1. Visão Geral da Solução

A solução proposta atende ao desafio descrito no documento e foi desenvolvida priorizando isolamento de domínios, resiliência, escalabilidade, observabilidade, aderência a requisitos não funcionais e funcionais.

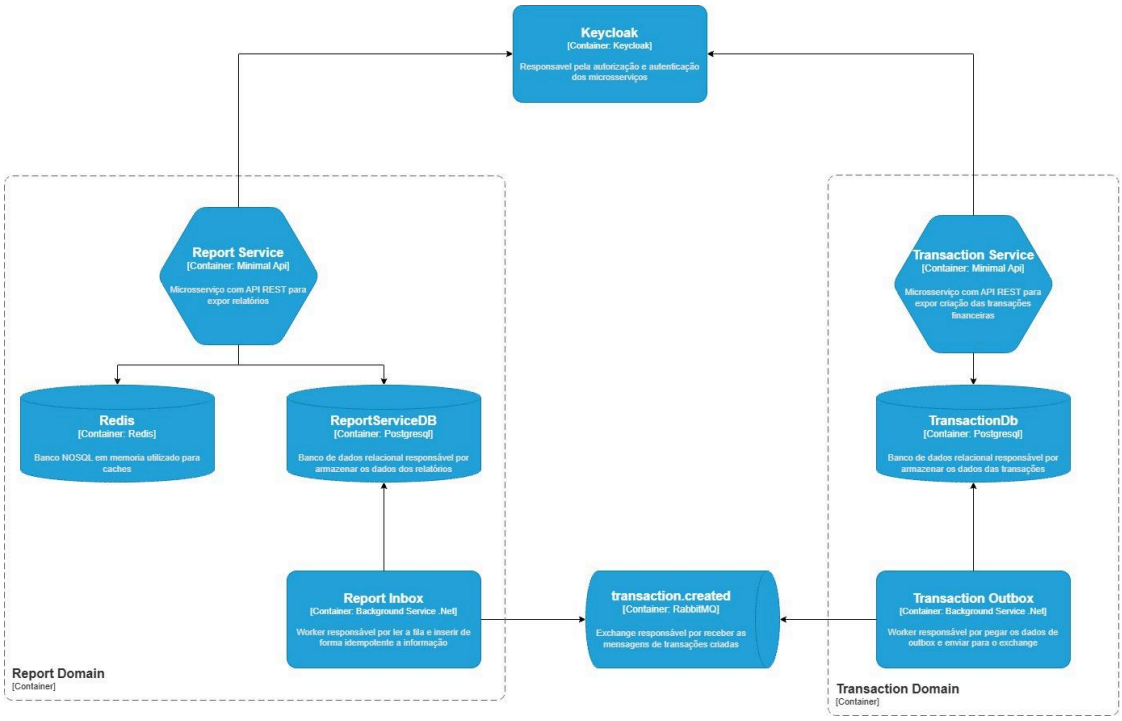
A arquitetura é baseada em microsserviços, orientados a eventos, onde se utilizam duas Minimal APIs em .NET:

- **Transaction Service**: responsável pelo controle de lançamentos financeiros (débitos e créditos).
- **Report Service**: responsável pelo cálculo e exposição do consolidado diário.

E dois background services, um para cada contexto dos microsserviços para utilização do padrão outbox:

- **Transaction Outbox**: responsável por ler a tabela de inbox no contexto da transaction e enviar para o RabbitMQ.
- **Report Inbox**: responsável por ler a fila do RabbitMQ e gravar as informações no contexto do report, levando em consideração a idempotência.

A comunicação entre os serviços é assíncrona, como já citado utilizando o RabbitMQ, garantindo desacoplamento entre os domínios.



2. Mapeamento de Domínios

2.1 Domínios Funcionais

Domínio de Transações (Transaction Domain)

- Registrar lançamentos de crédito e débito
- Garantir consistência e auditabilidade
- Publicar eventos de negócio

Domínio de Relatórios (Reporting Domain)

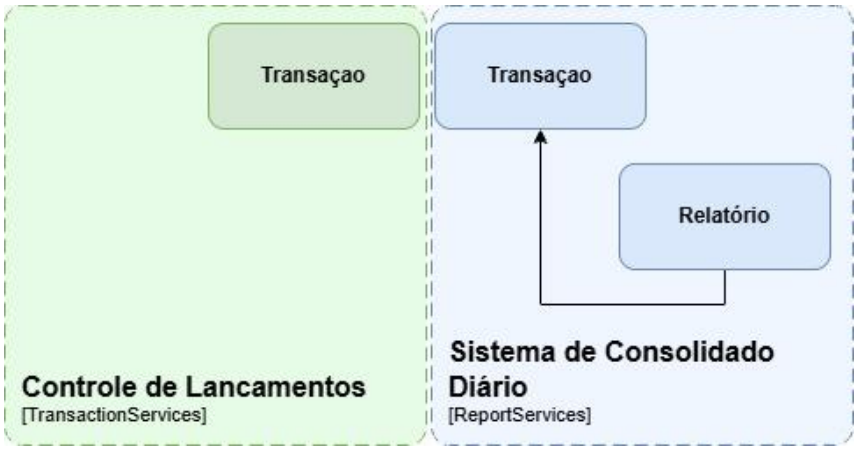
- Consolidar dados financeiros por dia
- Atender consultas de leitura com alta performance
- Expor saldo diário consolidado

O primeiro domínio funcional identificado é o Domínio de Transações. Ele engloba todas as funcionalidades relacionadas à captura, validação e persistência dos lançamentos financeiros. Esse domínio é responsável por garantir que um lançamento seja corretamente registrado como um fato imutável do negócio, obedecendo às regras de integridade e disponibilidade exigidas. Toda a lógica necessária para o registro de um lançamento pertence a esse domínio, e nenhuma funcionalidade analítica ou de agregação faz parte de sua responsabilidade.

O segundo domínio funcional é o Domínio de Relatórios. Ele concentra as funcionalidades responsáveis por consumir os lançamentos previamente registrados e transformá-los em informações agregadas por período. Esse domínio não cria fatos de negócio primários, mas trabalha exclusivamente sobre dados derivados, realizando cálculos, somatórios e agrupamentos que resultam em visões consolidadas.

2.2 Capacidades de Negócio

Capacidade	Serviço Responsável
Registro de lançamentos financeiros	Transaction Services
Publicação de eventos	Transaction Outbox
Consumo de eventos	Report Inbox
Consolidação diária	Report Services
Consulta de relatórios	Report Services



É possível identificar claramente a existência de responsabilidades de negócio distintas que exigem isolamento conceitual e operacional. O serviço de controle de lançamentos (Transaction Services) não pode ser impactado por falhas no sistema de consolidado diário (Report Services) e isso indica que essas duas capacidades não pertencem ao mesmo modelo de domínio.

O controle de lançamentos representa uma capacidade essencial do negócio, pois está diretamente relacionado à entrada e persistência dos fatos financeiros. Esse domínio não pode depender de processos analíticos ou de consolidação para executar suas funções primárias.

O sistema de consolidado diário possui uma natureza claramente distinta. Ele é responsável por processar informações já registradas, agregando dados por período e produzindo visões resumidas para análise ou tomada de decisão. Como o enunciado cita, em situações de pico, esse sistema pode operar com perda controlada de requisições. Sendo assim, consigo pensar que a consistência eventual é aceitável e que o processamento não precisa ocorrer de forma síncrona ou imediata.

A separação desses dois domínios leva naturalmente à definição de Bounded Contexts independentes. O contexto de Lançamentos possui seu próprio modelo, regras e linguagem, sendo responsável exclusivamente pela criação e manutenção dos registros financeiros. Já o contexto de Consolidação Diária mantém um modelo próprio, orientado à agregação e análise, sem interferir nas regras do domínio principal. Essa separação evita ambiguidades conceituais e reduz o acoplamento entre as partes do sistema.

3. Arquitetura Alvo

3.1 Visão de Alto Nível

- Arquitetura de Microsserviços
- Event-driven architecture com consistência eventual

- Serviços totalmente independentes em banco, deploy e escala

Componentes principais:

- Transaction Service (.NET Minimal API)
- Report Service (.NET Minimal API)
- PostgreSQL (por serviço)
- RabbitMQ
- Redis
- Keycloak
- OpenTelemetry

4. Padrões Arquiteturais Utilizados

4.1 Microsserviços

Motivação:

- Isolamento de falhas (requisito explícito do desafio)
- Escalabilidade independente
- Evolução desacoplada

4.2 Event Sourcing

Motivação:

- Rastreabilidade total dos lançamentos
- Base sólida para auditoria
- Facilidade de reconstrução de estados

4.3 Outbox Pattern (Report Outbox)

Motivação:

- Garantir entrega confiável de eventos
- Evitar dual write (banco + broker)
- Tolerância a falhas temporárias do RabbitMQ

Funcionamento:

- Eventos são persistidos no PostgreSQL
- Background Service publica eventos no RabbitMQ

4.4 Inbox Pattern + Idempotência (Transaction Inbox)

Motivação:

- Garantir processamento exatamente uma vez
- Evitar duplicidade em cenários de retry

Implementação:

- Tabela de controle de mensagens processadas
- Chave única por message_id

4.5 Cache Distribuído (Redis)

Motivação:

- Atender picos de até 50 req/s
- Reduzir carga no PostgreSQL

5. Segurança

5.1 Autenticação e Autorização

- Keycloak como Identity Provider
- OAuth 2.0 + OpenID Connect
- JWT com audiences distintas por API
- Single Sign-On (SSO)

6. Observabilidade

6.1 OpenTelemetry

Presente em **todos os 4 projetos**:

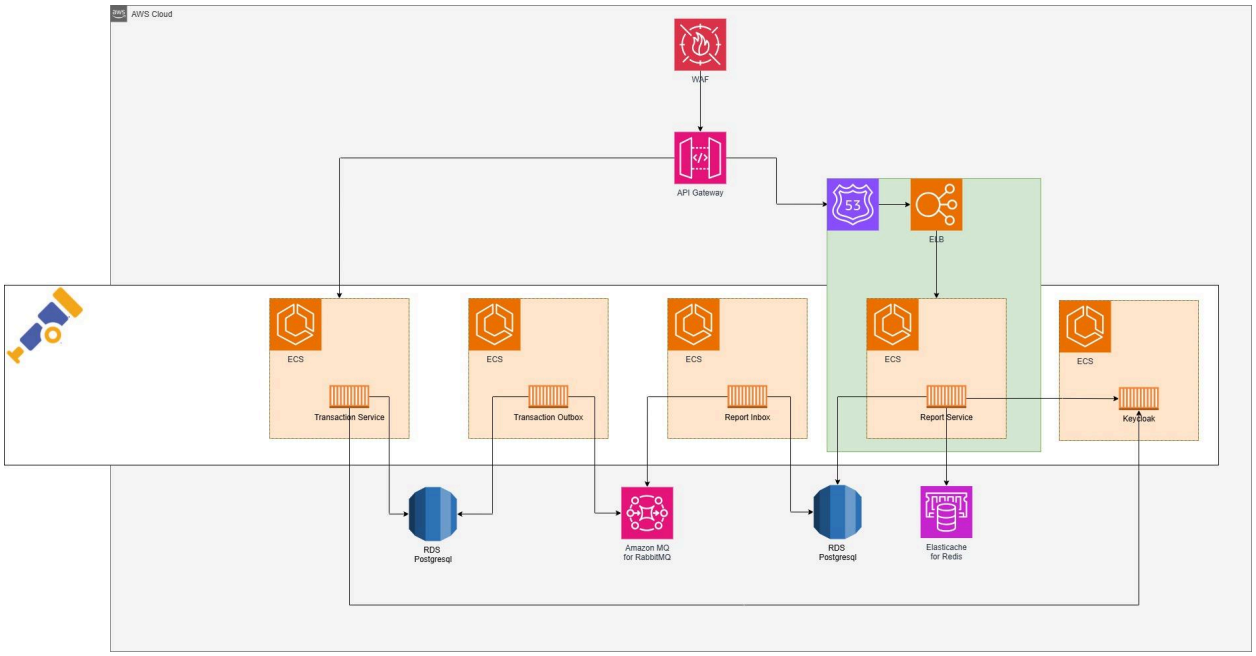
Coleta de:

- Logs estruturados
- Métricas
- Traces distribuídos

7. Infraestrutura em Nuvem (AWS)

7.1 Desenho Proposto

- **ECS**: execução dos microsserviços
- **RDS PostgreSQL**: banco por serviço
- **Amazon MQ (RabbitMQ)**
- **ElastiCache Redis**
- **ELB**: balanceamento
- **Keycloak** em EKS ou EC2
- **CloudWatch + OpenTelemetry Collector**



Temos uma Api Gateway responsável por gerenciar toda requisição para as duas APIs desenvolvidas e para garantir segurança, temos também um WAF que valida cada requisição realizada, além do Keycloak para autenticação e autorização de usuários. Foi adicionado um Load Balancer com foco no Report Service para garantir que as demandas de requisições em momentos de picos sejam atendidas com sucesso. Para garantir a performance, também foi pensado em um componente para cache de dados, utilizando o Redis. A questão da elasticidade dos containers, foi incluído o ECS, essa peça foi escolhida por ser mais simples que o EKS em questão de configuração e preço e atende o volume e complexidade da solução.

Toda comunicação entre os dois contextos é realizada por fila, nesse caso o RabbitMQ onde conseguimos desacoplar completamente os dois microsserviços. O envio de dados para o RabbitMQ é feito utilizando o padrão “outbox”, onde é feito o insert de forma atômica no postgresql e depois enviado para a fila através do Transaction Outbox. Para a ingestão dos

dados do lado do relatório, foi proposto um serviço de background no contexto do Report Inbox. A vantagem de não rodar essa ingestão de dados no mesmo processo do Report Service é que podemos ter um controle maior da elasticidade da ingestão de dados, totalmente desacoplado do serviço que suporta os relatórios da solução. Como é uma aplicação que vai rodar internamente, não senti necessidade de adicionar um CDN (Cloudfront). Como no requisito não deixa claro se o Report será acessado externo ou não, tomei a liberdade de adicionar um Route 53 principalmente para lidar com o tráfego externo em multi regiões.

Nota: Deve-se notar que o Redis e RabbitMQ no diagrama sugere que seja utilizado como recurso autogerenciado na AWS, assim como o Keycloak também seja feito deploy na própria AWS com o ECS..

7.2 Justificativa da solução

- **AWS**: a escolha da nuvem AWS foi devido a oferecer uma quantidade maior de serviços e recursos do que os concorrentes em questão.
- **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.
- **WAF**: é uma peça necessária para toda aplicação que está exposta na web e possui uma integração com o Gateway da AWS, além de possuir várias configurações para controlar e filtrar o fluxo.
- **Route 53:** é a peça opcional para esse cenário, mas que tem ganhos na sua utilização principalmente se as requisições para o Report Service vier externamente a AWS, além de possuir DNS global, alta disponibilidade e failover automático.
- **Load Balancer**: foi escolhido devido ao alto volume de acesso ao Report Service, desta forma conseguimos distribuir de forma correta as chamadas entre os containers.
- **ECS**: é a escolha mais que padrão para gerenciar containers e seus ciclos de vida quando buscamos algo com baixo custo (se comparado com o EKS) e com configurações relativamente simples.
- **Postgresql**: é um banco de dados relacional amplamente utilizado e adotado para o tipo de armazenamento simples que a solução precisa. Não achei necessário explorar opções mais caras ou NOSQL para essa questão.
- **Redis**: também é a escolha mais assertiva quando estamos falando de cache distribuído, este componente fornece todo o suporte a registros em memória que as instâncias do Report Service necessitam.
- **RDS**: possui um recurso excelente de alta disponibilidade, backup automáticos e Failover gerenciado, como esse banco sofre bastante com inserts e selects essa opção não poderia ser descartada.
- **RabbitMQ**: está como principal componente de integração entre os dois contextos. Este broker possui filas que são utilizadas para receber os eventos que acontecem no controle de lançamento através de exchanges disponibilizar as informações para o Report Service.

- **OpenTelemetry**: foi utilizado para implementar a parte de observabilidade de forma neutra, sem a dependência de nenhuma ferramenta. Nesse contexto, ela foi utilizada justamente por esse poder.
- **KeyCloak**: foi utilizado, pois essa ferramenta centraliza identidade, autenticação e autorização, removendo essa responsabilidade das aplicações.

8. Requisitos Não Funcionais

8.1 Disponibilidade

- Transaction Service não depende do Report Service, portanto se um sistema cair não afeta o outro. Além disso, conseguimos traçar estratégias de deploy e elasticidade diferentes para cada domínio.
- Comunicação assíncrona garante isolamento

8.2 Perda Máxima de Requisições

- RabbitMQ + dlq + idempotência: isso garante que nenhuma mensagem é perdida no processo.
- Perda inferior a 5% mesmo em picos

9. Arquitetura de Transição (Opcional)

Para migração de legado, seria importante ter um descritivo mais elaborado de como se encontra esse legado atualmente, como arquitetura utilizada, tecnologias, entre outros. Porém, de certa forma, eu sempre inicio o pensamento de uma arquitetura que precise lidar com o legado de duas formas:

- Padrão strangler (estrangulamento): A primeira etapa consiste em identificar individualmente os componentes monolíticos, considerando suas capacidades e limitações. Em seguida, é importante decidir qual será o primeiro componente a ser migrado como um microsserviço. Isso é um exemplo básico.
- Gradual migração para Event Sourcing: Uma abordagem válida seria alterar em pontos estratégicos de forma cirúrgica para o sistema legada emitir eventos para um broker e deixar cada contexto consumir os dados da melhor maneira.

Obviamente, pode-se utilizar outros tipos de tecnologias, como realizar verificações em alterações no banco de dados e criar eventos, rotinas ou processos ETLs. De qualquer forma, em uma abordagem sem muito conhecimento prévio, sempre tento utilizar o estrangulamento do legado ou adaptação para emitir eventos de negócio.

10. Evoluções Futuras

- DLQ para itens com problemas de processamento na fila do RabbitMQ
- Versionamento de eventos
- Data Lake para analytics
- Criação de microsserviço de auditoria

11. Conclusão

A arquitetura proposta demonstra:

- Aderência aos requisitos do desafio
- Uso consciente de padrões arquiteturais
- Forte preocupação com requisitos não funcionais

Trata-se de uma solução robusta, escalável, resiliente e preparada para evolução, alinhada às boas práticas modernas de arquitetura de microsserviços em nuvem.

ADRs – Architecture Decision Records

Todas as decisões importantes de arquitetura estão documentadas em ADR nesse documento. Cada ADR segue o formato: **Título → Contexto → Decisão → Alternativas consideradas → Consequências**

ADR-001: Adoção de Microsserviços com APIs Minimal (.NET)

Contexto:
Precisávamos isolar domínios transacionais e de relatórios para escalabilidade e confiabilidade.

Decisão:
Usar microsserviços em .NET Minimal APIs.

Alternativas Rejeitadas:

- O próprio desafio já direcionava para uma arquitetura em microsserviços, portanto não foi avaliado nenhum outro dia de arquitetura.

Consequências:
Alta autonomia e escalabilidade independente por serviço. O Minimal APIs foi utilizado devido sua performance superior e devido ao tamanho contido dos microsserviços.

ADR-002: Comunicação Assíncrona via Event-Driven

Contexto:
Transações financeiras devem propagar suas alterações sem bloqueios.

Decisão:
Foi decidido utilizar Event-Driven devido ao desacoplamento forte entre os microsserviços. Além da escalabilidade natural que ganhamos com esse tipo de arquitetura, somado com a resiliência e tolerância a falhas.

Alternativas Rejeitadas:

- Não foi sugerido outros tipos de alternativas com os requisitos obtidos

Consequências:
Garantia de desacoplamento, resiliência, escalabilidade e tolerância a falhas.

ADR-003: Outbox Pattern para Publicação Confiável

Contexto:
Dual-write entre DB e mensageria pode criar inconsistência.

Decisão:
Persistir eventos localmente e publicar via background service.

Alternativas Rejeitadas:

- Transação distribuída (2PC): geralmente mais complexo em realizar uma transação distribuída que envolve banco de dados e broker. Além disso, pode se tornar um problema em larga escala

Consequências:
Consistência eventual e tolerância a falhas com garantia de entrega. At-Least-Once o que garante confiabilidade na entrega.

ADR-004: Idempotência no Inbox

Contexto:
Como o producer dos dados utiliza o padrão Outbox e tem a característica de ser “at-least-once”, pode acontecer de ocorrer reprocessamento de eventos e isso pode causar duplicação de dados.

Decisão:
Armazenar identificador de cada evento e validar antes de processar. Nesse caso, estamos utilizando a própria base de dados do microsseviço de Report para gravar e validar, onde estamos usando um BD relacional para tornar a operação totalmente ACID.

Alternativas Rejeitadas:

- Apenas retry/ack: não garante idempotência.
- Redis: Cache em memória não é durável nem confiável; idempotência exige persistência transacional para evitar reprocessamento e inconsistências.

Consequências:
Evita duplicidade de dados.

ADR-005: Cache Distribuído com Redis (ElastiCache)

Contexto:
Relatórios são lidos frequentemente (com picos em determinados períodos) e devem responder rapidamente.

Decisão:
Usar Redis ElastiCache para cache de consultas.

Alternativas Rejeitadas:

- Cache local — inconsistência entre réplicas.
- Sem cache — alto custo de queries repetidas.

Consequências:
Melhora a performance e reduz carga no DB.

ADR-006: Observabilidade com OpenTelemetry

Contexto:
Múltiplos componentes distribuídos dificultam o rastreamento de falhas, traces e métricas.

Decisão:
Instrumentar todos os serviços com OpenTelemetry.

Alternativas Rejeitadas:

- Logs isolados sem trace → visão limitada.
- Ferramenta proprietária fixa → vendor lock-in.

Consequências:
Logs, métricas e traces forçando diagnósticos ponta-a-ponta.

ADR-007: Autenticação e Autorização com Keycloak

Contexto:
Precisávamos de uma ferramenta para controle de autenticação e autorização centralizado.

Decisão:
Usar Keycloak com JWT e OAuth2.

Alternativas Rejeitadas:

- ASP.NET Identity por serviço: não possui SSO e a implementação ficaria duplicada entre as APIs

Consequências:
SSO com chave forte e integração OIDC moderna.

ADR-008: Broker de Mensagens com RabbitMQ (Amazon MQ)

Contexto:
Precisávamos de uma ferramenta de mensageria para desacoplar a comunicação entre os microsserviços.

Decisão:
Usar RabbitMQ (Amazon MQ) para eventos orientados a mensagens. Essa ferramenta é ideal quando os eventos representam ações de negócio, além da garantia de entrega e confiabilidade

Alternativas Rejeitadas:

- Kafka: considerado complexo demais para a solução
- SNS/SQS: considerado uma solução mais simples que o RabbitMQ, com menos recursos de UI e troubleshooting.

Consequências:
Comunicação Event-Driven confiável (At-Least-Once), baixa latência para reações rápidas, roteamento flexível para múltiplos consumidores e facilidade de integração com .NET

Estimativa de Custos da Arquitetura na AWS

1. Objetivo

Este documento tem como objetivo documentar, de forma auditável e reproduzível, a estimativa de custos da arquitetura proposta, utilizando como base exclusiva a AWS Pricing Calculator e os preços públicos oficiais da AWS.

O conteúdo aqui descrito permite:

- Reproduza os valores diretamente na AWS Pricing Calculator;
- Entenda claramente as premissas arquiteturais utilizadas;
- Avalie o impacto financeiro das decisões de arquitetura para alta disponibilidade, multi-region e resiliência.

2. Premissas Gerais de Cálculo

- **Ambiente**: Produção
- **Modelo de cobrança**: On-Demand
- **Horas por mês**: 730
- **Moeda**: USD
- **Regiões utilizadas**:
    - us-east-1 (Região Primária)
    - us-west-2 (Região Secundária – somente leitura para Report API)

3. Arquitetura Considerada para o Cálculo

3.1 Serviços Single-Region

- Transaction API (ECS Fargate)
- Transaction Inbox Worker (ECS Fargate)
- Report Outbox Worker (ECS Fargate)
- Keycloak
- OpenTelemetry Collector
- Amazon MQ (RabbitMQ)
- PostgreSQL – Transaction DB

3.2 Serviços Multi-Region

- Report API (ECS Fargate)
- Application Load Balancer
- Redis (ElastiCache)
- PostgreSQL – Report DB (Read Replica cross-region)

O balanceamento global é realizado via Amazon Route 53, utilizando política de latência ou failover.

4. Inputs Utilizados na AWS Pricing Calculator

4.1 Amazon ECS – Fargate (Transaction API)

- Região: us-east-1
- Número de tasks: 2
- vCPU por task: 0.5
- Memória por task: 1 GB
- Sistema operacional: Linux
- Arquitetura: ARM64 (Graviton)
- Horas/mês: 730

4.2 Amazon ECS – Fargate (Report API – Região Primária)

- Região: us-east-1
- Número de tasks: 3
- vCPU por task: 1
- Memória por task: 2 GB
- Sistema operacional: Linux
- Arquitetura: ARM64
- Horas/mês: 730

4.3 Amazon ECS – Fargate (Report API – Região Secundária)

- Região: us-west-2
- Número de tasks: 3
- vCPU por task: 1
- Memória por task: 2 GB
- Sistema operacional: Linux
- Arquitetura: ARM64
- Horas/mês: 730

4.4 Amazon ECS – Fargate (Workers)

Report Outbox Worker

- Região: us-east-1
- Tasks: 1
- vCPU: 0.5
- Memória: 1 GB
- Horas/mês: 730

Transaction Inbox Worker

- Região: us-east-1
- Tasks: 1
- vCPU: 0.5
- Memória: 1 GB
- Horas/mês: 730

4.5 Amazon ECS – Fargate (Keycloak)

- Região: us-east-1
- Tasks: 2
- vCPU: 1
- Memória: 2 GB
- Horas/mês: 730

4.6 Amazon ECS – Fargate (OpenTelemetry Collector)

- Região: us-east-1
- Tasks: 1
- vCPU: 0.25
- Memória: 0.5 GB
- Horas/mês: 730

5. Banco de Dados – Amazon RDS PostgreSQL

5.1 Transaction Database

- Região: us-east-1
- Engine: PostgreSQL
- Instância: db.t4g.medium
- Deployment: Multi-AZ
- Armazenamento: 50 GB (gp3)
- Backup retention: 7 dias

5.2 Report Database – Primário

- Região: us-east-1
- Engine: PostgreSQL
- Instância: db.t4g.medium
- Deployment: Single-AZ
- Armazenamento: 50 GB

5.3 Read Replica Cross-Region

- Região: us-west-2
- Instância: db.t4g.medium
- Origem: Report DB (us-east-1)
- Replicação cross-region habilitada

6. Mensageria – Amazon MQ (RabbitMQ)

- Região: us-east-1
- Engine: RabbitMQ
- Tipo de instância: mq.m5.large
- Número de brokers: 3
- Deployment: Alta Disponibilidade

7. Cache Distribuído – Amazon ElastiCache (Redis)

Região Primária

- Região: us-east-1
- Engine: Redis
- Node type: cache.r6g.large
- Número de nós: 2
- Multi-AZ: habilitado

Região Secundária

- Região: us-west-2
- Engine: Redis
- Node type: cache.r6g.large
- Número de nós: 2
- Multi-AZ: habilitado

8. Balanceamento de Carga

Application Load Balancer

- Região: us-east-1
- Tipo: Application Load Balancer
- Horas/mês: 730
- LCUs: 1
- Região: us-west-2
- Tipo: Application Load Balancer
- Horas/mês: 730
- LCUs: 1

9. Balanceamento Global – Amazon Route 53

- Hosted Zones: 1
- Health Checks: 1
- Política de roteamento: Latency-based
- Volume estimado de queries: ~1 milhão/mês

10. Serviços Auxiliares

AWS Secrets Manager

- Quantidade de secrets: 5
- Volume de chamadas: baixo

Amazon CloudWatch

- Logs ingeridos: ~5 GB/mês
- Métricas customizadas: baixo volume

NAT Gateway

- Quantidade: 1
- Tráfego estimado: baixo

11. Resultado Consolidado

Com base nos inputs acima, o custo total mensal estimado da solução é de aproximadamente:

USD 2.570 / mês

Este valor reflete decisões arquiteturais voltadas para:

- Alta disponibilidade
- Isolamento de domínios
- Arquitetura orientada a eventos
- Escalabilidade horizontal
- Resiliência e tolerância a falhas