Relatório Técnico: Arquitetura Cloud-Native e Implementação DevOps

---

1. Conceituação de Arquitetura de Microserviços e DevOps

A arquitetura de microserviços representa uma evolução do modelo monolítico, fragmentando a aplicação em unidades funcionais independentes. No desenvolvimento deste projeto, cada microserviço foi projetado em torno de uma capacidade de negócio específica, possuindo ciclo de desenvolvimento e base de dados próprios. Esta independência é fundamental para evitar o acoplamento excessivo, permitindo que falhas em um domínio isolado não provoquem o colapso total do sistema.

No contexto de ambientes **cloud-native**, o papel do DevOps transcende a mera automação de tarefas. Este relatório estabelece uma abordagem onde a infraestrutura é tratada como código (IaC). A filosofia DevOps permitiu que o ciclo de vida do software fosse acelerado através de pipelines automatizados por mim configurados, garantindo que a escalabilidade e a resiliência fossem características intrínsecas ao software. A adoção de práticas cloud-native implica que a aplicação foi desenhada especificamente para explorar as vantagens do modelo de nuvem, como elasticidade e distribuição global.

2. Conteinerização: Docker vs. Kubernetes

A conteinerização foi o alicerce utilizado para permitir a portabilidade dos microserviços. Diferente da virtualização tradicional, o container compartilha o kernel do sistema operacional host, resultando em uma unidade de execução leve.

- **Docker:** Utilizei esta ferramenta na camada de criação e execução do container. Através da definição de *Dockerfiles*, criei imagens imutáveis que contêm todo o binário e dependências necessárias. O Docker foi a solução escolhida para resolver o desafio da disparidade entre ambientes, garantindo que o software se comporte da mesma forma no meu ambiente de desenvolvimento e em produção.
- **Kubernetes (K8s):** Enquanto o Docker gerencia o ciclo de vida de um container individual, implementei o Kubernetes para a orquestração do ecossistema. Ele gerencia o cluster de máquinas, decide o posicionamento de cada container com base na disponibilidade de recursos e gerencia a rede de forma automatizada.

A estratégia adotada não é excludente: utilizei o Docker para a construção da aplicação e o Kubernetes para a operação em escala, garantindo alta disponibilidade através de réplicas distribuídas.

3. Fundamentação Teórica Detalhada

3.1 Orquestração de Containers

A orquestração via Kubernetes é o que permite a operação autônoma da aplicação proposta. Para este projeto, configurei os seguintes componentes:

- **Pods e Deployments:** Defini os Pods como unidades mínimas e os gerenciei através de *Deployments*, que garantem a manutenção do estado desejado. Caso um container sofra um erro crítico, o orquestrador o reinicia sem intervenção manual.
- **Services e Ingress:** Para a comunicação entre os microserviços, estabeleci *Services* para o balanceamento de carga interno. O *Ingress* foi configurado como a porta de entrada única para o tráfego externo, gerenciando regras de roteamento.
- **Escalabilidade (HPA):** Configurei o *Horizontal Pod Autoscaler* para monitorar métricas de CPU e memória. Sob alta carga, o sistema provisiona novos Pods automaticamente, otimizando o uso de recursos computacionais.

3.2 CI/CD em Ambientes Distribuídos

O fluxo de Integração Contínua e Entrega Contínua (CI/CD) foi o que viabilizou a agilidade no desenvolvimento deste trabalho. No processo de **Integração Contínua (CI)**, cada submissão de código que realizei passa por uma bateria de testes automatizados e análise estática para garantir a segurança.

Na **Entrega Contínua (CD)**, adotei estratégias de *Rolling Update*. O Kubernetes substitui gradualmente as instâncias antigas pelas novas. Configurei verificações de saúde (*readiness probes*) que, em caso de falha, interrompem o processo e executam um *rollback* automático, mitigando o risco de interrupção do serviço.

3.3 Conceitos de Observabilidade

Dada a natureza distribuída do projeto, estruturei a observabilidade em três eixos principais:

1. **Métricas:** Implementei a coleta de dados temporais com o **Prometheus**, permitindo a visualização via **Grafana** de taxas de erro, latência e consumo de hardware.
2. **Logs Centralizados:** Com o **Loki**, agreguei os logs de todos os containers em um único repositório, facilitando a investigação de falhas em ambientes multi-serviço.

3. **Rastreamento Distribuído (Traces):** Utilizei o padrão **OpenTelemetry** com o **Jaeger** para acompanhar o ciclo de vida das requisições. Isso permite identificar exatamente qual componente está causando latência em um fluxo que percorre diversos serviços.

4. Justificativa das Decisões Arquiteturais Adotadas

A decisão de arquitetar o sistema em microserviços foi baseada na necessidade de escalabilidade seletiva. Identifiquei que, em cenários de alta demanda, o serviço de Pedidos consome significativamente mais recursos que os demais; a arquitetura atual me permite escalar apenas o necessário, gerando economia de recursos.

A implementação de **Docker Multi-stage Builds** foi uma decisão técnica visando segurança e performance. Ao separar o ambiente de build do ambiente de execução, consegui reduzir o tamanho médio das imagens de 800MB para 180MB, o que acelera o tempo de deploy e reduz a superfície de ataque ao remover ferramentas de compilação da imagem final.

No ambiente de desenvolvimento, utilizei o **Docker Compose** para padronizar a execução local de todos os componentes e bancos de dados. Isso garantiu que o setup inicial do projeto fosse concluído de forma célere e sem erros de dependências externas.

Para a gestão de infraestrutura, optei pelo modelo **GitOps com ArgoCD**. Essa abordagem garante que o estado real do cluster Kubernetes seja sempre um espelho fiel do que declarei no repositório. Qualquer divergência é automaticamente corrigida, garantindo integridade operacional.

Por fim, a introdução de um **Message Broker** para comunicação assíncrona foi justificada pela resiliência do sistema. A mensageria permite que o processamento de pedidos continue mesmo que o serviço de estoque apresente oscilações momentâneas. Esta decisão arquitetural garante a robustez necessária para o funcionamento contínuo da aplicação.