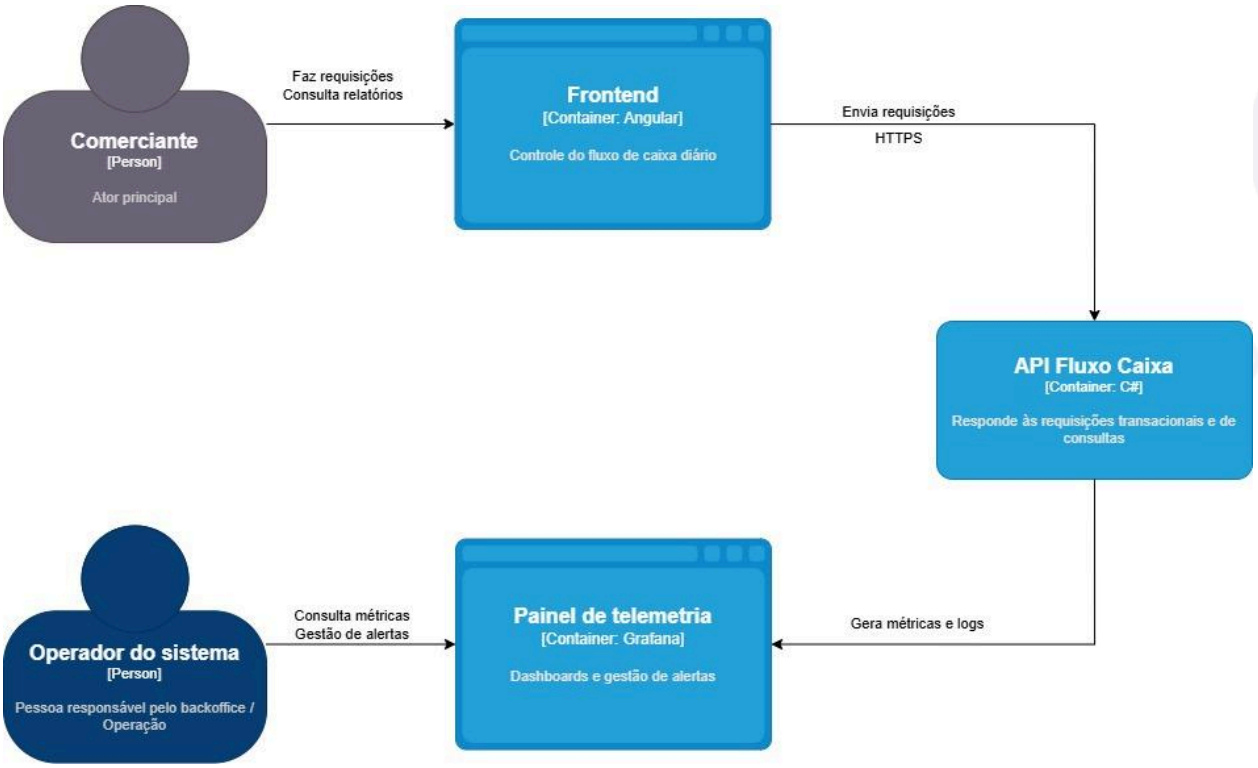
Desafio

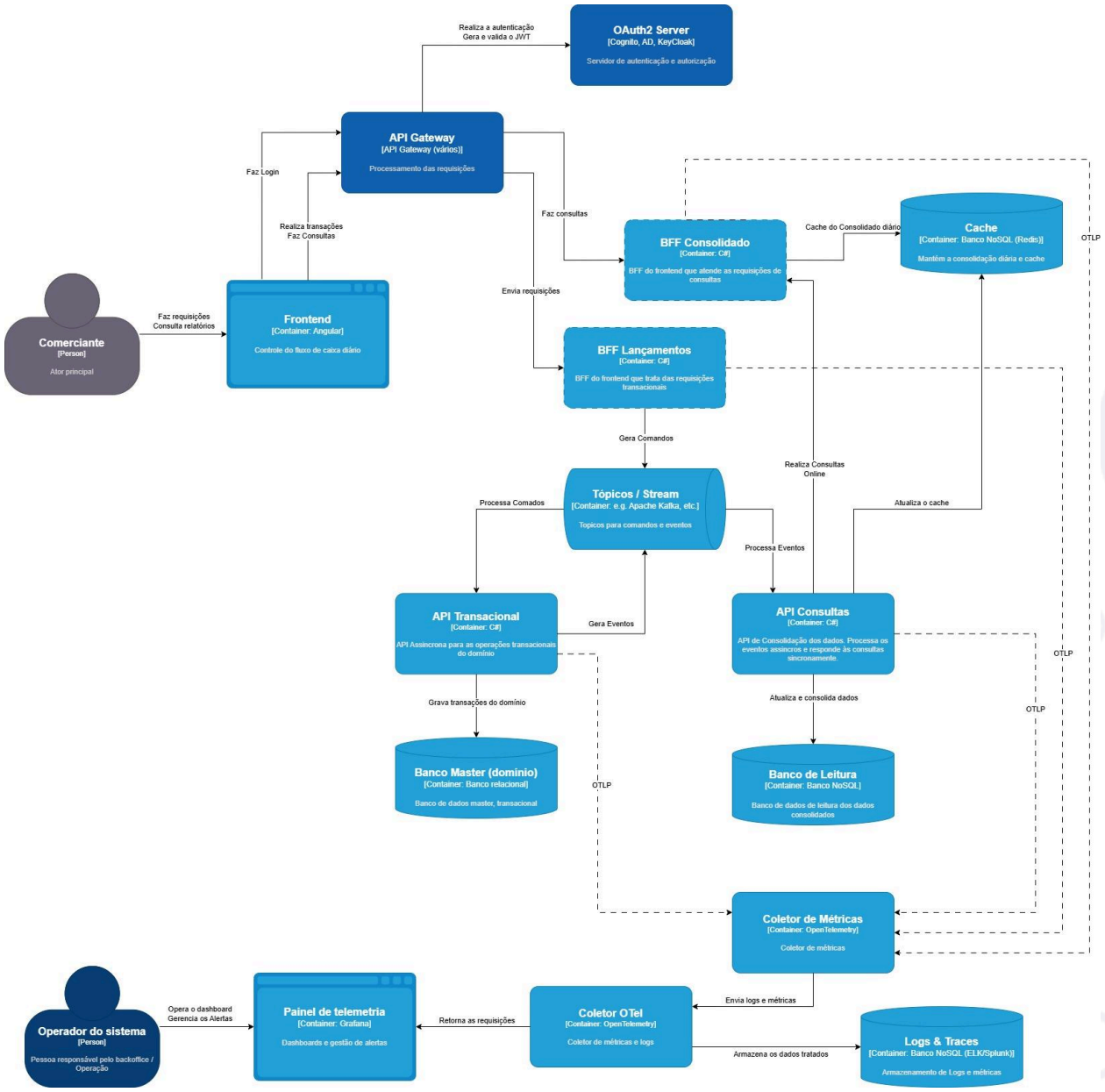
Carrefour

Visão geral da Arquitetura do software

Este sistema foi projetado para gerenciar e controlar o fluxo de caixa diário, atendendo às necessidades de comerciantes e operadores com eficiência, escalabilidade e segurança. Sua arquitetura moderna foi cuidadosamente elaborada para suportar altos volumes de transações, garantir a confiabilidade dos dados e oferecer uma experiência fluida para os usuários finais.



A solução adota um conjunto de fundamentos arquiteturais amplamente reconhecidos no mercado, como Microsserviços, Arquitetura Orientada a Eventos, DDD (Domain-Driven Design), CQRS (Command Query Responsibility Segregation) e o padrão BFF (Backend for Frontend). Esses pilares garantem que o sistema seja robusto, modular e flexível, permitindo que se adapte rapidamente às demandas do negócio e escale conforme necessário.



Principais Características da Arquitetura

- **Microsserviços**: Divisão do sistema em serviços independentes, cada um com responsabilidade única. Essa abordagem promove escalabilidade, resiliência e facilita a manutenção, além de permitir que equipes possam trabalhar de forma descentralizada.
- **Arquitetura Orientada a Eventos**: Comunicação entre os serviços por meio de eventos assíncronos, utilizando tecnologias como Kafka, o que garante desacoplamento, maior resiliência e suporte a escalabilidade horizontal.
- **DDD** (Domain-Driven Design): Modelagem do sistema baseada nos conceitos do domínio de negócios, com contextos delimitados que isolam responsabilidades e regras específicas, garantindo alinhamento com as necessidades da empresa.
- **CQRS (Command Query Responsibility Segregation)**: Separação entre comandos (escrita) e consultas (leitura), permitindo otimização para cada tipo de operação. Isso é

viabilizado por bancos de dados especializados: um relacional para operações transacionais e um banco NoSQL para dados consolidados.
- **Padrão BFF:** Implementação de Backends específicos para o Frontend, atendendo diretamente às necessidades de cada interface, reduzindo a complexidade no cliente e otimizando a comunicação com os serviços de backend.

Benefícios da Arquitetura

Essa abordagem arquitetural oferece benefícios significativos, como:

- **Escalabilidade**: Serviços independentes podem ser dimensionados separadamente conforme a demanda.
- **Resiliência**: O uso de eventos e fallback garante que o sistema continue operacional mesmo diante de falhas em componentes específicos.
- **Desempenho**: Consultas otimizadas com cache e bases de leitura reduzem a latência para os usuários finais.
- **Manutenibilidade**: A modularidade e a separação clara de responsabilidades facilitam o desenvolvimento contínuo e a implementação de novas funcionalidades.
- **Segurança**: Autenticação e autorização são gerenciadas de forma centralizada por meio de um servidor OAuth2, garantindo proteção robusta aos dados.
- **Observabilidade e monitoramento contínuos**: Garantem que o sistema seja confiável e que possíveis falhas sejam identificadas e corrigidas rapidamente, antes que impactem os usuários.

A apresentação a seguir detalha como os conceitos e tecnologias foram aplicados para alcançar esses objetivos.

Frontend

O frontend do sistema será desenvolvido utilizando Angular, proporcionando uma aplicação web moderna, responsiva e de alto desempenho. Ele será integrado a um BFF, que mediará as comunicações com os serviços de backend. As requisições transacionais serão processadas de forma assíncrona, enquanto as consultas e relatórios serão realizadas de forma síncrona, com os dados provenientes de uma base consolidada. Essa arquitetura garante uma experiência fluida e eficiente para os usuários, alinhada às demandas do negócio e ao volume esperado de operações.

Padrão BFF
Sugerimos que o padrão BFF seja adotado para essa solução de frontend. O padrão "Backend for Frontend" (BFF) oferece um maior desacoplamento para a construção do frontend . Aqui estão algumas das principais vantagens: • **Customização para o Cliente**: Cada BFF pode ser personalizado para atender às

necessidades específicas de cada tipo de cliente. Por exemplo, o BFF para um aplicativo Angular pode ser otimizado para entregar dados e funcionalidades que se alinham melhor com a interface de usuário web.

- **Redução de Carga no Cliente**: Ao mover a lógica e processamentos que normalmente seriam feitos no front para o BFF, reduz-se a carga nos navegadores. Isso pode melhorar a performance do front e proporcionar uma experiência de usuário mais fluida.
- **Minimização de Chamadas de Rede**: O BFF pode agir como um agregador de várias chamadas a sistemas ou micro serviços backend, reduzindo o número de chamadas de rede que o frontend precisa fazer. Isso é especialmente útil em ambientes onde a latência de rede é uma preocupação.
- **Segurança Aprimorada**: O BFF pode também implementar controles de segurança específicos, como autenticação e autorização, de maneira centralizada para o front, ajudando a reduzir a complexidade do código e garantir um manejo consistente das políticas de segurança.
- **Facilita a Adaptação a Mudanças**: Mudanças em lógicas de negócios, integrações ou mesmo alterações nas APIs do backend podem ser mais facilmente gerenciadas no BFF sem necessidade de grandes alterações no código do front.
- **Otimização de Dados**: O BFF pode formatar e otimizar dados para que sejam mais adequados para o consumo pelo front, como filtrar, enriquecer ou transformar dados de acordo com as necessidades específicas do front.

Embora o uso de BBFs traga benefícios para o front, alguns trade-offs devem ser considerados na decisão, tais como: aumento na complexidade geral do sistema, risco de duplicação de lógicas de negócio, uma camada adicional para ser testada.

Banco de leitura NoSQL

Sugerimos a adoção de um banco de leitura NoSQL em conjunto com o BFF para os módulos de frontend que preveem um alto volume de consultas ou cujas consultas em bancos relacionais sejam custosas.

A desnormalização de dados em um banco de leitura, especialmente em um sistema que utiliza tecnologias NoSQL, é uma prática estratégica para otimizar o desempenho das consultas. Aqui estão alguns pontos chave que justificam essa abordagem:

- **Redução do Número de Consultas (Join Operations):** Em bancos de dados relacionais tradicionais, os dados são normalmente armazenados em várias tabelas relacionadas, o que exige operações de junção (joins) para reconstruir a informação completa durante as consultas. Cada junção adiciona complexidade computacional e pode retardar significativamente o tempo de resposta. Em contraste, bancos de dados desnormalizados armazenam informações relacionadas em uma única estrutura ou

documento, eliminando a necessidade de junções frequentes e, consequentemente, acelerando as consultas.

- **Otimização para Leituras:** Os sistemas que dependem fortemente de leituras rápidas e eficientes, como aplicações de tracking em tempo real, podem se beneficiar enormemente da desnormalização. Isso porque os dados desnormalizados são organizados de maneira a servir a maioria das consultas de forma direta e rápida. Bancos de dados NoSQL são projetados para escalar horizontalmente, e sua estrutura flexível permite que sejam otimizados para leituras rápidas, manipulando grandes volumes de dados com eficiência.
- **Simplificação da Modelagem de Dados:** A desnormalização simplifica o modelo de dados, uma vez que relacionamentos complexos são aplainados em uma única estrutura. Isso reduz a complexidade no desenvolvimento de aplicações, pois o modelo de dados é mais fácil de entender e de manipular programaticamente.
- **Escala Horizontal:** Bancos de dados NoSQL, que frequentemente utilizam estruturas desnormalizadas, são projetados para uma fácil distribuição dos dados em vários servidores. Isso permite uma escalabilidade horizontal eficaz, essencial para sistemas com alta demanda de dados e consultas.
- **Consistência de Leitura**: Com dados desnormalizados, a consistência de leitura é melhorada porque todas as informações necessárias estão contidas em uma única entidade ou documento. Isso evita problemas de inconsistência que podem surgir devido à latência ou falhas em atualizar múltiplas tabelas relacionais simultaneamente.

A desnormalização em um banco de dados NoSQL para operações de leitura intensiva oferece vantagens significativas em termos de desempenho, escalabilidade e simplicidade. Contudo, é importante balancear esses benefícios com os potenciais trade-offs, como o aumento no uso de espaço de armazenamento e a complexidade adicional na manutenção da consistência durante as operações de escrita.

Backend

O backend do sistema será desenvolvido em C#, utilizando uma arquitetura baseada em microsserviços para garantir modularidade, escalabilidade e alta performance. O design segue o padrão CQRS (Command Query Responsibility Segregation), separando as operações de escrita (transacionais) das operações de leitura (consultas e relatórios). As transações serão processadas de forma assíncrona, enquanto as consultas serão realizadas de forma síncrona sobre uma base de dados consolidada, otimizando o desempenho e a experiência do usuário.

Para garantir visibilidade operacional e diagnóstico eficiente, será utilizada a OpenTelemetry para coleta de métricas, logs e traces de toda a arquitetura, integrando-se a ferramentas de monitoramento, como o Grafana. Essa abordagem oferece não apenas alta confiabilidade e rastreabilidade, mas também suporte para a identificação e resolução proativa de problemas, assegurando a robustez do sistema mesmo em cenários de alta volumetria.

Microsserviços

Sugerimos a adoção de uma arquitetura baseada em microsserviços.

A utilização de uma arquitetura de microsserviços oferece inúmeras vantagens, especialmente em ambientes que demandam escalabilidade, resiliência e flexibilidade. Abaixo estão os principais benefícios desta abordagem:

- **Modularidade e Isolamento de Serviços**: Cada microsserviço é projetado para realizar uma função específica do sistema. Isso promove uma modularidade significativa, onde cada serviço pode ser desenvolvido, implantado e escalado de forma independente, sem impactar os outros.
- **Escalabilidade Independente:** Com a arquitetura de microsserviços, é possível escalar apenas os serviços que apresentam maior demanda. Isso permite uma melhor utilização dos recursos, reduzindo custos e otimizando a performance.
- **Desenvolvimento e Implantação Ágil**: A separação em pequenos serviços facilita o trabalho de equipes distribuídas, que podem trabalhar simultaneamente em diferentes serviços sem causar conflitos.
- **Flexibilidade Tecnológica:** Cada microsserviço pode ser desenvolvido com a tecnologia que melhor atende às suas necessidades específicas. Isso permite que equipes escolham linguagens, frameworks e bancos de dados mais adequados para cada contexto.
- **Resiliência:** Um sistema baseado em microsserviços tende a ser mais resiliente, pois uma falha em um serviço não afeta diretamente o funcionamento dos demais. Isso permite maior disponibilidade do sistema como um todo.
- **Facilidade na Manutenção e Evolução:** A granularidade dos microsserviços facilita a identificação de problemas e a realização de mudanças sem comprometer todo o sistema, promovendo uma manutenção simplificada.
- **Integração com padrões modernos:** A arquitetura de microsserviços é naturalmente compatível com padrões como CI/CD, escalabilidade em nuvem e monitoramento distribuído. Ferramentas como Kubernetes, Docker e Istio tornam o gerenciamento e a orquestração mais eficientes.

No entanto, é importante considerar alguns desafios associados a esta arquitetura, como: gerenciamento da comunicação, complexidade operacional, consistência e latência.

Arquitetura Orientada a Eventos

Sugerimos a adoção de uma arquitetura orientada e eventos para este backend.

A adoção de uma arquitetura orientada a eventos pode trazer uma série de vantagens, especialmente em termos de escalabilidade, desacoplamento e eficiência. Aqui estão algumas das principais vantagens dessa abordagem:

- **Desacoplamento**: Em uma arquitetura orientada a eventos, os componentes do sistema se comunicam principalmente através de eventos, o que reduz as dependências diretas entre eles. Isso facilita a manutenção e evolução de cada componente de forma independente, sem impactar outros componentes do sistema.
- **Reatividade e Resiliência**: Sistemas orientados a eventos são intrinsecamente projetados para serem reativos, respondendo a eventos à medida que ocorrem. Isso pode melhorar a resiliência do sistema, pois permite que ele lide melhor com variações de carga e falhas. Por exemplo, se um serviço falhar, os eventos podem ser enfileirados e processados quando o serviço for restaurado.
- **Escalabilidade**: A arquitetura orientada a eventos facilita a escalabilidade horizontal do sistema. Como os componentes são desacoplados, eles podem ser escalados independentemente com base em suas próprias necessidades de carga. Além disso, o processamento de eventos pode ser distribuído entre múltiplas instâncias para balancear a carga.
- **Melhoria na performance**: Ao evitar chamadas síncronas bloqueantes, um sistema baseado em eventos pode melhorar a performance geral. Isso é particularmente útil em sistemas com integrações externas, onde as chamadas síncronas podem resultar em latências significativas.
- **Consistência Eventual**: Embora possa ser um desafio, a consistência eventual é uma característica comum em sistemas orientados a eventos. Ela permite que o sistema continue operando de maneira eficiente em cenários de alta disponibilidade e particionamento, garantindo que todas as partes do sistema eventualmente cheguem a um estado consistente.
- **Facilita a implementação de padrões avançados**: Como os Circuit Breaker, Backpressure, e outros padrões de resiliência que são mais fáceis de implementar e gerenciar em um sistema baseado em eventos devido ao seu design inerentemente desacoplado e assíncrono.

Ao adotar essa arquitetura deve-se considerar alguns trade-offs como: aumento na complexidade inicial do sistema, gerenciamento de consistência eventual e a garantia da entrega das mensagens.

CQRS

Sugerimos a adoção do padrão CQRS (Separação de Responsabilidades entre Comando e Consulta) para este backend.

A arquitetura CQRS é particularmente vantajosa em sistemas com alta complexidade de negócio ou grandes volumes de dados, onde as operações de leitura e escrita possuem

diferentes requisitos de performance, escalabilidade e consistência. Abaixo destacamos os principais benefícios dessa abordagem:

- **Separação de Responsabilidades**: O CQRS separa explicitamente as operações de escrita (comandos) das operações de leitura (consultas), permitindo que cada uma seja otimizada de forma independente, tanto em termos de modelo de dados quanto de infraestrutura.
- **Escalabilidade Direcionada**: Como comandos e consultas são tratados separadamente, é possível escalar cada parte do sistema de acordo com suas próprias necessidades. Por exemplo, um sistema com alta demanda por leituras pode escalar apenas os serviços de consulta.
- **Modelos de Dados Otimizados**: Permite o uso de diferentes modelos de dados para leitura e escrita, otimizando o desempenho de cada um. Por exemplo, enquanto o lado de escrita pode manter um modelo transacional detalhado, o lado de leitura pode usar um modelo desnormalizado para consultas rápidas.
- **Facilidade na Evolução**: Com o CQRS, alterações no modelo de leitura ou na lógica de escrita podem ser feitas de forma isolada, reduzindo o impacto no sistema como um todo.
- **Capacidade de Suportar Eventos**: O CQRS é frequentemente combinado com o Event Sourcing, permitindo que todas as mudanças no estado do sistema sejam capturadas como uma série de eventos imutáveis. Isso facilita a auditoria, o replay de eventos e a construção de novos modelos de leitura com base em eventos históricos.
- **Redução de Conflitos de Concorrência**: Ao separar leitura e escrita, o CQRS minimiza os bloqueios causados por transações concorrentes, o que melhora a experiência do usuário e reduz o risco de falhas em cenários de alta carga.
- **Resiliência e Flexibilidade**: A independência entre os lados de leitura e escrita permite maior resiliência. Por exemplo, se um serviço de consulta estiver fora do ar, os comandos podem continuar a operar, garantindo a integridade do sistema.

Apesar das vantagens, o CQRS também apresenta desafios que devem ser considerados, como: aumento da complexidade, consistência eventual, sincronização entre os bancos de escrita e leitura.

Monitoração e Observabilidade

Sugerimos a implementação de uma estratégia robusta de monitoração e observabilidade utilizando OpenTelemetry, Prometheus e Grafana para este backend.

OpenTelemetry: Padronização e Flexibilidade na Coleta de Dados

- **Coleta unificada:** OpenTelemetry é uma estrutura aberta que permite coletar métricas, logs e rastreamentos em um único formato, promovendo padronização e

interoperabilidade entre ferramentas.

- **Instrumentação facilitada:** Com SDKs e bibliotecas para diversas linguagens, a instrumentação do sistema torna-se mais simples, reduzindo o esforço de desenvolvimento.
- **Integração com outras ferramentas:** Os dados coletados pelo OpenTelemetry podem ser exportados para múltiplos backends, incluindo Prometheus e Grafana, garantindo flexibilidade.

Prometheus: Monitoramento de Métricas em Tempo Real

- **Armazenamento eficiente de métricas:** Prometheus é otimizado para armazenar séries temporais de métricas, permitindo monitoramento em tempo real de desempenho e utilização de recursos.
- **Alertas configuráveis:** Com o Alertmanager, Prometheus possibilita a criação de alertas personalizados com base em regras definidas, permitindo ações proativas em caso de falhas ou degradação.
- **Query language poderosa:** A PromQL (Prometheus Query Language) facilita análises avançadas e correlação de métricas.

Grafana: Visualização e Análise de Dados

- **Paineis personalizáveis:** Grafana permite criar dashboards ricos e interativos, adaptados às necessidades específicas do negócio e da operação.
- **Visualização centralizada:** Com Grafana, é possível consolidar dados de múltiplas fontes, incluindo Prometheus, logs e rastreamentos, em uma única interface.
- **Alertas visuais:** Além dos alertas de Prometheus, Grafana oferece opções para alertas visuais e acionáveis diretamente nos dashboards.

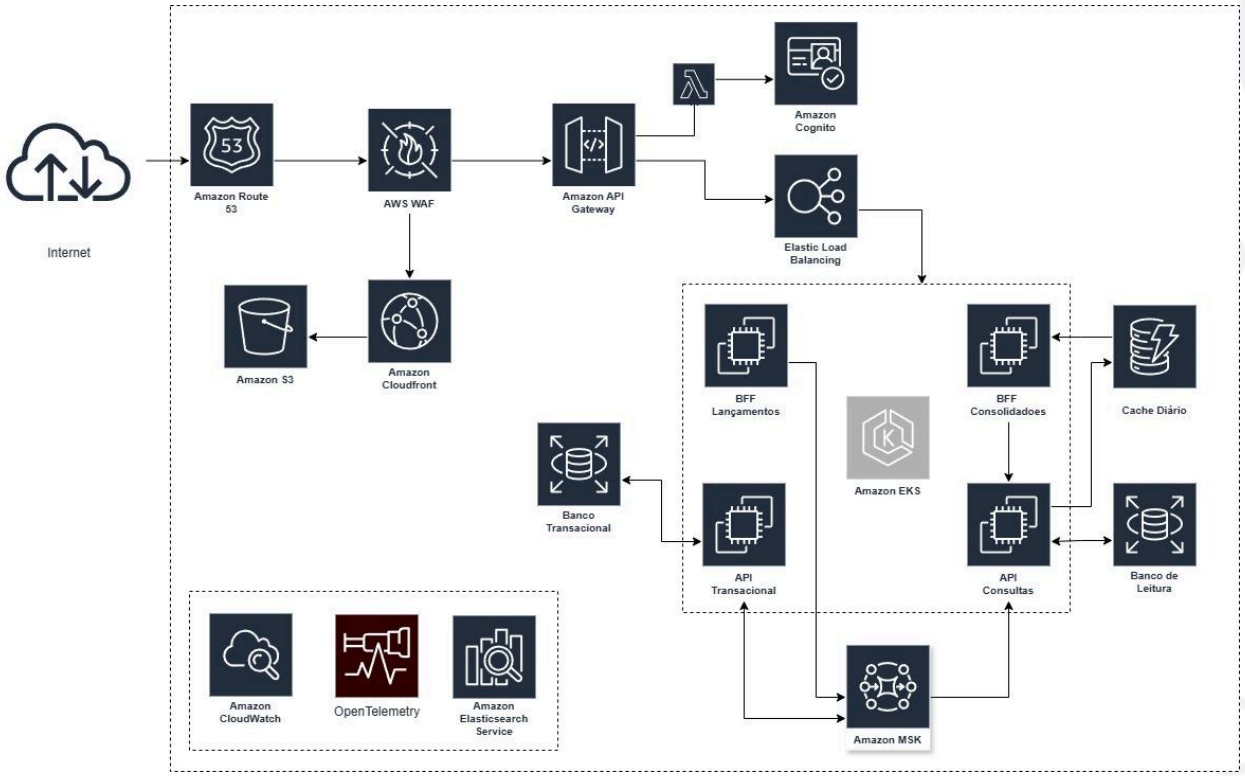
Principais Benefícios da Stack Completa

- **Observabilidade distribuída:** A integração de logs, métricas e rastreamentos distribuídos oferece uma visão abrangente do sistema, facilitando a detecção e resolução de problemas complexos.
- **Diagnóstico rápido:** Com dashboards intuitivos e alertas configuráveis, é possível identificar gargalos e pontos de falha rapidamente, minimizando o impacto nos usuários finais.
- **Escalabilidade e resiliência:** A stack é projetada para escalar horizontalmente, suportando sistemas de alta volumetria sem comprometer a performance.

A adoção dessa estratégia garante que o sistema seja monitorado de forma proativa, com uma observabilidade sólida que auxilia na tomada de decisões baseadas em dados. Porém alguns trade-offs devem ser considerados: Configuração inicial da stack é complexa, aumento no custo de infraestrutura e realização de treinamento das equipes para o bom uso dos SDK e ferramentas.

Visão geral da Arquitetura de infraestrutura em nuvem

Esta arquitetura em nuvem foi projetada para garantir alta disponibilidade, escalabilidade e segurança, utilizando serviços da AWS. Ela combina ferramentas avançadas como Route 53, CloudFront e WAF para proteção e entrega eficiente de conteúdo, além de APIs gerenciadas pelo API Gateway e funções serverless com AWS Lambda. O armazenamento e processamento são otimizados por S3, RDS e DynamoDB, enquanto serviços como CloudWatch e OpenTelemetry garantem monitoramento e análise de desempenho. Com uma abordagem modular e resiliente, essa solução é ideal para aplicações modernas e de alto desempenho.



A arquitetura na nuvem descrita no diagrama utiliza serviços da AWS, por conveniência, para criar uma solução robusta, escalável e segura. Aqui está um resumo:

- **Entrada de Rede**: A comunicação com o sistema começa pela Internet, direcionada por meio do Amazon Route 53 (serviço DNS gerenciado), permitindo alta disponibilidade e roteamento de tráfego.
- **Distribuição de Conteúdo**: O Amazon CloudFront é usado como CDN para melhorar a entrega de conteúdo estático e reduzir latências.
- **Camada de Segurança**: Um AWS WAF (Web Application Firewall) protege contra ataques como DDoS e injeções maliciosas.
- **Gerenciamento de Tráfego**: O tráfego é gerenciado pelo Amazon API Gateway, que funciona como ponto central para a integração com microsserviços e BFFs.
- **Autenticação**: O Amazon Cognito é responsável pela autenticação de usuários.

- **Serviços de Backend**: O Amazon Elastic Load Balancer (ELB) distribui solicitações para instâncias EC2 que rodam aplicações backend. Serviços adicionais, como o Amazon MSK (Managed Streaming for Kafka), são utilizados para processamento de mensagens e integração entre serviços.
- **Armazenamento**: O Amazon S3 armazena arquivos estáticos e objetos. Bancos de dados transacionais e de leitura são suportados pelo Amazon RDS.
- **Cache e Consultas**: Amazon DynamoDB é utilizado como cache diário para melhorar a performance de consultas frequentes. APIs específicas, como para transações e consultas, são geridas separadamente em instâncias EC2.
- **Monitoramento e Logs**: Amazon CloudWatch coleta métricas, logs e eventos para monitoramento. O OpenTelemetry é integrado para rastreamento distribuído e análise detalhada de desempenho. Amazon Elasticsearch Service é utilizado para busca e análise de logs.