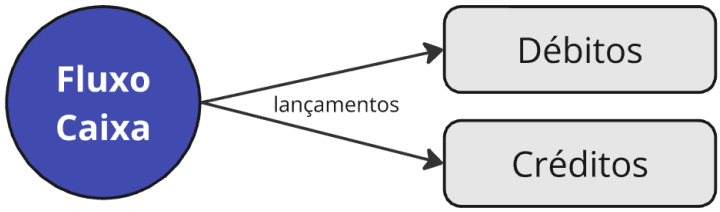
Solução do controle de fluxo de caixa

A solução proposta se baseia no conhecimento “geral” de um fluxo de caixa, onde são utilizados lançamento de crédito e débito para gerar o saldo diário.



Requisitos funcionais

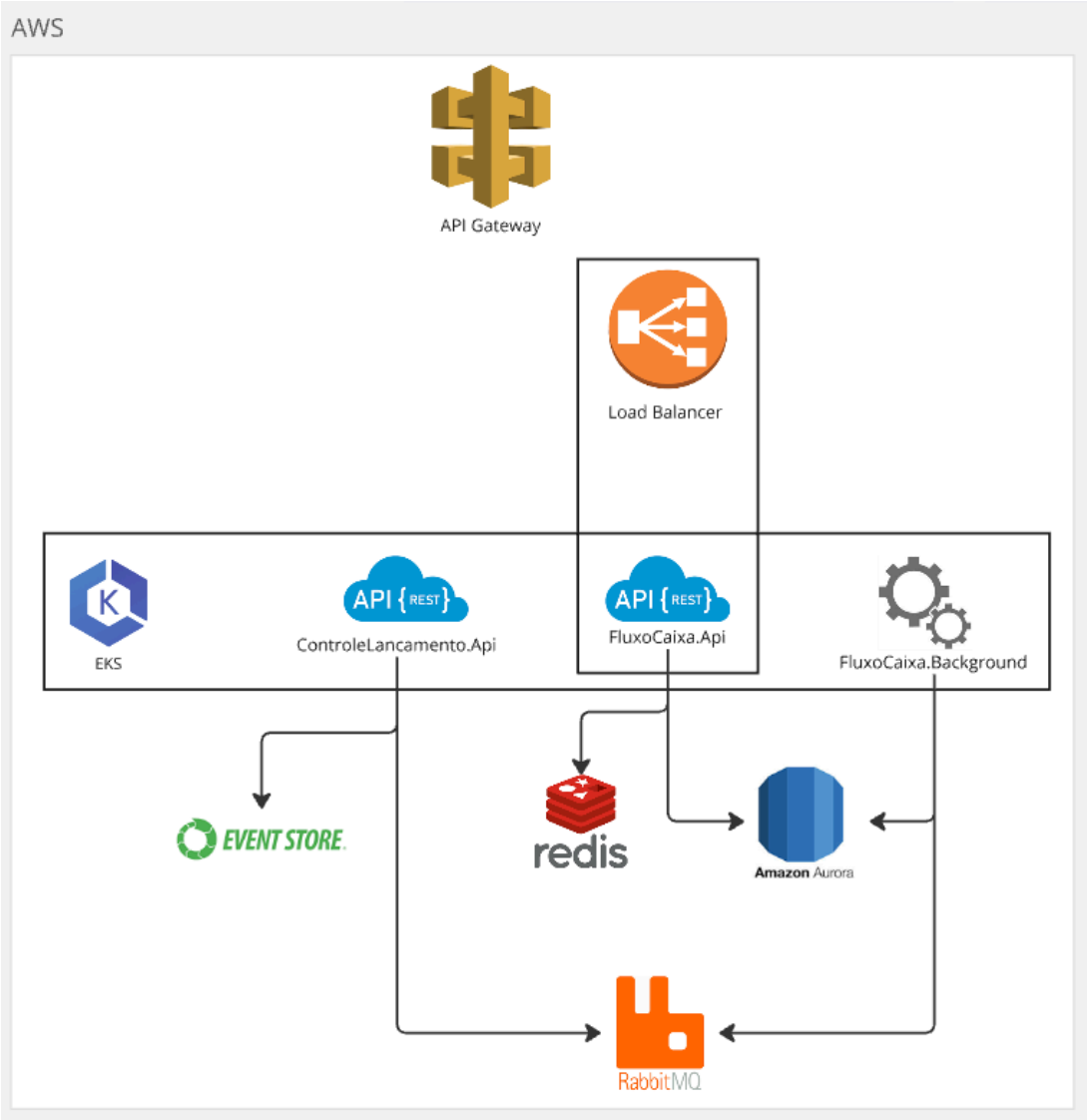
Controle de lançamentos de débitos e créditos
Relatórios diários sumarizado destes lançamentos

Requisitos não funcionais

Alta disponibilidade
Desempenho
Tolerante a falhas
Escalabilidade

Arquitetura da Solução

No próprio descritivo da solução já foi identificado a necessidade da criação de dois serviços, um para controle de lançamentos e outro para o resultado do fluxo de caixa. Partindo desta premissa, pensei numa solução com dois bounded context (ControleLancamento e FluxoCaixa), onde utilizei uma arquitetura baseada em eventos para realizar a comunicação entre eles. A solução é composta por três projetos, sendo ControleLancamento.Api, FluxoCaixa.Api e FluxoCaixaBackground.
O diagrama abaixo mostra os principais componentes da arquitetura da solução



Temos uma Api Gateway responsável por gerenciar toda requisição para as duas APIs desenvolvidas. E também foi adicionado um Load Balancer com foco na FluxoCaixa.Api para garantir que as demandas de requisições em momentos de picos sejam atendidas com sucesso, além dos recursos de cluster específicos para leitura disponíveis no AWS Aurora. Além disso, também foi pensado em um componente para cache de dados, utilizando o Redis. Para a questão da elasticidade dos containers, foi incluído o EKS.
Toda comunicação entre os dois contextos é realizada por fila, nesse caso o RabbitMQ onde conseguimos desacoplar completamente os dois serviços. Para ingestão dos dados, foi proposto um serviço de background no contexto do FluxoCaixa (FluxoCaixa.Background) rodando também no EKS. A vantagem de não rodar essa ingestão de dados no mesmo processo da FluxoCaixa.Api é que podemos ter um controle maior da elasticidade da ingestão de dados, totalmente desacoplado do serviço que suporta os relatórios da solução.

Justificativa da solução

A escolha da nuvem AWS foi devido a oferecer uma quantidade maior de serviços e recursos do que os concorrentes em questão.
A Api Gateway é peça fundamental em uma estrutura baseada em apis, basicamente utilizaremos todas as funções disponíveis como padronização de acesso aos serviços, segurança, limitação de taxa etc
O Load Balancer foi escolhido devido ao alto volume de acesso ao FluxoCaixa.Api, desta forma conseguimos distribuir de forma correta as chamadas entre os containers.
O Kubernetes é a escolha mais que padrão para gerenciar containers e seus ciclos de vida.
EventStore é um banco focado e especializado para event sourcing
O Redis também é a escolha mais assertiva quando estamos falando de cache distribuído, esse componente fornece todo o suporte a registros em memória que as instâncias do FluxoCaixa.Api necessitar.
O AWS Aurora possui um recurso excelente que disponibiliza vários clusters de leitura independente da escrita, como esse banco sofre bastante com inserts e selects essa opção não poderia ser descartada.
O RabbitMQ está como principal componente de integração entre os dois contextos (ControleLancamento e FluxoCaixa). Essa fila é utilizada para receber os eventos que acontecem no controle de lançamento e consequentemente disponibilizar as informações para o Fluxo de Caixa.

Projetos

ControleLancamento.Api
Mesmo sendo um projeto simples, quis utilizar DDD para demonstrar o valor que isso pode gerar em um projeto real. Onde temos a linguagem universal entre negócio e desenvolvimento, modelos ricos com ações claras e bem definidas, além de design patterns.
A arquitetura dessa API está em CQRS + Event Sourcing, mesmo não tendo implementações de “queries” (por falta de tempo hábil para codar) escolhi essa arquitetura porque, na minha visão, a separação lógica de escrita e leitura fazia total sentido para expressar as ações. Os eventos foram perfeitos para traduzir o mundo real em código e como bônus gerar desacoplamento com qualquer outro microsserviço.
Design Patterns utilizados: Mediator, Repository, DomainEvents

FluxoCaixa.Api
Também utilizado DDD para demonstrar os valores já citados e também uso de alguns design patterns.
Esta API está desenhada em Clean Architecture, ou seja, baseada em domínio. Mesmo sendo uma API com apenas “gets”, tentei criar um domínio rico de fluxo de caixa para demonstrar algumas ideias de implementações e padrões.
Design Patterns utilizados: ApplicationService, DTO, DomainService, Repository

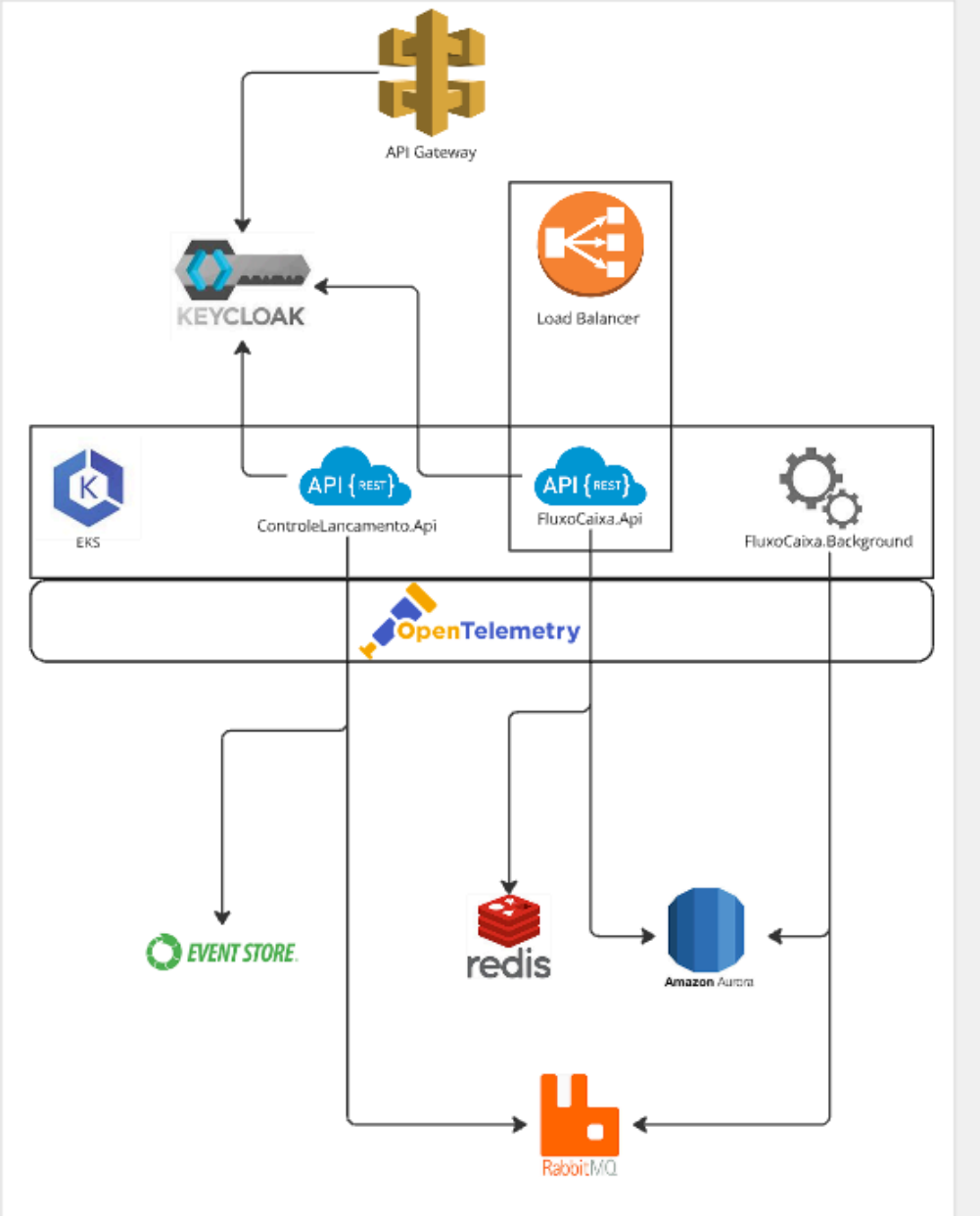
FluxoCaixaBackground
Esse componente poderia ser um simples “CRUD”, mas segui a mesma ideia de utilizar DDD e arquitetura baseada em Clean Architecture. Isso trouxe algumas vantagens, pois temos duas “threads” sendo executadas (uma para cada tipo de lançamento) buscando dados do RabbitMQ. Na minha visão, o código ficou simples e de fácil leitura.
Design Patterns utilizados: DAO, DomainService,

OBS: Fiz questão de criar modelos (domain) com visões diferentes entre os microsserviços para exemplificar o desacoplamento total entre as partes do sistema e como foi possível aplicar padrões diferentes e específicos para cada tipo de problema de domínio de forma eficiente.

Proximos Passos

No desafio não foi citado a questão da segurança da informação, mas entendo que isso é um fator importante para qualquer arquiteto pensar. Como um próximo passo, levaria em consideração a autenticação e autorização e também incluiria um componente para observabilidade. Para esses dois casos, utilizaria KeyCloak para suprir as necessidade de autenticação/autorização e para observabilidade utilizaria o OpenTelemetry (um pouco mais a nível de engenharia/codigo) para ter um recurso agnóstico ao fornecedor, sendo possível escolher entre Datadog, ELK Stack, Dynatrace etc
Segue desenho alterado do diagrama

AWS


API Gateway
KEYCLOAK
Load Balancer
EKS
API
REST
ControleLancamento.Api
API
REST
FluxoCaixa.Api
FluxoCaixa.Background
OpenTelemetry
EVENT STORE
redis
Amazon Aurora
RabbitMQ