Engineer

gutocarvalho.net

Guto Carvalho

Especialista em Infra as Code e Cloud Native. Foi um dos primeiros profissionais a escrever e falar sobre DevOps e automação (IaC) no Brasil.

- Containers
- Docker
- Cloud Native
- Curiosidades
- Dúvidas

Add

Add Your S

Por gentileza coloque seus dispositivos em modo silencioso.

Entendendo os Containers

Introdução

Entendendo containers

Para que servem?

Essencialmente os containers servem para que possamos isolar processos em um sistema operacional.

Em nosso caso aqui falaremos de containers que rodam em sistemas operacionais Linux.

Entendendo containers

Para que servem?

Antes de entender containers, vamos entender alguns princípios fundamentais da virtualização clássica.

Entendendo containers

O container é um tipo de virtualização?

Podemos dizer que sim, uma virtualização leve parecida com a **para-virtualização**.

Temos duas técnicas de virtualização bem características são elas a **para-virt** e a **full-virt**.

Na virtualização clássica temos o que chamamos de HOSPEDEIRO (**host**) e CONVIDADO (**guest**).

O sistema hospedeiro (**host**) é quem executa do software de virtualização, também chamado de "**hypervisor**".

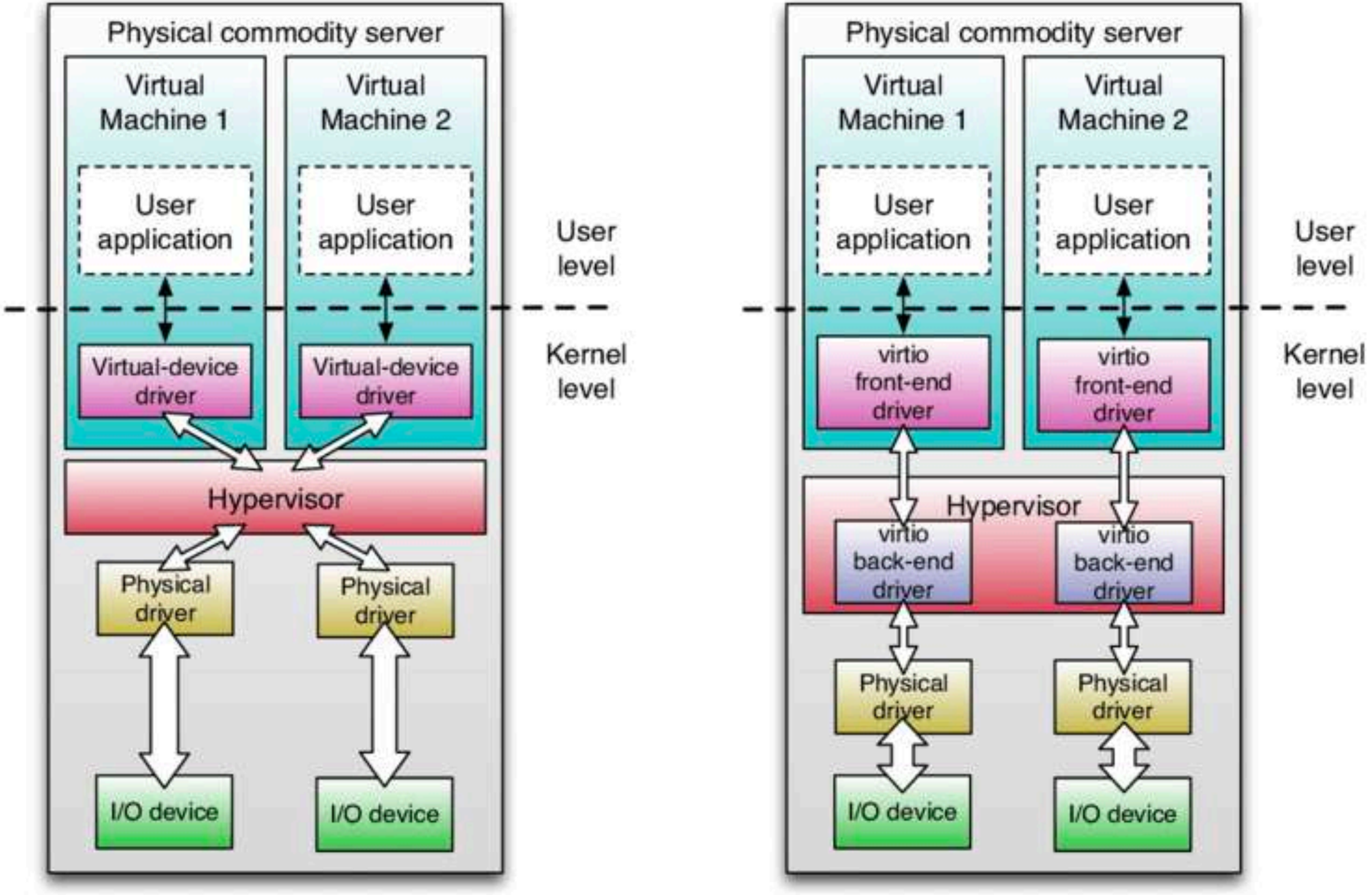
O sistema convidado (**guest**) é o sistema operacional que será executado dentro do hospedeiro (**host**).

Entendendo containers

O container é um tipo de virtualização?

Na para-virtualização (**para-virt**) o sistema operacional virtualizado usa um kernel modificado que consegue se comunicar de forma mais livre com o hypervisor, podendo até acessar alguns componentes do hardware diretamente.

Na virtualização completa (**full-virt**) o sistema operacional convidado (**guest**) utiliza um kernel normal que se comunica apenas com um hardware emulado pelo hypervisor, ele não fala diretamente com nenhum componente do hardware do hospedeiro (**host**).



(a) Full-virtualization

(b) Paravirtualization

Entendendo containers

Todos na mesma página?

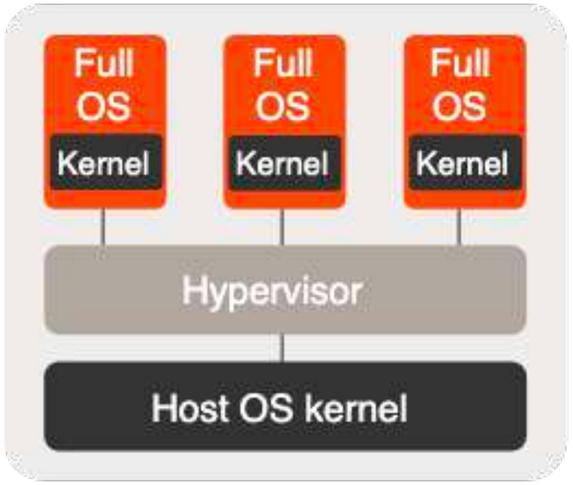
Conceitos estabelecidos?

Vamos lá!

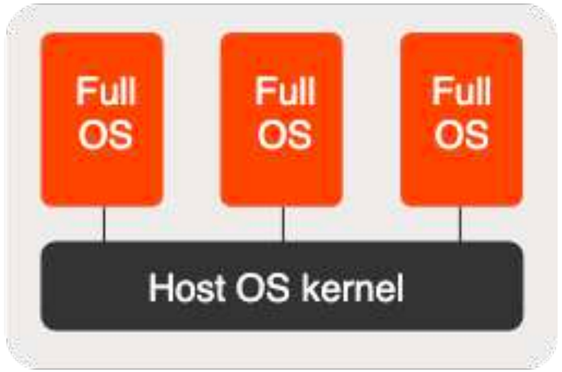
Entendendo containers

O container é um tipo de virtualização?

No uso de containers não temos um Hypervisor, o kernel linux se encarrega de isolar o processo utilizando recursos nativos.



Virtual machines



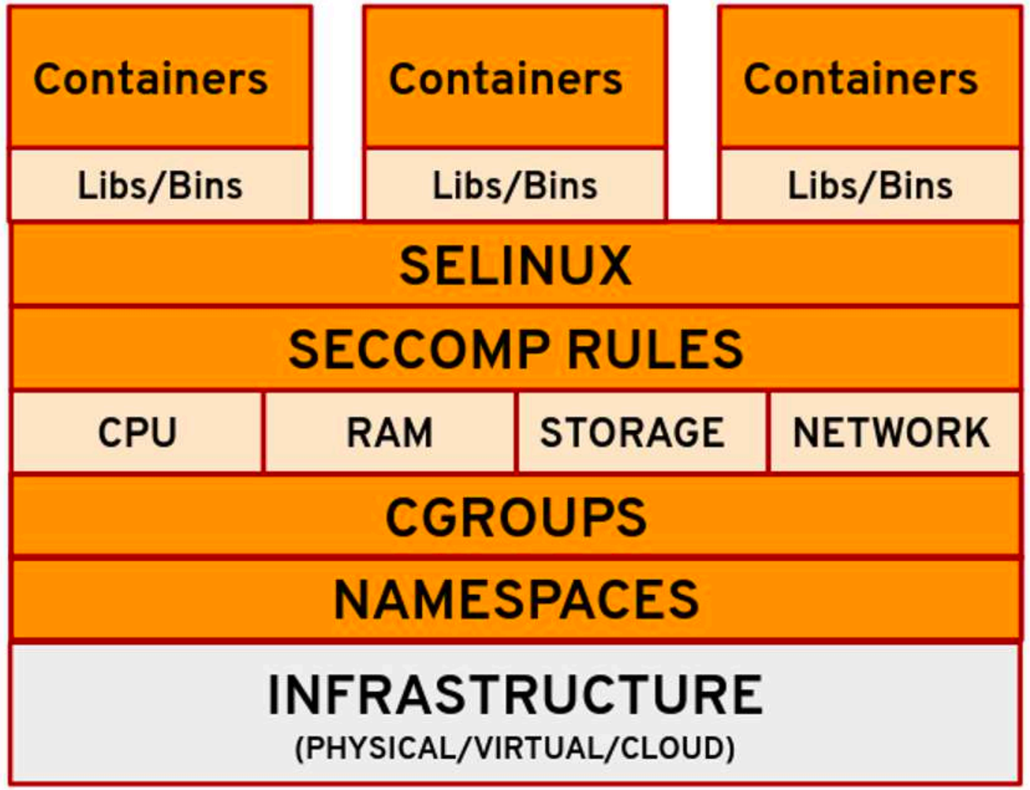
System containers

Entendendo containers

O container é um tipo de virtualização?

Os containers no linux são compostos por:

Namespaces
CGROUPS
SecComp
SELinux ou AppArmour



source image
https://opensource.com/article/21/8/container-linux-technology#:~:text=interface%20for%20communication.-,Control%20groups%20(cgroups),when%20the%20container%20is%20run.

Entendendo containers

O container é um tipo de virtualização?

Os **Namespaces** atuam no isolamento do container, eles dão ao container uma visão de um filesystem. Isso vai limitar o que um processo pode ver e os recursos que ele pode ou deve acessar.

Dentro dos namespaces podemos trabalhar com:

Users
Filesystem/Mount
Hostname
Processes
Network

Entendendo containers

O container é um tipo de virtualização?

O **CGROUPS** vai nos ajudar a definir os limites para os recursos utilizados por um containers.

Quais são esses limites?

CPU
Memória
Network I/O

Entendendo containers

O container é um tipo de virtualização?

O **SECCOMP** vai limitar as chamadas de sistema (syscalls) que um container pode fazer.

Entenda que no sistema operacional linux existem centenas de chamadas de sistemas que podem ser feitas, temos mais de 300 tipos de syscals.

O SECCOMP vai filtrar quais chamadas de sistemas o container pode fazer e liberar apenas aquelas necessárias para que ele funcione.

O SECCOMP funciona com conjuntos de configurações que se chamam profiles, cada tecnologia de containers tem seu próprio profile e syscalls liberados.

Entendendo containers

O container é um tipo de virtualização?

Existe ainda uma camada de segurança envolvida no uso de containers, essa camada pode ser o SELINUX ou APPArmour.

Geralmente o SELINUX é nativo em ambiente RedHat Like e o APPArmour de ambientes Debian-Like como ubuntu por exemplo.

Essa camada de segurança protege o sistema operacional **host** em caso do comprometimento do mesmo através de uma aplicação em execução.

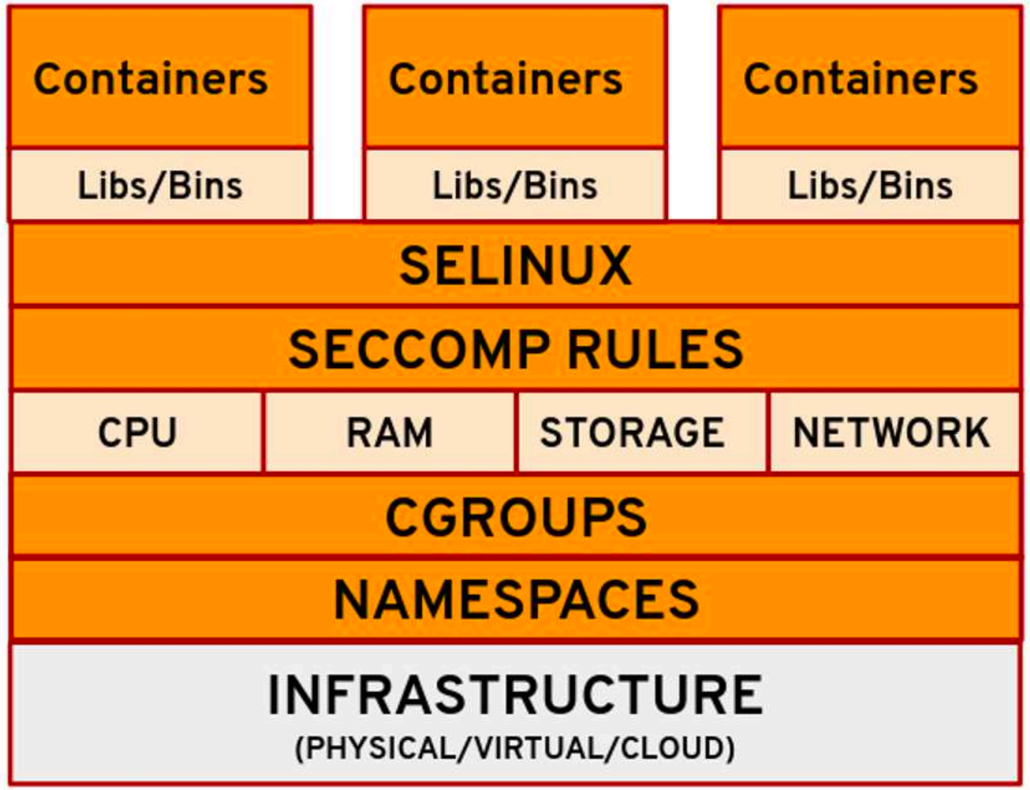
Isso significa que caso um APP seja comprometida, ainda assim ela não irá conseguir alterar ou modificar partes sensíveis do sistema operacional.

Entendendo containers

O container é um tipo de virtualização?

Revisando!

Namespaces
CGROUPS
SecComp
SELinux ou AppArmour



source image
https://opensource.com/article/21/8/container-linux-technology#:~:text=interface%20for%20communication.-,Control%20groups%20(cgroups),when%20the%20container%20is%20run.

Entendendo containers

O container e um CHROOT são coisas parecidas?

São parecidos, mas diferentes.

No caso do CHROOT nós conseguimos isolar o filesystem e processos de uma forma mais rústica, contudo, só conseguimos limitar os processos em nível de filesystem, todo o resto será compartilhado como rede, usuário, hostname, ip, cpu, memória, etc.

O processo vai ver o ROOT que voce designou, vai achar que está realmente na raiz, mas você não vai controlar os recursos, é um isolamento com nível de controle e segurança muito menor.

Entendendo containers

O container e um CHROOT são coisas parecidas?

O Docker é um tipo de tecnologia de containers, vamos chegar lá, segura mais uns minutinhos ;)

Um pouco de história

Raizes dos containers

História dos containers

Entendendo onde tudo começo

O primeiro e mais rudimentar sistema de isolamento foi o CHROOT criado em 1979 para os sistemas operacionais UNIX V7. Como já explicamos ele isola o processo em um filesystem diferente, o processo acha que está na raiz do host, mas está em um lugar reservado.

Em 1999 o projeto FreeBSD (Sistema Unix-Like) introduziu o Jails, que era um alternativa ao CHROOT com mais recursos. O Jails permitia a **virtualização** de usuários, processos e também de recursos de network oferecendo um isolamento mais completo e seguro.

História dos containers

Entendendo onde tudo começo

Em 2001 foi lançado o projeto **VServer** para o kernel Linux. Ele conseguia isolar e particionar filesystems, network e memória. Esse foi um patch experimental para o kernel linux que foi mantido entre 2001 e 2006.

O sistema Unix Like **Solaris** também criou seu próprio sistema de containers no final de 2004, chamado de **Zones**, em conjunto com o ZFS ele provia isolamento de processos através de virtualização além de snapshots e clones desses containers.

História dos containers

Entendendo onde tudo começo

Em 2005 foi lançado talvez um dos mais conhecidos e utilizados sistemas de containers para **Kernel Linux** chamado de **OpenVZ** (Open Virtuzzo). Ele era um patch externo aplicado ao **Kernel Linux**. Foi muito utilizado por provedores de hospedagem na época e até hoje está disponível para uso.

Em 2006 para atender demandas internas o Google criou o um projeto chamado **Process Containers** com objetivo de isolar e limitar o uso de recursos como CPU, Memória, Disco e Rede para um conjunto de processos. Esse era um patch externo para o **Kernel Linux** que acabou sendo incorporado rebatizado de **CGROUPs**, sendo oficialmente lançado no kernel **2.6.24**.

História dos containers

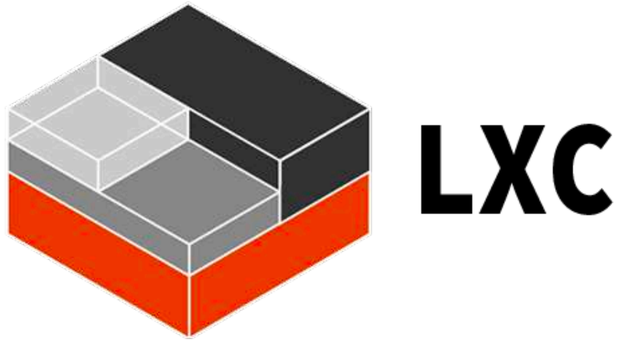
Entendendo onde tudo começo

Em 2008 foi lançado talvez o **LXC**, o primeiro projeto que implementou de forma decente o uso de containers utilizando CGROUPs.

Muitos provedores começaram a migrar do **OpenVZ** para **LXC** por este ser um projeto que usava CGROUPs diretamente.

O LXC segue sendo mantido e disponível.

Em 2010 foi criado o **LXD** um camada de abstração por **LXC** com uma experiência de usuário melhorada.



Era moderna dos containers

Entendendo a revolução Docker no mundo dos containers

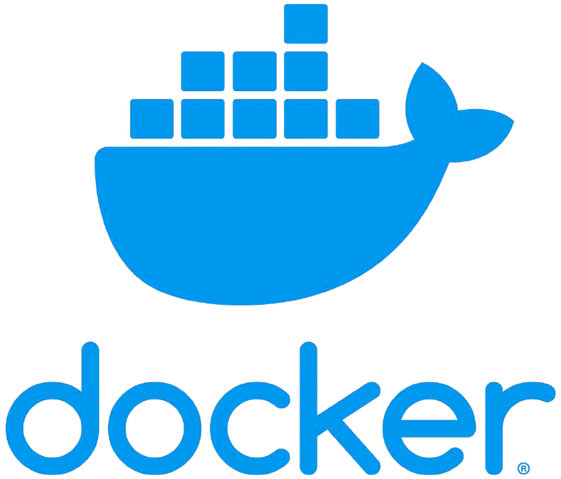
Docker

Revolução do ecossistema dos containers

O Docker simplificou o uso de containerairs dentro do sistema operacional Linux.

Ele trouxe um CLI amigável e fácil de usar, mas não apenas isso, ele trouxe um conceito de rodar sua aplicação de forma simples e em qualquer lugar.

Code > Build > Ship > Run



Docker

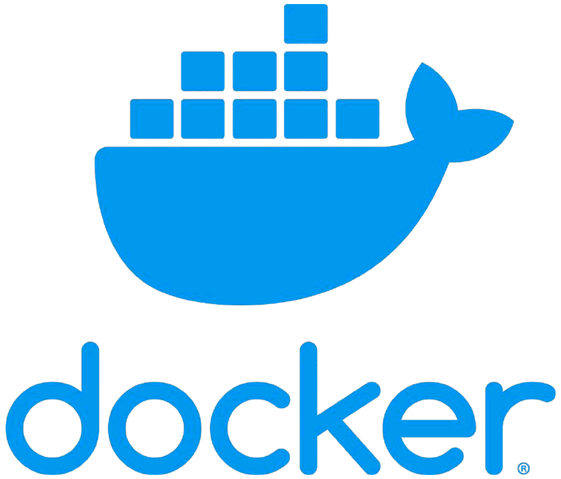
O que mudou?

Apenas um processo por container

Tudo que é preciso para rodar sua APP já está lá

Sua APP roda em qualquer lugar que suporte docker

Você executa sua APP com um único comando



Estratégia diferenciada

Entendendo o motivo do Docker ter ganhando o mercado

Estratégia

Entendendo o projeto Docker

Quando o Docker foi lançado em 2013, aquele era um momento em que ferramentas IaC de gerência de configurações e orquestração estavam se consolidando.

Ferramentas tais como Puppet, Chef, Salt traziam mecanismos de implementação de "gerência de estado" através de agentes (estratégia pull). Além destas, outras como o Ansible, traziam uma ideia similar mas usando a estratégia se de conectar diretamente para orquestrar e gerenciar nodes (estratégia push).

Era uma disputa acirrada com Puppet na frente nas ferramentas tipo PULL e o Ansible liderando nas ferramentas tipo PUSH.

Todos estavam investindo suas fichas nessas ferramentas.

Estratégia

Entendendo o projeto Docker

Essas ferramentas traziam algumas promessas importantes para operação:

➜ instalar e configurar aplicações de forma simples
➜ manter aplicações funcionando
➜ corrigir problemas a plataforma que roda as APPs
➜ corrigir problemas nas configurações das APPs
➜ criar novos ambientes de forma rápida simples
➜ retornar de falhas de forma mais rápida
➜ fazer deploy de forma mais rápida

E da mesma forma para os times de desenvolvimento:

➜ instalar ambiente do desenvolvedor
➜ instalar apps no ambiente do desenvolvedor

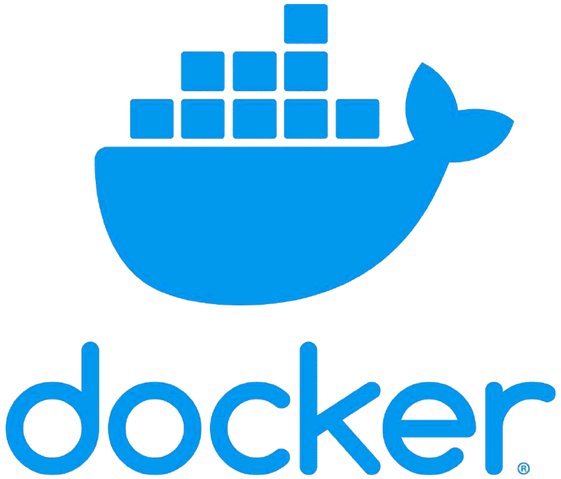
Estratégia

Entendendo o projeto Docker

Outra situação relevante na mesma época, era o movimento de adoção de infraestrutura em nuvem. Ambientes on-premisses estavam perdendo espaço devido ao alto custo de operação, então, além de manter estados, ferramentas de gerência de configuração começaram a se preocupar em manter estado na nuvem e até manter infra em nuvem, algo fora de seu escopo natural.

Foi nesse momento que Docker passou a fazer mais e mais sentido, afinal, se sua APP roda em Docker, e sua nuvem fala Docker, não haveria necessidade de ferramentas de gerência de estado, docker seria suficiente.

Para nuvem poderíamos usar algo mais especializado como **Terraform**, deixando toda a complexidade de rodar a APP dentro da imagem Docker que seria empurrada via Pipeline.



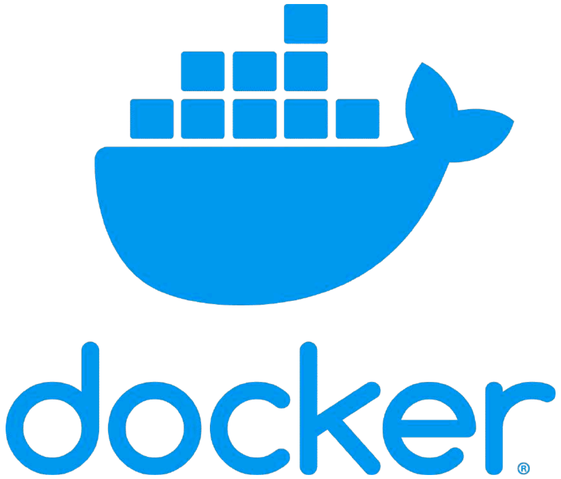
Estratégia

Entendendo o projeto Docker

Docker apareceu correndo por fora, trazendo ideias e conceitos divergentes das ferramentas de gerência de estado, gerando muita especulação e dúvidas.

A comunidade se dividiu e Docker virou um dos assuntos mais falados naquela época.

Em suma, enquanto ferramentas queriam manter estado e corrigir divergências de configurações (usando estratégias self-healing) o Docker queria ser mais simples e objetivo.



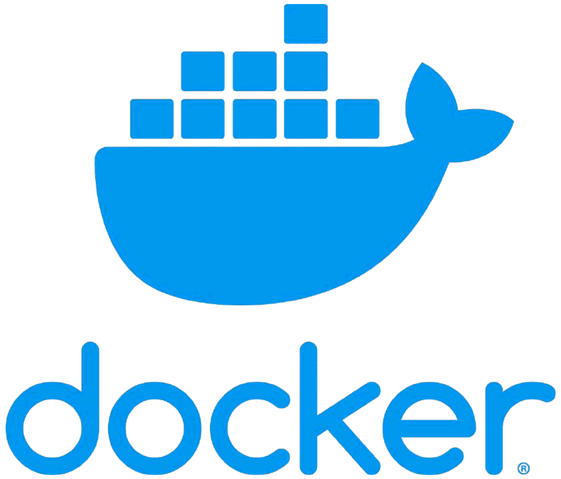
Estratégia

Entendendo o projeto Docker

A ideia do Docker era rodar a APP de forma isolada, um processo por container, e se houver um problema no container em execução, você tem apenas que remover o container com problema e executar outro igual.

A base destes containers é a mesma, uma imagem que contém a APP.

Essa imagem deve conter tudo que a APP precisa para ser executada, e isso basta.

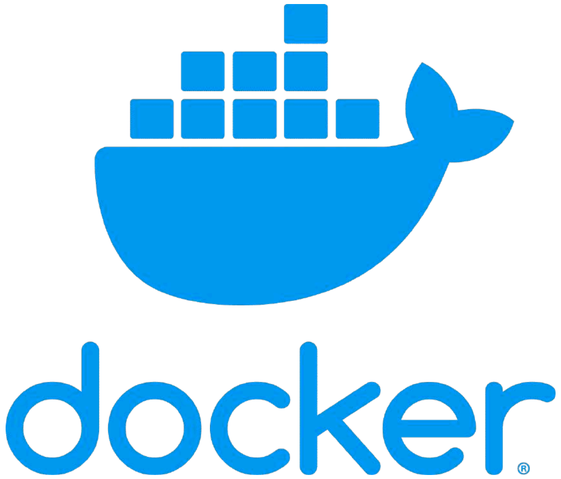


Estratégia

Entendendo o projeto Docker

A ideia é que a imagem seja construída pelo time responsável pelo produto, inserindo ali todas as dependências, configurações, ajustes e personalizações necessárias para que a aplicação funcione adequadamente.

A regra era clara, deu erro, deleta e sobe outro, não precisa corrigir, não precisa manter estado, você apenas verifica se está funcionando e caso não esteja, remove e coloca outro novo no lugar.



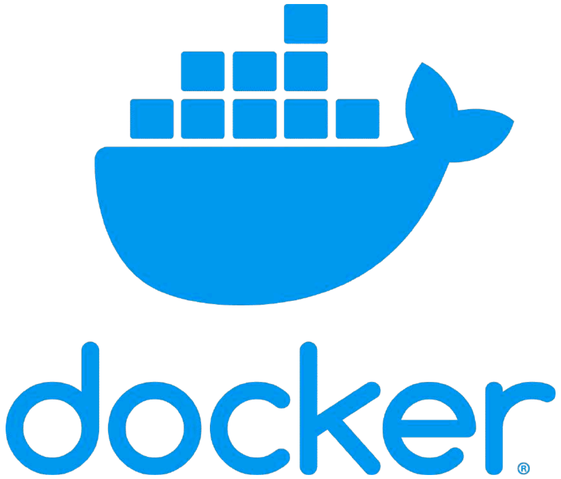
Estratégia

Entendendo o projeto Docker

Quem já estava acostumado com Gerência de Configuração, quem trabalhava mais com ambiente legado e onprem, levou algum tempo para adotar aceitar, acreditar aprender e realmente migrar para o modelo Docker.

Mas houve quem aproveitou o Docker desde seu início, uma vez que viviam em cenários que o uso do Docker era adequado, como projetos nascendo ou projetos que poderiam ser convertidos com uma certa facilidade para a lógica do Docker.

De qualquer forma o Docker acabou se provando uma abordagem diferente, algo que iniciou um novo capítulo na operação, administração e entrega de APPs, em especial na nuvem.



Estratégia

Entendendo o projeto Docker

Dentre os aspectos que mais conquistaram profissionais estavam:

➜ Uma vez feita a imagem, dá para rodar em qualquer provedor;

➜ Uma vez atualizada a imagem, basta substituir a antiga;

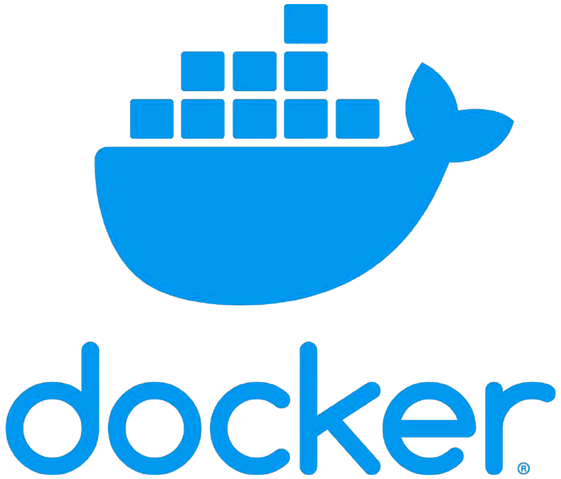
➜ A mesma imagem que roda no para o desenvolvedor, rodará na produção;

➜ A integração com pipelines é muito simples;

➜ Acelera o tempo de deploy incrivelmente;

➜ O servidor de produção só precisa ter Docker instalado e nada mais;

➜ Toda a complexidade está embutida na imagem.



Estratégia

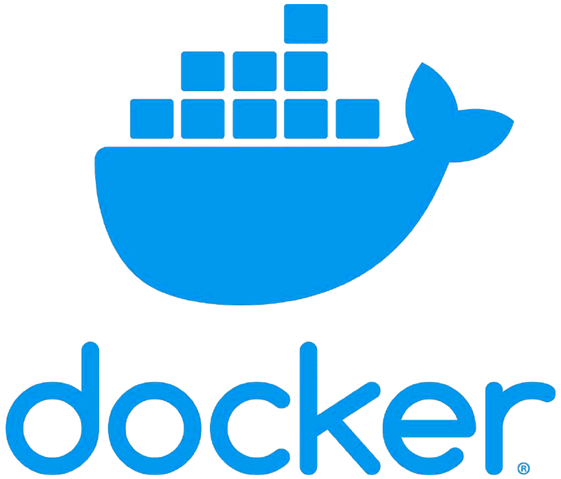
Entendendo o projeto Docker

Era o fim de situações desgastantes como:

➜ Funciona no meu notebook mas não funciona na produção
➜ Desenvolvedor sofrendo para colocar a APP para rodar em seu laptop
➜ Ambientes de Dev, Staging e Prod instalados de forma diferente
➜ Demora para criação ou reconstrução de novos ambientes

Algumas coisas pararam de fazer sentido:

➜ Rodar a APP em VMS
➜ Manter estado de VMs
➜ Manter estado de sua APP
➜ Usar gerência de configuração para fazer Deploy



Evoluções Docker

Entendendo as evoluções do projeto

Evoluções

Entendendo o projeto Docker

Ao longo dos anos o docker passou por melhorias diversas

➜ No início o Docker usava LXC internamente para gerenciamento dos containers

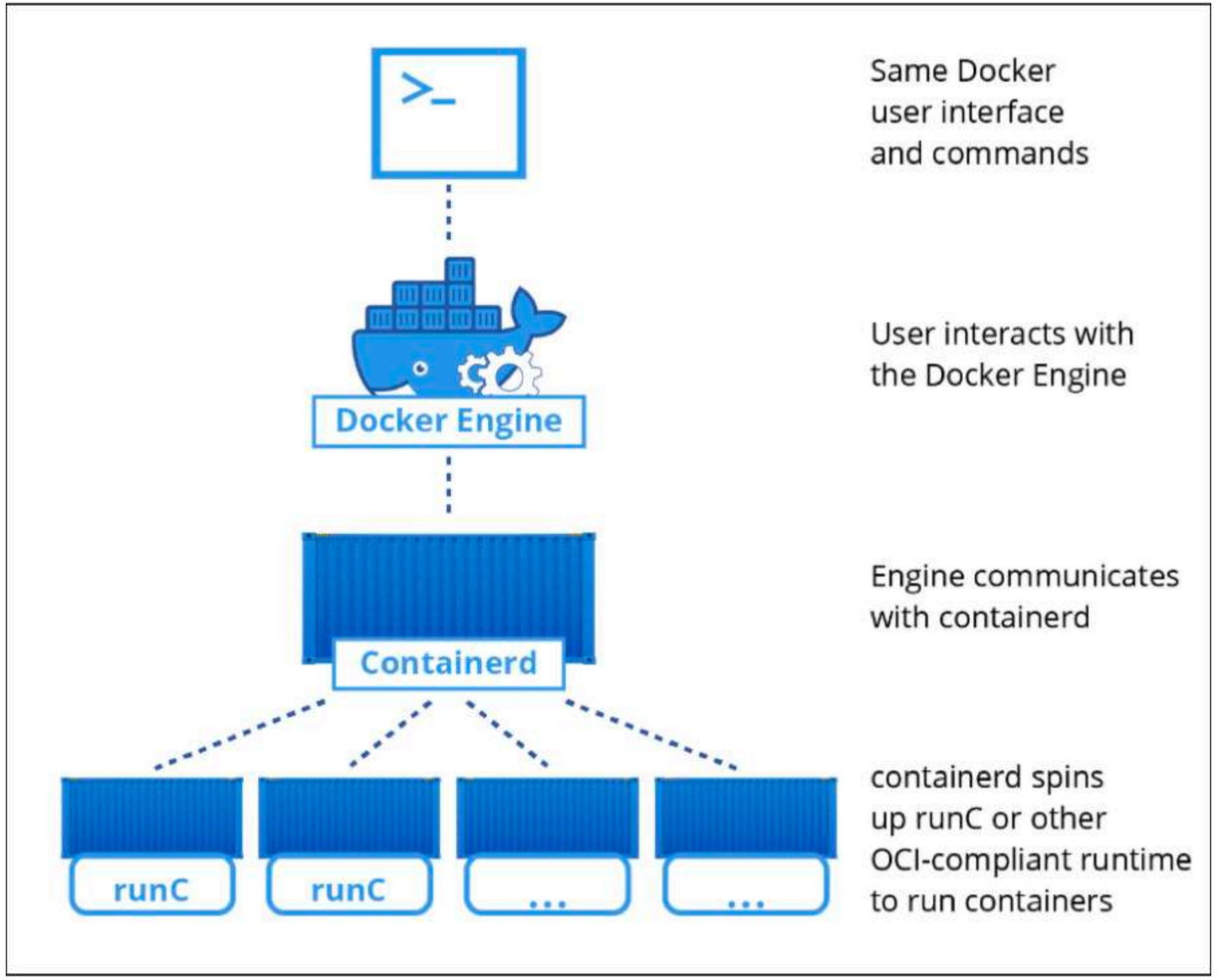
➜ Em 2014 criou o projeto **libcontainer** para usar **CGROUPS** diretamente sem necessidade do LXC

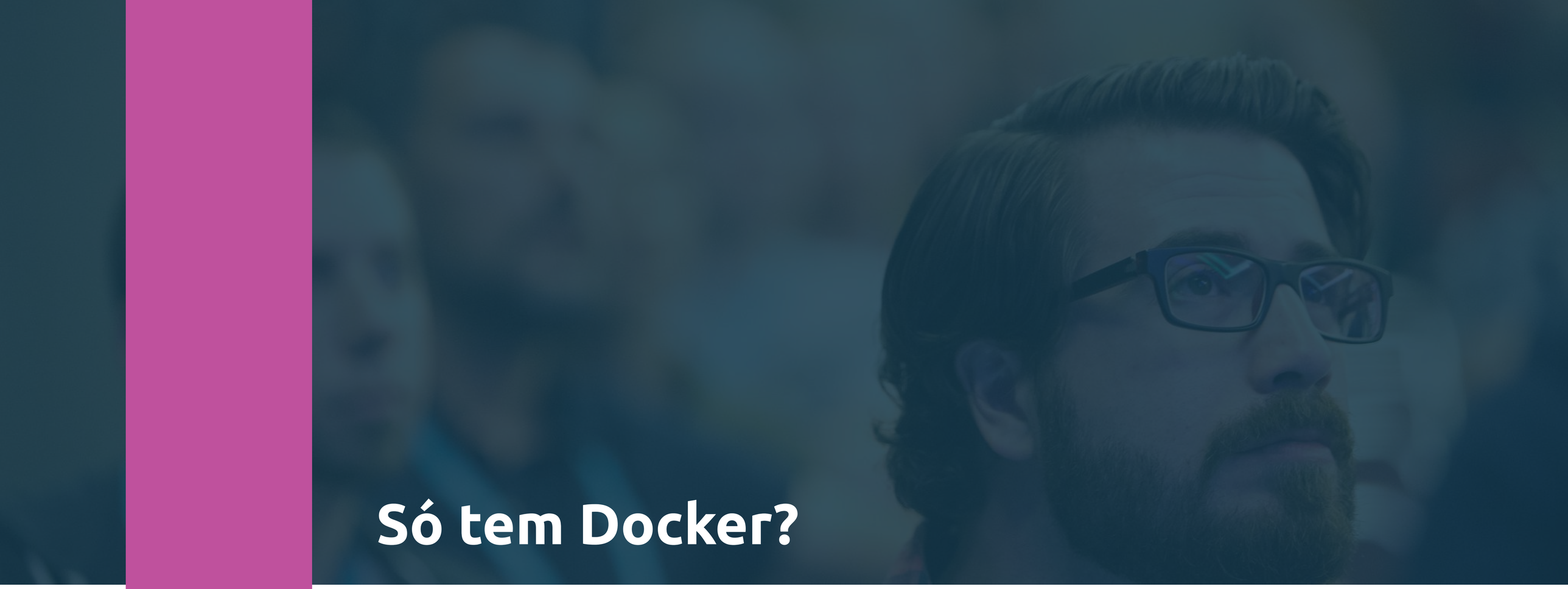
➜ Em 2015 eles desacoplaram alguns componentes e doaram parte do código para a CNFC

➜ O **libcontainer** por exemplo foi incubado na CNCF e virou o **RunC**

➜ Em 2016 a partir da versão 1.1, o Docker passou a usar **dockerd + containerd** + **runC**

E essas foram apenas as principais mudanças no projeto.


Same Docker
user interface
and commands
Docker Engine
User interacts with
the Docker Engine
Containerd
Engine communicates
with containerd
runC
runC
...
...
containerd spins
up runC or other
OCI-compliant runtime
to run containers



Só tem Docker?

Conhecendo alternativas e os padrões de mercado

Docker era único no início?

Entendendo o projeto Docker

Durante um tempo o Docker reinou sozinho, mas alternativas começaram a aparecer.

O primeiro a aparecer foi o **rkt** da **CoreOS** (empresa absorvida pela Redhat)

A comunidade começou a se envolver a cobrar que fosse criado um padrão para uso de containers, que permitisse o uso de diferentes **engines** com a mesma imagem/ dockerfile.

Inicialmente o projeto Docker não sinalizou uma abertura de padrões, mas foi vencido após a comunidade ameaçar a fazer um fork do projeto.

Todo esse movimento culminou na criação de uma iniciativa de padronização a OCI.

OCI significa Open Container Initiative

Open Container Initiative

Para que serve?

A OCI foi criada para definir um padrão para o uso de containers.

A especificação da OCI define padrões para criar imagens e executar containers.

Um padrão que permite que novos engines e runtimes sejam utilizados.

Um padrão permite que novas tecnologias de containers sejam desenvolvidas.

A vantagem disso tudo é que se você escrever o código de uma imagem seguindo os padrões OCI, essa imagem vai funcionar em qualquer CLI, Runtime ou Builder que siga as especificações da OCI.

Assim temos um padrão que não nos deixa amarrado a uma só ferramenta.

Com isso, temos um padrão que nos traz interoperabilidade e liberdade.

Alternativas atuais OCI

Quais são?

Image Build

➜ Buildah
➜ Buildkit
➜ Kaniko (Google)

Runtime Low Level

➜ runc (cncf)
➜ crun
➜ runsc (gvisor)
➜ runnc (nabla)

Runtime High Level

➜ rkt (redhat)
➜ railcar (oracle)
➜ mcr (mirantis)
➜ containerd

CLI/UI

➜ podman

K8S Runtimes

➜ CRI-O (cncf)
➜ containerd

Orquestração de containers

Necessidade natural para qualquer produção

Orquestrar para escalar

Entendendo containers em produção

O primeiro nível de orquestração para containers, em especial para Docker
foi o projeto Docker-compose.

Esse projeto permitiu configurar a criação de stacks inteiras para nossa aplicação, especificando diversos serviços e a forma como eles deveriam se comunicar e interagir.

Apesar do docker-compose ser um excelente projeto, em especial para rodar apps em pequenas empresas, o mercado de médias e grandes empresas sofria com a necessidade de um orquestração mais eficiente e robusta.

Orquestrar para escalar

Entendendo containers em produção

Quais as necessidades básicas de uma plataforma produtiva na internet?

➜ Alta disponibilidade

➜ Resiliência e tolerância a falhas

➜ Capacidade de escalar conforme demanda

➜ Facilidade modificar a versão de uma aplicação

Orquestrar para escalar

Entendendo containers em produção

Traduzindo as necessidades comuns

➜ Minha aplicação precisa rodar em diferentes hosts (VMs ou Metal)
➜ Preciso balancear o acesso entre os hosts e containers destes
➜ Caso meu container tenha um problema, ele deve ser substituído
➜ Preciso aumentar o número de containers com minha APP sob demanda
➜ Preciso diminuir o número de containers sob demanda
➜ Preciso de uma forma simples para atualizar a versão da minha APP em todos os hosts
➜ Preciso falar um rollout controlado da minha nova versão e combinar isso com o balanceador
➜ Caso um host pare, minha aplicação não pode parar
➜ Caso um container pare, minha aplicação não pode parar
➜ Preciso centralizar os logs da APP independente do host
➜ Preciso limitar o uso de CPU/MEM de um APP e isso deve valer para todos os hosts

Fora outras necessidades...

Orquestrar para escalar

Entendendo containers em produção

Como faço para ter isso?

Orquestrar para escalar

Entendendo containers em produção

Primeira geração de orquestradores

Docker Swarm
Rancher 1.x

Segunda geração de orquestradores

Kubernetes
Openshift
Rancher 2.x

Esse são alguns dos projetos mais relevantes, IMHO.

Antes de continuar....

Vamos fazer uma pausa estratégica para falar e entender o que é

Cloud Native & Kubernetes!

Cloud Native Computing Foundation

Nova era dos Containers na Nuvem impulsionada pela CNCF

Entendendo a CNCF

Evolução do Container para a Nuvem

O que é?

Iniciativa ligada a Linux Foundation que hospeda diversos projetos
ligados a computação em nuvem e infraestrutura de serviços

Qual seu objetivo?

Incubar projetos que são críticos do ponto vista de
de infraestrutura global de tecnologia, com grande foco em nuvem.

Qual seu alcance?

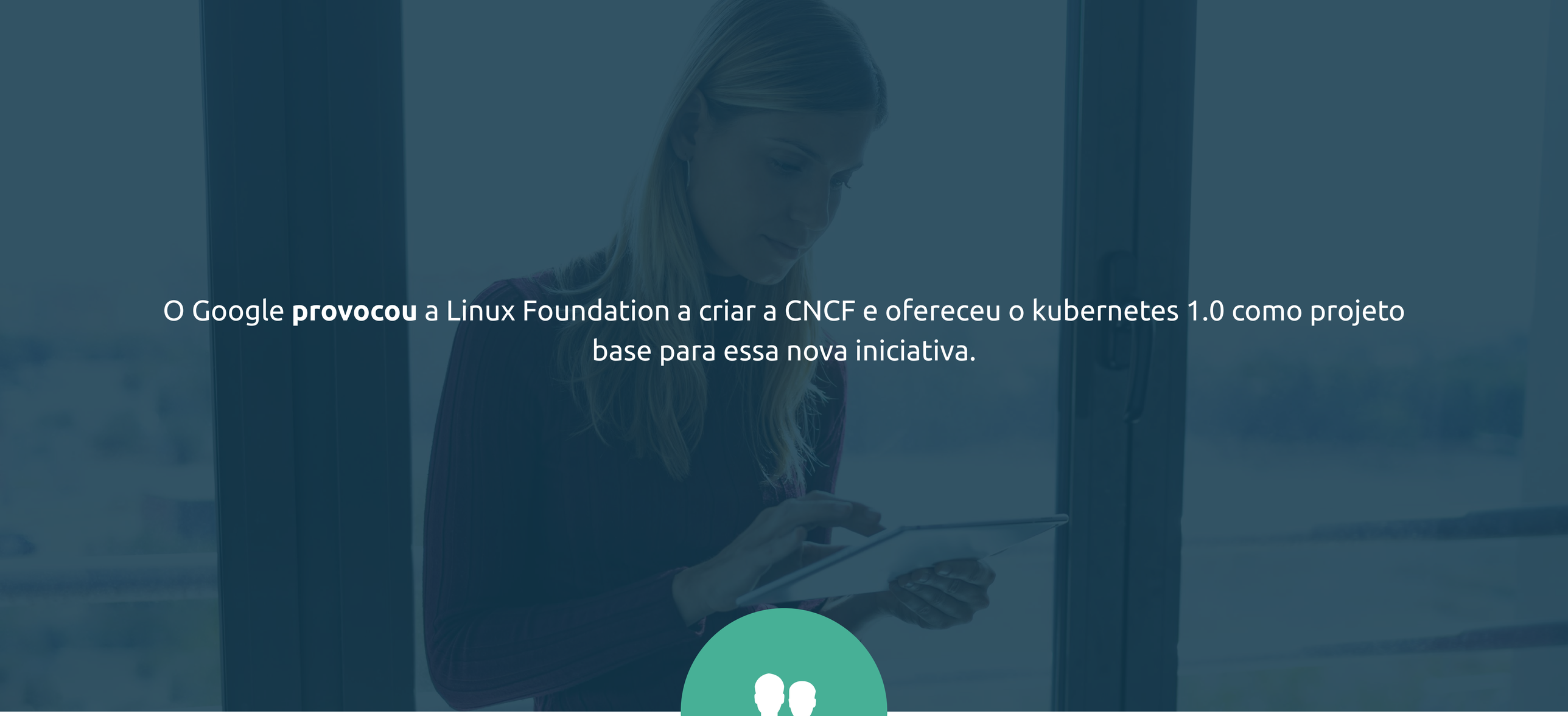
A CNCF conecta desenvolvedores, usuários, fabricantes e provedores, construindo assim uma comunidade de escala global que atua para manter e evoluir diversos projetos open source que são críticos para computação em nuvem

Membros fundadores

Google e Linux Fundation, juntamente com CoreOS, RedHat, Twitter, Huawei, Intel Cisco, IBM, Docker, VMware e Univa. Hoje possui mais de 450 membros.

CNFC

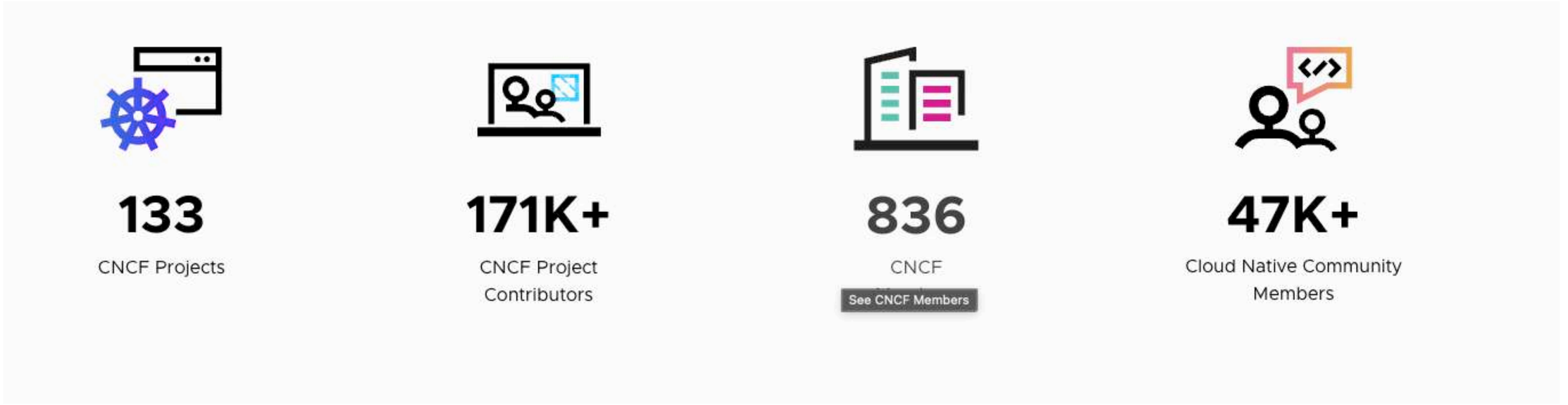
A CNCF foi fundada especialmente para acelerar as tecnologias de containers e alinhar a industria de tecnologia em volta de sua evolução.



O Google **provocou** a Linux Foundation a criar a CNCF e ofereceu o kubernetes 1.0 como projeto base para essa nova iniciativa.

Números

CNCF



Add your header here

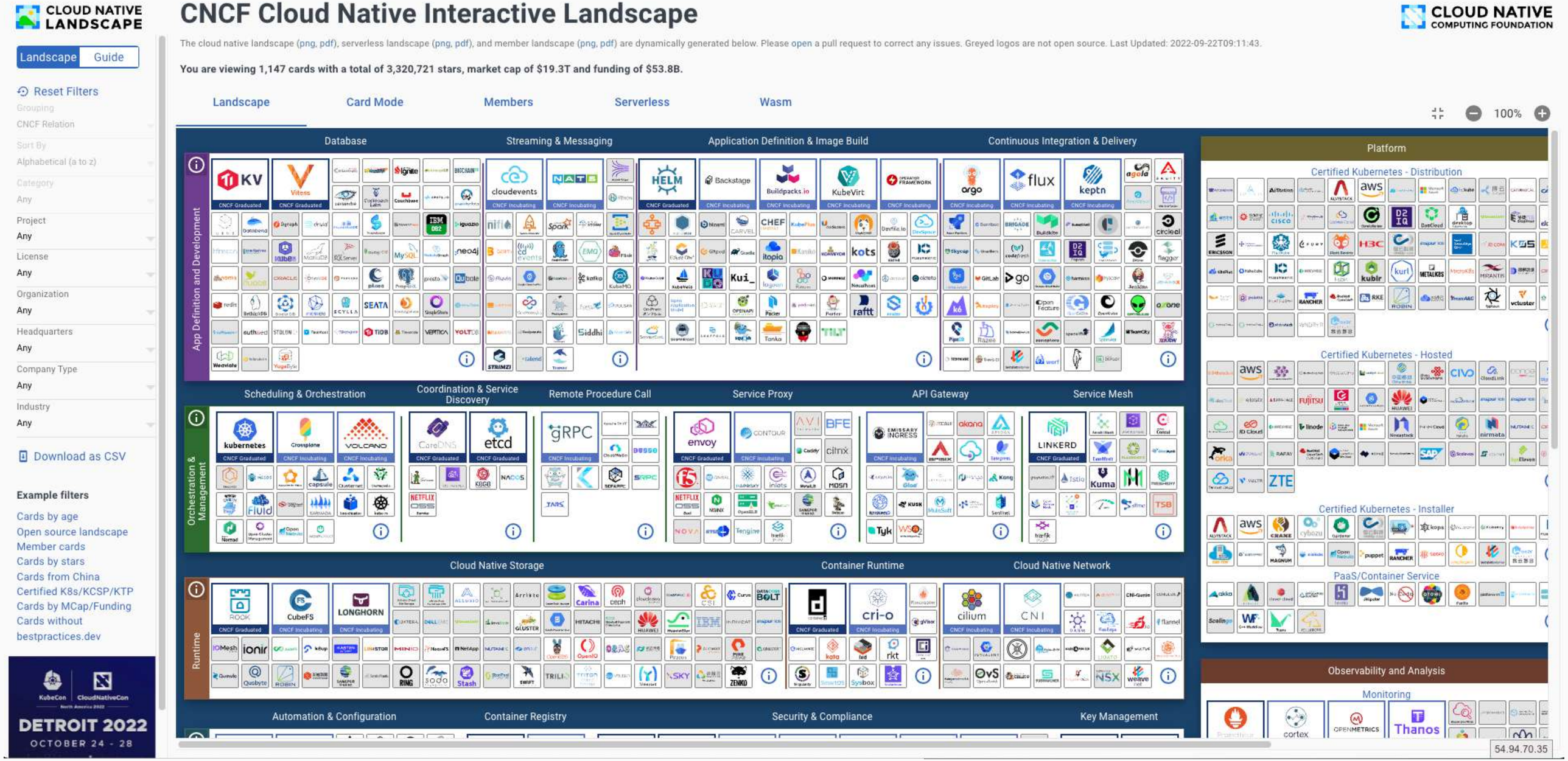
Add your sub-header here Lorem Ipsum

No Lock-in

Os projetos da CNFC são todos open source, interoperáveis através de API’s, e trabalham dentro de premissas e princípios de padrões abertos.

Portabilidade e compatibilidade

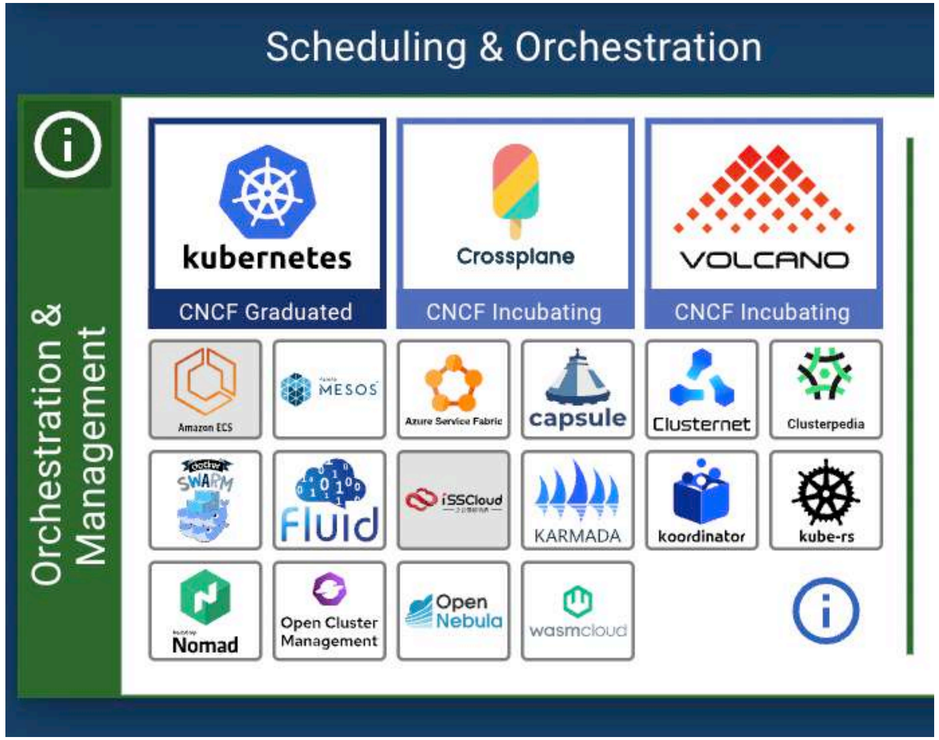
Se você utilizar estes projetos, conseguirá de forma muito simples transitar entre diferentes provedores de nuvem, ou mesmo infraestruturas on premisses com facilidade e transparência.

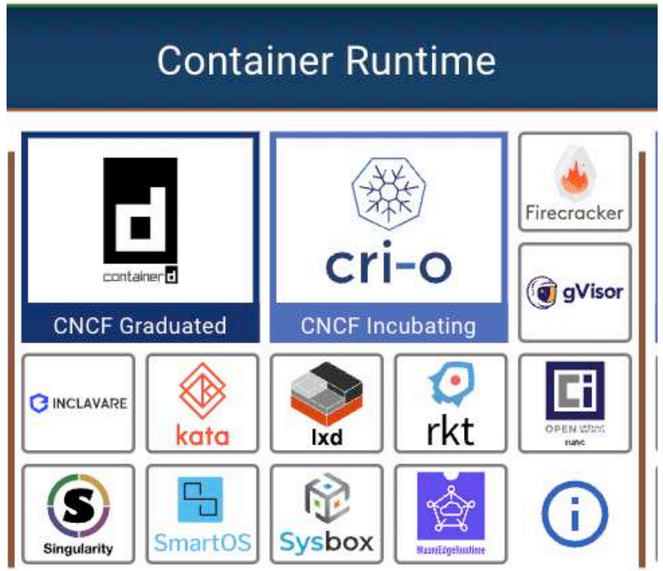


https://l.cncf.io/

Áreas interessantes

CNCF Landscape





https://l.cncf.io/

CLOUD NATIVE COMPUTING FOUNDATION

CLOUD NATIVE TRAIL MAP

The Cloud Native Landscape *l.cncf.io* has a large number of options. This Cloud Native Trail Map is a recommended process for leveraging open source, cloud native technologies. At each step, you can choose a vendor-supported offering or do it yourself, and everything after step #3 is optional based on your circumstances.

HELP ALONG THE WAY

A. Training and Certification

Consider training offerings from CNCF and then take the exam to become a Certified Kubernetes Administrator or a Certified Kubernetes Application Developer

cncf.io/training

B. Consulting Help

If you want assistance with Kubernetes and the surrounding ecosystem, consider leveraging a Kubernetes Certified Service Provider

cncf.io/kcsp

C. Join CNCF's End User Community

For companies that don't offer cloud native services externally

cncf.io/enduser

WHAT IS CLOUD NATIVE?

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

The Cloud Native Computing Foundation seeks to drive adoption of this paradigm by fostering and sustaining an ecosystem of open source, vendor-neutral projects. We democratize state-of-the-art patterns to make these innovations accessible for everyone.

l.cncf.io

v20200501

1. CONTAINERIZATION

- Commonly done with Docker containers
- Any size application and dependencies (even PDP-11 code running on an emulator) can be containerized
- Over time, you should aspire towards splitting suitable applications and writing future functionality as microservices

2. CI/CD

- Setup Continuous Integration/Continuous Delivery (CI/CD) so that changes to your source code automatically result in a new container being built, tested, and deployed to staging and eventually, perhaps, to production
- Setup automated rollouts, roll backs and testing
- Argo is a set of Kubernetes-native tools for deploying and running jobs, applications, workflows, and events using GitOps paradigms such as continuous and progressive delivery and MLops

argo — CNCF Incubating

3. ORCHESTRATION & APPLICATION DEFINITION

- Kubernetes is the market-leading orchestration solution
- You should select a Certified Kubernetes Distribution, Hosted Platform, or Installer: cncf.io/ck
- Helm Charts help you define, install, and upgrade even the most complex Kubernetes application

kubernetes — CNCF Graduated
HELM — CNCF Graduated

4. OBSERVABILITY & ANALYSIS

- Pick solutions for monitoring, logging and tracing
- Consider CNCF projects Prometheus for monitoring, Fluentd for logging and Jaeger for Tracing
- For tracing, look for an OpenTracing-compatible implementation like Jaeger

Prometheus — CNCF Graduated
fluentd — CNCF Graduated
JAEGER — CNCF Graduated
OPENTRACING — CNCF Incubating

5. SERVICE PROXY, DISCOVERY, & MESH

- CoreDNS is a fast and flexible tool that is useful for service discovery
- Envoy and Linkerd each enable service mesh architectures
- They offer health checking, routing, and load balancing

envoy — CNCF Graduated
CoreDNS — CNCF Graduated
LINKERD — CNCF Incubating

6. NETWORKING, POLICY, & SECURITY

To enable more flexible networking, use a CNI-compliant network project like Calico, Flannel, or Weave Net. Open Policy Agent (OPA) is a general-purpose policy engine with uses ranging from authorization and admission control to data filtering. Falco is an anomaly detection engine for cloud native.

CNI — CNCF Incubating
Open Policy Agent — CNCF Incubating
falco — CNCF Incubating

7. DISTRIBUTED DATABASE & STORAGE

When you need more resiliency and scalability than you can get from a single database, Vitess is a good option for running MySQL at scale through sharding. Rook is a storage orchestrator that integrates a diverse set of storage solutions into Kubernetes. Serving as the "brain" of Kubernetes, etcd provides a reliable way to store data across a cluster of machines. TiKV is a high performant distributed transactional key-value store written in Rust.

Vitess — CNCF Graduated
ROOK — CNCF Incubating
etcd — CNCF Incubating
KV — CNCF Incubating

8. STREAMING & MESSAGING

When you need higher performance than JSON-REST, consider using gRPC or NATS. gRPC is a universal RPC framework. NATS is a multi-modal messaging system that includes request/reply, pub/sub and load balanced queues. CloudEvents is a specification for describing event data in common ways.

gRPC — CNCF Incubating
NATS — CNCF Incubating
cloudevents — CNCF Incubating

9. CONTAINER REGISTRY & RUNTIME

Harbor is a registry that stores, signs, and scans content. You can use alternative container runtimes. The most common, both of which are OCI-compliant, are containerd and CRI-O.

containerd — CNCF Graduated
HARBOR — CNCF Incubating
cri-o — CNCF Incubating

10. SOFTWARE DISTRIBUTION

If you need to do secure software distribution, evaluate Notary, an implementation of The Update Framework.

TUF — CNCF Graduated
Notary — CNCF Incubating

https://github.com/cncf/trailmap

Orquestração Cloud Native

O melhor jeito de fazer, na minha opinião :)

Orquestração Cloud Native

Entendendo containers em produção

Kubernetes ganhou a corrida dos orquestradores de containers, fato.

Kubernetes é o novo padrão, isso é indiscutível, todo provedor sério suporta.

Orquestrar containers em produção sem Kubernetes é insensato em 2022, IMHO.

Quem ainda não mudou, está mudando.

Quem já mudou já colhe os resultados.

Orquestração Cloud Native

Entendendo containers em produção

Como entro nesse mundo?

Prepare sua APP para rodar em containers.

Leia, entenda e tente aplicar ao máximo os princípios e conceitos do 12factor.net

Sua APP tem que se **instrumentável**.

Sua APP tem que suportar e conseguir enviar dados para um APM.

Sua APP tem que oferecer logs úteis.

Sua APP tem que oferecer um endpoint para checar sua saúde **/health** (status code 200)

Seu APP tem que oferecer endpoints com métricas de negócio e operação.

Sua APP deve focar em ser stateless, se possível.

Use um framework de versionamento decente (ex: SEMVER).

Gerenciado ou não gerenciado?

Hoje eu particularmente acho loucura manter um K8S não gerenciado.

Dá para fazer?
Dá sim, mas tem vários pontos sensíveis.

Vale a pena?
Eu não acho que vale, meus projetos nos últimos 2 anos giram em torno do EKS.

Quais as opções?
GKS e EKS eu recomendaria com certa tranquilidade, se tivesse que escolher eu iria EKS sempre.

Alguma outra?
Nenhuma que eu tenha testado o suficiente para recomendar.

Orquestração OnPrem

Entendendo containers em produção

Na minha empresa somos OnPrem, pra onde correr?

O que eu usaria?

Distribuição Kubernetes RKE orquestrado por um Rancher.

Por qual razão?

Te oferece um bom gerenciador web com ótima compatibilidade para virar para alguma nuvem caso precise, sem dramas na migração de suas APPs.

Você pode provisionar nós em qualquer provedor ou hypervisor e estender seu cluster com certa facilidade.

É perfeito para multi-cloud ou cluster multi-região.

Experiência de K8S mais próximo do vanilla (k8s puro).

Usamos kubectl para administrar, não tem peculiaridades, abstrações ou uso de binários estranhos.

Pode importar diversos clusters para dentro dele e administrar, mesmo aqueles que não foram criados por ele.

Orquestração OnPrem

Entendendo containers em produção

Algo além?

Existem outras opções, mas na maioria essas alternativas geram dependência tecnológica pois a solução em si não é um Kubernetes, mas um produto com Kubernetes embarcado, e nesse caso ele modifica muito a experiência de uso do K8S com abstrações e binários adicionais. Por essa razão não recomendo.

Orquestração OnPrem

Entendendo containers em produção

O que dá mais trabalho em um cluster OnPrem?

Manter a consistência do seus nós master e do ETCd
Fazer backup do seu ETCd.
Fazer manutenção preventiva e reativa do seu ETCd.
Monitorar seu ETCd.
Atualizar o cluster pode ser um pouco mais complexo do que se imagina.

O que temos que ter quando mantemos um cluster OnPrem?

Um bom monitoramento de todo o seu cluster.
Um plano de backup bem feito para seu ETCd.
Um plano de backup via VELERO para suas APPs e Volumes.
Ter um bom plano de DR no caso de falha do seu ETCd.
Ter todo o provisionamento de um novo cluster no Terraform pronto, caso precise.
Testar seus backups regularmente.

Containers

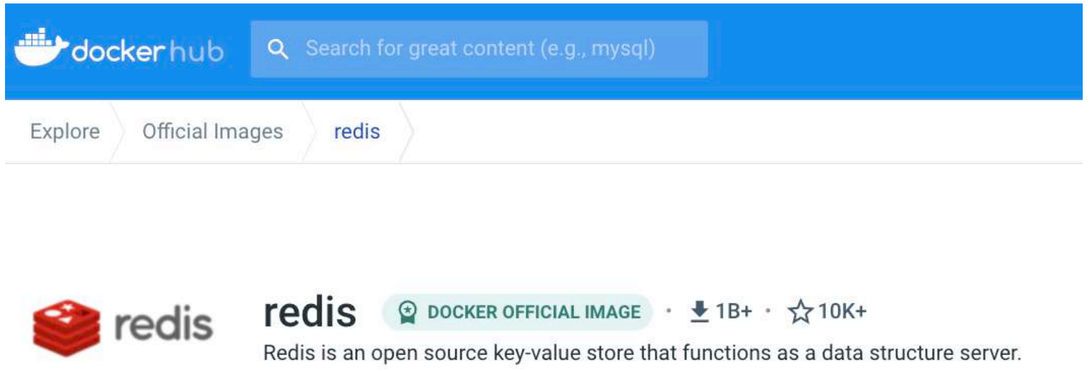
Trabalhando do jeito certo!

Imagens de containers

Dicas Dockerfile

Use sempre imagens oficiais e verificadas

```
FROM redis
```



Imagens de containers

Dicas Dockerfile

Defina de forma clara a versão da imagem em seu Dockerfile

```
FROM node:17.0.1
```

Evite usar versão latest ou não especificar a versão.
Sempre trabalhe com uma versão testada e homologada.

Imagens de containers

Dicas Dockerfile

Escolha a imagem certa para sua necessidade

```
FROM node:17.0.1-ubuntu
```

Cada projeto oferece diferentes tipos de imagens, com bases diferentes.

Imagens que tem bases em **distros** podem ser maiores e mais lentas de serem carregadas.

Quanto mais pacotes na imagem, maior a chance de vulnerabilidades em sua imagem.

As imagens com nome **SLIM** geralmente são as menores, mas teste com cuidado.

Imagens de containers

Dicas Dockerfile

Imagens menores são as melhores

```
FROM node:17.0.1-alpine
```

Quando menor mais rápido será criado o container e mais rápido sua APP será executada

Imagens menores reduzem sensivelmente a superfície de ataque e vulnerabilidades

Imagens de containers

Dicas Dockerfile

Tome cuidado com a quantidade de camadas

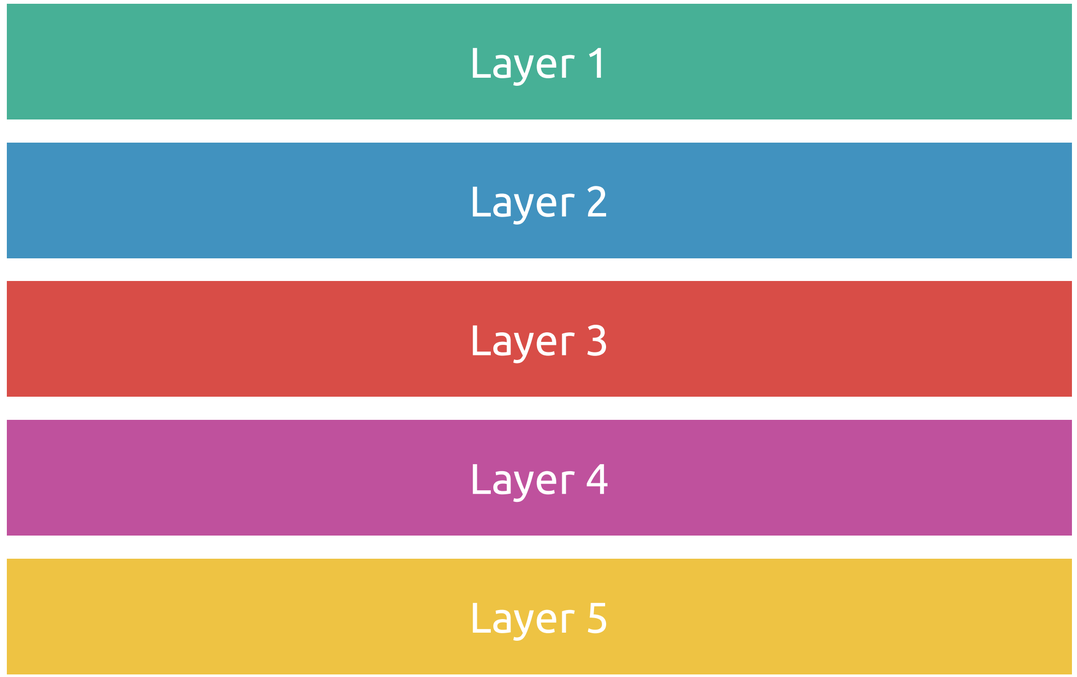
```
FROM node:17.0.5-apline

WORKDIR /app

COPY myapp /app

RUN npm install --production

CMD ["node", "src/index.js"]
```



Cada comando cria uma camada, além das camadas herdadas da imagem base.

Você pode tentar agrupar alguns comandos para reduzir a quantidade de layers.

Muitas camadas deixam o build lento e a imagem maior.

Imagens de containers

Dicas Dockerfile

Como reduzir camadas?

```
FROM ubuntu
RUN apt-get install -y wget
RUN wget https://url/file.tar
RUN tar xvzf file.tar
RUN rm file.tar
RUN apt-get remove wget
```

```
FROM ubuntu
RUN apt-get install wget \
   && wget https://url/file.tar \
   && tar xvzf file.tar \
   && rm file.tar \
   && apt-get remove wget
```

Você pode tentar agrupar alguns comandos para reduzir as camadas.

Isso vai agilizar o build de sua imagem e reduzir seu tamanho.

Imagens de containers

Dicas Dockerfile

Use multi-stage builds para reduzir o tamanho da sua imagem

```
# estágio temporário, apenas para fazer o build

FROM python:3.9 as builder
RUN pip wheel --no-cache-dir --no-deps --wheel-dir /wheels jupyter pandas

# estágio final usando o que foi buildado no primeiro estágio

FROM python:3.9-slim
WORKDIR /notebooks
COPY --from=builder /wheels /wheels
RUN pip install --no-cache /wheels/*
```

Observe que no primeiro estágio rodo os comandos que vão gerar o que eu preciso

No segundo estágio, eu uso apenas o que foi gerado para criar a imagem final

Com isso a imagem fica magrinha e rápida de ser executada, sem pacotes desnecessários

Imagens de containers

Dicas Dockerfile

Ordene os comandos corretamente lembrando que o docker faz cache de cada estágio

```
FROM python:3.9-slim

WORKDIR /app

COPY example.py .

COPY requirements.txt .

RUN pip install -r /requirements.txt
```

Esse primeiro dockerfile invalida o cache pois copia a APP antes de instalar as dependências, com isso, cada vez que houver uma mudança no **example.py** o build vai reinstalar os pacotes o invés de aproveitar o cache.

```
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install -r /requirements.txt

COPY example.py .
```

Alterando de forma simples, colocando a cópia da aplicação no final, caso a aplicação seja modificada o build vai reaproveitar o cache dos pacotes e construir a imagem de forma mais rápida e eficiente.

Imagens de containers

Dicas Dockerfile

Prefira COPY ao comando ADD quando for usar arquivos.

```
COPY /src/path /dest/path
COPY /src/file /dest/file
```

O comando COPY é mais previsível e retorna informações mais claras nos logs durante o BUILD.

```
ADD https://url/file.tar.gz /dest/path
```

Use o ADD quando estiver pegando um arquivo via URL ou quando precisar descompactar algo.

Imagens de containers

Dicas Dockerfile

Lembre-se de limpar o cache se instalou pacotes usando apk ou apt

```
FROM alpine
RUN apk update \
    && apk add curl nginx \
    && rm -rf /var/cache/apk/*
COPY index.html /var/www/html/
EXPOSE 80
CMD [“nginx”,“-g”,“daemon off;”]
```

```
FROM ubuntu
RUN apt update \
    && apt install curl nginx \
    && rm -rf /var/cache/apt/*
COPY index.html /var/www/html/
EXPOSE 80
CMD [“nginx”,“-g”,“daemon off;”]
```

Imagens de containers

Dicas Dockerfile

Evite instalar dependências desnecessárias em sua imagem

```
FROM ubuntu
RUN apt update \
    && apt install -y --no-install-recommends PACOTE
```

Imagens de containers

Dicas Dockerfile

Prefira **array**

```
CMD ["gunicorn", "-w", "4", "-k", "uvicorn.workers.UvicornWorker", "main:app"]
```

Ao invés de **string**

```
CMD "gunicorn -w 4 -k uvicorn.workers.UvicornWorker main:app"
```

Apesar de ambos estarem corretos a documentação do Docker recomenda sempre o uso de arrays.

Imagens de containers

Dicas Dockerfile

Defina um **healthcheck** para saber se seu container está funcionando bem

```
HEALTHCHECK CMD curl --fail http://localhost:8000 || exit 1
```

Quando isso quandop rodarmos um **docker ps** teremos algum contexto

```
STATUS
Up 8 seconds (health: starting)

STATUS
Up About a minute (health: unhealthy)
```

Imagens de containers

Dicas Dockerfile

Mais algumas dicas soltas!

➜ Rode apenas um processo por container

➜ Se possível rode o processo dentro do container com usuário comum

➜ Não armazene segredos no Dockefile, use variáveis para isso

➜ Use ARGs para passar argumentos durante o Build

➜ Use ENVs para passar argumentos durante a execução da imagem

➜ Verifique seu dockerfile com uma ferramenta linter como o Hadolint

➜ Verifique sua imagem docker com uma ferramenta de segurança como Trivy ou Clair

➜ Criei imagens base para suas pipelines e aplicações

➜ Use o arquivo dockerignore para não levar aquivos indesejados para imagem

➜ Assine suas imagens

Imagens de containers

O que não fazer?

Achar normal uma imagem de 3 gigas ou maior, algo não está certo aí.

Usar imagens não oficiais ou suspeitas.

Não manter suas imagens para pipelines.

Não verificar periodicamente a segurança de suas imagens.

Não atualizar periodicamente a versão das imagens utilizadas em suas pipelines.

Usar uma versão muito antiga do docker em produção.

Rodar em produção usando "docker run" ou "docker compose"

Usar um orquestrador velho e desatualizado.

Buildar a imagem a cada deploy ao invés de usar um Registry.

Banco em containers?

Posso rodar banco?

Claro sem problema, hoje em dia rodar banco em Container é algo bem normal.

A maioria dos bancos mais populares como PostgreSQL, MySQL, Redis, Mongo já tem suas versões de imagens oficiais para rodar em containers, estão bem testados e bem maduros para esse tipo de cenário.

Lembre-se de montar o volume persistente e cuidar direitinho dos backups, de resto é só correr para o abraço.

Banco em Kubernetes?

Posso rodar banco?

Claro sem problema, hoje em dia rodar banco em Kubernetes é tranquilo.

Vá no portal de landscape da CNCF e escolha o melhor projeto para rodar seu banco.

No kubernetes o ideal é utilizar um operador de banco.

:)

Prefira sempre banco gerenciado ao rodar banco no seu Cluster, se houver a possibilidade.

Java em containers

Posso rodar app java?

Pode sim :)

Use um framework java para containers como Quarks, Micronaut ou Springboot.

O Java mínimo recomendado para rodar em containers é 1.8+, ainda assim alguns comportamentos inesperados podem acontecer, teste bastante antes de colocar em produção.

O ideal é usar JAVA a partir da versão 11 que tem um gerenciamento de memória e CPU bem decente.

Lembre-se de fazer o tuning da JVM corretamente, desde o garbage collector até a memória heap, prefira um limite menor para memória, lembrando que com containers nós escalamos horizontalmente e não verticalmente.

Se você usa aqueles frameworks que sobem dezenas de APPs Java, pense novamente na sua estrutura se quiser rodar em containers, o ideal é um processo por container, ou no caso de JAVA – com um bom desconto pelo uso do framework – uma aplicação por container.

O que não roda bem em containers?

Tem regra?

Rodar roda!

Olha qualquer coisa dá para empacotar em uma imagem Docker e rodar.

A pergunta é roda bem?

Vai performar legal?

Vai usar os recursos do Docker ou K8S?

Vale a pena?

Resolve seu problema?

Tem que pensar nesses aspectos.

Amarrando as pontas!

Dicas finais e tira dúvidas :)

Amarrando as pontas!

Curiosidades

O que eu uso hoje para trabalhar?

Docker no laptop (mac.m1max)
Esteira usamos CircleCI ou GitHub Actions
Nosso orquestrador é o EKS e nossa cloud AWS
Nossa ferramenta de monitoração é o DataDog
Nossa ferramenta de APM é o NewRelic
Usamos ansible para configurar nossos sistemas e gerir nossos usuários
Usamos tailscale para se conectar em nossos servidores de forma segura
Usamos wazuh verificar aspectos de segurança em nosso servidores
Usamos terraform para provisionar nossa infra na nuvem

E o Podman, vai usar?

Já testei, mas confesso que ainda não precisei :)

Ainda usa Rancher?

Apenas em projetos pessoais no mini-datacenter de casa, o resto é tudo EKS.

Amarrando as pontas!

Perguntas

O que eu acho do movimento Cloud Native?

O que eu acho do Kubernetes e que esperar para o futuro?

O que acho do Docker e o que esperar para o futuro?

Alguma outra pergunta?

Amarrando as pontas!

Curiosidades

Onde moro?

Brasília/DF

Idade?

41

Tempo de profissão?

23++

Onde trabalho atualmente?

DNSFilter

Fazendo o que?

Ajudo a manter e administrar um SaaS que roda em mais de 30 países :)

Qual minha role?

Platform Engineer

Entre em contato

@gutocarvalho

guto@carvalho.life
gutocarvalho.net
curriculo.gutocarvalho.net

Redes Sociais

Acompanhe

/falagutera	/falagutera	/gutocarvalho	gutocarvalho.net	@gutocarvalho