Relatório Pedagógico: Guia para o Desenvolvimento de Sistemas Corporativos com Microserviços e o Método CERTO

Este relatório detalha um plano didático e pedagógico para a disciplina de Desenvolvimento de Sistemas Corporativos, focando na arquitetura de microserviços e na integração do Método CERTO como ferramenta essencial para o aprendizado e a gestão de projetos. O objetivo é capacitar os alunos com conhecimentos teóricos e práticos para projetar, desenvolver e implantar sistemas corporativos modernos e resilientes.

1. Introdução: A Jornada de Aprendizagem em Sistemas Corporativos Modernos

A disciplina de Desenvolvimento de Sistemas Corporativos no 6º Período visa proporcionar aos alunos uma imersão prática nas arquiteturas e metodologias mais relevantes do mercado atual. A complexidade e as demandas de escalabilidade, resiliência e agilidade dos negócios modernos exigem abordagens arquiteturais que superem as limitações dos sistemas monolíticos tradicionais. A arquitetura de microserviços surge como a resposta predominante a esses desafios, permitindo que as organizações inovem e se adaptem rapidamente.

Historicamente, as aplicações corporativas eram frequentemente construídas como monólitos, onde todas as funcionalidades eram fortemente integradas em uma única base de código. Embora essa abordagem simplificasse o desenvolvimento inicial, ela se tornava um gargalo para aplicações grandes e em evolução, dificultando atualizações frequentes e a escalabilidade.[1] A ascensão da arquitetura de microserviços representa uma resposta direta a esses desafios, propondo que uma única aplicação seja composta por componentes pequenos e independentes que se comunicam por meio de interfaces bem definidas.[1]

A transição para microserviços não é meramente uma escolha técnica, mas uma

resposta estratégica às crescentes exigências por agilidade, tempo de lançamento no mercado mais rápido e inovação contínua no cenário corporativo. Essa mudança implica uma reestruturação organizacional, movendo as equipes de uma organização focada em tecnologia (como equipes de frontend, backend, banco de dados) para equipes organizadas em torno de capacidades de negócio, cada uma responsável por um serviço específico.[1] Esta transformação sublinha que as decisões arquiteturais influenciam diretamente a estrutura organizacional e os processos de desenvolvimento, preparando os futuros profissionais para as profundas mudanças que acompanham tais transições.

Neste contexto, o projeto prático será o pilar central da aprendizagem, permitindo que os alunos apliquem conceitos teóricos em um cenário realista. Para otimizar a clareza na definição de requisitos, na comunicação entre equipes e na interação com ferramentas de Inteligência Artificial, integraremos o **Método CERTO** (Contexto, Exigências, Referências, Tarefas, Observações) como uma estrutura pedagógica fundamental. Este método, inicialmente concebido para criar prompts de IA eficazes [1], será expandido para atuar como uma ferramenta robusta de engenharia de requisitos e gestão de projetos. Ao ensinar os alunos a aplicar um método projetado para prompts claros de IA ao contexto mais amplo da definição de projetos de software e resolução de problemas, a disciplina os equipa com uma estrutura versátil para articular ideias complexas, identificar restrições, aproveitar o conhecimento pré-existente e especificar ações claras em qualquer domínio. Isso eleva o resultado do aprendizado além da mera proficiência técnica para abranger competências críticas de comunicação profissional e gestão de projetos.

2. Fundamentos Essenciais da Arquitetura de Microserviços

Para que os alunos compreendam a necessidade e os benefícios dos microserviços, é crucial estabelecer uma base sólida nos seus princípios e padrões.

2.1. Definição e Evolução dos Sistemas Corporativos

Conforme mencionado, sistemas corporativos historicamente foram construídos

como monólitos – aplicações onde todas as funcionalidades eram fortemente integradas em uma única base de código. Essa abordagem, embora mais simples para o desenvolvimento inicial, frequentemente se tornava um gargalo para aplicações grandes e em evolução, dificultando atualizações frequentes e a escalabilidade.[1] A ascensão da arquitetura de microserviços surge como uma resposta direta a esses desafios, propondo que uma única aplicação seja composta por componentes pequenos e independentes, que se comunicam entre si através de interfaces bem definidas.[1]

2.2. O Paradigma de Microserviços: Vantagens e Desafios

A arquitetura de microserviços oferece um conjunto distinto de vantagens que a tornam atraente para sistemas corporativos. Entre elas, destacam-se a escalabilidade independente, que permite alocar recursos de forma mais eficiente ao escalar apenas os serviços que realmente precisam, em vez de todo o sistema.[1] Promove também a resiliência e tolerância a falhas, pois a falha de um único serviço não derruba toda a aplicação.[1] O desenvolvimento e a implantação independentes aceleram a entrega de novas funcionalidades, e a flexibilidade tecnológica permite que diferentes serviços sejam desenvolvidos em diferentes linguagens e tecnologias.[1] Além disso, bases de código menores e mais focadas aprimoram a manutenibilidade.[1]

No entanto, a arquitetura de microserviços introduz novos desafios que exigem consideração cuidadosa. Aumenta a complexidade de gerenciamento devido ao maior número de serviços e à comunicação distribuída.[1] A observabilidade se torna mais difícil, pois rastrear requisições através de múltiplos serviços exige ferramentas robustas de monitoramento e rastreamento distribuído.[1] A consistência de dados distribuídos é mais intrincada, já que as transações ACID tradicionais não se aplicam em um ambiente com múltiplos bancos de dados.[1] Por fim, a superfície de ataque para segurança aumenta, com mais pontos de entrada e comunicação inter-serviços a serem protegidos.[1]

A promessa de escalabilidade independente, resiliência e agilidade dos microserviços é equilibrada por uma complexidade operacional e de desenvolvimento significativamente maior.[1] Isso significa que os benefícios não são automáticos; eles exigem um investimento substancial em ferramentas robustas, padrões de design sofisticados e uma cultura DevOps madura. Sem essas implementações, a complexidade pode rapidamente anular as vantagens, resultando em um sistema mais

difícil de gerenciar do que um monólito bem construído. Essa compreensão é vital para os alunos, pois os prepara para a realidade de que a adoção de microserviços exige uma abordagem holística e um compromisso com a excelência operacional.

2.3. Princípios Fundamentais e Padrões de Design em Microserviços

2.3.1. Decomposição de Domínios de Negócio e Limites de Serviço

Um dos princípios mais críticos no desenvolvimento de microserviços é a modelagem de serviços em torno de capacidades de negócio, e não de tecnologias.[1] Isso significa que cada microserviço deve encapsular uma funcionalidade de negócio específica e autônoma, como "Gerenciamento de Usuários" ou "Processamento de Pedidos", em vez de ser uma camada técnica como "Serviço de Banco de Dados". A aplicação de Domain-Driven Design (DDD) é fundamental para identificar contextos delimitados (Bounded Contexts) e definir limites de serviço claros.[1]

O sucesso de uma arquitetura de microserviços depende criticamente de uma decomposição de domínio eficaz, que é um desafio inerentemente de negócio e organizacional, e não apenas técnico.[1] Uma decomposição inadequada pode levar a "monólitos distribuídos", onde as alterações em um serviço exigem alterações em muitos outros, minando a implantação independente e anulando os benefícios dos microserviços.[1] A ênfase no DDD destaca que a compreensão do domínio de negócio é primordial, mesmo para arquitetos técnicos, pois influencia diretamente os limites do serviço e, consequentemente, a capacidade de evolução e manutenção do sistema. Essa interligação profunda entre o design arquitetural e a análise de negócios é um aprendizado essencial para os alunos.

2.3.2. Padrões Essenciais de Comunicação (Síncrona, Assíncrona, Orientada a Eventos)

A comunicação entre microserviços é um aspecto central da arquitetura distribuída, e

a escolha do padrão de comunicação impacta diretamente o desempenho, a escalabilidade e a resiliência do sistema.[1] Existem dois paradigmas fundamentais: síncrono e assíncrono.

A **comunicação síncrona (Request-Response)** envolve um serviço enviando uma requisição a outro e aguardando uma resposta antes de continuar seu próprio processamento, geralmente via APIs RESTful sobre HTTP/HTTPS ou gRPC.[1] Suas vantagens incluem simplicidade e facilidade de implementação para interações em tempo real. No entanto, introduz acoplamento temporal (serviços precisam estar disponíveis simultaneamente), latência e o potencial para falhas em cascata.[1]

A **comunicação assíncrona**, por outro lado, permite que o serviço solicitante envie uma mensagem ao receptor e continue suas próprias tarefas sem esperar uma resposta imediata, utilizando message brokers como RabbitMQ, Kafka ou AWS SQS.[1] As vantagens incluem desacoplamento (serviços não precisam estar disponíveis simultaneamente), melhor escalabilidade e maior resiliência. As desvantagens são o aumento da complexidade e desafios relacionados à ordenação e rastreamento de mensagens.[1] Uma variação é a

comunicação orientada a eventos (Event-Driven), onde microserviços se comunicam através de um message broker que facilita a troca de eventos, permitindo desacoplamento extremo e suporte a notificações em tempo real.[1]

A escolha entre padrões de comunicação síncronos e assíncronos é uma decisão arquitetural fundamental. Uma abordagem híbrida, utilizando ambos os padrões, é frequentemente ideal para sistemas corporativos complexos, exigindo uma consideração cuidadosa das compensações.[1] A decisão por uma interação de serviço específica deve ser impulsionada pelo requisito de negócio, como a necessidade de interação em tempo real versus consistência eventual. Esta perspectiva ensina os alunos a traduzir necessidades de negócio em decisões arquiteturais, compreendendo os

trade-offs práticos envolvidos no design de sistemas distribuídos.

Padrões complementares de comunicação incluem o **API Gateway**, que atua como um único ponto de entrada para todas as requisições de clientes, gerenciando tarefas como balanceamento de carga, roteamento e autenticação, simplificando a complexidade para os clientes.[1] O

Service Mesh (e.g., Istio) é uma camada de infraestrutura que aprimora a comunicação entre microserviços, fornecendo recursos como gerenciamento de

tráfego, segurança (mTLS) e observabilidade, lidando com preocupações transversais e liberando os desenvolvedores para focar na lógica de aplicação.[1] A emergência do API Gateway e do Service Mesh como componentes de infraestrutura críticos demonstra que a adoção de microserviços exige uma abordagem mais sofisticada para a comunicação inter-serviços.

2.3.3. Padrões de Resiliência e Tolerância a Falhas

Em sistemas distribuídos, as falhas são inerentes. Portanto, em vez de tentar construir sistemas perfeitamente livres de falhas, o foco se desloca para projetar sistemas que possam degradar graciosamente e se recuperar de falhas.[1] Essa filosofia impulsiona diretamente a necessidade de padrões como o

Circuit Breaker Pattern, que impede falhas em cascata interrompendo requisições adicionais para um serviço que está falhando até que ele se recupere.[1] O

Bulkhead Pattern, inspirado nas anteparas de um navio, isola falhas em pools separados, confinando o impacto da falha.[1] Outros padrões incluem o

Retry Mechanism para tentar novamente operações que falharam temporariamente, e **Timeouts** para definir limites de tempo para as respostas dos serviços, evitando que requisições fiquem penduradas indefinidamente.[1]

Além desses padrões, o conceito de "engenharia do caos" surge como uma prática crítica para testar e validar proativamente a resiliência do sistema, garantindo que ele se comporte como esperado sob estresse e falhas parciais.[1] A natureza distribuída dos microserviços torna o tratamento robusto de falhas uma preocupação primordial, expandindo os limites da engenharia de confiabilidade tradicional. Este aprendizado é fundamental para que os alunos compreendam que a robustez de um sistema distribuído reside na sua capacidade de gerenciar e se recuperar de falhas esperadas.

2.3.4. Padrões de Dados (Database per Service, Saga Pattern)

A gestão de dados em arquiteturas de microserviços difere significativamente dos

monólitos, principalmente devido ao princípio de "Database per Service".[1] Este padrão preconiza que cada microserviço possua seu próprio banco de dados, garantindo desacoplamento e gerenciamento de dados independente.[1] As vantagens incluem autonomia para cada serviço, permitindo que as equipes escolham a tecnologia de banco de dados mais adequada (persistência poliglota) para suas necessidades específicas.[1]

No entanto, a principal desvantagem são os desafios na consistência de dados entre serviços, pois as transações ACID tradicionais não se aplicam em um ambiente distribuído.[1] Para gerenciar a consistência de dados em transações distribuídas, o

Saga Pattern é empregado. Ele divide uma transação de negócio complexa em múltiplas transações locais. Cada transação local atualiza os dados dentro de um único serviço e publica um evento. Outros serviços escutam esses eventos e realizam suas próprias transações locais. Se uma transação local falhar, transações de compensação são executadas para desfazer as alterações, garantindo a consistência eventual.[1]

O padrão "Database per Service", embora crucial para a autonomia do serviço, altera fundamentalmente a abordagem da consistência de dados, passando de transações ACID para consistência eventual, o que exige novos padrões de design como Saga.[1] Essa mudança implica uma alteração fundamental na forma como os desenvolvedores raciocinam sobre a integridade dos dados e como os processos de negócio que abrangem vários serviços são projetados e implementados. Os alunos devem compreender que a flexibilidade arquitetural da persistência poliglota vem com a complexidade de gerenciar a consistência em armazenamentos de dados distribuídos, exigindo uma nova mentalidade para garantir a integridade dos dados.

2.3.5. Observabilidade e Monitoramento Distribuído

Em uma arquitetura de microserviços, ter visibilidade sobre o comportamento e o desempenho de serviços individuais e do sistema como um todo é essencial.[1] A observabilidade é garantida por meio de monitoramento robusto, logging e mecanismos de recuperação automatizados.[1] A coleta e agregação de logs de todos os serviços em um local centralizado (

Logging Centralizado) é crucial para depuração e auditoria.[1] O

Distributed Tracing (e.g., OpenTelemetry) permite rastrear uma única requisição de usuário à medida que ela atravessa múltiplos microserviços, identificando gargalos de desempenho e a causa raiz de falhas.[1] A coleta de

Métricas e Dashboards (ex: Prometheus com Grafana) de desempenho e saúde de cada serviço fornece *insights* em tempo real sobre o estado do sistema.[1]

A observabilidade não se trata apenas de coletar dados; trata-se de compreender o estado de um sistema distribuído sem acesso direto aos seus internos. Em uma aplicação monolítica, a depuração é relativamente simples, pois todos os componentes estão em um único local. Em microserviços, uma única requisição de usuário pode atravessar dezenas de serviços. Sem logging centralizado e rastreamento distribuído, identificar a causa raiz de um problema se torna incrivelmente difícil.[1] Esta complexidade demonstra que as ferramentas de observabilidade não são opcionais, mas requisitos fundamentais para operar microserviços em produção, impactando diretamente o tempo médio de recuperação (MTTR) de incidentes. Para os alunos, isso significa que a capacidade de diagnosticar e resolver problemas em ambientes distribuídos é uma competência tão crítica quanto a própria codificação.

2.3.6. Considerações de Segurança em Ambientes Distribuídos

A descentralização, embora benéfica para a agilidade, introduz desafios de segurança significativos que exigem uma abordagem de "defesa em profundidade" em todas as rotas de comunicação e limites de serviço.[1] A superfície de ataque aumenta devido a múltiplos pontos de entrada e à comunicação inter-serviços.[1] O

API Gateway atua como um ponto de aplicação de políticas de segurança, gerenciando a autenticação e autorização de requisições de clientes antes que elas cheguem aos serviços internos.[1]

É crucial proteger a comunicação entre os próprios microserviços, utilizando mecanismos como mTLS (mutual Transport Layer Security) para criptografia serviço-a-serviço.[1] A implementação de

Controle de Acesso Baseado em Função (RBAC) em cada serviço garante que apenas usuários ou outros serviços com as permissões corretas possam acessar funcionalidades específicas.[1] Em um monólito, a segurança pode ser frequentemente

gerenciada em um único perímetro. Com microserviços, cada serviço potencialmente expõe uma API, e a comunicação inter-serviços cria inúmeros novos vetores de ataque.[1] Isso exige uma mudança da segurança baseada em perímetro para controles de segurança granulares em cada limite de serviço. Esta abordagem demonstra que a segurança deve ser uma parte integrante do design desde o início ("segurança por design"), em vez de uma reflexão tardia, dada a natureza distribuída do sistema.

3. Tecnologias e Ferramentas Modernas para Desenvolvimento de Microserviços

A escolha das tecnologias é um componente vital do projeto, permitindo aos alunos experimentar o ecossistema atual de desenvolvimento corporativo.

3.1. Backend: Análise e Recomendação de Frameworks e Linguagens

A flexibilidade tecnológica é uma das vantagens dos microserviços, permitindo que as equipes escolham a ferramenta mais adequada para cada tarefa específica.[1] A diversidade de frameworks populares, especialmente o surgimento de frameworks otimizados para nuvem (como Quarkus e Micronaut) ao lado de opções estabelecidas (como Spring Boot), indica uma forte tendência da indústria em otimizar microserviços para ambientes conteinerizados e serverless, priorizando desempenho e eficiência de recursos.[1]

Para o **Ecossistema Java**, **Spring Boot** e **Spring Cloud** continuam sendo escolhas predominantes para o desenvolvimento rápido de aplicações, oferecendo um ecossistema extenso e configuração simplificada para serviços RESTful.[1]

Quarkus e **Micronaut** são frameworks modernos projetados para aplicações nativas de Kubernetes, com tempos de inicialização mais rápidos e menor uso de memória, otimizados para ambientes de nuvem.[1]

No **Ecossistema Python**, **FastAPI** é um framework web moderno para construir APIs com Python, projetado para serviços RESTful com esforço mínimo e gerando

automaticamente documentação de API interativa.[1]

Django + Django REST Framework (DRF) é um framework web de alto nível que promove o desenvolvimento rápido de sites seguros e de fácil manutenção, enquanto **Flask** é um micro-framework leve e popular, ideal para prototipagem rápida.[1]

Para o **Ecossistema Node.js**, **NestJS** é um framework recomendado por sua estrutura organizada, uso de TypeScript e aderência a padrões de arquitetura modular, injeção de dependências e *decorators*, facilitando a manutenção e testes.[1] Outras opções incluem

Express.js (minimalista, flexível), **Koa**, **Seneca** e **Feathers.js**.[1]

Outras linguagens e frameworks relevantes incluem **Go (Golang)** com **Go kit** para microserviços de alta performance e concorrência nativa, e **Kotlin** com **Ktor** para aplicações de servidor assíncronas.[1]

Enquanto Spring Boot permanece uma escolha dominante, a ascensão de Quarkus e Micronaut é uma resposta direta às demandas do desenvolvimento nativo da nuvem.[1] Seu foco em tempos de inicialização mais rápidos e menor pegada de memória é crucial para a utilização eficiente de recursos em Kubernetes e funções serverless, onde a escalabilidade rápida e a otimização de custos são fundamentais. Isso demonstra que a ampla adoção de Docker e Kubernetes impulsiona a evolução e a popularidade dos frameworks de backend adaptados para esses ambientes, um ponto crucial para os alunos compreenderem a interconexão entre infraestrutura e desenvolvimento de software.

A Tabela 1 oferece uma visão comparativa dos frameworks de backend mais relevantes para o desenvolvimento de microserviços, auxiliando na tomada de decisão informada.

Tabela 1: Comparativo de Frameworks Backend para Microserviços

Linguagem Principal	Framework/Estilo	Vantagens Chave para Microserviços	Casos de Uso Comuns	Nível de Maturidade/Comunidade
Java	Spring Boot	Ecossistema maduro, dev. rápido, vasta	Aplicações corporativas, RESTful APIs,	Alta (predominante) [1]

		documentação, integração fácil [1]	microsserviços complexos	
Java	Quarkus	Otimizado para cloud-native, menor consumo de memória, boot rápido [1]	Serverless, Kubernetes, microsserviços de alta performance	Crescendo rapidamente [1]
Java	Micronaut	Compilação AOT, baixo overhead, inicialização rápida [1]	Microsserviços de alta performance, funções serverless	Crescendo rapidamente [1]
Python	FastAPI	Alta performance, documentação automática, validação de dados, assíncrono [1]	APIs RESTful, WebSockets, microsserviços leves	Alta (moderno) [1]
Python	Django + DRF	Dev. rápido, "batteries-included", admin interface, comunidade forte [1]	Aplicações web complexas, APIs robustas, sistemas de gerenciamento	Alta (maduro) [1]
Node.js	Express.js	Minimalista, flexível, assíncrono [1]	APIs RESTful, microsserviços leves, aplicações em tempo real	Alta (maduro) [1]
Go	Go kit	Simplicidade, alta performance, concorrência nativa [1]	Microsserviços de alta performance, sistemas distribuídos	Moderada a Alta [1]

3.2. Frontend: Frameworks e Abordagens para Interfaces de Usuário

No contexto de microserviços, o frontend também evoluiu para lidar com a complexidade de sistemas distribuídos. A necessidade de desacoplar o desenvolvimento do frontend de estruturas monolíticas de UI, espelhando a abordagem de microserviços do backend, é cada vez mais evidente.[1]

Entre os **Frameworks Modernos**, **React** é um dos mais populares globalmente, com um vasto ecossistema (React Router para roteamento, Redux para gerenciamento de estado) e forte suporte da Meta (Facebook), sendo uma escolha confiável para aplicações de grande escala.[1]

Vue.js possui um ecossistema rico e crescente, oferecendo um equilíbrio entre simplicidade e escalabilidade.[1]

Angular tem uma forte presença em ambientes corporativos, ideal para projetos de nível empresarial.[1] Outros frameworks notáveis incluem

Svelte (rápido crescimento, desempenho), **Solid.js**, **Qwik** (arquitetura resumível, carregamento instantâneo) e **Astro** (agnóstico a frameworks, excelente para geração de sites estáticos).[1]

As **Abordagens para Micro Frontends** são cruciais para a modularidade da UI. O padrão **Backends for Frontends (BFF)** envolve o design de backends distintos para lidar com requisições de diferentes clientes (como mobile e web), ajudando a desacoplar o frontend dos detalhes de implementação do backend e otimizar para necessidades específicas do cliente.[1]

Micro Frontends é uma arquitetura onde a interface do usuário é composta por múltiplas aplicações frontend independentes, cada uma gerenciada por um microserviço ou equipe. Isso permite que diferentes partes da UI sejam de propriedade de equipes distintas, alinhando-se com a natureza descentralizada dos microserviços.[1]

Em uma arquitetura de microserviços, ter um único frontend monolítico pode se tornar um gargalo, reintroduzindo acoplamento forte na camada de UI.[1] O padrão BFF aborda isso, fornecendo APIs específicas para o cliente, reduzindo a necessidade de agregação complexa no lado do cliente. Isso se estende naturalmente ao conceito de "Micro Frontends", onde diferentes partes da UI são de propriedade de diferentes equipes, alinhando-se com a natureza descentralizada dos microserviços. Esta

extensão dos princípios arquiteturais dos microserviços para o lado do cliente demonstra que, para colher todos os benefícios da agilidade, a camada de UI também deve adotar estratégias de decomposição semelhantes, evitando que se torne um novo gargalo monolítico.

A Tabela 2 apresenta uma comparação dos frameworks de frontend modernos, auxiliando na escolha para projetos de microserviços.

Tabela 2: Comparativo de Frameworks Frontend Modernos

Framework	Popularidade/Comunidade	Ecossistema/Extensibilidade	Curva de Aprendizagem	Performance	Adequação para Micro Frontends	Casos de Uso Comuns
React	Mais popular, vasta comunidade [1]	Rico (Redux, Router), flexível [1]	Moderada	Boa	Alta (com ferramentas como Webpack Module Federation)	Aplicações de grande escala, SPAs, dashboards [1]
Vue.js	Crescendo, forte comunidade [1]	Rico (Vuex, Router, CLI), extensível [1]	Baixa a Moderada	Boa	Alta	Aplicações de médio porte, SPAs, prototipagem [1]
Angular	Forte em empresas [1]	Completo (CLI, RxJS), opinioso [1]	Alta	Boa	Moderada (requer planejamento)	Aplicações empresariais complexas, dashboards [1]
Svelte	Rápido crescimento [1]	Crescendo, menos dependên	Baixa	Excelente (compilação) [1]	Alta	Aplicações pequenas,

		cias [1]				componentes leves, alta performance [1]
Astro	Rápido crescimento [1]	Agnóstico a frameworks, integrável [1]	Baixa	Excelente (static site gen) [1]	Alta (para sites compostos)	Sites de conteúdo, e-commerce, blogs [1]

3.3. Bancos de Dados: Escolha de Soluções Relacionais e NoSQL

O princípio de "Database per Service" é fundamental na arquitetura de microserviços, onde cada serviço possui seu próprio banco de dados.[1] Isso permite a escolha da tecnologia de banco de dados mais adequada para as necessidades específicas de cada serviço, um conceito conhecido como persistência poliglota.[1]

Para **Bancos de Dados Relacionais (SQL)**, **PostgreSQL** é um poderoso banco de dados de código aberto, conhecido por seu excelente desempenho em memória, confiabilidade, escalabilidade e flexibilidade, sendo o líder em popularidade.[1]

MySQL é amplamente utilizado para aplicações web, valorizado por sua confiabilidade e simplicidade.[1] Outras opções incluem

MariaDB, **Microsoft SQL Server**, **Oracle Database** e **SQLite**.[1]

Para **Bancos de Dados Não Relacionais (NoSQL)**, **MongoDB** é um banco de dados de documentos que armazena dados em documentos flexíveis semelhantes a JSON, ideal para lidar com dados não estruturados, big data e projetos de IoT, conhecido por sua velocidade e escalabilidade.[1]

Redis é um banco de dados em memória de alta performance, ideal para aplicações em tempo real, caching e armazenamento de sessões.[1] Outras opções incluem

Firebase, **Amazon DynamoDB**, **Cassandra** e **ClickHouse**.[1]

O padrão "Database per Service" permite diretamente a persistência poliglota. Em vez

de forçar todos os serviços a usar um único banco de dados relacional, um serviço que lida com perfis de usuário pode usar PostgreSQL para forte consistência, enquanto um catálogo de produtos pode usar MongoDB para esquemas flexíveis, e um serviço de cache pode usar Redis para velocidade.[1] Essa otimização para casos de uso específicos maximiza o desempenho e a escalabilidade para serviços individuais. No entanto, introduz o desafio de gerenciar diversas tecnologias de banco de dados, exigindo experiência e ferramentas especializadas para cada uma, e complica a consistência de dados entre serviços (abordada pelo padrão Saga).[1] Para os alunos, isso demonstra que o gerenciamento de banco de dados em um ambiente de microserviços se torna um desafio distribuído em si, exigindo uma compreensão das compensações entre otimização e complexidade operacional.

A Tabela 3 detalha as opções de bancos de dados e seus casos de uso típicos em microserviços.

Tabela 3: Opções de Bancos de Dados para Microserviços e Seus Casos de Uso

Tipo de Banco de Dados	Nome do Banco de Dados	Principais Vantagens	Casos de Uso Típicos em Microserviços	Considerações para Escolha
Relacional (SQL)	PostgreSQL	Robustez, ACID, extensibilidade, suporte a JSON [1]	Gerenciamento de usuários, pedidos, transações financeiras	Para dados estruturados e alta integridade [1]
Relacional (SQL)	MySQL	Popularidade, facilidade de uso, escalabilidade [1]	Aplicações web gerais, blogs, e-commerce	Para dados estruturados, boa para web apps [1]
Não Relacional (NoSQL)	MongoDB	Flexibilidade de esquema, escalabilidade horizontal, desempenho [1]	Catálogos de produtos, perfis de usuário, dados de IoT, CMS	Para dados semi-estruturados ou não estruturados, agilidade [1]
Não Relacional (NoSQL)	Redis	In-memory, alta velocidade, estruturas de	Cache, sessões, filas de mensagens,	Para dados voláteis, alta performance de

		dados ricas [1]	leaderboards, real-time analytics	leitura/escrita [1]
Não Relacional (NoSQL)	Cassandra	Escalabilidade distribuída, alta disponibilidade, tolerância a falhas [1]	Big data, dados de sensor, sistemas de mensagens, logs	Para grandes volumes de dados distribuídos, alta escrita [1]
Não Relacional (NoSQL)	ClickHouse	Analítica em tempo real, processamento colunar [1]	Dashboards analíticos, relatórios, dados de eventos	Para cargas de trabalho analíticas pesadas [1]

3.4. Containerização e Orquestração: Docker e Kubernetes como Pilares de Implantação

Docker e Kubernetes não são apenas ferramentas de implantação; eles são facilitadores fundamentais do paradigma de microserviços, fornecendo a infraestrutura necessária para a implantação independente, escalabilidade e resiliência que os microserviços prometem.[1]

Docker (Containerização) padroniza o ambiente de execução para microserviços, encapsulando o código da aplicação, suas dependências e configurações em um contêiner leve e portátil.[1] Isso garante que o serviço funcione de forma consistente em diferentes ambientes (desenvolvimento, teste, produção), eliminando problemas de "funciona na minha máquina".[1]

Kubernetes (Orquestração) é uma plataforma de código aberto para gerenciar e automatizar a implantação, escalonamento e operação de aplicações em contêineres.[1] Ele oferece recursos essenciais como balanceamento de carga, atualizações contínuas (rolling updates) e auto-recuperação (self-healing), que são vitais para gerenciar a complexidade de múltiplos microserviços em produção.[1] Frameworks como Quarkus são projetados para serem nativos do Kubernetes, aproveitando ao máximo seus recursos.[1]

Microserviços inerentemente exigem implantação e escalabilidade independentes.[1]

Sem a conteinerização (Docker) para empacotar serviços com suas dependências em unidades isoladas e portáteis, e a orquestração (Kubernetes) para automatizar sua implantação, escalabilidade e gerenciamento em um cluster, a sobrecarga operacional dos microserviços seria proibitiva.[1] Os recursos do Kubernetes, como balanceamento de carga, atualizações contínuas e auto-recuperação, suportam diretamente os objetivos de resiliência e alta disponibilidade dos microserviços.[1] Esta forte relação de causa e efeito demonstra que Docker e Kubernetes são pré-requisitos essenciais para a realização eficaz dos benefícios de uma arquitetura de microserviços em produção, e seu surgimento influenciou o desenvolvimento de frameworks.

3.5. Automação de CI/CD e Estratégias de Deploy

A automação de CI/CD não é meramente uma boa prática operacional, mas um facilitador fundamental da agilidade prometida pelos microserviços.[1]

A **Continuous Integration (CI)** envolve a integração frequente de código em um repositório compartilhado, seguida por testes automatizados para detectar problemas de integração precocemente.[1] A

Continuous Delivery/Deployment (CD) automatiza a entrega e a implantação de novas funcionalidades e atualizações para os ambientes de teste e produção, acelerando a entrega de valor aos usuários e reduzindo o risco de implantações.[1]

As **Estratégias de Deploy** são cruciais para minimizar o impacto das atualizações. O **Blue-Green Deployment** envolve a manutenção de dois ambientes de produção idênticos ("azul" e "verde"), onde a nova versão é implantada no ambiente inativo e o tráfego é rapidamente alternado para ele, minimizando o tempo de inatividade e o risco.[1] A

Canary Release lança a nova versão gradualmente para um pequeno subconjunto de usuários, permitindo testar a nova versão em produção com risco controlado.[1]

Rolling Updates implantam novas instâncias de um serviço incrementalmente, substituindo as antigas e garantindo que o serviço permaneça disponível durante a atualização.[1]

A capacidade de implantar serviços de forma independente e frequente é uma

vantagem fundamental dos microserviços sobre os monólitos.[1] Essa agilidade só é alcançável por meio de pipelines de CI/CD robustos que automatizam testes e implantação. Sem automação, a sobrecarga de implantar vários serviços pequenos anularia os benefícios do desenvolvimento independente.[1] Esta observação demonstra que a arquitetura de microserviços exige um pipeline de CI/CD maduro para liberar todo o seu potencial de entrega rápida de funcionalidades e capacidade de resposta às necessidades de negócios em mudança.

4. O Método CERTO: Uma Ferramenta Didática para a Engenharia de Requisitos e Comunicação

O Método CERTO, embora explicitamente projetado para prompts de IA [1], possui uma abordagem estruturada para coleta de informações e formulação de requisições que é inerentemente transferível e altamente eficaz para definir qualquer escopo de projeto complexo, incluindo o desenvolvimento de software.[1] Ele funciona como uma estrutura universal para comunicação clara. Os elementos centrais do CERTO (Contexto, Exigências, Referências, Tarefas, Observações) são essencialmente uma forma formalizada de perguntar "quem, o quê, por que, quando, como e o que mais?", que são precisamente as perguntas que um arquiteto de software ou gerente de projeto faz ao coletar requisitos e definir um escopo de projeto.[1]

4.1. Revisão Detalhada dos Componentes do Método CERTO

Cada letra do acrônimo CERTO representa um elemento essencial para criar uma comunicação clara e eficaz:

- **C - Contexto:** O pano de fundo que oferece informações sobre a situação, o cenário ou a persona envolvida.[1] É essencial para garantir que a resposta esteja alinhada com o objetivo desejado.[1]
- **E - Exigências:** Requisitos ou restrições que a resposta deve seguir, como limitações financeiras, de tempo ou de recursos.[1] Elas ajudam a limitar e direcionar a solução, garantindo que seja prática e aplicável.[1]
- **R - Referências:** Dados, exemplos ou informações prévias que ajudam a contextualizar ainda mais o pedido.[1] Elas mostram o que já foi feito e quais

resultados foram obtidos, servindo como base para novas ideias.[1]

- **T - Tarefas:** As ações que se deseja que a ferramenta execute.[1] Elas precisam ser claras, diretas e objetivas para garantir que a resposta seja precisa e atinja o objetivo final.[1]
- **O - Observações:** Detalhes adicionais, preferências pessoais ou critérios específicos que se deseja que sejam considerados.[1] Elas são úteis para personalizar ainda mais a resposta.[1]

4.2. Exemplos Práticos de Como Usar o CERTO para Elaborar Requisitos para o Projeto de Microserviços

Ao estruturar os requisitos do projeto usando o CERTO, o projeto acadêmico pode servir como uma demonstração prática de como aplicar metodologias de comunicação estruturada a desafios reais de engenharia de software.[1] Esse processo não apenas esclarece o projeto para a equipe de desenvolvimento, mas também serve como uma excelente ferramenta pedagógica para os estudantes, ensinando-os a decompor problemas complexos, identificar restrições, aproveitar o conhecimento existente e especificar entregáveis claramente. Isso demonstra que o método CERTO pode ser integrado ao currículo do curso não apenas como uma ferramenta de escrita de prompts, mas como uma habilidade fundamental para a engenharia de requisitos e a comunicação de projetos em um ambiente profissional.

- **C - Contexto do Projeto:** "Somos uma equipe de estudantes de Ciência da Computação desenvolvendo uma plataforma de gestão de eventos corporativos para uma disciplina de bacharelado. A plataforma atual é monolítica e tem problemas de escalabilidade e manutenção. Nosso objetivo é modernizá-la usando microserviços para suportar o crescimento da empresa e facilitar a adição de novas funcionalidades." [1]
- **E - Exigências do Projeto:** "A aplicação deve ser desenvolvida em um prazo de um semestre acadêmico (aproximadamente 4 meses), utilizando tecnologias de código aberto (open-source) e sem custos de licenciamento significativos. Deve ser capaz de suportar 5.000 usuários simultâneos e ter uma disponibilidade mínima de 99.9% para serviços críticos. A solução deve ser implantável em um ambiente de nuvem." [1]
- **R - Referências (Lições Aprendidas/Melhores Práticas):** "Nossa experiência anterior com monólitos resultou em dificuldades de deploy e escalabilidade. Referências de arquiteturas de sucesso incluem plataformas como Netflix e

Amazon, que utilizam microserviços extensivamente. Considerar padrões como Database per Service, Circuit Breaker, API Gateway e comunicação assíncrona para resiliência e escalabilidade. A documentação do Método CERTO de Thais Martan [1] serve como base para a estruturação dos requisitos." [1]

- **T - Tarefas Principais do Projeto:** "Desenvolver os microserviços de Usuários, Eventos e Inscrições, incluindo suas respectivas APIs e bancos de dados. Implementar um API Gateway como ponto de entrada unificado. Configurar o ambiente de desenvolvimento e implantação utilizando Docker e Docker Compose para conteinerização. Definir e implementar estratégias de comunicação assíncrona para notificações. Desenvolver um frontend web utilizando um framework moderno." [1]
- **O - Observações Adicionais:** "Preferimos usar Java (Spring Boot/Quarkus) e Python (FastAPI) para o backend e React para o frontend. A documentação da arquitetura e das APIs deve ser detalhada, seguindo padrões como OpenAPI. O projeto deve ser implantável em um ambiente Kubernetes simulado (Minikube ou Kind) para demonstrar a orquestração de contêineres. Priorizar a observabilidade com logging centralizado e rastreamento distribuído." [1]

4.3. Benefícios do CERTO para a Clareza e Assertividade na Colaboração de Equipes

A aplicação do Método CERTO na definição do projeto oferece múltiplos benefícios [1]:

- **Redução de Ambigüidades:** A estrutura força a explicitação de todos os detalhes, minimizando mal-entendidos entre os membros da equipe e as partes interessadas.
- **Melhora na Assertividade das Entregas:** Ao alinhar claramente as expectativas desde o início, as entregas do projeto tendem a ser mais precisas e alinhadas aos objetivos.
- **Facilita a Colaboração:** Fornece uma linguagem comum e uma estrutura para discutir requisitos e decisões de design, facilitando a colaboração entre diferentes equipes ou módulos de uma equipe.
- **Ferramenta para Documentação:** Serve como um framework robusto para a criação de documentação de requisitos e especificações de design, garantindo que todas as informações cruciais sejam capturadas e comunicadas.

A Tabela 5 ilustra a aplicação prática do Método CERTO na definição de requisitos do

projeto de microserviços.

Tabela 5: Aplicação Prática do Método CERTO na Definição de Requisitos do Projeto

Componente CERTO	Definição (Thais Martan) [1]	Aplicação no Projeto de Microserviços (Exemplos Detalhados)	Justificativa/Benefício para o Projeto
C - Contexto	Pano de fundo, situação, cenário, persona. [1]	"Somos uma equipe de estudantes de Ciência da Computação desenvolvendo uma plataforma de gestão de eventos corporativos para uma disciplina de bacharelado. A plataforma atual é monolítica e tem problemas de escalabilidade e manutenção. Nosso objetivo é modernizá-la usando microserviços para suportar o crescimento da empresa e facilitar a adição de novas funcionalidades." [1]	Alinha a equipe e as partes interessadas sobre o propósito e a motivação do projeto, estabelecendo a base para todas as decisões.
E - Exigências	Requisitos ou restrições que a resposta deve seguir. [1]	"A aplicação deve ser desenvolvida em um prazo de um semestre acadêmico (aprox. 4 meses), utilizando tecnologias de código aberto (open-source) e sem custos de licenciamento	Define os limites e as metas mensuráveis do projeto, orientando as escolhas tecnológicas e arquiteturais e permitindo a avaliação do sucesso.

		significativos. Deve ser capaz de suportar 5.000 usuários simultâneos e ter uma disponibilidade mínima de 99.9% para serviços críticos. A solução deve ser implantável em um ambiente de nuvem." [1]	
R - Referências	Dados, exemplos ou informações prévias para contextualizar o pedido. [1]	"Nossa experiência anterior com monólitos resultou em dificuldades de deploy e escalabilidade. Referências de arquiteturas de sucesso incluem plataformas como Netflix e Amazon, que utilizam microserviços extensivamente. Considerar padrões como Database per Service, Circuit Breaker, API Gateway e comunicação assíncrona para resiliência e escalabilidade. A documentação do Método CERTO de Thais Martan [1] serve como base para a estruturação dos requisitos." [1]	Fornece uma base de conhecimento, evitando a reinvenção da roda e direcionando a equipe para soluções comprovadas e melhores práticas da indústria.
T - Tarefas	Ações que se deseja que a ferramenta execute. [1]	"Desenvolver os microserviços de Usuários, Eventos e Inscrições, incluindo	Esclarece as entregas e atividades principais, permitindo o planejamento do

		suas respectivas APIs e bancos de dados. Implementar um API Gateway como ponto de entrada unificado. Configurar o ambiente de desenvolvimento e implantação utilizando Docker e Docker Compose para conteinerização. Definir e implementar estratégias de comunicação assíncrona para notificações. Desenvolver um frontend web utilizando um framework moderno." [1]	trabalho e a atribuição de responsabilidades de forma clara.
O - Observações	Detalhes adicionais, preferências pessoais, critérios específicos. [1]	"Preferimos usar Java (Spring Boot/Quarkus) e Python (FastAPI) para o backend e React para o frontend. A documentação da arquitetura e das APIs deve ser detalhada, seguindo padrões como OpenAPI. O projeto deve ser implantável em um ambiente Kubernetes simulado (Minikube ou Kind) para demonstrar a orquestração de contêineres. Priorizar a observabilidade com logging centralizado e	Permite a personalização do projeto com base em preferências e requisitos específicos, garantindo que nuances importantes sejam consideradas e integradas ao design.

		rastreamento distribuído." [1]	

5. Roteiro Didático do Projeto: Etapas de Desenvolvimento com Aplicação do CERTO

O projeto proposto para a disciplina de Desenvolvimento de Sistemas Corporativos visa criar uma aplicação corporativa fazendo uso de microserviços, seguindo boas arquiteturas e boas práticas, utilizando Docker e diversas tecnologias.[1] Para este plano, o estudo de caso detalhado será uma

"Plataforma de Gestão de Eventos Corporativos".[1]

Contexto do Estudo de Caso: Uma empresa de médio porte organiza diversos eventos, como conferências, workshops e feiras, para seus clientes e funcionários. A plataforma atual é monolítica e enfrenta problemas de escalabilidade, manutenção e integração com novos serviços (por exemplo, pagamentos, notificações). O objetivo é modernizar essa plataforma, desenvolvendo uma nova baseada em microserviços para gerenciar todo o ciclo de vida dos eventos, desde a criação até a participação e o feedback.[1]

Definir requisitos não funcionais (NFRs) detalhados antecipadamente é de suma importância em microserviços, pois eles frequentemente ditam as escolhas arquiteturais (por exemplo, comunicação assíncrona para alta escalabilidade, padrões de resiliência específicos para alta disponibilidade).[1] Em um monólito, os NFRs são tipicamente definidos para todo o sistema. Em microserviços, os NFRs como escalabilidade e desempenho frequentemente se aplicam no nível do serviço (por exemplo, o Serviço de Notificações precisa de alta

throughput, enquanto o Serviço de Pagamentos precisa de alta consistência e baixa latência).[1] Essa aplicação granular dos NFRs influencia diretamente a escolha da tecnologia, os padrões de comunicação e as estratégias de implantação para cada serviço. Por exemplo, um NFR de alta disponibilidade para o Serviço de Pagamentos exigiria padrões como Circuit Breaker e Bulkhead, e potencialmente um banco de dados altamente resiliente. Isso demonstra que os NFRs são mais complexos de definir e alcançar em um sistema distribuído, exigindo uma abordagem mais matizada.

A Tabela 4 apresenta uma decomposição detalhada dos serviços propostos para o estudo de caso, mapeando tecnologias, padrões de design e requisitos não funcionais.

Tabela 4: Decomposição de Serviços para o Estudo de Caso Proposto

Nome do Microserviço	Responsabilidades Principais	Tecnologias Backend Sugeridas	Banco de Dados Sugerido	Padrões de Design Aplicáveis	Requisitos Não Funcionais Chave
Usuários/Autenticação	Gerenciar perfis, autenticação, autorização	Spring Boot (Java) [1]	PostgreSQL [1]	API Gateway, Database per Service	Alta disponibilidade (99.9%), Segurança (RBAC) [1]
Eventos	Criar, editar, listar eventos	FastAPI (Python) [1]	MongoDB [1]	Database per Service, Comunicação Síncrona	Escalabilidade horizontal, Flexibilidade de esquema
Inscrições	Gerenciar inscrições, vagas	Micronaut (Java) [1]	PostgreSQL [1]	Saga Pattern, Database per Service	Consistência de dados, Baixa latência
Pagamentos	Processar pagamentos, histórico	Go kit (GoLang) [1]	PostgreSQL/CockroachDB [1]	Circuit Breaker, Saga Pattern	Alta segurança, Confiabilidade transacional
Notificações	Enviar e-mails, SMS, push	Express.js (Node.js) [1]	Redis (cache/fila) [1]	Comunicação Assíncrona (Message Broker)	Alta *throughput*, Baixa latência de envio
Feedback/Avaliação	Coletar e gerenciar feedback	Flask (Python) [1]	MongoDB [1]	Database per Service	Escalabilidade, Flexibilidade

					de esquema
API Gateway	Roteamento, autenticação, LB	Spring Cloud Gateway/Nginx [1]	-	API Gateway Pattern [1]	Alta disponibilidade, Baixa latência
Frontend Web	Interface de usuário (Web)	React/Vue.js [1]	-	Micro Frontends (opcional), BFF	Usabilidade, Performance de carregamento

Segue o roteiro de artefatos e entregas ao longo da disciplina:

5.1. Etapa 1: Definição do Escopo e Requisitos (Semanas 1-2)

- **Objetivo:** Escolher o tema da aplicação corporativa (ex: Plataforma de E-commerce, RH, Gestão de Eventos) [1] e levantar requisitos de alto nível.
- **Conteúdo a Trabalhar:**
  - **Introdução aos Sistemas Corporativos:** Discussão sobre monólitos vs. microserviços, e a relevância de cada abordagem.
  - **Levantamento de Requisitos:** Técnicas para identificar requisitos funcionais e não funcionais.
  - **Aplicação do Método CERTO (C, E, R, T, O):** Os alunos aprenderão a estruturar suas ideias e necessidades.
    - **C - Contexto:** Definir a empresa, o problema a ser resolvido e o público-alvo do projeto.
    - **E - Exigências:** Listar as restrições do projeto (prazo, equipe, tecnologias open-source, requisitos de performance e disponibilidade).[1]
    - **R - Referências:** Analisar lições aprendidas de projetos anteriores (monólitos) e buscar inspiração em arquiteturas de sucesso (Netflix, Amazon).[1]
    - **T - Tarefas:** Esboçar as funcionalidades principais desejadas.
    - **O - Observações:** Definir preferências tecnológicas iniciais e critérios de qualidade.
- **Artefatos Esperados:** Documento de Visão e Escopo (incluindo requisitos funcionais e não-funcionais preliminares), possíveis diagramas de caso de uso, e

divisão preliminar de domínios para microserviços.[1]
- **Atividades Didáticas:** Brainstorming em grupo, sessões de Q&A, e exercícios práticos de aplicação do CERTO para formular requisitos.

5.2. Etapa 2: Modelagem de Domínio e Design da Arquitetura (Semanas 3-4)

- **Objetivo:** Elaborar diagramas e modelos detalhando a solução, com foco na arquitetura de microserviços.
- **Conteúdo a Trabalhar:**
  - **Princípios de Decomposição:** Como identificar contextos delimitados (Bounded Contexts) e modelar serviços em torno de capacidades de negócio.[1]
  - **Padrões de Comunicação:** Discussão aprofundada sobre comunicação síncrona (REST/gRPC) e assíncrona (Message Brokers - RabbitMQ/Kafka).[1]
  - **Padrões de Dados:** Conceito de "Database per Service" e persistência poliglota.[1] Introdução ao Saga Pattern para consistência distribuída.[1]
  - **Padrões de Resiliência:** Circuit Breaker, Bulkhead, Retry, Timeouts.[1]
  - **Aplicação do Método CERTO (C, E, R, T, O):** Os alunos usarão o CERTO para refinar o design.
    - **C - Contexto:** Descrever o design arquitetural atual e os serviços identificados.
    - **E - Exigências:** Listar requisitos não-funcionais específicos para cada serviço (escalabilidade, disponibilidade, performance).
    - **R - Referências:** Citar arquiteturas de referência do mercado e padrões de design estudados.
    - **T - Tarefas:** Solicitar validação da arquitetura, identificar pontos fracos, sugerir melhorias.
    - **O - Observações:** Focar em aspectos como segurança de dados, facilidade de manutenção e observabilidade.
- **Artefatos Esperados:** Diagrama de Domínio/Classes, Diagrama de Arquitetura de Microserviços (com API Gateway, serviços de domínio, bancos de dados descentralizados, message broker) [1], Diagramas UML adicionais (ex: Diagrama de Sequência para fluxos críticos), e um documento textual descrevendo decisões arquiteturais.[1]
- **Atividades Didáticas:** Workshops de design de domínio, exercícios de aplicação de padrões, configuração de Dockerfile e docker-compose básico para "Hello

World" de cada serviço.[1]

5.3. Etapa 3: Protótipo de Interface e Contratos de API (Semanas 5-6)

- **Objetivo:** Desenvolver protótipos de interface e definir os contratos de comunicação entre os serviços.
- **Conteúdo a Trabalhar:**
  - **Design de UI/UX:** Princípios de design de interfaces para aplicações corporativas.
  - **Padrões de Frontend:** React/Next.js para web, Flutter para mobile. Abordagens como BFF e Micro Frontends.[1]
  - **Especificação de APIs:** Uso de OpenAPI/Swagger para documentar endpoints REST (métodos, URIs, formatos JSON de request/response).[1]
  - **Aplicação do Método CERTO (C, E, R, T, O):** O CERTO será usado para especificar detalhes de UI e API.
    - **C - Contexto:** Descrever a funcionalidade da interface ou do endpoint API.
    - **E - Exigências:** Definir requisitos de usabilidade, performance da UI, e formatos de dados esperados na API.
    - **R - Referências:** Utilizar exemplos de interfaces de usuário bem-sucedidas e padrões de API REST.
    - **T - Tarefas:** Criar protótipos navegáveis da UI, especificar endpoints e modelos de dados.
    - **O - Observações:** Preferências de estilo, considerações de acessibilidade, e detalhes sobre validação de dados.
- **Artefatos Esperados:** Wireframes ou protótipo navegável da UI, especificações detalhadas das APIs (documentação OpenAPI/Swagger).[1] Diagrama de Sequência focado em integração.
- **Atividades Didáticas:** Sessões de prototipagem, revisão de design de APIs, e exercícios de escrita de especificações OpenAPI.

5.4. Etapa 4: Implementação Iteração 1 (MVP dos Microserviços) (Semanas 7-9)

- **Objetivo:** Entregar uma primeira versão funcional mínima do sistema (MVP),

integrando as partes essenciais.

- **Conteúdo a Trabalhar:**
  - **Desenvolvimento Backend:** Implementação dos microserviços de Usuários, Eventos e Inscrições (ex: Spring Boot/Quarkus, FastAPI, NestJS).[1]
  - **Integração de Banco de Dados:** Configuração do PostgreSQL para cada serviço, uso de ORMs (TypeORM, Prisma).[1]
  - **Autenticação e Autorização:** Implementação de JWT (JSON Web Tokens) para autenticação *stateless*.[1]
  - **Comunicação Assíncrona:** Exemplo básico de uso de RabbitMQ/Kafka para eventos (ex: "pedido realizado").[1]
  - **Aplicação do Método CERTO (C, E, R, T, O):** O CERTO será uma ferramenta para resolução de problemas de código e integração.
    - **C - Contexto:** Descrever um problema de código ou integração.
    - **E - Exigências:** Definir requisitos de correção (ex: resolver sem quebrar outras funcionalidades).
    - **R - Referências:** Fornecer trechos de código, logs de erro, *stacktraces*.
    - **T - Tarefas:** Identificar a causa raiz do problema, sugerir correções/refatoração.
    - **O - Observações:** Detalhes do ambiente de execução, tentativas de solução anteriores.
- **Artefatos Esperados:** Código-fonte nos repositórios, docker-compose funcional para o ambiente local, demonstração do MVP em aula.[1] Relatório de testes implementados.
- **Atividades Didáticas:** *Pair programming*, sessões de depuração assistida, e feedback contínuo do professor.

5.5. Etapa 5: Iteração 2 (Funcionalidades Avançadas e Hardening) (Semanas 10-12)

- **Objetivo:** Expandir funcionalidades para cobrir todo o escopo planejado e melhorar a robustez do sistema.
- **Conteúdo a Trabalhar:**
  - **Implementação de Módulos Restantes:** Serviço de Pagamentos (integração com *gateways*, Circuit Breaker) [1], Serviço de Notificações (Kafka/RabbitMQ para alta
    throughput).[1]
  - **Otimização e Caching:** Uso de Redis para cache e sessões.[1]

    - **Tratamento de Erros e Validações:** Retorno de códigos de erro apropriados, validações mais completas.
    - **Segurança Aprimorada:** Sanitização de *inputs*, criptografia de dados, mTLS para comunicação inter-serviços.[1]
    - **Observabilidade:** Implementação de logging centralizado, rastreamento distribuído e métricas (ELK Stack, Prometheus/Grafana).[1]
- **Artefatos Esperados:** Nova versão do sistema rodando em ambiente de *staging*, conjunto mais amplo de testes automatizados.[1] Relatório de progresso e mudanças arquiteturais.
- **Atividades Didáticas:** Simulações de falha, exercícios de otimização de performance, e sessões de revisão de segurança.

5.6. Etapa 6: Testes Finais e Qualidade (Semana 13)

- **Objetivo:** Realizar testes integrados completos do sistema e garantir a qualidade final.
- **Conteúdo a Trabalhar:**
    - **Testes Automatizados:** Testes de unidade, integração, *end-to-end* (E2E), testes de contrato de APIs.[1]
    - **Testes de Carga/Performance:** Uso de ferramentas como JMeter ou k6.[1]
    - **Monitoramento e Saúde:** Verificação de logs, métricas e comportamento do sistema sob estresse.[1]
    - **Controle de Qualidade:** *Code review*, análise estática de código (SonarQube).[1]
- **Artefatos Esperados:** Relatório de Testes (cobertura de testes, resultados, ajustes finais).[1] Dados fictícios para demonstração.
- **Atividades Didáticas:** Sessões de teste em grupo, simulações de cenários de uso real, e discussões sobre métricas de qualidade.

5.7. Etapa 7: Documentação Final e Deploy (Semanas 14-15)

- **Objetivo:** Consolidar toda a documentação do sistema e preparar o *deploy* de produção.
- **Conteúdo a Trabalhar:**

    - **Documentação Contínua:** Manual do usuário, manual de desenvolvedor, documentação de arquitetura atualizada (incluindo ADRs).[1]
    - **Organização de Repositórios:** Estrutura clara (*monorepo* vs. *polyrepo*), uso de GitHub *organization*.[1]
    - **Gerenciamento de Configuração:** Uso de variáveis de ambiente.[1]
    - **Estratégias de Deploy:** Revisão de Blue-Green, Canary Release, Rolling Updates.[1]
- **Artefatos Esperados:** Pacote final do projeto (código-fonte, docker-compose.yml para *deploy*), documentação completa (arquitetura, APIs, uso), versão *release* no Git (*tag*).[1]
- **Atividades Didáticas:** Workshops de documentação, exercícios de empacotamento para *deploy*, e preparação para a apresentação final.

5.8. Etapa 8: Apresentação Final e Retrospectiva (Semana 16)

- **Objetivo:** Apresentar o sistema funcionando e refletir sobre o processo de desenvolvimento.
- **Conteúdo a Trabalhar:**
    - **Comunicação de Projeto:** Estrutura de apresentação de um projeto técnico.
    - **Lições Aprendidas:** Análise crítica do que funcionou e o que poderia ser melhorado no processo.
- **Artefatos Esperados:** Apresentação e slides, vídeo demonstrativo.[1]
- **Atividades Didáticas:** Demonstração ao vivo do sistema, discussão sobre desafios e soluções, e sessão de retrospectiva do projeto.

6. Boas Práticas de Engenharia de Software e DevOps na Disciplina

Além das tecnologias, a disciplina deve enfatizar boas práticas de engenharia de software que são esperadas em projetos corporativos profissionais.[1] A adoção dessas práticas é fundamental para que os alunos desenvolvam competências essenciais de engenharia de software, especialmente no contexto de microserviços. Projetos de microserviços exigem rigor na organização devido à complexidade distribuída, por

isso práticas de CI/CD, versionamento e testes são cruciais para o sucesso.[1]

- **Controle de Versão e Workflow Git:** Todo o código deve residir em repositórios Git (por exemplo, no GitHub ou GitLab da turma). Recomenda-se definir um fluxo de trabalho claro – como Git Flow (ramificações *develop, main, feature branches*, etc.) ou o modelo *trunk-based development* com *feature flags*. Cada nova funcionalidade ou correção deve ser desenvolvida em *branch* separado e integrada via *pull requests*, permitindo *code review* pelos colegas/professor antes de mesclar.[1]
- **Integração Contínua (CI):** Configurar um *pipeline* de integração contínua que seja acionado a cada *push* no repositório. Ferramentas como GitHub Actions, GitLab CI ou Jenkins podem rodar tarefas automatizadas: *build*/compilação do serviço, execução de testes automatizados e *build* de imagem Docker. A CI garante que erros de compilação ou testes quebrados sejam detectados imediatamente.[1]
- **Entrega Contínua (CD):** Embora um *deploy* contínuo automatizado a cada mudança possa não ser factível em ambiente acadêmico, pode-se simular ou implementar entrega contínua ao final de *sprints*. Por exemplo, configurar o *pipeline* para, após passar nos testes, gerar imagens Docker e publicá-las em um *registry* local ou em *cloud* (Docker Hub, GitHub Container Registry), e até automatizar o *deploy* em um servidor de *staging*.[1]
- **Testes Automatizados:** Enfatizar a escrita de testes em múltiplos níveis: testes de unidade para funções/métodos de cada serviço, testes de integração para verificar comunicação entre componentes, e testes *end-to-end* (E2E) para fluxos completos no sistema. Praticar *Test-Driven Development* (TDD) pode ser incentivado em partes críticas.[1]
- **Documentação Contínua:** Documentar o projeto ao longo do desenvolvimento, não apenas no final. Isso inclui: Documentação de Arquitetura (com ADRs), Documentação de API (Swagger/OpenAPI), READMEs e Wiki do projeto.[1]
- **Gerenciamento de Configuração:** As configurações sensíveis (credenciais, *strings* de conexão) não devem ficar *hardcoded* no código ou no controle de versão. Usar variáveis de ambiente e arquivos de configuração separados por ambiente (*dev, test, prod*).[1]
- **Deployment e Organização de Ambientes:** Se possível, manter pelo menos três ambientes: Desenvolvimento (cada aluno roda local com Docker Compose), Staging (um servidor ou VM onde periodicamente a versão integrada do sistema é implantada para testes finais) e Produção (simulada para apresentação final).[1]
- **Organização dos Repositórios:** Definir uma estrutura clara. Se múltiplos repositórios (microserviços separados), usar um repositório GitHub *organization*

para agrupar, com nomenclatura consistente. Se *monorepo*, separar por pasta e talvez por *workspace*.[1]

- **Metodologia Ágil e Gestão de Projeto:** Adotar práticas ágeis no decorrer da disciplina: definir *sprints* ou iterações para entrega dos artefatos, realizar pequenas reuniões de acompanhamento (*daily stand-ups*), e manter um *backlog* de tarefas priorizado. Ferramentas como Trello, Jira ou GitHub Projects podem ajudar a distribuir e acompanhar tarefas.[1]
- **Controle de Qualidade e Revisões:** Além dos testes automatizados, implementar rotinas de *code review* (cada PR deve ser revisado por pelo menos um colega), e se possível fazer pares (*pair programming*) em partes críticas. Também planejar auditorias internas do código e usar ferramentas de análise estática de código (SonarQube).[1]
- **CI para Qualidade:** Integrar ao *pipeline* ferramentas de qualidade: rodar *linter* e formatador no CI, rodar análise estática e verificar cobertura de testes. Definir metas, como, por exemplo, >80% de cobertura de código testado.[1]
- **Documentação do Processo:** Incentivar que os alunos documentem não só o produto final, mas o processo de desenvolvimento. Isso pode ser parte da avaliação: um diário de bordo ou relatório de evolução, comentando desafios encontrados, como foram resolvidos, quais práticas funcionaram ou não.[1]

Ao adotar essas boas práticas, os alunos desenvolverão competências fundamentais de engenharia de software. Projetos de microserviços exigem rigor na organização devido à complexidade distribuída, por isso práticas de CI/CD, versionamento e testes são cruciais para o sucesso.[1] Esta abordagem demonstra que estas "boas práticas" não são meramente aprimoramentos opcionais, mas competências críticas que se tornam requisitos inegociáveis no contexto de microserviços distribuídos. A complexidade inerente de gerenciar numerosos serviços independentes exige um alto grau de rigor, automação e organização.

7. Conclusão e Recomendações Finais

7.1. Síntese dos Pontos Essenciais

A arquitetura de microserviços representa uma solução poderosa e flexível para o desenvolvimento de sistemas corporativos modernos, oferecendo escalabilidade, resiliência e agilidade sem precedentes. No entanto, a adoção dessa arquitetura exige um entendimento profundo de seus princípios fundamentais, padrões de design e desafios inerentes.[1] A decomposição de domínios de negócio, a escolha estratégica entre comunicação síncrona e assíncrona, a implementação de padrões de resiliência (como Circuit Breaker e Bulkhead), a abordagem de dados distribuídos (Database per Service e Saga Pattern), a priorização da observabilidade (logging centralizado,

distributed tracing) e a automação de CI/CD são pilares essenciais para o sucesso.[1] A seleção criteriosa de tecnologias modernas de backend (como Spring Boot, Quarkus, FastAPI), frontend (React, Vue.js), bancos de dados (PostgreSQL, MongoDB, Redis) e ferramentas de conteinerização e orquestração (Docker, Kubernetes) é crucial para construir uma aplicação robusta e eficiente.[1]

7.2. Orientações para a Implementação do Projeto Acadêmico

Para a implementação do projeto acadêmico de gestão de eventos corporativos, recomenda-se uma abordagem iterativa e incremental. Iniciar com um conjunto mínimo de funcionalidades (MVP) e expandir gradualmente, adicionando novos serviços e refinando os existentes. A ênfase na importância de testes automatizados e integração contínua desde o início é fundamental para garantir a qualidade e a estabilidade do sistema à medida que ele cresce. O uso de ferramentas de versionamento de código, como Git, e plataformas de colaboração, é indispensável para o trabalho em equipe. Encoraja-se a exploração e experimentação com diferentes tecnologias e padrões dentro do escopo definido, permitindo uma compreensão prática das compensações envolvidas em cada escolha arquitetural.[1]

7.3. Perspectivas Futuras e Tendências em Arquiteturas Corporativas

O futuro do desenvolvimento de sistemas corporativos com microserviços está cada vez mais ligado à evolução sinérgica de paradigmas nativos da nuvem, integração de IA/ML e estratégias sofisticadas de gerenciamento de dados, caminhando para sistemas altamente autônomos, inteligentes e resilientes.[1] As tendências apontam

para a evolução contínua de frameworks

cloud-native e *serverless*, que otimizam o desempenho e o consumo de recursos em ambientes conteinerizados. Avanços em *service meshes* e plataformas de orquestração continuarão a simplificar a gestão da comunicação e da infraestrutura.[1] A crescente integração de IA/ML em operações (AIOps) e desenvolvimento promete automatizar aspectos do gerenciamento de serviços, reduzindo a carga operacional.[1] A adoção de arquiteturas baseadas em eventos e dados, como o

data mesh, que se baseia no princípio de "database per service", tratará os dados como produtos, permitindo maior autonomia e governança.[1] A segurança e a conformidade em ambientes distribuídos permanecerão como preocupações primordiais, exigindo abordagens de defesa em profundidade e ferramentas avançadas.[1]

Bancos de dados com recursos avançados de IA, como o Oracle Database 23ai, e o crescimento de soluções nativas da nuvem e *serverless* são indicativos de um futuro onde os microserviços não são apenas unidades implantáveis independentemente, mas também entidades autogerenciáveis e orientadas a dados.[1] Este cenário em constante evolução do desenvolvimento de sistemas corporativos, onde diferentes paradigmas convergem e se influenciam sinergicamente, significa que as habilidades aprendidas hoje exigirão atualização contínua. Consequentemente, uma conclusão fundamental para os alunos é a necessidade imperativa de aprendizado contínuo e adaptabilidade. A abordagem pedagógica deve incutir não apenas o conhecimento atual, mas também a mentalidade e as ferramentas para a autoeducação contínua, preparando-os para uma carreira de constante mudança e inovação.

Referências citadas

1. Thais Martan - Método CERTO.pdf