ChatGPT

Plano de Desenvolvimento da Disciplina Desenvolvimento de Sistemas Corporativos (6º Período)

1. Escopo da Aplicação (Projetos Corporativos Sugeridos)

Para aplicar arquitetura de microserviços e boas práticas modernas, sugere-se escolher um projeto **corporativo realista** que tenha múltiplos domínios de negócio. Abaixo estão algumas ideias de escopo de aplicação:

- **Plataforma de E-commerce Corporativo:** Uma loja online completa (venda de produtos) com módulos de catálogo de produtos, carrinho/pedidos, pagamentos, usuários e administração. Esse escopo permite divisão em microserviços (produtos, pedidos, pagamentos, etc.) e uso de eventos (ex.: atualização de estoque após vendas).
- **Sistema de Gestão de Recursos Humanos (RH):** Aplicação interna para gerenciar funcionários, folhas de pagamento, férias e recrutamento. Pode ser dividido em serviços como *Usuários/Autenticação*, *Folha de Pagamento*, *Gestão de Férias*, *Recrutamento*, cada um independente.
- **Portal Corporativo de Projetos e Tarefas:** Uma plataforma estilo **ERP** simplificado ou gestão de projetos, incluindo módulos de projetos, tarefas, times/colaboradores e relatórios. Em microserviços, poderia ter serviços separados para projetos, tarefas e usuários, comunicando-se via APIs.

Cada uma dessas ideias de projeto possibilita implementar uma arquitetura **orientada a microserviços** e adotar práticas de mercado. Por exemplo, no caso do e-commerce, poderíamos ter serviços independentes para catálogo, carrinho/pedido e pagamentos, cada qual focado em um contexto de negócio específico (seguindo a ideia de *single responsibility* e baixo acoplamento) [1]. Para fins deste plano, assumiremos a primeira opção (plataforma de e-commerce) como exemplo base para detalhar arquitetura e tecnologias, mas o planejamento geral serve a qualquer dos escopos sugeridos.

2. Arquitetura Proposta da Aplicação

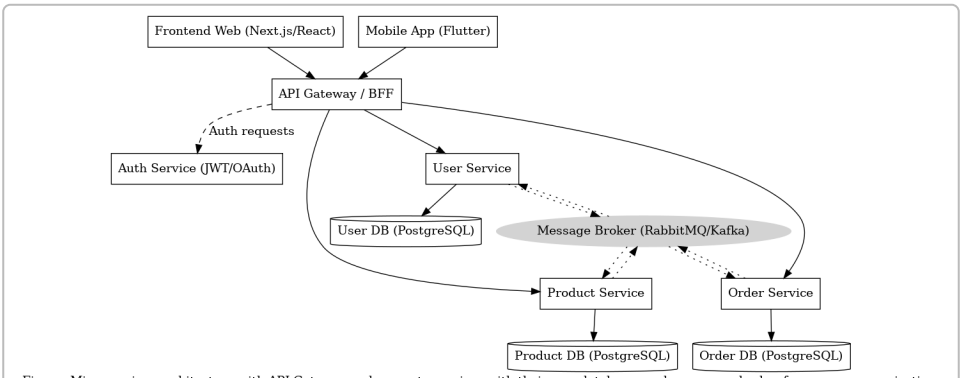

Figure: Microservices architecture with API Gateway and separate services with their own databases and a message broker for async communication.

Arquitetura geral proposta em microserviços (clientes no topo, gateway e serviços de domínio independentes, cada um com seu banco de dados, e comunicação assíncrona via mensageria).

A arquitetura seguirá o modelo de **microserviços**: a aplicação será composta de vários serviços pequenos e independentes, em vez de um único monolito. Cada microserviço corresponderá a um subsistema de negócio (por exemplo, *User Service*, *Product Catalog Service*, *Order Service*, etc.), comunicando-se uns com os outros por meio de APIs leves (geralmente REST/JSON) [2] ou mensageria. Os principais componentes propostos são:

- **Frontend Gateway/BFF:** Os clientes (aplicação web Next.js/React e aplicativo Flutter) não acessam diretamente os microserviços; as requisições passam primeiro por um **API Gateway** ou Backend-for-Frontend (BFF). Esse gateway centraliza as chamadas de frontend e as roteia para os serviços adequados, simplificando o consumo de múltiplos serviços e implementando preocupações transversais (ex.: autenticação, rate limiting).
- **Serviços de Domínio (Microserviços):** Cada serviço é **autônomo e de implantação independente** [1], cuidando de uma funcionalidade de negócio. Exemplos no e-commerce: um **Serviço de Usuários/Autenticação** (gerencia cadastro, login, perfis), **Serviço de Produtos** (CRUD de catálogo), **Serviço de Pedidos** (carrinho, ordem de compra) e possivelmente serviços de **Pagamentos** ou **Notificações**. Os serviços comunicam-se de forma síncrona via REST (ou gRPC) para consultas imediatas e de forma assíncrona via **eventos/mensageria** para integrações desacopladas (ex.: após a confirmação de um pedido, um evento de "`PedidoConfirmado`" é enviado para atualização de estoque ou envio de e-mail).
- **Bancos de Dados Descentralizados:** Cada microserviço terá seu próprio banco de dados privado, adequado aos dados que manipula (seguindo o princípio *Database per Service* para baixo acoplamento) [3]. Por exemplo, o serviço de produtos gerencia apenas o **Banco de Dados de Produtos**, o de pedidos gerencia o **Banco de Pedidos**, e assim por diante, todos preferencialmente usando **PostgreSQL** conforme sugerido (podendo usar esquemas separados). Essa separação garante independência e escalabilidade de cada módulo, evitando um único ponto de falha e facilitando a manutenção de cada esquema de dados.
- **Comunicação e Mensageria:** Para operações assíncronas e integração entre serviços, será adotado um **Message Broker** (como **RabbitMQ** ou **Apache Kafka**). Por exemplo, no e-commerce, quando um pedido é realizado no *Order Service*, ele publica um evento na fila ("pedido realizado"), consumido pelo *Product Service* para dar baixa no estoque e por um serviço de notificações para enviar confirmação ao cliente. Essa troca de eventos torna os serviços **fracamente acoplados** e resilientes a picos (o processamento pode ser distribuído e realizado em segundo plano).
- **Serviço de Autenticação (Auth)**: A autenticação pode ficar a cargo de um serviço dedicado (implementando JWT/OAuth2). O Auth Service emitirá tokens JWT para usuários autenticados e validará credenciais. O API Gateway consultará este serviço (ou um servidor OAuth externo, se preferir) para autenticar solicitações (seta tracejada na figura acima). Isso centraliza a segurança e permite SSO caso a aplicação cresça.
- **Infraestrutura de Suporte:** Outros componentes complementares podem fazer parte da arquitetura: um serviço de **cache em memória** (Redis) para armazenar dados frequentemente acessados (por exemplo, resultados de busca de produtos, sessões de usuário), melhorando performance; um sistema de *config server* ou variáveis de ambiente centralizadas para gerenciar configurações de cada microserviço; e ferramentas de observabilidade (logging centralizado, monitoramento) para acompanhar a saúde dos serviços em produção.

Descrições Adicionais: A figura acima ilustra a arquitetura proposta, com os clientes (web e mobile) consumindo a aplicação via um gateway único, que por sua vez se comunica com vários microserviços de domínio. Cada serviço tem seu próprio repositório de dados e todos compartilham um barramento de eventos (fila) para tarefas assíncronas e integração. Essa arquitetura traz benefícios de escalabilidade e isolamento de falhas: cada serviço pode ser escalado horizontalmente de forma independente conforme a demanda daquele domínio específico [4]. Vale notar que o design em microserviços

adiciona complexidade (por exemplo, necessidade de gerenciar comunicação inter-serviços e monitoração distribuída), mas segue padrões adotados em grandes empresas como Amazon, Netflix, etc., que migraram de monolitos para microserviços visando escalabilidade e agilidades de implantação [5] [6].

3. Tecnologias Recomendadas por Camada

Para implementar a arquitetura acima, são recomendadas tecnologias modernas em cada camada da aplicação, alinhadas com as competências dos alunos e tendências de mercado:

- **Frontend (Interface do Usuário):** A camada de front-end web pode ser desenvolvida em **Next.js** (React) – uma vez que Next.js suporta renderização server-side e geração de sites estáticos, útil para melhorar SEO e performance em aplicações corporativas. O React (com Next) permite componentização e uma boa experiência de desenvolvimento. Para estilo, pode-se usar CSS-in-JS ou frameworks como **Tailwind CSS**. No caso do projeto envolver também mobile, o uso de **Flutter** é adequado (os alunos já têm experiência), permitindo criar um app híbrido para Android/iOS consumindo as APIs do backend. A comunicação do front-end com o back será feita via chamadas HTTP aos endpoints do gateway/BFF ou diretamente a microserviços (no caso de Next.js, podendo usar **API Routes** internas como BFF). Pode-se considerar também o padrão **Micro frontends** se houvesse múltiplas equipes front-end, mas provavelmente não é necessário aqui.
- **Backend/Serviços:** Para implementar os microserviços, recomenda-se usar **Node.js** juntamente com o framework **NestJS**. O Node.js é indicado pelo seu modelo assíncrono e orientado a eventos, ótimo para lidar com muitas requisições concorrentes de forma não bloqueante [7]. Já o NestJS traz uma estrutura organizada, em TypeScript, seguindo padrões de **arquitetura modular**, injeção de dependências e decorators, facilitando a manutenção e testes [8]. Com NestJS, é fácil criar controladores HTTP para APIs REST, além de gateways GraphQL ou microserviços gRPC, se desejado. Alternativamente, os alunos com forte base em C# poderiam implementar alguns serviços em **ASP.NET Core Web API**, aproveitando a familiaridade com C# – uma abordagem poliglota é possível, mas para uniformidade e simplicidade de aprendizado, Node/Nest tende a ser mais didático neste contexto. Cada serviço exporá APIs RESTful (JSON) claras para seu domínio. Ferramentas adicionais no backend podem incluir **ORMs** para acesso a dados (por ex., TypeORM ou Prisma no Node, Entity Framework no .NET) e bibliotecas especializadas (ex.: para pagamentos, envio de e-mail, etc., conforme as features do projeto).
- **APIs e Comunicação:** O padrão principal de comunicação síncrona será **REST API** usando JSON sobre HTTP – um estilo simples e amplamente suportado. Os serviços podem também expor documentação dessas APIs usando **OpenAPI/Swagger**, o que facilita testes e integração entre times. Para comunicação assíncrona, como citado, usar-se-á **mensageria**: **RabbitMQ** (fila tradicional que implementa AMQP) ou **Apache Kafka** (stream de eventos) são boas escolhas. RabbitMQ pode ser mais simples de começar, oferecendo filas e tópicos para pub/sub; Kafka é mais robusto para altíssimo throughput de eventos. A escolha pode depender do escopo: por exemplo, para um e-commerce com eventos de pedido e estoque, RabbitMQ é suficiente. O *API Gateway* pode ser implementado usando um **Node.js Express** simples ou até mesmo através do próprio NestJS (um serviço dedicado atuando como gateway). Em cenários avançados, poderia-se usar um gateway pronto como **Kong**, **AWS API Gateway** ou **NGINX** ingress (se em K8s), mas dentro da disciplina é válido implementar um gateway customizado ou usar o próprio front-end Next.js como BFF roteando as chamadas para microserviços apropriados [9].
- **Banco de Dados:** O SGDB principal recomendado é **PostgreSQL**, por sua robustez, aderência a padrões SQL e extensões úteis (JSONB, etc.). Cada microserviço terá instância/tabelas próprias no Postgres. Além disso, conforme a necessidade, podem-se introduzir bancos NoSQL ou especializados: por exemplo, usar um **MongoDB** ou **Redis** para armazenar sessões de usuário

ou cache de produtos populares (Redis também pode servir como broker simples para pub/sub, ou cache de queries para aliviar o Postgres). Porém, manteremos o foco no PostgreSQL para a maior parte dos dados transacionais. Ferramentas de migração de schema (como Prisma Migrate ou Flyway) devem ser utilizadas para versionar a evolução do banco durante o projeto.

- **Mensageria e Streaming:** Como mencionado, **RabbitMQ** (fila de mensagens) ou **Kafka** (log de eventos) são recomendados para comunicação entre serviços de forma desacoplada. Por exemplo, RabbitMQ pode orquestrar eventos de criação de pedido, notificação de e-mail, atualização de estoque, de forma que cada serviço escute apenas as mensagens relevantes. Isso segue a tendência de arquiteturas orientadas a eventos, comum em sistemas corporativos atuais para garantir escalabilidade e tolerância a falhas.
- **Autenticação e Autorização:** Utilize **JWT (JSON Web Tokens)** para autenticação stateless entre front-end e back-end. O serviço de Auth gerará um JWT assinado para cada login, e o front-end incluirá esse token nos headers das requisições subsequentes. Os microserviços (ou o gateway) validarão o token (assinatura e claims) para autorizar cada chamada. Opcionalmente, pode-se integrar uma solução de SSO corporativa ou **OAuth2** (por exemplo, usando **Keycloak**, **Auth0** ou ASP.NET Identity), mas implementar um serviço de auth simples com JWT já cobre o necessário na disciplina. A autorização (permissões) pode ser controlada via claims nos tokens ou via um serviço de *ACL* central, dependendo do escopo.
- **Infraestrutura DevOps:** Recomenda-se uso de **Docker** para conteinerizar todos os serviços e componentes (bancos de dados, broker, etc.). Cada microserviço terá um `Dockerfile` e, durante o curso, pode-se utilizar **Docker Compose** para subir todo o ambiente localmente (útil para testes de integração entre serviços). A conteinerização garante que o ambiente de desenvolvimento e produção sejam consistentes, além de ser um conhecimento alinhado ao mercado [10]. Em fases avançadas, pode-se discutir **Kubernetes** para orquestração de contêineres, auto-scaling de instâncias de microserviço e deploy blue/green, mas se não houver tempo, manter no Docker Compose já exercita o essencial (K8s pode ser apenas conceitualmente apresentado ou usado caso a infraestrutura da instituição permita um cluster de teste).
- **Monitoramento e Logging:** Embora talvez fora do escopo de implementação prática completa, é bom mencionar ferramentas modernas: usar logs estruturados (JSON) em cada serviço, agregados via, por exemplo, **ELK Stack** (Elasticsearch, Logstash, Kibana) ou **Grafana Loki**; monitorar métricas com **Prometheus/Grafana**; e implementar *health checks* em cada serviço. Isso demonstra preocupação com **observabilidade**, importante em microserviços devido à complexidade distribuída.
- **Outras Tecnologias Auxiliares:** Para testes de carga, poderia-se usar **JMeter** ou **k6**; para documentação de APIs, **Swagger UI**; para cliente HTTP no front, a biblioteca **Axios**; e para o aplicativo Flutter, pacotes como **http** ou **Dio** para consumir as APIs. Se for implementado streaming de dados em tempo real (ex.: notificações em tempo real de pedidos), pode-se usar **WebSockets** via bibliotecas (Socket.io para Node, por exemplo). Essas adições ficam a critério do escopo final escolhido e do interesse dos alunos em explorar tecnologias extras.

Em resumo, a pilha tecnológica recomendada é **moderna e alinhada ao mercado**: front-end React/Next e Flutter, back-end Node/NestJS (ou ASP.NET Core) com APIs REST, banco PostgreSQL, mensageria RabbitMQ/Kafka, autenticação JWT, tudo orquestrado em contêineres Docker. Essa configuração permitirá aos alunos experimentar várias ferramentas e conceitos atuais de desenvolvimento corporativo.

4. Boas Práticas a Adotar na Disciplina

Para além da escolha de tecnologias, a disciplina deve enfatizar **boas práticas de engenharia de software** que são esperadas em projetos corporativos profissionais. Dentre as práticas a serem incorporadas durante o desenvolvimento, destacam-se:

- **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. Versionamento semântico pode ser adotado para releases dos serviços (ex.: v1.0.0, v1.1.0). Caso cada microserviço seja um repositório distinto, manter consistência de nomenclatura e organização em uma organização Git. Alternativamente, se optar por um **monorepo** (código de todos microserviços em um único repositório), estruturar pastas por serviço e usar ferramentas como Nx, Lerna ou turbo repo para gerenciar dependências, garantindo isolamento adequado. Independentemente do modelo, **commits frequentes e etiquetados** (tags de versão) são encorajados.
- **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. Por exemplo, cada microserviço Node pode rodar `npm run build` e `npm test` em pipelines. Essa prática reflete a necessidade de dominar CI/CD em projetos de microserviços modernos [11].
- **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. O objetivo é expor os alunos ao conceito de CD, onde cada incremento potencialmente geraria uma nova versão implantável. No mínimo, fazer deploy manual frequente (quinzenal, por exemplo) do sistema para teste integrado é recomendável, evitando deixar integração apenas para o fim.
- **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 (por exemplo, um teste que sobe o serviço de pedidos e de produtos em ambiente de teste e verifica se a chamada de baixar estoque funciona), e testes end-to-end (E2E) para fluxos completos no sistema (usando ferramentas como Postman/Newman, Cypress, etc.). Praticar **Test-Driven Development (TDD)** pode ser incentivado em partes críticas, aumentando a confiabilidade do código [12]. Cada microserviço deve ter um suite de testes rodando no CI. Também incluir testes de contrato de APIs (para garantir que mudanças em um serviço não quebrem consumidores) e, se possível, testes de carga/performance para avaliar escalabilidade.
- **Documentação Contínua:** Documentar o projeto ao longo do desenvolvimento, não apenas no final. Isso inclui:
- **Documentação de Arquitetura:** manter um documento (ou wiki) descrevendo decisões arquiteturais (por exemplo, usar NestJS, optar por RabbitMQ, modelagem de dados, etc.), possivelmente utilizando *ADR (Architecture Decision Records)* para registrar as razões de escolhas técnicas.
- **Documentação de API:** como mencionado, utilizar Swagger/OpenAPI para cada serviço expor automaticamente os endpoints e modelos de dados esperados. Assim, outros times (ou os próprios alunos de outra equipe) podem entender como consumir cada microserviço.

- **README e Wiki:** Cada repositório deve ter um README com instruções de setup, build e uso. Pode-se manter um wiki central do projeto com visão geral do sistema, diagrama de casos de uso, diagrama de arquitetura (como o apresentado acima) e divisão de responsabilidades das equipes. Ferramentas como **Mermaid** (para diagramas markdown) ou Draw.io podem ser usadas para criar diagramas claros.
- **Comentários e Padrões de Código:** Incentivar código claro, com comentários somente quando necessário (código bem escrito muitas vezes se autoexplica). Adotar um guia de estilo (ex.: padrão Airbnb Style Guide para JavaScript/TypeScript) e utilizar linters/formatters (ESLint, Prettier) para padronização.
- **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). Uma boa prática é ter um exemplo de `.env.example` no repo e, em produção, injetar variáveis de ambiente via pipeline. Também convém externalizar configs comuns (p. ex., URL do broker) para evitar retrabalho.
- **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 integradado sistema é implantada para testes finais) e Produção (simulada para apresentação final). Automatizar parte do deploy nesses ambientes via scripts ou CI (ex.: um job de CD que faz `docker-compose pull & up` no server de staging). Mesmo que a produção final seja apenas uma demonstração local, pensar nesses ambientes ajuda a estruturar o pipeline e praticar versionamento (por ex., somente releases aprovadas vão para "prod").
- **Organização dos Repositórios:** Conforme mencionado no controle de versão, definir uma estrutura clara. Se múltiplos repositórios (microserviços separados), usar um repositório GitHub *organization* para agrupar, com nomenclatura consistente (ex.: corp-app-gateway, corp-app-users, corp-app-orders, etc.). Se monorepo, separar por pasta e talvez por workspace (no Node, usar workspaces Yarn/NPM para dependências isoladas). Manter também repositórios para *infraestrutura* comum, se aplicável (ex.: arquivos de orquestração Docker Compose, charts de Kubernetes se houver, etc.). Uma boa organização reflete profissionalismo e facilita a manutenção.
- **Metodologia Ágil e Gestão de Projeto:** Adotar práticas ágeis no decorrer da disciplina: definir sprints ou iterações para entrega dos artefatos (ver seção 5), realizar pequenas reuniões de acompanhamento (daily stand-ups rápidos durante as aulas para remover impedimentos), e manter um **backlog de tarefas** priorizado. Ferramentas como Trello, Jira ou GitHub Projects podem ajudar a distribuir e acompanhar tarefas entre os alunos (por exemplo, colunas "A fazer / Em andamento / Concluído"). Isso dá aos alunos experiência em colaboração e gestão de projetos real, preparando-os para ambientes corporativos.
- **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 a cada marco importante: por exemplo, ao terminar a camada de persistência, revisar se estão seguindo padrões de segurança (SQL injection, validação de entrada) e boas práticas de POO (código limpo, princípios SOLID). Ferramentas de análise estática de código (SonarQube, etc.) podem ser usadas se houver facilidade, para detectar bugs comuns ou vulnerabilidades.
- **CI para Qualidade:** Integrar ao pipeline ferramentas de qualidade: rodar linter e formatador no CI, rodar análise estática (por exemplo, **SonarCloud** gratuito para open source) e até verificar cobertura de testes. Definir metas, como, p.ex., >80% de cobertura de código testado, apenas merges permitidos se passar em todos testes e linter. Isso força a equipe a manter a qualidade consistente.
- **Documentação do Processo:** Por fim, 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. Essa reflexão contínua cria aprendizado sobre processos (Scrum, DevOps) além da tecnologia em si.

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 sucesso [11]. Espera-se que ao final da disciplina eles dominem não apenas construir o sistema, mas também **entregar** um produto de qualidade de forma profissional.

5. Roteiro de Artefatos e Entregas ao Longo da Disciplina

Para garantir o progresso organizado do projeto durante o semestre, sugere-se um **cronograma de entregas (sprints)** com artefatos específicos. Abaixo um roteiro possível (podendo ser ajustado conforme a duração exata do curso e carga horária):

1. **Semana 1-2: Definição do Escopo e Requisitos** – Entrega: *Documento de Visão e Escopo*. Os alunos (ou equipes) devem escolher o tema da aplicação corporativa (e-commerce, RH, etc.) e levantar os requisitos de alto nível. Isso inclui descrição do problema a ser resolvido, principais funcionalidades desejadas, atores (tipos de usuários) e algumas **user stories** exemplares. Artefatos esperados: lista de requisitos funcionais e não-funcionais preliminares; possíveis diagramas de caso de uso; e divisão preliminar de domínios para microserviços. Nesta fase define-se também a **tecnologia a ser usada** em cada parte (front, back, BD...), já alinhando com as recomendações dadas. *(Dica: utilizar o método CERTO – ver seção 6 – para consultar o ChatGPT e obter ideias iniciais de escopo ou validação de requisitos, fornecendo o contexto do projeto e exigências do curso.)*
2. **Semana 3-4: Modelagem de Domínio e Design da Arquitetura** – Entrega: *Modelo Conceitual e Design Arquitetural*. Os alunos elaboram diagramas e modelos detalhando a solução. Espera-se aqui: um **Diagrama de Domínio/Classes** representando as entidades principais e relações; um **Diagrama de Arquitetura de Microserviços** (como o apresentado na seção 2) mostrando os serviços identificados e como se comunicam; e possivelmente diagramas UML adicionais (Diagrama de Sequência ilustrando um fluxo end-to-end de uso, por ex. "cliente faz pedido"). Além dos diagramas, um pequeno documento textual descrevendo as decisões de arquitetura – por que certos serviços foram definidos, quais tecnologias para cada camada, e como serão tratados requisitos não-funcionais (escalabilidade, segurança, etc.). Nesta entrega, definir também a **estrutura de repositórios** e criar os projetos base (ex.: criar o esqueleto de cada microserviço – mesmo vazio – no Git, configurar pipeline de CI inicial, etc.). *Boas práticas:* já configurar o Dockerfile de cada serviço e um docker-compose básico nesta fase, para garantir que a arquitetura proposta é viável de subir; rodar um "Hello World" de cada serviço.
3. **Semana 5-6: Protótipo de Interface e Contratos de API** – Entrega: *Protótipo Navegável da UI e Especificações de API*. Enquanto a equipe de back-end começa implementação básica, a equipe de front-end pode produzir protótipos das telas principais (ferramentas como Figma ou o próprio código Flutter/React com dados mock). Entregar wireframes ou protótipo navegável mostrando a experiência do usuário. Em paralelo, definir os **contratos de API** entre front-end e back-end: por exemplo, especificar os endpoints REST de cada microserviço (métodos, URIs, formatos JSON de request/response) – isso pode ser feito através de documentação OpenAPI ou mesmo uma tabela descritiva. Essa especificação serve como contrato para que front e back trabalhem em paralelo. Poderia também ser apresentado um **Diagrama de Sequência** focado em integração, ilustrando como uma requisição do cliente atravessa o gateway e aciona diferentes serviços.
4. **Semana 7-9: Implementação Iteração 1 (MVP dos Microserviços)** – Entrega: *Versão MVP Integrada (parcial)*. Nesse marco, espera-se ter uma primeira versão funcional mínima do

sistema, integrando todas as partes em um fluxo simples. Por exemplo, no e-commerce: permitir cadastro/login de usuário, listagem de produtos (cadastrados manualmente direto no banco ou via endpoint administrativo), e realização de um pedido simples de um produto. Isso exige que pelo menos 2-3 serviços estejam já codificados (usuários, produtos, pedidos) com suas operações principais, e que o front-end seja capaz de conversar com eles via o gateway. A integração com o banco de dados deve estar configurada (ex.: usar PostgreSQL local via Docker). É desejável já incluir um exemplo de comunicação assíncrona via mensageria se possível – por exemplo, ao criar um pedido, publicar um evento em uma fila (mesmo que nenhuma ação complexa seja tomada ainda). Também, configurar autenticação básica (ex.: proteger endpoints com JWT). A entrega consistirá do código-fonte nos repositórios, instruções para rodar (docker-compose funcionando), e uma pequena **demonstração em aula** do MVP. Os alunos devem relatar quais testes já implementaram e problemas encontrados. *Após essa entrega, espera-se feedback do professor para ajustes.*

5. **Semana 10-12: Iteração 2 (Funcionalidades Avançadas e Hardening)** – Entrega: *Sistema Completo em Staging*. Na segunda iteração de desenvolvimento, os alunos expandem as funcionalidades para cobrir todo o escopo planejado. Isso inclui tratar regras de negócio mais complexas em cada serviço, implementar quaisquer módulos restantes (ex.: serviço de pagamentos integrado a um sandbox, serviço de notificações por e-mail, etc.), e melhorar a robustez. É o momento de **otimização**: adicionar caching (ex.: implementar uso do Redis para dados de leitura frequente), implementar validações e tratamentos de erro mais completos (ex.: retornar códigos de erro apropriados nas APIs, fazer *circuit breaker* se algum serviço falhar – usando biblioteca como Istio se avançar para K8s, ou Node retry strategies). Nesta fase também deve-se aprimorar a **segurança** (sanitização de inputs, verificar autenticação em todas rotas, encriptar senhas no banco, etc.). Ao final da semana 12, o objetivo é ter **todas as features implementadas** conforme o escopo inicial. A entrega seria uma nova versão do sistema rodando no ambiente de *staging*, com um conjunto mais amplo de testes automatizados cobrindo os novos cenários. Fornecer um relatório breve do progresso, destacando o que foi adicionado e quaisquer mudanças de arquitetura necessárias (por ex., “descobrimos necessidade de separar serviço X em dois”, etc.).
6. **Semana 13: Testes Finais e Qualidade** – Entrega: *Relatório de Testes e Ajustes Finais*. Reservar um tempo para **testes integrados completos** do sistema. Os alunos devem executar casos de teste end-to-end cobrindo todos os requisitos (podendo utilizar testadores externos ou scripts automatizados). Monitorar performance básica (tempo de resposta das principais requisições, comportamento com cargas pequenas). Qualquer bug crítico encontrado deve ser corrigido. Verificar também logs e monitoração (simular falhas de um serviço, ver se os outros continuam funcionando – ex.: se o serviço de pagamentos cair, o pedido ainda é registrado e marcado como pendente). Os critérios de aceitação definidos lá no começo (semana 1-2) devem ser revisitados para garantir que todos foram atendidos. A entrega dessa fase é um **Relatório de Testes**, contendo: cobertura de testes automatizados (%), resultados dos testes manuais ou de usabilidade, e lista de **ajustes finais** realizados para estabilização. Opcionalmente, preparar dados fictícios para demonstrar o sistema (ex.: 10 produtos de exemplo, 5 usuários cadastrados).
7. **Semana 14-15: Documentação Final e Deploy** – Entrega: *Pacote Final do Projeto*. Nesta etapa, os alunos consolidam toda a documentação do sistema: manual do usuário (se aplicável), manual de desenvolvedor (como rodar, estrutura do código), e documentação de arquitetura atualizada (incluindo eventuais mudanças desde a fase de design inicial). Incluir todos os diagramas finais refinados. Além disso, preparar o **deploy de produção** do sistema para apresentação final – isso pode ser empacotar tudo em contêineres e fornecer um `docker-compose.yml` que sobe toda a aplicação, ou realizar deploy em um servidor/cloud da faculdade. O importante é que a configuração final esteja reproduzível. Também é interessante gerar uma versão *release* no Git (tag) representando a versão final 1.0 do produto.

8. **Semana 16: Apresentação Final e Retrospectiva** – Entrega: *Apresentação e Slides*. Na última semana, os grupos apresentam o sistema funcionando para a turma/banca. A apresentação deve cobrir: demonstração ao vivo das principais funcionalidades pela interface; explicação resumida da arquitetura (mostrando que cada parte funciona, possivelmente derrubando um serviço de propósito para mostrar resiliência, etc.); e discussão das lições aprendidas. Realizar também uma **retrospectiva** final: o que funcionou bem no planejamento, o que seria feito diferente? Esse feedback consolida o aprendizado. Os slides ou vídeo demonstrativo são entregues como artefato final complementar.

Este roteiro distribui as entregas de forma incremental, garantindo que já na metade do curso os alunos tenham uma versão mínima do sistema operando, e depois possam iterar adicionando complexidade. Assim eles praticam integração contínua e evitam o risco de deixar tudo para o final. Cada entrega serve como um marco para avaliação parcial e feedback rápido, em sintonia com metodologias ágeis.

6. Aplicação do Método CERTO no Projeto

Para potencializar o planejamento e a execução do projeto, propõe-se integrar o uso de **IA (ex: ChatGPT)** como ferramenta de apoio, utilizando o **método CERTO** para elaborar prompts eficazes. Desenvolvido por Thais Martan, o método CERTO sugere estruturar cada consulta à IA com cinco elementos – **Contexto, Exigências, Referências, Tarefas e Observações** [13] – de forma a fornecer máximo de clareza e informação ao modelo. Seguindo essa estrutura, garante-se que a ferramenta (IA) receba todos os dados necessários para gerar respostas mais assertivas e úteis [14].

Como aplicar na disciplina? Ao longo do projeto, os alunos (ou o professor) podem empregar o método CERTO em diversas situações, tais como: planejamento de funcionalidades, resolução de problemas de código, ou busca de sugestões de arquitetura. Por exemplo:

- **Planejamento do Escopo (Início do Projeto):** Durante a definição de requisitos, os estudantes podem consultar o ChatGPT para obter ideias adicionais ou validar o escopo. Usando CERTO: fornecer **Contexto** (descrição do projeto escolhido, público-alvo, objetivos), listar **Exigências** (ex.: "a solução deve usar microserviços, precisa ser feita em 4 meses por 5 alunos, deve usar React e NestJS"), incluir **Referências** (ex.: "já estudamos projetos X e Y semelhantes" ou citar requisitos já definidos), definir a **Tarefa** claramente ("gerar 5 ideias de funcionalidades inovadoras para nosso sistema de e-commerce que se beneficiem de microserviços") e adicionar **Observações** (ex.: preferências como "respostas em português", "considerar tecnologias open-source apenas"). Essa abordagem estruturada orienta a IA a dar sugestões focadas e viáveis, atuando quase como um *brainstorming* assistido.
- **Design da Arquitetura:** Caso os alunos queiram validação da arquitetura proposta, podem preparar um prompt descrevendo o desenho atual (Contexto), incluindo restrições (Exigências, p. ex. "precisa ser escalável, baixo custo"), Referências (talvez citar arquiteturas padrão do mercado, como a da Netflix ou um diagrama estudado em aula), Tarefa (pedir melhorias ou identificar possíveis pontos fracos na arquitetura) e Observações (por ex.: "focar em segurança de dados e facilidade de manutenção"). A resposta da IA, guiada por essas informações, pode apontar melhorias arquiteturais ou confirmar que estão no caminho certo.
- **Implementação e Codificação:** Ao escrever código, os alunos podem usar o ChatGPT para esclarecer dúvidas de sintaxe ou obter exemplos de implementação, sempre estruturando o prompt em CERTO. Exemplo: *Contexto:* "Estou desenvolvendo um microserviço em NestJS para a funcionalidade X"; *Exigências:* "Preciso usar PostgreSQL via TypeORM, seguindo repositório e serviço conforme arquitetura Nest, e manter performance"; *Referências:* "Segue trecho do código atual que não está funcionando..." (colando stacktrace ou função específica); *Tarefa:* "Identifique

o erro e sugira correções/refatoração"; *Observações:* "O erro ocorre ao salvar entidade Y; já tentei solução Z sem sucesso". Com isso, a IA consegue analisar com mais precisão o cenário específico, atuando como um pair programming virtual.

- **Geração de Testes e Documentação:** Os alunos também podem recorrer à IA para ajuda em escrever casos de teste ou documentar alguma parte do sistema. Por exemplo, pedindo "*Sugira casos de teste unitário para a classe X*" passando o contexto da funcionalidade e exigências (cobrir cenário feliz e de erro, etc.). Ou então "*Ajude a escrever a documentação do endpoint Y*" fornecendo o contexto do endpoint, referência de formatos de resposta esperados, etc. A IA, bem instruída, pode produzir rascunhos que os alunos refinam, agilizando tarefas repetitivas.
- **Resolução de Problemas e Depuração:** Quando enfrentarem um bug ou impedimento, os alunos podem explicar o **Contexto** (onde o bug ocorre, em qual serviço, qual a funcionalidade), dar **Exigências** (resolver sem quebrar outra coisa, por exemplo), mostrar **Referências** (log de erro, código relevante), definir a **Tarefa** (encontrar causa raiz e sugerir solução) e **Observações** (já tentaram X e Y, ambiente de execução, etc.). Muitas vezes, ao articular o problema nesse formato, a solução já fica mais clara, e a IA pode identificar o ponto cego rapidamente.

O importante é que o uso do método CERTO seja incentivado como uma **ferramenta de apoio**, não para fazer o trabalho dos alunos, mas para **ampliar a capacidade de pesquisa e resolução** deles. O professor pode inclusive requerer que, em relatórios, os alunos citem se usaram a IA para determinada tarefa e como foi a experiência, fomentando discussão crítica sobre os resultados. Isso torna a disciplina inovadora ao integrar IA no workflow de desenvolvimento de software.

Em suma, o método CERTO (Contexto, Exigências, Referências, Tarefas, Observações) [13] fornece um guia para criar prompts claros e eficazes, garantindo que a IA entenda exatamente o que se precisa. Conforme descrito por Martan, seguir essa estrutura mnemônica aumenta a assertividade das respostas do ChatGPT [15] [14]. Aplicado ao projeto, ele pode auxiliar no planejamento e na execução, desde a concepção do sistema até a depuração final, sempre de forma ética e construtiva. Essa integração do "método CERTO" no curso também prepara os alunos para usar ferramentas de IA de maneira profissional e responsável, uma competência cada vez mais valorizada no mercado de desenvolvimento de sistemas.

Referências Utilizadas: As recomendações e melhores práticas aqui descritas foram baseadas em literaturas e casos modernos de desenvolvimento. A importância de microserviços independentes e de baixo acoplamento é destacada por Maguire [1]. Exemplos de projetos reais (e-commerce, RH) e seus benefícios pedagógicos foram inspirados em compilações de projetos de microserviços [3] [4]. As tecnologias sugeridas refletem tendências atuais – por exemplo, Node.js/NestJS pela sua eficiência em I/O concorrente [7] [8] – e alinhamento com conhecimentos prévios dos alunos (C#, Flutter). Boas práticas de DevOps e QA estão em consonância com diretrizes encontradas em fontes como MindMajix [11], enfatizando CI/CD e testes no contexto de microserviços. Por fim, a estratégia de prompts com método CERTO foi embasada no guia de Thais Martan [13] [14], mostrando-se adequada para melhorar interação com IA no domínio de projetos de software. Com esse plano estruturado, espera-se que a disciplina forneça uma experiência abrangente de **desenvolver um sistema corporativo moderno**, unindo teoria e prática de forma integrada e preparando os discentes para desafios do mundo real.

---

[1] [2] [5] [6] Microservices Architecture Diagram Examples - DevTeam.Space
https://www.devteam.space/blog/microservice-architecture-examples-and-diagram/

[3] [4] [10] [11] [12] ▷ Microservices Projects Ideas for Beginners
https://mindmajix.com/microservices-projects

7 8 Building Microservices using NodeJS + React [Detailed Guide]
https://code-b.dev/blog/microservices-with-nodejs-and-react

9 Building Scalable Microservice Architecture in Next.js - DEV Community
https://dev.to/hamzakhan/building-scalable-microservice-architecture-in-nextjs-1p21

13 14 15 Thais Martan - Método CERTO.pdf
file://file-3yHPtw7sCXFii4BvLtMn8h