Developer Evaluation

Uma solução estratégica e moderna para MVP de e-commerce.

Sumário Executivo

Esta seção oferece uma visão concisa e de alto nível do projeto, destacando a arquitetura, o valor de negócio e a estratégia de tecnologia, projetada para uma compreensão rápida e impactante do projeto.

Propósito e Visão Geral

O sistema é uma API de e-commerce construída com microsserviços para gerenciar vendas, produtos e usuários. A arquitetura foi projetada para ser escalável e resiliente.

Valor de Negócio e Impacto Estratégico

A solução permite o controle completo de vendas, desde a criação até o cancelamento, aplicando regras de negócio e descontos. A arquitetura modular e desacoplada facilita futuras integrações e o desenvolvimento de novas funcionalidades de forma independente.

Escopo de Alto Nível

O escopo inicial (MVP) abrange as operações CRUD (Criar, Ler, Atualizar, Deletar) para usuários, produtos e vendas. A API de vendas implementa regras de desconto específicas com base na quantidade de itens.

Principais Tecnologias e Abordagem Arquitetural

Arquitetura: Microsserviços, DDD (Domain-Driven Design), CQRS (Command Query Responsibility Segregation) e Event Sourcing .

Tecnologias: C#/.NET 8+, PostgreSQL (para persistência de dados transacionais), RabbitMQ (como barramento de eventos), e Redis (para cache e gerenciamento de estado).

Boas Práticas: O projeto segue rigorosamente os princípios SOLID, Clean Architecture e TDD (Test-Driven Development) .

Observabilidade: A solução prioriza a implementação de OpenTelemetry (logs, métricas e traces) para ser compatível com ferramentas como Datadog, Elastic e Dynatrace.

Cloud & Infraestrutura: O sistema é conteinerizado (Docker/Kubernetes) e preparado para ambientes multi-cloud, com automação via Terraform (IaC).

Próximos Passos

As próximas fases do projeto envolve o detalhamento da arquitetura de software (Nível Contêineres e Componentes do C4 Model), o design aprofundado dos microsserviços, a validação contínua dos requisitos com as áreas de negócio e o planejamento detalhado das atividades de desenvolvimento e implantação.

Introdução ao Projeto

Esta seção descreve o cenário atual, os desafios enfrentados e a visão do produto para o sistema de E-commerce, definindo os objetivos a serem alcançados.

Cenário Atual e Desafios

O cenário atual demanda uma plataforma de e-commerce robusta e moderna, capaz de lidar com um alto volume de transações de maneira eficiente e segura.

Os desafios incluem a necessidade de um sistema altamente disponível, escalável e fácil de manter, que possa ser rapidamente adaptado a novas regras de negócio e integrações. A arquitetura monolítica tradicional é inadequada para esses requisitos.

A complexidade do negócio de vendas, com regras específicas de descontos e a necessidade de rastrear eventos de domínio, exige uma abordagem de design que priorize a clareza e a consistência do modelo de negócio.

Visão do Produto

A visão do produto é ser uma plataforma de e-commerce de alto desempenho, escalável e resiliente, que fornece APIs para gerenciar o catálogo de produtos, usuários e o ciclo de vida das vendas.

A plataforma deve ser a base para futuras expansões, como a integração com sistemas de estoque, logística e faturamento, utilizando um modelo de comunicação assíncrona baseado em eventos.

Objetivos do Projeto (SMART)

Específico (Specific): Criar um sistema de API para vendas que gerencie usuários, produtos e transações, com capacidade de cálculo de descontos.

Mensurável (Measurable): Alcançar uma latência de processamento de transações abaixo de 100ms, com uma taxa de sucesso de 99,9%. A cobertura de testes unitários e de integração deve ser superior a 80%.

Alcançável (Achievable): Utilizar uma arquitetura de microsserviços com DDD, CQRS e Event Sourcing, baseada em tecnologias como C#/.NET 8+, PostgreSQL e RabbitMQ, para atender aos requisitos de desempenho e escalabilidade.

Relevante (Relevant): A implementação da plataforma é crucial para a digitalização dos processos de vendas e para a futura expansão do negócio de e-commerce.

Objetivos Funcionais

Os objetivos funcionais do sistema são divididos por domínio de negócio, refletindo a arquitetura de microsserviços.

Serviço de Produtos (`Products-API`)

Este microsserviço é responsável por gerenciar o catálogo de produtos e as informações de estoque.

- **Gestão de Produtos:**
  - **Cadastro (`POST /products`):** Permitir a criação de um novo produto com informações como `nome`, `descrição`, `preço` e `quantidade` em estoque. A validação assegura que todos os campos obrigatórios estejam preenchidos corretamente.
  - **Consulta por ID (`GET /products/{id}`):** Recuperar os detalhes de um produto específico.
  - **Listagem (`GET /products`):** Permitir a consulta paginada e filtrada de todos os produtos do catálogo.
  - **Atualização (`PUT /products/{id}`):** Modificar as informações de um produto existente.
  - **Exclusão (`DELETE /products/{id}`):** Remover um produto do catálogo.
- **Controle de Estoque:**
  - A API deve notificar, via eventos de domínio, quando o estoque de um produto atingir um nível baixo (`ProductLowStockEvent`). A lógica de negócio para essa notificação é implementada no domínio (`Product`). A implementação atual do publicador de eventos é um console log, com a intenção de ser substituída por uma solução de mensageria (RabbitMQ).

Serviço de Usuários (`Users-API`)

Este microsserviço é responsável por gerenciar as informações de usuários e a autenticação.

- **Gestão de Usuários:**
  - **Registro (`POST /users`):** Criar um novo usuário. A lógica de domínio e validação garantem que e-mail, senha e outros dados sejam únicos e válidos.
  - **Consulta por ID (`GET /users/{id}`):** Recuperar o perfil de um usuário específico. A autenticação com token é necessária para esta operação.
  - **Exclusão (`DELETE /users/{id}`):** Excluir um usuário.
- **Autenticação:**
  - **Login (`POST /auth`):** Autenticar um usuário fornecendo um token JWT (JSON Web Token) válido para autorização em outras APIs.

Serviço de Vendas (`Sales-API`)

Este microsserviço gerencia o ciclo de vida das vendas, incluindo criação, listagem e cancelamento.

- **Gestão de Vendas:**
  - **Criação (`POST /sales`):** Registrar uma nova venda, que pode conter vários itens. O serviço deve aplicar a lógica de negócio de descontos na camada de domínio.
  - **Listagem (`GET /sales`):** Consultar a lista de vendas do usuário autenticado.
  - **Listagem para Gerentes (`GET /sales/manager`):** Consultar todas as vendas do sistema, uma funcionalidade restrita a usuários com o papel `Manager`.
  - **Cancelamento (`DELETE /sales/{id}`):** Permitir o cancelamento de uma venda existente. A regra de negócio garante que apenas o usuário que criou a venda ou um gerente possa cancelar.
- **Regras de Negócio e Descontos:**
  - A lógica de desconto é implementada usando o padrão `Specification`.
  - **Desconto de 10%:** Aplicado a vendas com 4 a 9 itens de um mesmo produto.
  - **Desconto de 20%:** Aplicado a vendas com 10 a 20 itens de um mesmo produto.
  - Não é permitido adicionar mais de 20 itens idênticos a uma única venda.

Objetivos Não-Funcionais

Performance

O sistema deve ser capaz de lidar com a carga esperada de requisições de forma eficiente, garantindo uma experiência de usuário fluida.

- **Latência da API:** As operações de escrita (e.g., `POST /sales`) devem ter uma latência média de resposta inferior a 200ms. As operações de leitura (e.g., `GET /products`) devem ter uma latência média de resposta inferior a 100ms.
- **Tempo de Processamento de Eventos:** Eventos de domínio publicados (e.g., `SaleCreatedEvent`) devem ser processados por consumidores em no máximo 1 segundo após a sua publicação.
- **Escalabilidade Horizontal:** A arquitetura em microsserviços, com a separação dos domínios (`Products-API`, `Sales-API`, `Users-API`), permite que cada serviço seja escalado de forma independente para lidar com picos de tráfego específicos. O uso de Docker e Kubernetes (mencionado na documentação de infraestrutura) é fundamental para essa escalabilidade.

Resiliência e Disponibilidade

O sistema deve ser capaz de resistir a falhas e manter a disponibilidade dos serviços essenciais.

- **Tolerância a Falhas:** A utilização de um message broker (RabbitMQ) desacopla os serviços, de modo que a indisponibilidade temporária de um consumidor não impede a criação de uma venda.
- **Health Checks:** Todos os microsserviços devem expor endpoints de `health check` para que o orquestrador (Kubernetes) possa monitorar e, se necessário, reiniciar instâncias com falha, garantindo a alta disponibilidade.
- **Autenticação e Autorização:** O mecanismo de autenticação baseado em JWT é stateless (não mantém estado no servidor), o que aumenta a resiliência e a escalabilidade, pois qualquer instância de serviço pode validar o token sem depender de uma sessão centralizada.

Segurança

A segurança é um pilar crítico, garantindo a proteção dos dados e o acesso controlado ao sistema.

- **Autenticação:** Todos os endpoints protegidos devem exigir um token JWT válido, emitido pelo `Users-API`.
- **Autorização:** A autorização deve ser implementada com base em papéis (`UserRole`), restringindo o acesso a funcionalidades sensíveis (e.g., `GET /sales/manager` é acessível apenas para usuários com o papel `Manager`).
- **Gerenciamento de Senhas:** As senhas dos usuários devem ser armazenadas de forma segura usando hashing criptográfico (`BCryptPasswordHasher`), protegendo contra vazamentos de dados.

Manutenibilidade e Qualidade de Código

O código deve ser fácil de entender, modificar e estender, aderindo a padrões de alta qualidade.

- **Clean Architecture & DDD:** A arquitetura em camadas (`Domain`, `Application`, `Infrastructure`, `WebApi`) isola a lógica de negócio das preocupações técnicas, facilitando a manutenção e a adaptação do código a mudanças.
- **Padrões de Projeto:** O uso de padrões como CQRS e `Specification` na camada de domínio torna o código mais expressivo e focado no problema de negócio, por exemplo, a validação de regras de desconto na API de vendas.
- **Testes Automatizados:** O projeto segue uma abordagem de TDD, com uma suíte abrangente de testes unitários e de integração, garantindo que as mudanças não introduzam regressões. A meta de cobertura de código é de 80%.

Observabilidade

O sistema deve ser facilmente monitorado para diagnosticar problemas e entender o comportamento em produção.

- **Logs Estruturados:** O sistema utiliza `Serilog` para logs estruturados, facilitando a análise e o diagnóstico de problemas.

- **Métricas:** O sistema deve expor métricas (e.g., tempo de resposta, quantidade de requisições, erros) para dashboards de monitoramento.
- **Traces:** A futura implementação de traces, mencionada na documentação de logging, permitirá o rastreamento de requisições através dos microsserviços, facilitando a identificação de gargalos de desempenho em um ambiente distribuído.

Escopo Detalhado (MVP)

O escopo do MVP está focado na implementação das funcionalidades essenciais para o core do negócio: usuários, produtos e a gestão de vendas.

Serviço de Usuários (`Users-API`)

Este microsserviço gerencia as informações dos usuários, incluindo registro, autenticação e perfil.

- **Endpoints da API:**
  - **`POST /api/v1/auth/login`**: Endpoint para autenticação de um usuário existente.
    - **Responsabilidade:** Validar as credenciais do usuário e, em caso de sucesso, gerar um token JWT para ser usado em requisições futuras.
    - **Implementação:** Utiliza o `BCryptPasswordHasher` para a verificação de senhas de forma segura e o `JwtTokenGenerator` para criar o token.
  - **`POST /api/v1/users`**: Endpoint para o registro de novos usuários.
    - **Responsabilidade:** Validar os dados de entrada, criar o usuário no banco de dados e garantir a unicidade de e-mail e telefone. As validações de dados (e-mail, senha, telefone) são feitas com `FluentValidation`.

Serviço de Produtos (`Products-API`)

Este microsserviço é responsável por gerenciar o catálogo de produtos.

- **Endpoints da API:**
  - **`POST /api/v1/products`**: Cadastrar um novo produto.
    - **Responsabilidade:** Receber os dados do produto, persistir no banco de dados e, em um futuro próximo (no escopo de expansão), publicar um evento de domínio (`ProductCreatedEvent`).
  - **`GET /api/v1/products`**: Listar produtos com paginação.
    - **Responsabilidade:** Consultar o banco de dados de produtos, aplicando filtros e paginação para otimizar o desempenho da consulta.
  - **`GET /api/v1/products/{id}`**: Obter os detalhes de um produto específico.

- **`PUT /api/v1/products/{id}`**: Atualizar as informações de um produto existente.
- **`DELETE /api/v1/products/{id}`**: Remover um produto do catálogo.

Serviço de Vendas (`Sales-API`)

Este microsserviço é o núcleo da aplicação, gerenciando as transações de venda.

- **Endpoints da API:**
  - **`POST /api/v1/sales`**: Criar uma nova venda.
    - **Responsabilidade:** Receber a lista de itens da venda, aplicar as regras de negócio de desconto (10% para 4-9 itens, 20% para 10-20 itens) na camada de domínio (`Sale.cs`), e salvar a transação no banco de dados. Publica um evento de domínio (`SaleCreatedEvent`) ao finalizar o processo.
  - **`GET /api/v1/sales/{id}`**: Consultar os detalhes de uma venda específica, restrita ao usuário que a criou.
  - **`GET /api/v1/sales`**: Listar as vendas de um usuário, com paginação.
    - **Responsabilidade:** Consultar as vendas do usuário autenticado, aplicando filtros e paginação.
  - **`DELETE /api/v1/sales/{id}`**: Cancelar uma venda.
    - **Responsabilidade:** Alterar o status da venda para cancelada, respeitando as regras de negócio. Essa ação só é permitida pelo usuário criador da venda ou por um gerente. Publica um evento de domínio (`SaleCancelledEvent`).
  - **`GET /api/v1/manager/sales`**: Endpoint exclusivo para o perfil `Manager`.
    - **Responsabilidade:** Permitir que usuários com o papel de gerente visualizem a lista completa de todas as vendas do sistema.
  - **`GET /api/v1/manager/sales/{id}`**: Consultar os detalhes de uma venda específica para gerentes.
  - **`DELETE /api/v1/manager/sales/{id}`**: Cancelar uma venda, endpoint exclusivo para o perfil `Manager`.

Próximos Passos (Roadmap) e Escopo Futuro

A arquitetura atual foi projetada para permitir a evolução do sistema de forma incremental, adicionando novos microsserviços sem a necessidade de reestruturar a solução existente. Os próximos passos incluem a implementação dos seguintes domínios:

- **Microsserviço de Pagamentos (`Payments-API`):**
  - **Responsabilidade:** Gerenciar todo o ciclo de vida de transações financeiras.
  - **Integração:** A `Sales-API` publicaria um evento como `SaleCreatedEvent` no barramento de eventos (RabbitMQ). O novo `Payments-API` consumiria este evento, iniciaria a transação de pagamento com um gateway de

pagamento externo e, em seguida, publicaria um evento de status de pagamento (`PaymentProcessedEvent`) de volta ao barramento.

- **Microsserviço de Entrega (`Delivery-API`):**
    - **Responsabilidade:** Gerenciar o fluxo de entrega dos produtos.
    - **Integração:** O `Delivery-API` consumiria eventos como `PaymentProcessedEvent` (indicando que a venda foi paga com sucesso) e `ProductShippedEvent` (publicado pela `Products-API` ao ter o estoque deduzido) para iniciar o processo de logística.
- **Microsserviço de Fraude (`Fraud-API`):**
    - **Responsabilidade:** Avaliar o risco de fraude de cada transação de venda.
    - **Integração:** O `Fraud-API` consumiria o evento `SaleCreatedEvent` imediatamente após a criação da venda. Ele realizaria uma análise de risco e publicaria um evento de status de fraude (`SaleFraudStatusUpdatedEvent`) no RabbitMQ, permitindo que a `Sales-API` (ou outro serviço) reagisse, por exemplo, suspendendo a venda até a confirmação da análise.

Glossário de Termos

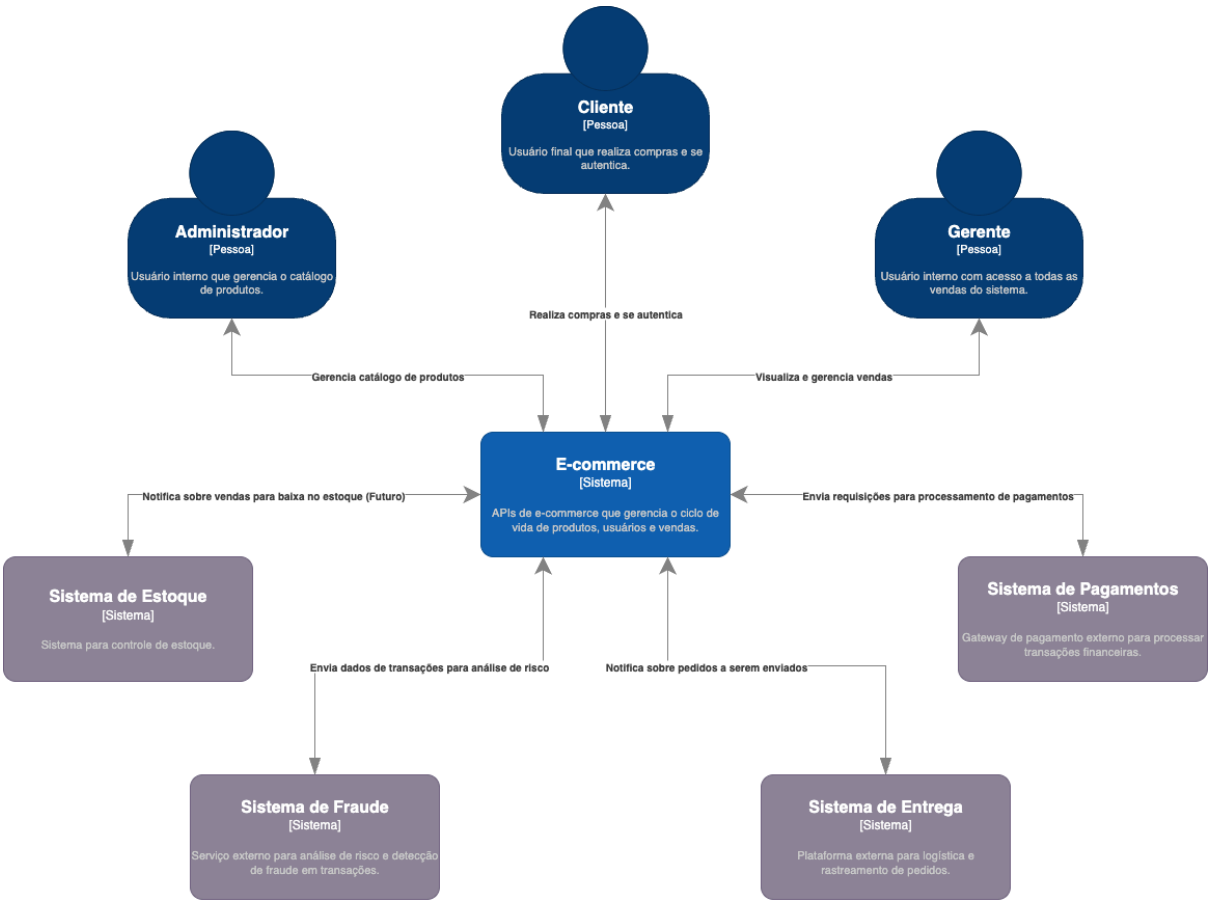
- **API (Application Programming Interface):** Um conjunto de definições e protocolos que permite que diferentes sistemas de software se comuniquem entre si. No contexto do projeto, refere-se aos microsserviços (`Sales-API`, `Users-API`, `Products-API`).
- **Boas Práticas de Desenvolvimento:** Conjunto de técnicas e regras de design que buscam melhorar a qualidade, a manutenibilidade e a escalabilidade do código. O projeto adota SOLID, Clean Architecture e TDD.
- **C4 Model:** Uma abordagem de modelagem de arquitetura de software que permite descrever o sistema de forma hierárquica, do nível mais alto (Contexto) ao mais baixo (Código).
- **Clean Architecture:** Um padrão arquitetural que separa as responsabilidades do sistema em camadas concêntricas (entidades, casos de uso, adaptadores e frameworks), garantindo que a lógica de negócio seja independente de tecnologias externas.
- **CQRS (Command Query Responsibility Segregation):** Padrão de arquitetura que separa as operações de leitura (Queries) das operações de escrita (Commands) em um sistema. Isso permite otimizar e escalar cada operação de forma independente, o que é implementado no projeto com o uso do `MediatR`.
- **DDD (Domain-Driven Design):** Uma abordagem de desenvolvimento de software que foca na modelagem de um domínio de negócio complexo, colaborando com especialistas para criar um modelo rico e expressivo. O projeto usa entidades, agregados e especificações para modelar o domínio.
- **Docker:** Uma plataforma de conteinerização que empacota aplicações e suas dependências em contêineres, garantindo que a aplicação funcione de forma consistente em qualquer ambiente.
- **Event Sourcing:** Um padrão arquitetural onde o estado de um sistema é determinado por uma sequência de eventos. No projeto, eventos de domínio como

`SaleCreatedEvent` são usados para comunicar alterações entre os microsserviços.
- **IaC (Infrastructure as Code):** A prática de gerenciar e provisionar a infraestrutura de TI através de arquivos de código, em vez de processos manuais. O projeto planeja usar Terraform para essa automação.
- **JWT (JSON Web Token):** Um padrão de token de acesso que permite transmitir informações de forma segura entre as partes. É usado para autenticação e autorização no sistema.
- **Kubernetes (K8s):** Uma plataforma de orquestração de contêineres open-source que automatiza a implantação, o escalonamento e o gerenciamento de aplicações conteinerizadas.
- **Microsserviços:** Um estilo de arquitetura de software que estrutura uma aplicação como uma coleção de serviços pequenos e autônomos, cada um com seu próprio domínio de negócio.
- **MVP (Minimum Viable Product):** A versão de um novo produto que permite à equipe coletar a quantidade máxima de aprendizado validado sobre clientes com o mínimo esforço.
- **RabbitMQ:** Um message broker que permite a comunicação assíncrona entre os microsserviços, desacoplando o emissor do receptor de mensagens.
- **Redis:** Um armazenamento de dados em memória, frequentemente utilizado para cache, gerenciamento de sessões e filas de mensagens, com o objetivo de melhorar o desempenho da aplicação.
- **SOLID:** Um conjunto de cinco princípios de design de software (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) que visam a criação de software mais robusto e manutenível.
- **TDD (Test-Driven Development):** Uma metodologia de desenvolvimento de software que consiste em escrever um teste automatizado primeiro, antes de escrever o código de produção necessário para passar no teste.
- **Terraform:** Uma ferramenta de IaC que permite definir e provisionar a infraestrutura de nuvem de forma declarativa e automatizada.

Arquitetura da Solução - Nível Contexto

Visão Geral do Sistema E-commerce

A solução consiste em um sistema de e-commerce com uma arquitetura de microsserviços. A interação com os usuários é feita através de uma interface de frontend, que por sua vez se comunica com o sistema de APIs. A comunicação interna entre os microsserviços é mediada por um barramento de eventos, garantindo que os serviços permaneçam desacoplados, resilientes e escaláveis.



Usuários (Pessoas)

- **Cliente (`Customer`):** Interage com o **Frontend** para criar e visualizar suas vendas. O `Cliente` também realiza seu registro e autenticação através do `Frontend`.
- **Gerente (`Manager`):** Interage com o **Frontend** para acessar a interface de gerenciamento de vendas.
- **Administrador (`Admin`):** Interage com o **Frontend** para gerenciar o catálogo de produtos.
- **Sistema de Estoque Externo:** Um sistema externo que, em um futuro próximo (no escopo de expansão), consumirá eventos do barramento de eventos.

Sistemas (Software)

- **Frontend:** Uma aplicação web (não inclusa neste repositório) que serve como a interface de usuário para `Clientes`, `Gerentes` e `Administradores`.
- **API Gateway (BFF - Backend for Frontend):** Atua como o ponto de entrada seguro para todas as requisições do frontend.
- **Barramento de Eventos (RabbitMQ):** O message broker para comunicação assíncrona entre os microsserviços.
- **Sales-API:** Microsserviço responsável por todo o processo de vendas.
- **Users-API:** Microsserviço responsável pelo gerenciamento de usuários.
- **Products-API:** Microsserviço que gerencia o catálogo de produtos.

- **PostgreSQL:** Banco de dados relacional dedicado e isolado para cada microsserviço (`products-db`, `sales-db`, `users-db`).

Descrição Detalhada das Interações

1. **Interação do Usuário**: Todos os usuários (`Cliente`, `Gerente`, `Administrador`) interagem diretamente com o Frontend, que é a única interface de usuário da solução.
2. **Autenticação**: O `Frontend` envia as credenciais de login para o API Gateway, que as encaminha para a `Users-API`. Após a validação, a `Users-API` retorna um token JWT para o API Gateway, que o repassa para o `Frontend`.
3. **Fluxo de Vendas**: O `Cliente` autenticado, através do `Frontend`, envia uma requisição de criação de venda para o API Gateway. O Gateway valida o token e encaminha para a `Sales-API`. A `Sales-API` processa a requisição, persiste a venda e publica um evento de domínio no RabbitMQ.
4. **Gestão de Produtos**: O `Administrador`, através do `Frontend`, envia requisições de CRUD para o API Gateway, que as encaminha para a `Products-API`.
5. **Tecnologias de Apoio**: PostgreSQL é usado para persistência, e o Redis (mencionado nos requisitos) será utilizado para cache.

Orquestração de Processos - Padrão Saga

Para garantir a consistência dos dados em transações que se estendem por múltiplos microsserviços, o sistema utilizará o padrão de **Saga**. Uma Saga é uma sequência de transações locais onde cada transação local atualiza o banco de dados de um serviço e publica um evento para acionar a próxima transação local na sequência. Se uma transação falhar, a Saga executa uma série de transações de compensação para reverter as alterações feitas pelas transações anteriores.

Fluxo de Transação (Exemplo de Criação de Venda)

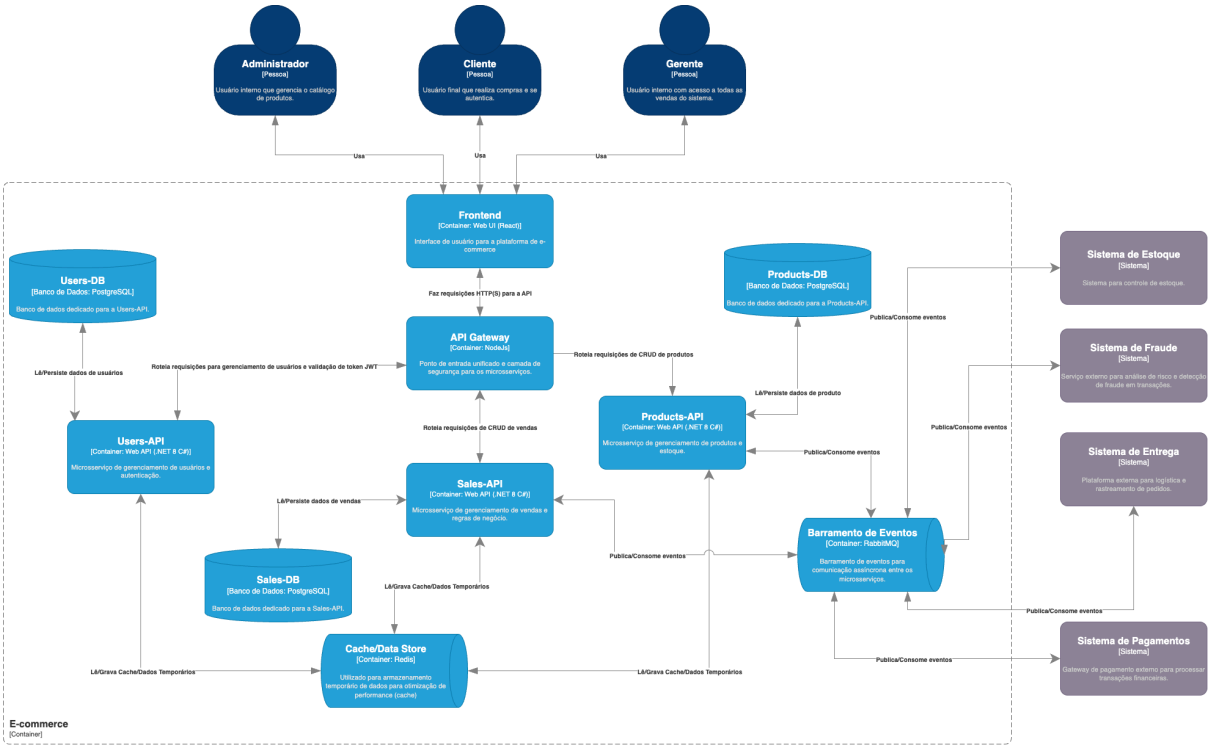
O padrão Saga será aplicado no fluxo de vendas, que envolve a coordenação de vários serviços:

1. **Início da Transação:** A transação de venda é iniciada pelo microsserviço `Sales-API`. Após a persistência local da venda, ele publica um evento para notificar os serviços downstream.
2. **Transações em Serviços Participantes:** Outros serviços, como os de Pagamentos e Estoque, consomem o evento publicado e executam suas próprias transações locais. Cada serviço, ao finalizar sua operação, publica um novo evento para que a sequência continue.
3. **Fluxo de Compensação (Rollback):** Se qualquer serviço na cadeia de transações falhar, ele deve publicar um evento de compensação. Esse evento atua como um sinal para os serviços que já executaram suas transações para que iniciem suas operações de rollback, revertendo as alterações para o estado inicial.

Arquitetura da Solução - Nível Contêineres

Esta seção detalha os principais contêineres que compõem o sistema, as tecnologias que os sustentam e suas interações. Cada contêiner é um componente de software que pode ser implantado e escalado de forma independente.

Visão Geral dos Contêineres



O sistema é composto por cinco contêineres principais: um Frontend, o API Gateway, os três microsserviços (`Users-API`, `Products-API`, `Sales-API`), o barramento de eventos e os bancos de dados.

- **Frontend (Aplicação Web)**
  - **Tecnologia:** React
  - **Responsabilidade:** Interface de usuário para `Clientes`, `Gerentes` e `Administradores`.
  - **Comunicação:** Interage com o sistema de backend através do **API Gateway**.
- **API Gateway (BFF - Backend for Frontend)**
  - **Tecnologia:** NodeJs
  - **Responsabilidade:** Ponto de entrada unificado, validação de segurança (JWT) e roteamento de requisições.
  - **Comunicação:** Recebe requisições do `Frontend` e as encaminha para os microsserviços apropriados (`Users-API`, `Products-API`, `Sales-API`). Também se comunica com a `Users-API` para autenticação.
- **Microsserviço Users-API**
  - **Tecnologia:** C#/.NET 8, PostgreSQL, Redis (cache).

    - **Responsabilidade:** Gerenciamento do ciclo de vida de usuários, autenticação via JWT e autorização baseada em papéis.
    - **Comunicação:** Recebe requisições do **API Gateway** e persiste dados em seu banco de dados dedicado (`users-db`).
- **Microsserviço Products-API**
    - **Tecnologia:** C#/.NET 8, PostgreSQL, RabbitMQ (barramento de eventos).
    - **Responsabilidade:** Gerenciamento do catálogo de produtos e controle de estoque.
    - **Comunicação:** Recebe requisições do **API Gateway**, persiste dados em seu banco de dados (`products-db`) e publica eventos de domínio (`ProductLowStockEvent`) no **RabbitMQ**.
- **Microsserviço Sales-API**
    - **Tecnologia:** C#/.NET 8, PostgreSQL, RabbitMQ (barramento de eventos).
    - **Responsabilidade:** Gerenciamento do ciclo de vida das vendas e aplicação de regras de negócio de desconto.
    - **Comunicação:** Recebe requisições do **API Gateway**, persiste dados em seu banco de dados (`sales-db`) e orquestra transações com o padrão Saga, publicando eventos de domínio (`SaleCreatedEvent`) no **RabbitMQ**.
- **Barramento de Eventos (RabbitMQ)**
    - **Tecnologia:** RabbitMQ.
    - **Responsabilidade:** Facilitar a comunicação assíncrona entre os microsserviços. Atua como um message broker, garantindo o desacoplamento e a resiliência do sistema.
- **Bancos de Dados (PostgreSQL)**
    - **Tecnologia:** PostgreSQL.
    - **Responsabilidade:** Persistência de dados transacionais. Cada microsserviço de negócio possui uma instância dedicada para garantir isolamento e autonomia.
- **Cache (Redis)**
    - **Tecnologia:** Redis.
    - **Responsabilidade:** Armazenamento em memória para dados de acesso frequente, como sessões de usuário ou informações de produtos, visando melhorar o desempenho da aplicação.

Fluxo de Interação de Vendas (Exemplo de Cenário)

1. O `Cliente` interage com o **Frontend**.
2. O `Frontend` envia a requisição para o **API Gateway**, que valida o token JWT.
3. O **API Gateway** roteia a requisição para a **`Sales-API`**.
4. A **`Sales-API`** processa a requisição, persiste a venda em seu banco de dados PostgreSQL e publica o evento `SaleCreatedEvent` no **RabbitMQ**.
5. O **RabbitMQ** encaminha o evento para os consumidores interessados (por exemplo, um futuro serviço de Pagamentos).
6. A **`Sales-API`** retorna uma resposta imediata ao **API Gateway**, que a repassa ao `Frontend`.

7. O sistema se torna **eventualmente consistente**, pois a continuidade da transação (pagamento, baixa de estoque) ocorre de forma assíncrona.

Decisões Arquiteturais de Alto Nível

As decisões de arquitetura foram tomadas para construir um sistema robusto, escalável, resiliente e seguro, focado em alta manutenibilidade. A escolha de cada tecnologia e padrão de design está diretamente alinhada com esses objetivos.

Justificativa para a Arquitetura de Microsserviços

A adoção de microsserviços é suportada por múltiplos fatores, com foco em resiliência, escalabilidade e otimização de recursos:

Microsserviços e Domínios Isolados

- **Decisão:** Adotar a arquitetura de microsserviços.
- **Motivo:** A complexidade de um sistema de e-commerce justifica a divisão em serviços menores e autônomos. Isso permite o desenvolvimento, a implantação e a escalabilidade de cada serviço (`Users-API`, `Products-API`, `Sales-API`) de forma independente. O isolamento de domínios garante que uma falha em um serviço não derrube todo o sistema, aumentando a resiliência.

DDD (Domain-Driven Design)

- **Decisão:** Utilizar o DDD para modelar a lógica de negócio.
- **Motivo:** Em um sistema com regras de negócio complexas (como os descontos de vendas e a gestão de estoque), o DDD garante que a linguagem de negócio esteja refletida diretamente no código. Isso torna o sistema mais compreensível para os especialistas de domínio e mais fácil de manter, especialmente na camada de `Domain` de cada microsserviço.

CQRS (Command Query Responsibility Segregation)

- **Decisão:** Separar as operações de escrita (Commands) das operações de leitura (Queries).
- **Motivo:** Ao separar `CreateSaleCommand` de `ListSalesQuery`, por exemplo, é possível otimizar e escalar cada operação de forma independente. A escrita pode ser focada na consistência transacional, enquanto a leitura pode ser otimizada para performance, com a possibilidade de usar caches ou réplicas de leitura, melhorando a experiência do usuário.

Padrão Saga para Transações Distribuídas

- **Decisão:** Utilizar o padrão Saga (Saga Pattern) para gerenciar a consistência de transações que envolvem múltiplos microsserviços.
- **Motivo:** Em uma arquitetura de microsserviços, não é possível usar transações distribuídas (2PC) para garantir a consistência. O padrão Saga, usando eventos

para orquestrar o fluxo e transações de compensação para lidar com falhas, garante a consistência eventual do sistema de forma resiliente, como no fluxo de uma venda que envolve pagamentos e estoque.

Barramento de Eventos (RabbitMQ)

- **Decisão:** Implementar um message broker (RabbitMQ) como barramento de eventos.
- **Motivo:** O barramento de eventos é crucial para o desacoplamento dos microsserviços. Ele permite a comunicação assíncrona, sendo a base para a implementação do padrão Saga e para a resiliência do sistema, pois um serviço pode estar indisponível, mas os eventos serão enfileirados e processados quando ele voltar.

Tecnologias de Persistência e Cache

- **Decisão:** PostgreSQL para persistência de dados transacionais e Redis para cache.
- **Motivo:** PostgreSQL é uma escolha robusta e confiável para dados transacionais e garante que cada microsserviço tenha um banco de dados isolado. O Redis é um complemento ideal para armazenar dados em memória, melhorando a performance de requisições de leitura e reduzindo a carga nos bancos de dados primários.

Observabilidade com OpenTelemetry

- **Decisão:** Priorizar a implementação de uma arquitetura de observabilidade baseada no OpenTelemetry.
- **Motivo:** Embora a implementação inicial possa ser simplificada (logs com `Serilog`), a adoção de OpenTelemetry como padrão arquitetural garante que o sistema seja "plug-and-play" com ferramentas de mercado (Datadog, Elastic, Dynatrace), facilitando o monitoramento de logs, métricas e, principalmente, traces distribuídos, que são essenciais para diagnosticar problemas em ambientes de microsserviços.
- *Nota:* Conforme solicitado, a prioridade de implementação pode ser ajustada, mas a decisão de design de arquitetura permanece, sendo um objetivo futuro importante.

Boas Práticas de Desenvolvimento

- **Decisão:** Aplicar rigorosamente os princípios SOLID, Clean Architecture e TDD.
- **Motivo:** Essas práticas garantem que o código seja de alta qualidade, fácil de testar, manutenível e extensível. A separação de responsabilidades e o foco em testes (unitários, de integração) são pilares para a construção de um software confiável e de longo prazo.

Preparação para Cloud, Containers e Infraestrutura como Código (IaC)

O MVP, embora focado na funcionalidade, será projetado com uma visão de longo prazo para implantação em ambientes de produção modernos:

- **Containers (Docker) e Orquestração (Kubernetes):** Toda a solução será conteinerizada usando Docker. Será preparada para ser implantada em clusters Kubernetes, permitindo escalabilidade elástica e gerenciamento automatizado, seja localmente (para desenvolvimento), em *clusters on-premise*, ou em ambientes *multi-cloud*.
- **Preparação para Multi-Cloud:** Embora não haja um foco imediato na implantação multi-cloud para o MVP, a arquitetura e as diretrizes de IaC (Infraestrutura como Código) com Terraform serão estabelecidas para permitir essa flexibilidade no futuro. Isso incluirá a colaboração próxima com o **Time de Infraestrutura** para desenvolver scripts Terraform que suportem essa capacidade, sem acoplar a solução a um único provedor de nuvem.
- **Swagger (OpenAPI):** Será utilizado internamente para documentar e expor os endpoints das APIs dos microsserviços, facilitando o desenvolvimento e a integração interna entre os serviços da célula estratégica e, futuramente, outros times.

Padrões Organizacionais e Diretrizes

Esta seção estabelece os padrões, diretrizes e boas práticas que guiarão o desenvolvimento, a operação e a manutenção do projeto.

A adesão a esses padrões é fundamental para garantir a consistência, a qualidade, a segurança e a manutenibilidade da solução, alinhando-se às políticas e estratégias tecnológicas da organização.

Visão Geral dos Padrões

O Projeto será desenvolvido e mantido com base em um conjunto rigoroso de padrões e diretrizes organizacionais.

Isso assegura não apenas a excelência técnica, mas também a facilidade de integração com o ecossistema de TI existente e a capacidade de evolução sustentável. A governança será um pilar central, garantindo que as decisões sejam documentadas e que os times sigam as melhores práticas estabelecidas.

Ciclo de Vida do Desenvolvimento de Software (SDLC) e Metodologia Ágil

O desenvolvimento do projeto seguirá uma abordagem ágil, com foco na entrega incremental de valor e na adaptação contínua.

Metodologia de Trabalho

- Será adotada uma metodologia ágil híbrida (Scrum/Kanban), com ciclos curtos de desenvolvimento (sprints) e entregas frequentes.

- O gerenciamento de tarefas e o progresso do projeto serão realizados utilizando a ferramenta Azure Boards ou Jira, garantindo transparência e visibilidade para todos os *stakeholders*.

Ambientes de Desenvolvimento e Implantação

- Serão estabelecidos ambientes dedicados para Desenvolvimento (Dev), Qualidade (QA), Homologação (UAT) e Produção (Prod), garantindo um fluxo de trabalho seguro e controlado.
- A promoção de código entre ambientes será automatizada via pipelines de CI/CD.

Padrões de Código e Qualidade de Software

A qualidade do código-fonte é prioritária para a manutenibilidade e escalabilidade do Flow Wise.

Boas Práticas de Codificação C#/.NET 8+

- **Princípios SOLID:** Aplicação mandatória dos princípios de Responsabilidade Única, Aberto/Fechado, Substituição de Liskov, Segregação de Interface e Inversão de Dependência para um design modular e testável.
- **Domain-Driven Design (DDD):** Modelagem do domínio de negócio com Agregados, Entidades e Value Objects, garantindo que o código reflita a complexidade do negócio.
- **CQRS e Event Sourcing:** Implementação rigorosa desses padrões para desacoplamento de responsabilidades e garantia de auditabilidade e resiliência, conforme detalhado na arquitetura.
- **Injeção de Dependência (DI):** Uso extensivo de DI para promover o baixo acoplamento e facilitar a testabilidade.
- **Tratamento de Exceções:** Implementação de uma estratégia centralizada e padronizada para tratamento de exceções, com registro adequado em logs.
- **Convenções de Nomenclatura:** Aderência às convenções de nomenclatura C# (PascalCase para tipos e membros públicos, camelCase para parâmetros e variáveis locais).
- **Uso de LINQ:** Preferência pelo uso de LINQ para operações de consulta, promovendo código mais legível e conciso.

Qualidade de Código e Revisão

- **Análise Estática de Código:** Utilização de ferramentas de análise estática (ex: SonarQube, Roslyn Analyzers) integradas ao pipeline de CI para identificar e corrigir débitos técnicos e vulnerabilidades.
- **Métricas de Cobertura de Testes:** Definição de um limiar mínimo de **60% de cobertura de código por testes unitários** para a lógica de negócio crítica, garantindo a validação da funcionalidade e a segurança em refatorações.

- **Revisão de Código (Pull Requests):** Todo código deve passar por revisão por pares através de Pull Requests (PRs) no GitHub, garantindo a qualidade, aderência aos padrões e compartilhamento de conhecimento.

Padrões de Teste e Qualidade Assegurada (QA)

A estratégia de testes visa garantir a qualidade, a funcionalidade correta e o atendimento aos NFRs do Flow Wise.

Estratégia de Testes Automatizados

- **Testes Unitários:** Foco em testes granulares para a lógica de negócio individual de classes e métodos.
- **Testes de Integração:** Verificação da comunicação correta entre os microsserviços e suas dependências (bancos de dados, *message brokers*, APIs de terceiros).
- **Testes de Aceitação (End-to-End):** Validação de fluxos de usuário completos, simulando cenários reais de uso do sistema.
- **Testes de Performance e Carga:** Realização de testes de carga regulares para garantir performance para os serviços.
- **Testes de Resiliência:** Simulação de falhas (ex: latência, indisponibilidade de dependências) para validar a eficácia dos *circuit breakers*, *retries* e mecanismos de *fallback*.

Qualidade Assegurada Contínua

- Integração de testes automatizados ao pipeline de CI/CD, permitindo antecipar novas falhas com as alterações.
- Criação de massa de dados de teste representativa e gerenciamento adequado dos dados de teste.

Padrões de Segurança no Desenvolvimento (Secure SDLC)

A segurança é um pilar fundamental do Flow Wise e será incorporada em todas as fases do SDLC (Secure by Design).

Diretrizes de Desenvolvimento Seguro

- **Mitigação do OWASP Top 10:** Aplicação de práticas de codificação segura para prevenir as vulnerabilidades mais comuns (injeção, quebra de controle de acesso, etc.).
- **Validação de Entradas:** Rigorosa validação de todas as entradas de usuário e dados recebidos de sistemas externos para prevenir ataques.
- **Gerenciamento de Segredos (Secrets Management):** Credenciais e chaves sensíveis não serão hardcoded, mas gerenciadas através de soluções de *secrets management* (ex: Azure Key Vault, AWS Secrets Manager, HashiCorp Vault), com acesso restrito e auditável.

- **Criptografia:** Garantia da criptografia de dados em trânsito (TLS 1.2+) e em repouso (criptografia de banco de dados/armazenamento).

Ferramentas e Processos de Segurança

- **Análise de Segurança de Código (SAST/DAST):** Integração de ferramentas de SAST (Static Application Security Testing) e DAST (Dynamic Application Security Testing) aos pipelines de CI/CD para detecção proativa de vulnerabilidades.
- **Análise de Composição de Software (SCA):** Verificação de dependências de terceiros para vulnerabilidades conhecidas (ex: ferramentas como Snyk ou WhiteSource).
- **Treinamento Contínuo:** Capacitação e conscientização contínua dos desenvolvedores sobre as melhores práticas de segurança.
- **Colaboração com Times de Segurança:** Colaboração próxima com o **Time de Blue e Red Team** da organização para revisão de arquitetura de segurança, validação de controles e definição de estratégias avançadas de pentest e *bug bounty* (pós-MVP).

Padrões de Documentação e Conhecimento

A documentação é um artefato vivo e central para o sucesso do Flow Wise, garantindo que o conhecimento seja compartilhado e acessível a todos.

Abordagem C4 Model para Diagramas Arquiteturais

- Utilização padronizada do **C4 Model** para representação da arquitetura em múltiplos níveis de abstração:
    - **Contexto (Nível 1):** Onde o sistema se encaixa no panorama geral da organização.
    - **Contêineres (Nível 2):** Os principais componentes executáveis (microsserviços, bancos de dados, message brokers) do sistema.
    - **Componentes (Nível 3):** Detalhes internos de um contêiner (lógica de negócio, interfaces).
    - **Código (Nível 4):** Diagramas de classes ou fluxo de código específicos (quando necessário).
- **Ferramenta:** Todos os diagramas serão criados e mantidos no **Draw.io**, com os arquivos-fonte (.xml) versionados no repositório GitHub.

Controle de Versão e Colaboração (Git/GitHub)

O GitHub será a plataforma central para o controle de versão do código-fonte e a colaboração da equipe.

Fluxo de Trabalho (Workflow)

- Será adotado o GitHub Flow ou Git Flow simplificado, com branches de *feature* para o desenvolvimento de novas funcionalidades e *branches* de *release* para lançamentos, garantindo um processo de desenvolvimento e implantação claro.
- A branch *main* será sempre mantida em um estado "implantável".

Processo de Pull Request (PR)

- Todos os PRs exigirão pelo menos um revisor aprovador, além de *checks* de CI/CD (testes automatizados, análise estática de código) passando antes do *merge*.
- As mensagens de commit e os títulos dos PRs devem ser claros e seguir um padrão predefinido para facilitar a rastreabilidade.

Diretrizes de Contribuição

Um documento *`CONTRIBUTING.md`* será disponibilizado no repositório do GitHub, detalhando como outros times e desenvolvedores podem contribuir com o projeto. Este documento incluirá:

- **Setup do Ambiente:** Instruções para configurar o ambiente de desenvolvimento local.
- **Execução de Testes:** Guia para rodar os testes automatizados.
- **Padrões de Commits Semânticos:** Diretrizes claras para a criação de mensagens de commit para garantir a clareza do histórico de versão e a automação de *release notes*.
- **Diretrizes para Abertura de Pull Requests (PRs):** Processo de criação e revisão de PRs, critérios de merge e responsabilidades.

Diretrizes de DevOps e CI/CD

A automação é fundamental para a entrega contínua de valor e a operação eficiente do Flow Wise.

Pipelines de Integração e Entrega Contínua (CI/CD)

Serão implementados pipelines de CI/CD automatizados (ex: utilizando GitHub Actions, Azure DevOps Pipelines) para cada microsserviço, que incluirão:

- Build do código.
- Execução de testes unitários e de integração.
- Análise estática de código e segurança.
- Geração de artefatos (imagens Docker).
- Deployment automatizado para ambientes de desenvolvimento e QA.

Estratégias de Deployment: Para produção, o projeto se preparará para estratégias de *deployment* avançadas como *Canary Releases* (liberação gradual para um pequeno grupo de usuários) ou *Blue/Green Deployments*, para garantir a segurança nas implantações, minimizar o risco de impacto em produção e garantir que novas versões não quebrem a que está em execução. Esta será a abordagem para **garantir que testes de regressão funcionem e impedir a quebra do projeto com novas mudanças ou novos *deploys***.

Infraestrutura como Código (IaC) e Multi-Cloud Ready

- Toda a infraestrutura e recursos de nuvem (ex: Kubernetes, RabbitMQ, Redis, bancos de dados) serão gerenciados via **Terraform**. Os scripts Terraform serão preparados para serem **multi-cloud ready**, permitindo a implantação em diferentes provedores de nuvem no futuro, caso seja uma decisão estratégica.
- O **Time de Infraestrutura** terá um papel fundamental no apoio à criação e manutenção desses scripts IaC.
- **Conteinerização:** A solução será totalmente conteinerizada com Docker, permitindo a execução consistente em ambientes locais (para desenvolvimento), *clusters* Kubernetes *on-premise* ou em qualquer provedor de nuvem.

Monitoramento e Observabilidade

- Conforme NFRs de Observabilidade, o MVP já incorporará a instrumentação necessária para logs estruturados, métricas e *tracing* distribuído (via *Correlation ID*), permitindo uma integração plug-and-play com ferramentas de observabilidade corporativas (Datadog, Elastic, Dynatrace) em fases futuras. O foco será na preparação, não na implementação completa no MVP.
- Será feita uma documentação de fluxo do Correlation ID, detalhando como ele será propagado em todos os pontos de interação (APIs, mensageria, eventos de negócio, webhooks), sendo crucial para a tolerância a falhas, *rollback* de operações e monitoramento.