UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

HOLLIVER DE OLIVEIRA COSTA

MICROSSERVIÇOS E ORQUESTRAÇÃO DE CONTÊINERES: UMA ABORDAGEM PRÁTICA.

CAMPINA GRANDE - PB

2023

HOLLIVER DE OLIVEIRA COSTA

MICROSSERVIÇOS E ORQUESTRAÇÃO DE CONTÊINERES: UMA ABORDAGEM PRÁTICA.

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador : Fábio Jorge Almeida Morais

CAMPINA GRANDE - PB

2023

HOLLIVER DE OLIVEIRA COSTA

MICROSSERVIÇOS E ORQUESTRAÇÃO DE CONTÊINERES: UMA ABORDAGEM PRÁTICA.

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Fábio Jorge Almeida Morais

Orientador – UASC/CEEI/UFCG

Wilkerson de Lucena Andrade

Examinador – UASC/CEEI/UFCG

Francisco Vilar Brasileiro

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 28 de Junho de 2023.

CAMPINA GRANDE - PB

MICROSERVICES AND CONTAINER ORCHESTRATION: A PRACTICAL APPROACH.

With the evolution of software and distributed communication, multiple microservices are becoming increasingly prevalent and in high demand in large companies. This architectural format is widely used nowadays, thus necessitating qualified professionals for its implementation. Therefore, this paper aims to present and analyze methods and strategies for systems that utilize microservices in a distributed platform, examining the positives, benefits, and drawbacks. This is accomplished by employing a microservices architecture and deploying it in a distributed infrastructure based on Kubernetes and Docker. The objective of this work is to provide a conceptual and technical foundation for the reader to understand the microservices architecture and be able to implement and deploy a standard application using microservices in a Kubernetes environment with Docker.

MICROSSERVIÇOS E ORQUESTRAÇÃO DE CONTÊINERES: UMA ABORDAGEM PRÁTICA.

Trabalho de Conclusão de Curso

Holliver de Oliveira Costa (Aluno), Fabio Jorge Almeida Morais (Orientador)

Departamento de Sistemas e Computação

Universidade Federal de Campina Grande

Campina Grande, Paraíba - Brasil

RESUMO

Com a evolução dos *softwares* e comunicação distribuída, os múltiplos microsserviços estão cada vez mais presentes e com grande demanda nas macros empresas. Esse formato arquitetural é amplamente utilizado atualmente, portanto, se faz necessário profissionais qualificados para a implantação dessa arquitetura. Assim, o presente trabalho tem como objetivo apresentar e analisar métodos e estratégias de sistemas que utilizam microsserviços em uma plataforma distribuída, analisando os pontos positivos, benefícios e desvantagens. Isto é feito usando uma arquitetura de microsserviços e implantação em uma infraestrutura distribuída baseada em Kubernetes e Docker. Este trabalho tem como objetivo fornecer base conceitual e técnica para que o leitor entenda a arquitetura em microsserviços e consiga implementar e implantar uma aplicação padrão utilizando microsserviços em um ambiente Kubernetes com Docker.

PALAVRAS-CHAVE

Microsserviços, arquitetura, aplicação, kubernetes, docker.

Repositório

https://github.com/HolliverCosta/WAITERAPP

1. INTRODUÇÃO

A evolução das aplicações de *software* tem sido impulsionada pela demanda crescente por sistemas escaláveis, resilientes e flexíveis. Nesse contexto, os microsserviços e a orquestração de *containers* têm se destacado como abordagens inovadoras e poderosas para a construção e implantação de arquiteturas distribuídas.

Os microsserviços representam uma abordagem arquitetônica que divide uma aplicação monolítica em serviços independentes e altamente especializados. Cada microsserviço é responsável por uma funcionalidade específica, facilitando o desenvolvimento, testes e manutenção, além de permitir a escalabilidade horizontal dos serviços individualmente, ou seja, cada serviço pode ser escalado independentemente dos outros, de acordo com a demanda específica que ele enfrenta. Dessa forma, é possível dimensionar apenas os serviços necessários, sem afetar todo o sistema. Essa abordagem traz benefícios como a autonomia de equipes de desenvolvimento, maior flexibilidade na implantação de novas funcionalidades e maior tolerância a falhas. No entanto, é importante ressaltar que, embora sejam independentes em termos de desenvolvimento e implantação, os microsserviços ainda podem ter dependências entre si. Portanto, a queda de um microsserviço pode impactar o funcionamento correto dos serviços que dependem dele.

Já a orquestração de *containers* refere-se à administração e gerenciamento eficiente de múltiplos *containers* que hospedam microsserviços. Os *containers*, através do uso de virtualização (por exemplo, via Docker), oferecem um ambiente isolado e portátil para o empacotamento de microsserviços e suas dependências. No entanto, a escalabilidade e o gerenciamento manual de *containers* podem ser complexos e demandar recursos significativos. Nesse contexto, o Kubernetes é uma plataforma *open-source* usada na orquestração de *containers*, que automatiza a implantação, dimensionamento e gerenciamento de aplicações em *containers*. Ele fornece uma infraestrutura escalável e robusta para executar e monitorar os *containers*, lidando com o balanceamento de carga, gerenciamento de recursos, recuperação de falhas e escalabilidade horizontal dos serviços. O Kubernetes oferece recursos avançados, como descoberta de serviços, roteamento de tráfego e atualizações sem tempo de inatividade, tornando-o uma escolha ideal para a orquestração de microsserviços em larga escala.

Neste trabalho, exploraremos o poder da combinação entre microsserviços e orquestração de *containers* utilizando as tecnologias Kubernetes e Docker. A escolha do Kubernetes e Docker baseia-se em sua maturidade, ampla adoção na indústria de tecnologia e suporte ativo da comunidade. Essas tecnologias são reconhecidas por sua confiabilidade, flexibilidade e escalabilidade, sendo amplamente utilizadas por empresas de diversos segmentos para a construção e operação de arquiteturas modernas baseadas em microsserviços.

Será apresentado um estudo de uma aplicação de arquitetura monolítica, que foi reestruturada para uma arquitetura de microsserviços, durante um mês, cujo nome é WaiterApp (traduzido, aplicativo do garçom), no qual se deu através da integração de um aplicativo móvel e uma aplicação *web*, os quais consomem um conjunto de microsserviços que acessam um banco de dados. Como também, serão discutidas ao decorrer do trabalho as práticas usadas para a implementação dessa arquitetura, considerando aspectos como o empacotamento de microsserviços em *containers* Docker, a configuração e gerenciamento do *cluster* Kubernetes, a escalabilidade dinâmica dos serviços e a automação de tarefas de implantação e manutenção.

Ao final deste estudo, espera-se obter uma visão aprofundada do poder dos microsserviços e da orquestração de *containers* utilizando o Kubernetes e Docker, fornecendo *insights* para profissionais e pesquisadores interessados na construção de aplicações baseadas em arquitetura de microsserviços.

2. MONÓLITOS

Um sistema de *software* com arquitetura de monólito funciona de modo que todas as funcionalidades e componentes estão interligados dentro de um único código fonte e são executados como um único processo em um único servidor, esse tipo de arquitetura é chamada de monolítica. A estrutura de um sistema monolítico é composta por uma única base de código, que inclui a interface do usuário, a lógica de negócios e o banco de dados. Como apresentado na figura 1 a seguir:

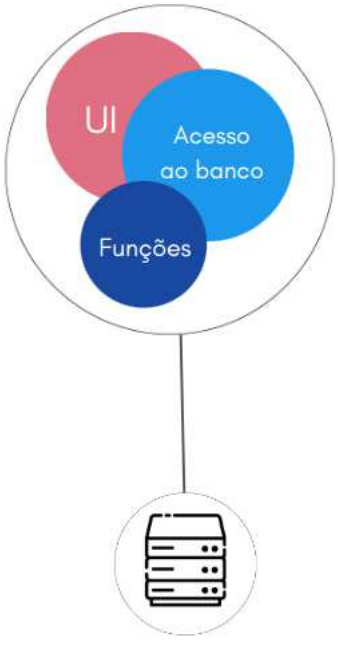

Figura 1: Exemplo de uma arquitetura monolítica

Quando um usuário interage com o sistema, a solicitação é encaminhada para o servidor, onde o código do aplicativo é executado. O servidor processa a solicitação, realiza as operações necessárias e envia a resposta de volta ao usuário. Toda a lógica de negócios, desde a validação dos dados até a execução de operações, é realizada dentro do monólito.

Dentro do código do sistema monolítico, as diferentes partes do aplicativo (componentes) estão diretamente conectadas, ou seja, não há interfaces externas ou camadas de separação definidas. Isso permite que as diferentes partes do sistema compartilhem dados e funcionalidades de forma direta e rápida.

O banco de dados também faz parte do sistema monolítico. Os dados são armazenados em uma única estrutura de bancos de dados, geralmente um banco de dados relacional. Todas as operações de leitura e escrita no banco de dados são realizadas diretamente pelo código do aplicativo. Mesmo que os sistemas monolíticos incluam o banco de dados, não é necessário que faça parte dessa arquitetura. Muitas empresas optam por criar uma arquitetura distribuída, separando a camada do banco de dados do código do aplicativo.

Uma das vantagens de um sistema monolítico é que o desenvolvimento e o *deployment* são simplificados. Como todo o código está em um único lugar, os desenvolvedores podem trabalhar no mesmo ambiente e não há necessidade de comunicação entre componentes separados.

No entanto, existem desvantagens em relação à manutenção e evolução do sistema. Como todas as partes são interligadas, é difícil fazer alterações em um componente específico sem afetar outras partes do sistema. Isso pode levar a um código base complexo e difícil de entender. Além disso, o tempo de compilação e implantação pode ser maior, já que qualquer mudança no código requer a compilação e implantação de todo o sistema.

Em suma, um sistema de *software* monolítico é uma aplicação em que todas as partes do aplicativo estão interligadas em um único código fonte e são executadas como um único processo. Embora o desenvolvimento e o *deployment* possam ser simplificados, a manutenção e a evolução do sistema podem ser mais desafiadoras devido à falta de separação e à dependência entre os componentes

3. ARQUITETURAS BASEADAS EM MICROSSERVIÇOS

Microsserviços é uma abordagem de arquitetura de *software* que permite que as aplicações sejam divididas em componentes independentes que se comunicam entre si por meio de APIs. Cada microsserviço é responsável por executar uma única tarefa específica e pode ser desenvolvido, implantado e escalado de forma independente dos outros serviços, como apresentado na figura 2. Isso oferece várias vantagens, como flexibilidade, escalabilidade e resiliência, uma vez que os serviços podem ser dimensionados de forma individual para atender a demandas específicas, sem afetar outros componentes do sistema.

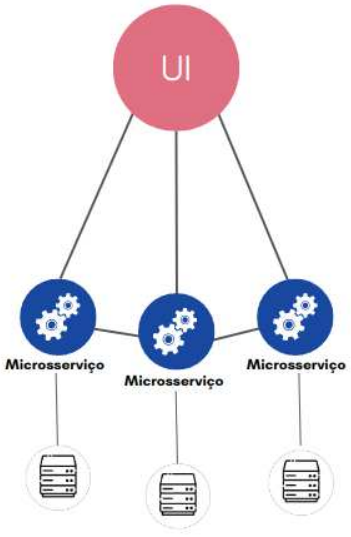

Figura 2: Exemplo de uma arquitetura de microsserviços

Para implementar a arquitetura de microsserviços, é necessário adotar ferramentas e tecnologias que permitam a criação, o gerenciamento e o escalonamento de serviços independentes. Uma tecnologia chave nesse contexto são os *containers*. Um *container* é uma unidade de *software* que encapsula um microsserviço e todas as suas dependências, como bibliotecas e outras ferramentas necessárias para executar o serviço.

4. OS CONTAINERS E SEU GERENCIAMENTO

Os *containers* são leves, portáteis e isolados do ambiente de hospedagem, o que garante que o serviço seja executado de forma consistente em diferentes ambientes, independentemente das configurações de *hardware* e *software* do ambiente de hospedagem. Além disso, os *containers* podem ser facilmente implantados, gerenciados e dimensionados, o que facilita o trabalho dos desenvolvedores e dos administradores de sistemas.

São criados a partir de imagens, que são como moldes para os *containers*. Uma imagem é um pacote que contém todos os componentes necessários para executar um microsserviço, incluindo o próprio código do serviço, bibliotecas e outras dependências. As imagens são criadas uma única vez e, em seguida, podem ser usadas para implantar e executar o serviço em vários *containers* em diferentes ambientes. Como ilustrado na figura 3 a seguir:

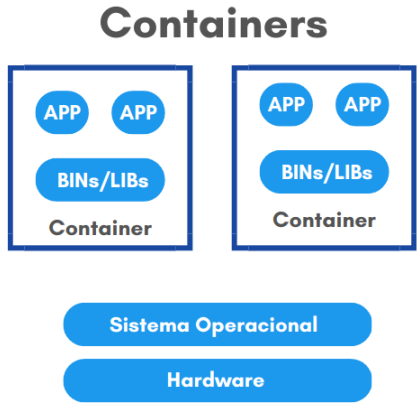

Figura 3: Ilustração de containers

Isto posto, o gerenciamento dos *containers* se dá por meio do Docker, cujo é uma plataforma de *software* que permite a criação, implantação e execução de aplicativos em *containers*. Os *containers* Docker são um tipo de virtualização de nível de sistema operacional que fornece uma camada de abstração entre o aplicativo e o sistema operacional. Cada *container* é uma instância isolada do aplicativo, com sua própria cópia do sistema operacional, bibliotecas e outros componentes necessários. Isso significa que diferentes aplicativos podem ser executados em diferentes *containers*, sem que haja interferência entre eles.

Uma vantagem do Docker é que ele simplifica o processo de implantação. Em vez de implantar todo o sistema monolítico de uma vez, os desenvolvedores podem implantar *containers* individuais, permitindo que diferentes partes do aplicativo sejam atualizadas separadamente. Isso torna mais fácil adicionar novos recursos, corrigir *bugs* e atualizar o aplicativo como um todo.

Logo, o Docker é uma tecnologia de virtualização que permite empacotar aplicativos inteiros. Oferece vantagens de portabilidade e facilidade de implantação, permitindo que os desenvolvedores executem aplicativos em diferentes ambientes sem se preocupar com a compatibilidade com outros sistemas.

5. ORQUESTRAÇÃO DE CONTAINERS

Kubernetes é uma plataforma *open-source* de orquestração de *containers* que permite aos desenvolvedores implantar, escalar e gerenciar aplicativos em um ambiente de *container*. Ele automatiza a implantação, o dimensionamento e a manutenção de aplicativos, tornando mais fácil e eficiente gerenciar aplicativos em ambientes complexos e distribuídos.

O Kubernetes usa conceitos como *Pods*, serviços e controladores para gerenciar e orquestrar aplicativos em *containers*. Os *Pods* são a menor unidade de implantação em Kubernetes e representam um ou mais *containers* que são implantados juntos em um mesmo nó. Os serviços são responsáveis por expor um conjunto de *Pods* como um endpoint de rede único e consistente, permitindo que outros aplicativos os acessem facilmente. Os controladores são

responsáveis por garantir que o estado desejado do sistema seja mantido, gerenciando e escalando os *Pods* e serviços conforme necessário. Segue um exemplo na figura 4 da arquitetura do Kubernetes.

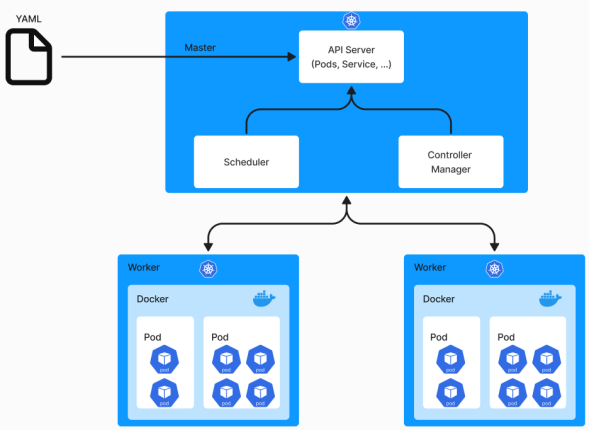

Figura 4: Arquitetura Kubernetes

Uma das principais vantagens do Kubernetes é que ele oferece alta disponibilidade e resiliência para aplicativos em *containers*. Se um *container* falhar ou um nó inteiro do *cluster* cair, o Kubernetes pode detectar automaticamente o problema e reprogramar o *container* em um nó diferente. Isso significa que os aplicativos podem ser executados de forma contínua e sem interrupções.

Desta maneira, o Kubernetes é uma plataforma de orquestração de *containers* de código aberto que ajuda os desenvolvedores a automatizar a implantação, à escala e o gerenciamento de aplicativos em *containers* em ambientes de produção. Ele oferece alta disponibilidade e resiliência para aplicativos em *containers*, além de permitir que os desenvolvedores implantem aplicativos de maneira consistente e previsível em diferentes ambientes.

6. A APLICAÇÃO

Inicialmente, foi desenvolvido uma aplicação que será utilizada para estudo, cujo nome é WaiterApp (traduzido, aplicativo do garçom), no qual é composto por três microsserviços principais, cada um com uma função específica, são eles: o *Login Service*, o *Payment Service* e o *Request Service*.

A Figura 5 apresenta o diagrama que representa a arquitetura da aplicação desenvolvida. Essa arquitetura consiste na integração de um aplicativo móvel e uma aplicação *web*, os quais consomem um conjunto de microsserviços que acessam um banco de dados.

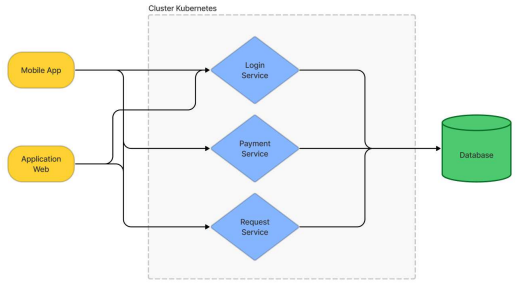

Figura 5: Arquitetura da aplicação

O primeiro deles é o *Login Service* (Serviço de Autenticação), desempenha um papel fundamental na gestão da autenticação e autorização dos garçons e dos restaurantes no aplicativo. Ele garante que apenas usuários autorizados tenham acesso aos recursos e funcionalidades do sistema, protegendo a privacidade e a segurança das informações.

O segundo serviço é o *Payment Service* (Serviço de Pagamento), que é responsável por processar e gerenciar todas as transações financeiras realizadas pelos clientes. Esse serviço garante a segurança e a integridade das transações.

Por fim, temos o terceiro serviço denominado *Request Service* (Serviço de Pedidos), que desempenha um papel central na gestão e processamento de todos os pedidos realizados pelos garçons. Esse serviço permite que o estabelecimento acompanhe e gerencie todos os pedidos de forma eficiente, garantindo um fluxo de trabalho suave e a satisfação dos clientes.

Desse modo, esses três microsserviços trabalham em conjunto para oferecer uma experiência completa e eficiente aos usuários. De modo que os garçons podem realizar seus pedidos por meio do aplicativo, o qual são encaminhados ao microsserviço de pedidos. A cozinha, por sua vez, recebe as notificações destes pedidos em seus dispositivos, agilizando o processo de atendimento. Com isso, o microsserviço de pagamento garante que todas as transações sejam seguras e confiáveis, enquanto o microsserviço de *login* protege o acesso às contas dos garçons.

Além disso, foi utilizado apenas um único banco de dados com o objetivo de simplificar a infraestrutura, uma vez que utilizar apenas um reduz a complexidade da infraestrutura necessária para a aplicação. Como também, o compartilhamento de dados, no qual, se os microsserviços precisam compartilhar alguns dados entre si, utilizar um banco de dados centralizado facilita o acesso e a consistência dessas informações. Dessa forma, os serviços podem consultar e atualizar os dados relevantes sem a necessidade de comunicação adicional complexa entre bancos de dados separados.

Para executar a aplicação em um ambiente virtualizado, foram utilizadas na aplicação as ferramentas Docker e Kubernetes. O Docker para empacotar cada microsserviço juntamente com suas dependências em *containers*. Essa abordagem nos permite ter ambientes de execução consistentes, independentemente do sistema operacional e das configurações de infraestrutura, garantindo que cada microsserviço seja facilmente replicado e implantado em

diferentes ambientes, simplificando o processo de desenvolvimento e implantação da aplicação.

Por outro lado, o Kubernetes foi utilizado como uma plataforma de orquestração de *containers*, uma vez que foi criado um *cluster* de nós no Kubernetes, no qual os *containers* dos microsserviços foram implantados. Com isso, essa plataforma cuida do monitoramento e recuperação de falhas, garantindo, através dos Kubernetes a disponibilidade contínua dos serviços, mesmo em momentos de alta demanda.

Nessa lógica, o uso do Docker e dos Kubernetes na aplicação trouxe benefícios significativos, como a simplificação do processo de implantação e atualização dos microsserviços, a escalabilidade automática de acordo com a demanda e a alta disponibilidade dos serviços. Essas tecnologias nos permitiram criar uma infraestrutura flexível, confiável e fácil de gerenciar, garantindo um desempenho consistente e uma experiência fluida para os usuários da aplicação.

7. IMPLEMENTAÇÃO

7.1 Construindo a imagem Docker

Para construir uma aplicação conteinerizada é necessário criar um arquivo denominado *Dockerfile* no repositório da aplicação. Neste arquivo, deve-se adicionar a imagem base para a aplicação, considerando que o serviço foi desenvolvido em Node.js. Essa definição é feita utilizando a instrução *FROM* seguida da versão da imagem base.

Para instalar as dependências do projeto no Node.js, é preciso copiar o arquivo package.json para a imagem, a fim de identificar as dependências e suas respectivas versões. A instrução *COPY* é utilizada para realizar essa cópia. Após a cópia do arquivo de dependências, é necessário baixar e instalar as dependências na imagem. Essa etapa é realizada por meio da instrução *RUN*, seguida do comando para baixar e instalar as dependências. Uma vez que as dependências estão instaladas, é preciso copiar todos os arquivos do diretório atual para dentro da imagem que será criada. Isso é feito utilizando a instrução *COPY*.

Posteriormente, a imagem recém-criada deve ser construída usando o comando *RUN*, seguido do comando de *build* específico para a aplicação. A próxima etapa consiste em expor a porta na qual a aplicação está aguardando conexões. Isso é realizado através da instrução *EXPOSE*, seguida do número da porta. Por fim, para executar a aplicação, utiliza-se a instrução CMD seguida do comando para rodar a aplicação.

Portanto, o processo de criação do *Dockerfile* para uma aplicação Node.js envolve a definição da imagem base, cópia e instalação de dependências, cópia dos arquivos da aplicação, construção da imagem, exposição da porta e execução da aplicação. A Figura 6 apresenta o arquivo de *Dockerfile* usado.

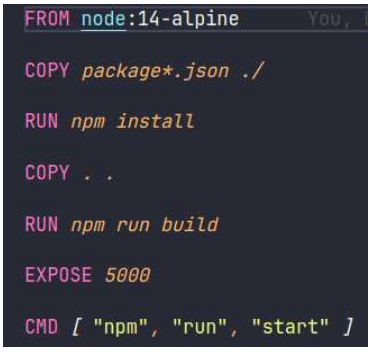

Figura 6: Dockerfile do serviço de pedidos

É importante ressaltar que esse modelo de *Dockerfile* foi aplicado aos três microsserviços, com a única modificação sendo a porta de exposição especificada na instrução "*EXPOSE*".

7.2 Orquestrando as imagens com o kubernetes

A priori, foi criado um arquivo no formato YAML para a configuração. Nele, são definidas as configurações utilizando diferentes campos, sendo definidas no campo "*spec*", no qual, é especificado o número de réplicas desejadas para o *Pod*.

A configuração do *Pod* é realizada no campo "*containers*" especificado, onde são definidos os detalhes do *container*, incluindo o nome e a imagem a ser utilizada. A imagem é especificada através do nome e da URL correspondente, hospedada no Docker Hub. Além disso, foram definidos limites para o uso de *CPU* e memória.

No mais, a exposição da porta necessária para o funcionamento do serviço é definida por meio do campo "*ports*". Por fim, variáveis de ambiente podem ser configuradas utilizando o campo "env", o que permite que informações relevantes sejam transmitidas para o *container*. A Figura 7 apresenta o arquivo de *Deployment* usado.

Figura 7: Arquivo de deployment utilizado no serviço de pedidos

Foi criado um objeto HorizontalPodAutoscaler (HPA) no arquivo YAML para escalonar o *Deployment* da aplicação. O HPA recebeu o nome "hpa-api" para identificação no *cluster*. Nas especificações (*spec*), foram configurados os seguintes aspectos. A referência do alvo de escala (*scaleTargetRef*) foi definida, selecionando o *Pod* a ser escalonado com base no rótulo (*label*) "*name*". Os valores mínimo e máximo de réplicas permitidas foram definidos, sendo um mínimo de uma réplica e um máximo de cinco réplicas.

A métrica utilizada para o escalonamento foi a utilização da *CPU*, sendo configurado para escalonar quando a utilização atingir 50%. Métricas também foram configuradas para criação ou exclusão de *Pods*. No caso do "*scaleDown*", quando a utilização da *CPU* estiver abaixo de 50% por 120 segundos, os *Pods* excedentes serão excluídos, verificando a cada 10 segundos. Já o "*scaleUp*", quando a utilização da *CPU* ultrapassar 50%, um novo *Pod* será criado imediatamente, sem janela de estabilização, e a verificação das métricas é feita a cada 10 segundos. A Figura 8 apresenta o arquivo de HorizontalPodAutoscaler usado.

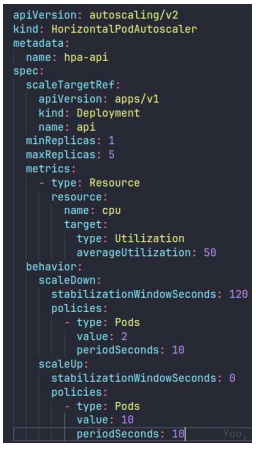

Figura 8: Arquivo de horizontalPodAutoscaler utilizado no serviço de pedidos

Em seguida, um novo objeto *Service* foi criado em formato YAML para expor a porta do *Pod* do *Deployment*. O *Service* possui configurações no campo "*spec*". Nele, o campo "*selector*" especifica a seleção do *Pod* desejado com base nos rótulos (*labels*) atribuídos. No caso, o rótulo usado é "api" com o valor correspondente ao nome do *Pod*.

A exposição da porta é configurada no campo "*ports*". A porta é nomeada como "http" e utiliza o protocolo TCP. A porta é definida como 80, com a porta alvo (*targetPort*) definida como 5000, correspondendo à porta definida no arquivo de *Deployment*.

Além disso, o tipo de porta escolhido para o *Service* é "*NodePort*", com a porta especificada como 30080. Essa configuração permite que o serviço seja acessível externamente através do IP do nó do *cluster*, seguido pela porta *NodePort*. Segue abaixo o arquivo de configuração de um *service*:

Figura 9: Arquivo de service utilizado no serviço de pedidos

7.3 Aplicando os arquivos

De antemão, o Minikube consiste em uma ferramenta que permite executar um *cluster* Kubernetes localmente em uma única máquina, facilitando o desenvolvimento e o teste de aplicativos.

Após instalar o Minikube, com os arquivos YAML prontos, contendo as informações necessárias para criar um *Pod*, aplicou-se o manifesto ao *cluster* Minikube digitando o comando "kubectl apply -f nome_do_arquivo.yaml" no terminal. Esse processo foi repetido para cada arquivo criado. Para verificar se os *Pods* foram criados corretamente, utilizando o comando "kubectl get Pods", como é mostrado na Figura 10.

Figura 10: Pods rodando

Além disso, para acessar o Kubernetes *Dashboard* foi utilizado o comando "minikube dashboard" no terminal e o

painel de controle será aberto no navegador padrão. Assim, encontrará informações sobre os *Pods*, serviços e outros recursos do seu *cluster*.

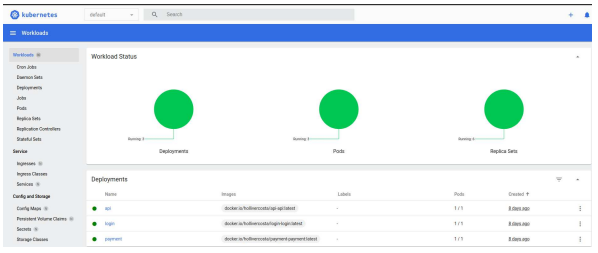

Figura 11: Dashboard do kubernetes

Vale ressaltar que o Minikube é melhor usado para fins de desenvolvimento e teste local. Para ambientes de produção, é recomendado o uso de um *cluster* Kubernetes completo em um ambiente de nuvem.

7.4 Acessando os serviços

Inicialmente, para acessar externamente uma aplicação implantada no Kubernetes através do Minikube, utilizando o recurso de encaminhamento de porta (*port forwarding*) do Minikube.

Com os *Pods* rodando, foi empregado o comando "kubectl get services" para acessar todos os serviços que estão rodando e depois usando o comando "minikube service" seguido do nome do serviço, o Minikube cuidará de redirecionar a porta para que você possa acessar a aplicação implantada no Kubernetes externamente.

Após executar o comando, o navegador padrão será aberto e você poderá acessar a aplicação.

8. DISCUSSÃO

Esta seção apresenta discussão que tem como objetivo descrever a aplicação WaiterApp, a qual adotou a arquitetura de microsserviços utilizando Docker e Kubernetes. A análise discute os impactos em termos de escalabilidade, disponibilidade e eficiência operacional em um cenário hipotético.

A implementação dos microsserviços, utilizando a abordagem de *containers* com Docker e a orquestração pelo Kubernetes, trouxe desafios e benefícios significativos para a aplicação em questão. Um dos desafios encontrados foi a necessidade de dividir a aplicação em serviços independentes e especializados. Essa demanda exigiu uma análise minuciosa das funcionalidades e dependências existentes, bem como uma reestruturação do código e da lógica de negócios para se adequarem à arquitetura de microsserviços.

No contexto deste trabalho, outro desafio relevante foi o processo de configuração e gerenciamento do *cluster* Kubernetes. Foi necessário definir a configuração dos *Pods*, estabelecendo limites para cada um deles, a fim de garantir a disponibilidade, escalabilidade e resiliência dos microsserviços.

No entanto, os benefícios obtidos com a implementação dessa arquitetura foram consideráveis. A escalabilidade dos microsserviços possibilitou dimensionar cada serviço de forma independente e automatizada. No estudo em questão, utilizou-se o consumo de *CPU* como parâmetro para ajustar a quantidade de *Pods*, resultando em uma utilização mais eficiente dos recursos disponíveis e maior eficiência operacional. Além disso, a disponibilidade e resiliência foram aprimoradas, uma vez que o Kubernetes foi capaz de detectar e recuperar automaticamente falhas nos serviços, criando novos *Pods* para tolerar as falhas e manter a aplicação em funcionamento, mesmo em situações de instabilidade. Tais benefícios foram alcançados sem a necessidade de configurações adicionais para a tolerância a falhas.

Outro benefício importante foi a flexibilidade na implantação de novas funcionalidades. Com a arquitetura de microsserviços, tornou-se possível adicionar, modificar ou remover serviços de forma independente, seja através da adição de parâmetros nos arquivos de configuração YAML, ou pela criação de novos arquivos. Essa flexibilidade permitiu maior agilidade no desenvolvimento e implantação da aplicação, facilitando a entrega contínua de novas funcionalidades.

Em resumo, a implementação de microsserviços utilizando Docker e Kubernetes apresentou desafios relacionados à divisão da aplicação em serviços independentes e ao gerenciamento do *cluster* Kubernetes. No entanto, os benefícios obtidos, como escalabilidade, disponibilidade, resiliência e flexibilidade na implantação, justificam a adoção dessa arquitetura, contribuindo para a eficiência e agilidade no desenvolvimento e operação da aplicação.

9. CONCLUSÃO

Neste trabalho, com o objetivo de implementar e estudar a arquitetura de microsserviços foram desenvolvidos três microsserviços para a aplicação WaiterApp. Utilizou-se o Docker para criar as imagens dos microsserviços. Em seguida, o Kubernetes foi adotado como a plataforma de orquestração para gerenciar os containers, configurar os Pods e garantir a escalabilidade, disponibilidade e resiliência dos microsserviços.

Verificou-se que a implementação dos microsserviços na aplicação demonstrou os benefícios da arquitetura de microsserviços utilizando Docker e Kubernetes proporcionando escalabilidade, disponibilidade, resiliência e eficiência operacional para a aplicação. A adoção dessa arquitetura permitiu a divisão da aplicação em serviços independentes e especializados, resultando em uma melhor organização e flexibilidade no desenvolvimento e implantação.

A utilização do Docker possibilitou empacotar os microsserviços em *containers*, facilitando o processo de implantação e garantindo a consistência do ambiente de execução. A orquestração pelo Kubernetes foi fundamental para a gestão do *cluster*, permitindo a configuração dos *Pods* e garantindo a escalabilidade, disponibilidade e resiliência dos microsserviços.

Além disso, a capacidade do Kubernetes em detectar e recuperar falhas automaticamente pode contribuir para

manter a aplicação em funcionamento, mesmo em situações de instabilidade, aumentando sua confiabilidade.

Ainda assim, habilidades adquiridas durante o curso, como a criação e orquestração de um *container*, foram essenciais para o desenvolvimento do trabalho, no qual pôde-se ter um breve contato com as ferramentas utilizadas na aplicação.

Todavia, o estudo também revelou desafios a serem superados, uma análise criteriosa das funcionalidades e dependências existentes foi necessária, assim como a reestruturação do código e da lógica de negócios para adequação à arquitetura de microsserviços. Além disso, a configuração e o gerenciamento do *cluster* Kubernetes exigiram atenção para garantir o funcionamento adequado da aplicação.

Para aprimorar ainda mais a arquitetura de microsserviços do WaiterApp, podem ser explorados recursos como o uso de um balanceador de carga para distribuir o tráfego de forma equilibrada entre os diferentes serviços. Além disso, adotar bancos de dados separados para cada serviço proporcionará maior isolamento, flexibilidade e escalabilidade. Práticas avançadas de monitoramento e observabilidade, como o uso de métricas, logs e ferramentas como o Prometheus e Grafana, devem ser implementadas para análise de desempenho, detecção de problemas e otimização contínua. Essas medidas contribuirão para uma melhor utilização dos recursos disponíveis e para a resolução proativa de problemas.

10. AGRADECIMENTOS

Primeiramente, gostaria de agradecer aos meus pais, que sempre estiveram ao meu lado, apoiando e investindo na minha educação. Sua dedicação em proporcionar uma base sólida e uma educação de qualidade foi fundamental para meu crescimento pessoal e profissional. Sem o apoio incondicional deles, eu não estaria aqui hoje, celebrando a conclusão deste importante capítulo da minha vida.

Também gostaria de expressar minha profunda gratidão à Rebeca Braga. Durante toda a minha jornada acadêmica, ela tem sido um pilar de força, um suporte incansável e uma fonte constante de motivação. Sua confiança em mim e seu constante encorajamento foram essenciais para superar os desafios que surgiram ao longo do caminho. Agradeço por sempre acreditar no meu potencial e me mostrar que sou capaz de alcançar grandes conquistas. Te amo muito.

Além disso, gostaria de agradecer aos professores e corpo docente do curso de Ciência da Computação, que dedicaram seu tempo e conhecimento para nos ensinar e nos inspirar. Especialmente ao meu orientador Fabio Jorge Almeida Morais, por toda atenção, dedicação e disponibilidade para/comigo.

Com gratidão em meu coração, encerro este trabalho de conclusão de curso reconhecendo que cada pessoa que cruzou meu caminho contribuiu de alguma forma para o meu crescimento e sucesso. Estou confiante de que o conhecimento adquirido e as experiências vividas durante o curso de Ciência da Computação serão a base sólida para minha carreira profissional e me guiarão ao longo de toda a minha vida. Obrigado a todos que fizeram parte dessa jornada e acreditaram em mim.

11. REFERÊNCIAS

[1] Morais, F (2022). Provisionamento e Operação de infraestrutura, Universidade Federal de Campina Grande, Paraíba.

[2] Documentação do Kubernetes https://kubernetes.io/pt-br/docs/home/

[3] Documentação do Docker https://www.docker.com/

[4] Deploying Microservices on Kubernetes https://medium.com/aspnetrun/deploying-microservices-on-kubernetes-35296d369fdb

[5] Microservice architecture style https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices