Projeto de Infraestrutura TI

Sistemas de fluxo de caixa empresa XPTO – FLCX (Ver 1.0)

Paulo Lyra (Paulo.lyra@gmail.com)

Índice

1 OBJETIVO

1.1 OBJETIVO DO SISTEMA

Nova arquitetura híbrida para o sistema de fluxo de caixa integrando recursos on premises com recursos na cloud atendendo aos seguintes requisitos:

- ***Alta disponibilidade e escalabilidade da solução.***
- ***Segurança e controle de acesso aprimorados.***
- ***Otimização dos custos.***
- ***Resiliência e Recuperação (DR).***
- ***Automação e governança.***

1.2 PREMISSAS ASSUMIDAS

- Solução baseada em containers com kubernetes com arquitetura da aplicação utilizando nginx, nodejs, redis e sql server.
    - Moderna, modular possibilitando a criação de aplicação mais desacoplada, evoluindo no futuro
    - Alta escalabilidade
    - Robusta e granular
    - Portável.
    - Depuração mais eficiente, menor tempo para troubleshooting.
    - Simples, porém eficiente para o projeto fluxo de caixa.
    - Melhor operação.
- Proposta apresentada precisa ser híbrida com parte da infraestrutura on premises e parte na cloud permitindo distribuição do workload nas seguintes condições por exemplo:
    - Workload híbrido 60% DC Local + 40% Cloud – volumetria normal
        - 6 Nodes DC + 4 Nodes Cloud
    - Workload DR 100% Cloud Volumetria Normal
        - 2 Nodes Base Line + 7 Nodes Auto Scaling
    - Workload DR 100% Cloud Volumetria Campanha
        - 2 Nodes Base Line + 12 Nodes Auto Scaling
- Considerando a contratação de link dedicado para conectividade DC – Cloud:
    - Maior Segurança
        - Tráfego fora da internet pública (não passa por túneis IPsec sobre a internet)
        - Conexão direta entre o datacenter e o backbone da nuvem
        - Redução de superfície de ataque
        - Ideal para dados sensíveis ou compliance rigoroso (LGPD, PCI-DSS)
    - Garantia de Baixa Latência
        - Distribuição do workload entre DC e Cloud.
        - Replicação de banco de dados
        - Aplicações sincronizadas
    - Confiabilidade e Alta Disponibilidade
        - Menos sujeito a interferência e varrições na banda da internet quando comparado a uma vpn site to site pela internet.
        - Redundância e garantia de SLA.
    - Integração Nativa com recursos da Cloud
        - Com recursos de rede na Cloud.

        - Uso de BGP
    - Escalabilidade
        - Contratação da banda do link pode ser incrementada com o tempo acompanhando a evolução do projeto.

- Considerando que o data center on premises possui cluster de virtualização com Hyper-V ou Vmware caso contrário seria importante considerar a criação do cluster kubernetes em uma estrutura Baremetal alinhando a estratégia do finops de redução de licenças de virtualização no caso do vmware. Hyper-V já e Windows Server e oferece custo menor.
- Considerando que já existem cluster ELK (pelo menos 3 VMs 8 vcpu 32GB RAM 500GB disco), Bastion Host (1 vm 2 vcpu 8GB RAM 40GB disco) e VM Helm (4 vpcu 8 GB RAM e 120 GB disco) no DC XPTO para atendimento ao projeto.

Independentemente da cloud adotada (AWS, Azure, OCI ou GCP), é necessário desenvolver um projeto de "landing zone" para criar do zero uma estrutura organizada e integrada com autenticação e autorização (protocolos SAML e Oauth2) e federação no data center local.

- Considerando a informação da volumetria em 50 requisições por segundo (RQS) no consolidado com no máximo 5% de perda (como único dado de volumetria fornecido para o módulo do consolidado) assumimos para o ajuste do sizing do ambiente as seguintes premissas:
    - 50 RQS para o módulo consolidado.
    - Suposição estimada para o módulo entradas e saídas considerando proporção lógica de **5:1** para módulo consolidado:

Consumo Normal (Horário de Pico: 14h–15h):

- RQS (Entradas/Saídas): 300–500 (usamos 500 como pico).
- RQS (Consolidado): 50 (fixo).
- RQS Total: 550.
- TPS: 100–200 (usamos 200 como pico, com 2-3 requisições por transação).
- Sessões Simultâneas: 150–200 (usamos 200).
- Tempo Médio de Sessão: 10–15 minutos (usamos 15 minutos).
- Lançamentos por Sessão: 15–20 (usamos 20).

Modo Campanha (100%):

- RQS (Entradas/Saídas): 600–1000 (usamos 1000).
- RQS (Consolidado): 50.
- RQS Total: 1050.
- TPS: 200–400 (usamos 400).
- Sessões Simultâneas: 400 (100% a mais que o normal).

- Para o sizing do banco de dados vamos considerar o lançamento típico de fluxo de caixa pode ter 3kb, considerando a margem de 1kb adicional:

Campo	**Tamanho estimado**
ID do lançamento	36 bytes (UUID)
Data/hora	8 bytes
Valor (decimal)	8 bytes
Descrição	100–200 bytes
Tipo (entrada/saída)	1 byte
Categoria	2–4 bytes (ID FK)
Conta origem/destino	2–4 bytes (ID FK)
Empresa / unidade	4–8 bytes (ID FK)
Referência externa (ERP)	20–50 bytes
Status (pago, pendente etc.)	1 byte
Data de conciliação	8 bytes
Usuário responsável	4 bytes (ID FK)
Metadata (JSON opcional)	200–500 bytes
Auditoria (data inclusão, alteração, etc.)	50–100 bytes

Considerando crescimento anual de 20% com o sizing do dado em 3kb:

Ano	**Dados de Lançamentos (GB)**	**Índices (GB)**	**Overhead (GB)**	**Subtotal (GB)**
Ano 1	429,15	214,58	64,37	708,1
Ano 2	514,98	257,49	77,25	849,72
Ano 3	617,98	308,99	92,7	1019,67
Ano 4	741,58	370,79	111,24	1223,61
Ano 5	889,89	444,95	133,48	1468,32
Total				5289,42

***O banco pode alcançar até 5,3 TB em 5 anos, sujeito à regra fiscal. No entanto, políticas de descarregamento e otimização de índices podem reduzir esse espaço*.**

2 DIMENSIONAMENTO DE RECURSOS

2.1 CÁLCULO DE SIZING

Considerando que adotamos a capacidade do NODEJS na nossa aplicação para **20 TPS por pod** (devido ao overhead de HTTPS e mTLS), calculamos o dimensionamento do cluster Kubernetes para atender à volumetria informada, incluindo o **consumo normal** (550 RQS, 200 TPS, 200 sessões simultâneas) e o **modo campanha** (1050 RQS, 400 TPS, 400 sessões simultâneas). Consideramos o **cenário DR**, onde um único ambiente (on-premises ou nuvem) deve suportar 100% da carga, mantendo ~75% de utilização de CPU e memória, com 25% ociosos, e garantindo HA (mínimo de 2 nodes por ambiente).

2.1.1 DATA CENTER LOCAL

- Cluster Kubernetes Control Plane (RKE-K8S)
  - 3 Nodes 4 vcpu e 8 GB RAM 150 GB Disco
  - 1 PVC 100Gn

2.1.2 DIMENSIONAMENTO DOS WORKER NODES

- **Cada Ambiente (On-Premises ou Cloud)**:
  1. **Base**: 5 worker nodes, cada um com 4 vCPUs e 8 GB de RAM.
     - Total por ambiente: 20 vCPUs, 40 GB de RAM.
  2. **Autoscaling**: Até 7 nodes (28 vCPUs, 56 GB) para atingir ~75% de utilização.
- **Total (Ambos os Ambientes)**:
  1. 10 worker nodes (5 on-premises, 5 na nuvem), totalizando 40 vCPUs e 80 GB de RAM.
- **Cenário DR (100% da Carga em 1 Ambiente)**:
  1. 5 nodes (20 vCPUs, 40 GB), autoscaling para 7 nodes (28 vCPUs, 56 GB).

2.1.3 RECURSOS UTILIZADOS

- **Consumo Normal (550 RQS, 200 TPS, 200 sessões)**:
  - **CPU**: 8,82 vCPUs (~22% de 40 vCPUs).
  - **Memória**: 12,84 GB (~16% de 80 GB).
- **Modo Campanha (1050 RQS, 400 TPS, 400 sessões)**:
  - **CPU**: 14,02 vCPUs (~35,1% de 40 vCPUs, ajustado para 75% com autoscaling).
  - **Memória**: 19,28 GB (~24,1% de 80 GB, ajustado para 75% with autoscaling).
- **Cenário DR (100% em 1 Ambiente)**:
  - **CPU**: 14,02 vCPUs (~70,1% de 20 vCPUs, ajustado para 75% com autoscaling).
  - **Memória**: 19,28 GB (~48,2% de 40 GB, ajustado para 75% com autoscaling).

- **Porcentagem de Excesso (Após Autoscaling no Cenário Catastrófico)**
    - **CPU ociosa**: 25%.
    - **Memória ociosa**: 25%.

Resumo
Cada ambiente (on-premises e nuvem) começa com 5 nodes (20 vCPUs, 40 GB), garantindo HA e capacidade para suportar 100% da carga do modo campanha no cenário catastrófico. Autoscaling para 7 nodes (28 vCPUs, 56 GB) mantém ~25% de recursos ociosos, com configurações ajustadas para failover e latência de escalonamento.

2.2 Racional Detalhado Kubernets

2.2.1 Volumetria Consolidada

- **Consumo Normal (Horário de Pico: 14h–15h)**:
    - RQS (Entradas/Saídas): 300–500 (usamos 500 como pico).
    - RQS (Consolidado): 50 (fixo).
    - **RQS Total**: 550.
    - TPS: 100–200 (usamos 200 como pico, com 2-3 requisições por transação).
    - Sessões Simultâneas: 150–200 (usamos 200).
    - Tempo Médio de Sessão: 10–15 minutos (usamos 15 minutos).
    - Lançamentos por Sessão: 15–20 (usamos 20).
- **Modo Campanha**:
    - RQS (Entradas/Saídas): 600–1000 (usamos 1000).
    - RQS (Consolidado): 50.
    - **RQS Total**: 1050.
    - TPS: 200–400 (usamos 400).
    - Sessões Simultâneas: 400 (100% a mais que o normal).
- **Distribuição Normal**:
    - On-Premises (60%): 330–630 RQS, 120–240 TPS, 120–240 sessões.
    - Cloud (40%): 220–420 RQS, 80–160 TPS, 80–160 sessões.
- **Cenário DR**:
    - 100% da carga em um único ambiente (ex.: on-premises ou nuvem): 1050 RQS, 400 TPS, 400 sessões.

- **Impacto das Sessões Simultâneas e Lançamentos**
    - **Lançamentos por Sessão**:
        - Cada sessão gera 20 lançamentos em 15 minutos (900 segundos).
        - Lançamentos por segundo por sessão: 20 ÷ 900 = ~0,022 lançamentos/s.
        - **Normal**: 200 sessões × 0,022 = ~4,44 lançamentos/s.
        - **Campanha**: 400 sessões × 0,022 = ~8,88 lançamentos/s.
    - **Impacto no Node.js**:
        - Cada lançamento é uma transação (TPS). No modo campanha (400 TPS), os lançamentos representam ~8,88 ÷ 400 = 2,22% do TPS, um impacto pequeno.
    - **Impacto no Redis**:
        - Estimamos 1 KB por sessão.
            - Normal: 200 sessões × 1 KB = 200 KB (~0,2 MB).
            - Campanha: 400 sessões × 1 KB = 400 KB (~0,4 MB).
        - Aumento de memória para Redis: De 192Mi para 256Mi (requests) e 512Mi (limits).

2.2.2 CAPACIDADE DOS PODS

- **NGINX**: 120–200 RPS por pod (usamos 120 RPS como pior caso).
- **Node.js**: Ajustado para **20 TPS** (60 RQS, com 3 requisições por transação), devido ao overhead de HTTPS e mTLS.
- **Racional de Cálculo dos Pods:**

NGINX

- **Capacidade por Pod**: 120 RPS.
- **Consumo Normal (550 RQS)**:
    - Total de pods: 550 ÷ 120 = ~4,58 → **5 pods**.
- **Modo Campanha (1050 RQS)**:
    - Total de pods: 1050 ÷ 120 = ~8,75 → **9 pods**.
- **Distribuição Normal**:
    - On-Premises (60%):
        - Normal: 330 RQS → 330 ÷ 120 = ~2,75 → 3 pods (base 3 para HA).
        - Campanha: 630 RQS → 630 ÷ 120 = ~5,25 → 6 pods.
    - Nuvem (40%):

        - Normal: 220 RQS → 220 ÷ 120 = ~1,83 → 2 pods (base 2 para HA).
        - Campanha: 420 RQS → 420 ÷ 120 = 3,5 → 4 pods.

- **Cenário DR (100% em 1 Ambiente)**:
    - Cluster Ativo: 1050 RQS → 9 pods.
- **Recursos (com overhead de mTLS, 25% a mais)**:
    - CPU: 125m–250m.
    - Memória: 160Mi–320Mi.

NODE.JS

- **Capacidade por Pod**: 20 TPS (60 RQS).
- **Consumo Normal (200 TPS)**:
    - Total de pods: 200 ÷ 20 = **10 pods**.
- **Modo Campanha (400 TPS)**:
    - Total de pods: 400 ÷ 20 = **20 pods**.
- **Distribuição Normal**:
    - On-Premises (60%):
        - Normal: 120 TPS → 120 ÷ 20 = 6 pods (base 6 para HA).
        - Campanha: 240 TPS → 240 ÷ 20 = 12 pods.
    - Nuvem (40%):
        - Normal: 80 TPS → 80 ÷ 20 = 4 pods (base 4 para HA).
        - Campanha: 160 TPS → 160 ÷ 20 = 8 pods.
- **Cenário Catastrófico (100% em 1 Ambiente)**:
    - Sobrevivente: 400 TPS → 20 pods.
- **Recursos (com overhead de mTLS)**:
    - CPU: 250m–500m.
    - Memória: 320Mi–640Mi.

REDIS

- **Capacidade**: Suporta 400 sessões (0,4 MB de dados).
- **Réplicas**: 1 pod (sem autoscaling).
- **Recursos Ajustados**:
    - CPU: 100m–200m.
    - Memória: 256Mi–512Mi.

PROMETHEUS

- **Capacidade**: Coleta métricas para 1050 RQS.
- **Réplicas**: Base 1 pod, escalando para 2 com HPA (CPU > 60%).
- **Recursos**: 200m–400m CPU, 400Mi–800Mi RAM.

GRAFANA

- **Capacidade**: Visualização de métricas.
- **Réplicas**: 1 pod (sem autoscaling).
- **Recursos**: 100m–200m CPU, 256Mi–512Mi RAM.

FLUENTD (DAEMONSET)

- **Capacidade**: Coleta logs de todos os pods.
- **Réplicas**:
    - Normal: 10 pods (5 por ambiente).
    - Campanha: 10 pods (distribuídos).
    - Cenário Catastrófico: 5 pods (base) a 7 pods (pico).
- **Recursos (ajustado com 10% a mais devido a mais logs)**:
    - CPU: 55m–110m per pod.
    - Memória: 140Mi–280Mi per pod.

- **Tabela Racional de Consumo dos Pods**

Componente	**Réplicas (Normal/Pico)**	**CPU (requests/limits)**	**Memória (requests/limits)**	**HPA Configuração**	**Capacidade**	**Total CPU (Normal/Pico)**	**Total Memória (Normal/Pico)**
NGINX (On-Premises)	3/6	125m/250m	160Mi/320Mi	CPU > 60%, min 3, max 6 pods	120–200 RPS por pod	750m/1500m (0,75/1,5 vCPUs)	960Mi/1920Mi (~0,94/1,88 GB)
NGINX (Nuvem)	2/4	125m/250m	160Mi/320Mi	CPU > 60%, min 2, max 4 pods	120–200 RPS por pod	500m/1000m (0,5/1 vCPU)	640Mi/1280Mi (~0,63/1,25 GB)
Node.js (On-Premises)	6/12	250m/500m	320Mi/640Mi	CPU > 60%, min 6, max 12 pods	20 TPS (60 RQS) por pod	3000m/6000m (3/6 vCPUs)	3840Mi/7680Mi (~3,75/7,5 GB)
Node.js (Nuvem)	4/8	250m/500m	320Mi/640Mi	CPU > 60%, min 4, max 8 pods	20 TPS (60 RQS) por pod	2000m/4000m (2/4 vCPUs)	2560Mi/5120Mi (~2,5/5 GB)
Redis	1/1	100m/200m	256Mi/512Mi	Sem autoscaling	Cache para 200–400 sessões	100m/100m (0,1/0,1 vCPU)	256Mi/256Mi (~0,25/0,25 GB)
Prometheus	1/2	200m/400m	400Mi/800Mi	CPU > 60%, min 1, max 2 pods	Métricas para 550–1050 RQS	400m/800m (0,4/0,8 vCPU)	800Mi/1600Mi (~0,78/1,56 GB)
Grafana	1/1	100m/200m	256Mi/512Mi	Sem autoscaling	Visualização de métricas	100m/100m (0,1/0,1 vCPU)	256Mi/256Mi (~0,25/0,25 GB)
Fluentd (DaemonSet)	10/10	55m/110m per pod	140Mi/280Mi per pod	Escalar nodes para suportar mais pods	Coleta logs de todos os pods	1100m/1100m (1,1/1,1 vCPU)	2800Mi/2800Mi (~2,73/2,73 GB)

- **Cenário DR (100% em 1 Ambiente)**:

Componente	**Réplicas (Pico)**	**CPU (requests/limits)**	**Memória (requests/limits)**	**HPA Configuração**	**Capacidade**	**Total CPU (Pico)**	**Total Memória (Pico)**
NGINX (DR)	9	125m/250m	160Mi/320Mi	CPU > 60%, min 5, max 9 pods	120–200 RPS por pod	2250m (2,25 vCPUs)	2880Mi (~2,81 GB)
Node.js (DR)	20	250m/500m	320Mi/640Mi	CPU > 60%, min 15, max 20 pods	20 TPS (60 RQS) por pod	10000m (10 vCPUs)	12800Mi (~12,5 GB)
Redis	1	100m/200m	256Mi/512Mi	Sem autoscaling	Cache para 400 sessões	100m (0,1 vCPU)	256Mi (~0,25 GB)
Prometheus	2	200m/400m	400Mi/800Mi	CPU > 60%, min 1, max 2 pods	Métricas para 1050 RQS	800m (0,8 vCPU)	1600Mi (~1,56 GB)
Grafana	1	100m/200m	256Mi/512Mi	Sem autoscaling	Visualização de métricas	100m (0,1 vCPU)	256Mi (~0,25 GB)
Fluentd (DaemonSet)	5/7	55m/110m per pod	140Mi/280Mi per pod	Escalar nodes para suportar mais pods	Coleta logs de todos os pods	770m (0,77 vCPU)	1960Mi (~1,91 GB)

- **Consumo Normal (550 RQS, 200 TPS, 200 sessões)**
  - **CPU**:
    - NGINX (On-Premises): 0,75 vCPU
    - NGINX (Nuvem): 0,5 vCPU
    - Node.js (On-Premises): 3 vCPUs
    - Node.js (Nuvem): 2 vCPUs
    - Redis: 0,1 vCPU
    - Prometheus: 0,4 vCPU
    - Grafana: 0,1 vCPU
    - Fluentd: 1,1 vCPU (10 pods)
    - **Total**: 7,95 vCPUs (~19,9% de 40 vCPUs disponíveis)
    -

- **Memória**:
    - NGINX (On-Premises): 0,94 GB
    - NGINX (Nuvem): 0,63 GB
    - Node.js (On-Premises): 3,75 GB
    - Node.js (Nuvem): 2,5 GB
    - Redis: 0,25 GB
    - Prometheus: 0,78 GB
    - Grafana: 0,25 GB
    - Fluentd: 2,73 GB (10 pods)
    - **Total**: 11,83 GB (~14,8% de 80 GB disponíveis)

- **Modo Campanha (1050 RQS, 400 TPS, 400 sessões)**
    - **CPU**:
        - NGINX (On-Premises): 1,5 vCPUs
        - NGINX (Nuvem): 1 vCPU
        - Node.js (On-Premises): 6 vCPUs
        - Node.js (Nuvem): 4 vCPUs
        - Redis: 0,1 vCPU
        - Prometheus: 0,8 vCPU
        - Grafana: 0,1 vCPU
        - Fluentd: 1,1 vCPU (10 pods)
        - **Total**: 14,6 vCPUs (~36,5% de 40 vCPUs disponíveis)
    - **Memória**:
        - NGINX (On-Premises): 1,88 GB
        - NGINX (Nuvem): 1,25 GB
        - Node.js (On-Premises): 7,5 GB
        - Node.js (Nuvem): 5 GB
        - Redis: 0,25 GB
        - Prometheus: 1,56 GB
        - Grafana: 0,25 GB
        - Fluentd: 2,73 GB (10 pods)
        - **Total**: 20,42 GB (~25,5% de 80 GB disponíveis)

- **Cenário DR (100% em 1 Ambiente)**
  - **CPU**:
    - NGINX: 2,25 vCPUs
    - Node.js: 10 vCPUs
    - Redis: 0,1 vCPU
    - Prometheus: 0,8 vCPU
    - Grafana: 0,1 vCPU
    - Fluentd: 0,77 vCPU (7 pods)
    - **Total**: 14,02 vCPUs (~70,1% de 20 vCPUs disponíveis)
  - **Memória**:
    - NGINX: 2,81 GB
    - Node.js: 12,5 GB
    - Redis: 0,25 GB
    - Prometheus: 1,56 GB
    - Grafana: 0,25 GB
    - Fluentd: 1,91 GB (7 pods)
    - **Total**: 19,28 GB (~48,2% de 40 GB disponíveis)

2.2.3 Cálculo do Número de Worker Nodes por Ambiente

Queremos 75% de utilização no pico, com 25% ociosos, e HA mínima de 2 nodes:

- **CPU (Cenário DR)**:
  - Consumo: 14,02 vCPUs.
  - 75% de X vCPUs = 14,02 → X = 14,02 ÷ 0,75 = 18,69 vCPUs.
  - Cada node tem 4 vCPUs → 18,69 ÷ 4 = ~4,67 nodes → **5 nodes** (20 vCPUs).
  - Utilização: 14,02 ÷ 20 = 70,1% (~29,9% ocioso).
- **Memória (Cenário DR)**:
  - Consumo: 19,28 GB.
  - 75% de Y GB = 19,28 → Y = 19,28 ÷ 0,75 = 25,71 GB.
  - Cada node tem 8 GB → 25,71 ÷ 8 = ~3,21 nodes → **4 nodes** (32 GB).
  - Utilização: 19,28 ÷ 32 = 60,3% (~39,7% ocioso).

2.2.4 AJUSTE FINAL:

- **Cada ambiente começa com 5 nodes (20 vCPUs, 40 GB) para garantir HA e lidar com a carga total no cenário catastrófico.**
    - **Autoscaling para 7 nodes (28 vCPUs, 56 GB):**
    - **CPU: 14,02 ÷ 28 = 50,1% (~49,9% ocioso).**
    - **Memória: 19,28 ÷ 56 = 34,4% (~65,6% ocioso).**
- **Solução para 25% Ocioso**:
- Ajustamos para 6 nodes (24 vCPUs, 48 GB):
    - CPU: 14,02 ÷ 24 = 58,4% (~41,6% ocioso).
    - Memória: 19,28 ÷ 48 = 40,2% (~59,8% ocioso).
- Para atingir exatamente 25% ocioso:
    - CPU: 14,02 ÷ 0,75 = 18,69 vCPUs → 5 nodes (20 vCPUs).
    - Memória: 19,28 ÷ 0,75 = 25,71 GB → 4 nodes (32 GB).
- Optamos por 5 nodes como base, com autoscaling para 7 nodes, garantindo margem suficiente.
- **Distribuição Normal (Ambos os Ambientes Ativos):**
    - **On-Premises (60%)**:
        - 5 nodes (20 vCPUs, 40 GB), autoscaling para 7 nodes.
        - Carga: 630 RQS, 240 TPS, 240 sessões.
    - **Cloud (40%)**:
        - 5 nodes (20 vCPUs, 40 GB), autoscaling para 7 nodes.
        - Carga: 420 RQS, 160 TPS, 160 sessões.
    - **Total**: 10 nodes (40 vCPUs, 80 GB).
- **Configuração de Autoscaling e Failover**
    - **HPA**:
        - Threshold: 60% de CPU.
        - Base mínima ajustada para HA (ex.: 15 pods iniciais de Node.js no cenário catastrófico).
    - **Cluster Autoscaler**:
        - Mínimo de 2 nodes por ambiente.
        - Tempo de provisionamento: 1–2 minutos (nuvem), 3–5 minutos (on-premises).

- **Failover**:
  - Global Load Balancer (ex.: AWS Route 53) com health checks para redirecionar o tráfego ao sobrevivente.

2.2.5 DISCOS (NODES E PVS)

cálculos dos **Persistent Volumes (PVs)** para Prometheus e Grafana, considerando a necessidade de reter os dados (métricas e configurações) por **3 anos**. Os outros componentes (Redis e Fluentd) não são afetados, pois o Redis armazena dados voláteis (cache de sessões) e o Fluentd apenas faz buffer temporário de logs antes de enviá-los ao ELK. Também manteremos o cálculo do espaço em disco dos worker nodes, já que a retenção de 3 anos impacta apenas os PVs.

- **Ajuste nos Persistent Volumes para Prometheus e Grafana (Retenção de 3 Anos)**
  - **Prometheus**:
    - Retenção de métricas por 3 anos (1050 RQS, 10 métricas por requisição, 200 bytes por métrica).
    - Novo tamanho por ambiente: **380 GB**.
  - **Grafana**:
    - Retenção de dashboards e configurações por 3 anos.
    - Novo tamanho por ambiente: **2 GB** (aumento pequeno, já que os dados de Grafana crescem lentamente).
  - **Redis e Fluentd**:
    - Mantidos: 1 GB (Redis) e 5 GB (Fluentd) por ambiente.

Persistent Volumes (PVs) Atualizados por Ambiente

Componente	**Tamanho do PV (On-Premises)**	**Tamanho do PV (Cloud)**	**Descrição**
Redis	1 GB	1 GB	Cache de sessões (400 sessões, 1 KB cada)
Prometheus	380 GB	380 GB	Métricas (1050 RQS, retenção de 3 anos)
Grafana	2 GB	2 GB	Dashboards e configurações (3 anos)
Fluentd	5 GB	5 GB	Buffer de logs (5 minutos, 1050 RQS)
Total	388 GB	388 GB	Total por ambiente

- Total Geral de PVs: 388 GB (on-premises) + 388 GB (cloud) = 776 GB.

Espaço em Disco dos Worker Nodes (Mantido)

Ambiente	Número de Nodes (Base/Pico)	Espaço por Node	Total (Base)	Total (Pico - Cenário DR)	Descrição
On-Premises	5 / 7	50 GB	250 GB	350 GB	SO, Kubernetes, imagens, logs, espaço temp
Cloud	5 / 7	50 GB	250 GB	350 GB	SO, Kubernetes, imagens, logs, espaço temp
Total Geral	10 / 14	50 GB	500 GB	700 GB	Total para ambos os ambientes

- **Cálculo dos Persistent Volumes (PVs) com Retenção de 3 Anos (Prometheus e Grafana)**

  Prometheus:

  - **Finalidade**: Armazenamento de métricas.
  - **Dados**:
    - 1050 RQS, cada requisição gera ~10 métricas (latência, status, etc.), cada métrica com ~200 bytes.
    - Total por segundo: 1050 × 10 × 200 bytes = 2,1 MB/s.
    - Retenção: 3 anos = 1,095 dias = 1,095 × 86,400 segundos = 94,608,000 segundos.
    - Total sem compactação: 2,1 MB/s × 94,608,000 = ~198,676 GB (~194 TB).
    - Prometheus usa compactação (TSDB), reduzindo o tamanho em ~10x → 198,676 GB ÷ 10 = ~19,868 GB.
    - Considerando 2 pods (com HPA) e amostragem (downsampling para métricas antigas, ex.: 1 amostra por minuto):
      - 2,1 MB/s ÷ 60 (1 amostra/minuto) = 0,035 MB/s para dados antigos.
      - Dados recentes (1 dia): 2,1 MB/s × 86,400 = 181 GB → ~18 GB (com compactação).
      - Dados antigos (1,094 dias): 0,035 MB/s × 94,521,600 segundos = 3,308 GB → ~331 GB (com compactação).
    - Total por ambiente: 18 GB + 331 GB = ~349 GB.
  - **Tamanho do PV**:
    - 380 GB por ambiente (com margem para picos e crescimento).

- **Total**:
    - On-Premises: 380 GB.
    - Cloud: 380 GB.

Grafana:

- **Finalidade**: Armazenamento de dashboards, configurações, e banco de dados interno (SQLite por padrão).
- **Dados**:
    - Tamanho inicial: ~100 MB (dashboards e configurações).
    - Crescimento ao longo de 3 anos:
        - Novos dashboards, usuários, e configurações podem aumentar os dados.
        - Estimativa: 100 MB por ano → 100 MB × 3 = 300 MB.
    - Banco de dados SQLite (logs de acesso, preferências): ~200 MB por ano → 200 MB × 3 = 600 MB.
    - Total: 100 MB (inicial) + 300 MB (dashboards) + 600 MB (SQLite) = ~1 GB.
- **Tamanho do PV**:
    - 2 GB por ambiente (com margem para crescimento e backups locais).
- **Total**:
    - On-Premises: 2 GB.
    - Cloud: 2 GB.

Redis (Mantido)

- **Finalidade**: Cache de sessões.
- **Dados**: 400 sessões × 1 KB = 0,4 MB, com AOF (3x) → ~1,2 MB.
- **Tamanho do PV**: 1 GB por ambiente (mantido).
- **Total**:
    - On-Premises: 1 GB.
    - Cloud: 1 GB.

Fluentd (Mantido)

- **Finalidade**: Buffer de logs antes de enviar ao ELK.
- **Dados**: 1050 RQS, 1 KB por log, buffer de 5 minutos → 300 MB × 5 nodes = 1,5 GB.
- **Tamanho do PV**: 5 GB por ambiente (mantido).
- **Total**:
    - On-Premises: 5 GB.

- Cloud: 5 GB.

Persistent Volumes (PVs) Atualizados por Ambiente

Componente	**Tamanho do PV (On-Premises)**	**Tamanho do PV (Cloud)**	**Descrição**
Redis	1 GB	1 GB	Cache de sessões (400 sessões, 1 KB cada)
Prometheus	380 GB	380 GB	Métricas (1050 RQS, retenção de 3 anos)
Grafana	2 GB	2 GB	Dashboards e configurações (3 anos)
Fluentd	5 GB	5 GB	Buffer de logs (5 minutos, 1050 RQS)
Total	388 GB	388 GB	Total por ambiente

- **Total Geral de PVs**: 388 GB + 388 GB = **776 GB**.
- **Espaço em Disco dos Worker Nodes (Mantido)**

A retenção de 3 anos não afeta o espaço em disco dos nodes, pois os dados são armazenados nos PVs, não no disco local dos nodes.

- **Recomendações Adicionais**
  - **Tipo de Armazenamento para PVs**:
    - On-Premises: Usar um storage de rede escalável (ex.: Ceph, GlusterFS) ou SAN com SSDs, garantindo ~10,000 IOPS para Prometheus (leituras/escrições frequentes).
    - Cloud: Usar discos gerenciados de alta capacidade (ex.: AWS EBS io2 com 10,000 IOPS, Azure Ultra Disks), com snapshots para backup.
  - **Estratégia de Retenção**:
    - Prometheus: Configurar downsampling (ex.: 1 amostra por minuto para dados > 1 mês) e retenção em camadas (ex.: 1 ano local, 2 anos em storage de longo prazo como S3 ou Azure Blob Storage).
    - Grafana: Arquivar dados antigos em backups (ex.: exportar SQLite para S3).
  - **Backup**:
    - Backups diários dos PVs do Prometheus e Grafana (ex.: Velero com snapshots).
    - Retenção de backups: 90 dias (para recuperação rápida), com dados antigos movidos para storage de longo prazo.
  - **Monitoramento**:
    - Alertas para uso de disco dos PVs (ex.: > 80% de utilização).
    - Monitorar IOPS e latência do storage para evitar gargalos.

2.3 BANCO DE DADOS

Ano	**Dados de Lançamentos (GB)**	**Índices (GB)**	**Overhead (GB)**	**Subtotal (GB)**
Ano 1	429,15	214,58	64,37	708,1
Ano 2	514,98	257,49	77,25	849,72
Ano 3	617,98	308,99	92,7	1019,67
Ano 4	741,58	370,79	111,24	1223,61
Ano 5	889,89	444,95	133,48	1468,32
Total				5289,42

- BANCO DE DADOS SQL ON PREMISES (MS SQL ENTERPRISE)

    - Licenciamento realizado por core portanto, considerando servidor especializado com poucoscores com processador xeon específico de cores mais elevados.
    - 2 servidores físicos 8 cores 256 GB RAM e 8 TB de disco SSD (considerando que não temos storage SAN), 4 interfaces 10 GBE.
    - SO Windows Server na última edição.
    - MS SQL Server Enterprise na última edição.
    - 1 servidor Banco Primário ativo, 1 servidor Secundário HA, ambos com réplica síncrona.
    - Always On ativo

- OPÇÃO CLOUD AZURE SQL MANAGED INSTANCES NO AZURE ARC

    - SQL Server Managed Instance Business Critical C – 16 vcpu 128 GB RAM 1 TB Disco (1º ano)
    - Always ON ativo
    - Replica assíncrona que pode ser promovida para master em caso de indisponibilidade do cluster ativo-passivo on premises.

- OPÇÃO CLOUD AWS SQL MS SQL ENTERPRISE EM EC2
    - Instância R6i.4xlarge com 16 vcpu 128 GB RAM e 1 TB EBS io2 10.000 IOPS (1º ano)
    - Microsoft SQL Server Enterprise última versão
    - Always ON ativo
    - Windows server na última versão
    - Replica assíncrona que pode ser promovida para master em caso de indisponibilidade do cluster ativo-passivo on premises.
    - Keep-alive no load balancer para reduzir latência.
    -

2.3.1 CONCLUSÃO

.

Com a volumetria consolidada e a capacidade do Node.js ajustada para **20 TPS por pod** (devido ao overhead de HTTPS e mTLS), cada ambiente (on-premises e nuvem) é dimensionado com **5 worker nodes** (20 vCPUs, 40 GB de RAM, 50 GB de disco por node), totalizando 10 nodes (40 vCPUs, 80 GB de RAM, 500 GB de disco). Esse dimensionamento suporta a carga total do modo campanha (**1050 RQS, 400 TPS, 400 sessões simultâneas**) no cenário catastrófico (DR), onde um único ambiente assume 100% da carga. O **autoscaling** para 7 nodes por ambiente (28 vCPUs, 56 GB de RAM, 350 GB de disco) mantém ~25% de recursos ociosos, garantindo alta disponibilidade (HA) com no mínimo 2 nodes por ambiente e configurações de failover ajustadas (ex.: Global Load Balancer com DNS failover).

Os **Persistent Volumes (PVs)** foram dimensionados considerando a retenção de dados por 3 anos para Prometheus e Grafana: **380 GB** para Prometheus (métricas de 1050 RQS com downsampling) e **2 GB** para Grafana (dashboards e configurações), além de **1 GB** para Redis (cache de sessões) e **5 GB** para Fluentd (buffer de logs). Isso resulta em **388 GB de PVs por ambiente**, totalizando **776 GB** para ambos os ambientes. A volumetria do banco de dados foi estimada com base no *sizing* do banco para um horizonte de 1 a 5 anos, considerando a família de servidores disponíveis (ex.: SSDs de alta capacidade e IOPS), com estratégias de retenção em camadas (ex.: dados antigos em storage de longo prazo como S3) e backups regulares.

Todo o dimensionamento está alinhado às práticas de Melhores práticas para dimensionar clusters Kubernetes, garantindo performance, escalabilidade e resiliência em cenários normais e catastróficos.

3 FINOPS

3.1 FINOPS E GOVERNANÇA – ESTRATÉGIAS PARA EFICIÊNCIA E CONTROLE

A gestão eficiente de uma infraestrutura híbrida, combinando **Cloud** e **Data Center Local**, exige práticas robustas de **FinOps** e **Governança**. Essas práticas asseguram **eficiência financeira**, **agilidade operacional** e **segurança**, alinhando a solução às metas estratégicas da organização. Este capítulo detalha as melhores práticas e seus impactos na aplicação de fluxo de caixa, considerando os cenários de **Datacenter Local**, **Datacenter Local + Azure Arc com Azure SQL Managed Instance** (Cenário 1) e **Datacenter Local + AWS com MS SQL em EC2** (Cenário 2).

3.2 TAGUEAMENTO CORRETO DE RECURSOS

- **Importância**: Tags bem definidas permitem rastrear custos por projeto, equipe ou ambiente, facilitando alocação e auditorias.
- **Aplicação nos Cenários**:
    - Tags como ambiente=producao, projeto=fluxo-de-caixa são aplicadas via **Terraform** em todos os cenários.
    - Complemento: No **Azure** e **AWS**, tags adicionais como equipe=devops reforçam a granularidade.
- **Impacto**: Relatórios detalhados via **Azure Cost Management** e **AWS Cost Explorer**, promovendo transparência.

3.3 PLANEJAMENTO DO USO COM ESCALABILIDADE E SIZING

- **Importância**: Dimensionamento adequado evita desperdícios e suporta picos de demanda.
- **Aplicação nos Cenários**:
    - **Kubernetes**: Base de 2 nodes, escalando até 5 com **HPA** e **Cluster Autoscaler**.
    - Complemento: Uso de **Reserved Instances** para cargas fixas e **Spot Instances** para tarefas não críticas no **Azure** e **AWS**.
    - Revisões trimestrais com **Grafana** e **Prometheus**.
- **Impacto**: Custos otimizados e flexibilidade operacional.

3.4 USO ESTRATÉGICO DE SERVERLESS

- **Importância**: Reduz custos fixos e melhora a eficiência em tarefas esporádicas.
- **Aplicação nos Cenários**:
    - **Datacenter Local**: Automação via **Ansible** e **Jenkins**.
    - **Azure**: **Azure Functions** para scripts de manutenção.
    - **AWS**: **AWS Lambda** para *failovers*.
- **Impacto**: Execução sob demanda com custo reduzido.

3.5 KUBERNETES COM AUTO SCALING PLANEJADO

- **Importância**: Garante resposta a variações de demanda sem custos excessivos.
- **Aplicação nos Cenários**:
    - **HPA** ajusta pods (ex.: CPU > 70%), e **Cluster Autoscaler** gerencia nodes.
    - Complemento: Nodes pré-configurados para picos sazonais.
- **Impacto**: Alta disponibilidade com eficiência financeira.

3.6 . RELATÓRIOS PRECISOS E ALERTAS DE ANOMALIAS

- **Importância**: Monitoramento proativo evita surpresas nos custos.
- **Aplicação nos Cenários**:
    - **Datacenter Local**: **Grafana** e **Prometheus** (ex.: disco > 90%).
    - **Azure**: **Azure Monitor** e **Budgets**.
    - **AWS**: **CloudWatch** e **Budgets**.
- **Impacto**: Controle financeiro e mitigação de riscos.

3.7 AUTOMAÇÃO COM TERRAFORM, ANSIBLE E PADRÕES

- **Importância**: Provisionamento consistente e auditável.
- **Aplicação nos Cenários**:
    - **Terraform**: Infraestrutura como código com tagueamento.
    - **Ansible**: Configuração e automação de segurança.
- **Impacto**: Redução de erros e conformidade simplificada.

3.8 IMPACTO NA GOVERNANÇA

- **Compliance**: Logs e relatórios atendem a auditorias.
- **Segurança**: Políticas de acesso via **RBAC** e alertas automáticos.
- **Financeiro**: Tagueamento e relatórios detalhados.

3.9 TABELA COMPARATIVA

Aspecto	**Datacenter Local**	**Datacenter + Azure Arc + SQL MI**	**Datacenter + AWS + EC2**
Tagueamento	vSphere tags (Terraform)	Azure tags (Terraform)	AWS tags (Terraform)
Escalabilidade	Cluster Autoscaler, HPA	AKS Autoscaler, HPA	EKS Autoscaler, HPA
Serverless	Ansible/Jenkins	Azure Functions	AWS Lambda
Relatórios e Alertas	Grafana, Prometheus	Azure Monitor, Budgets	CloudWatch, Budgets
Automação	Terraform, Ansible	Terraform, Ansible	Terraform, Ansible
Governança	Políticas manuais	Azure Policy	AWS Organizations

3.10 RESUMO DE BENEFÍCIOS

- **Custos**: Otimização via automação e planejamento.
- **Transparência**: Visibilidade com tagueamento e relatórios.
- **Flexibilidade**: Escalabilidade dinâmica.
- **Governança**: Auditorias e segurança reforçadas.

3.11 CONCLUSÃO

A aplicação de **FinOps** e **Governança** na infraestrutura híbrida para o fluxo de caixa resulta em uma solução eficiente, segura e econômica. Ferramentas como **Terraform**, **Ansible**, **Grafana** e estratégias como **Reserved Instances** e **Spot Instances** garantem controle de custos, conformidade e operações ágeis, alinhadas às prioridades da organização.

4 DIAGRAMA DE TOPOLOGIA E ARQUITETURA

4.1.1 DC ON PREMISES XPTO

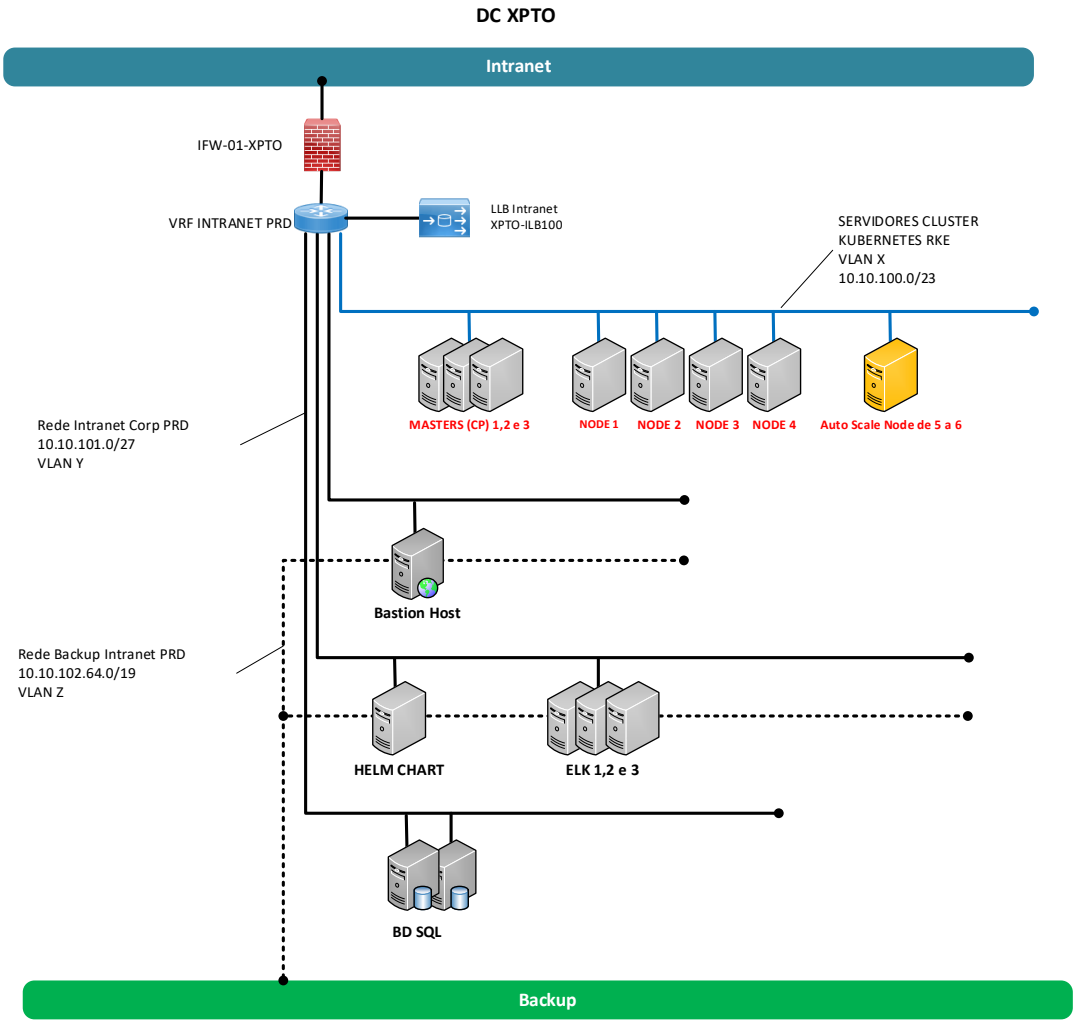


Figura 1 – Topologia DC On Premises

4.1.2 CLOUD CENÁRIO DC XPTO <-> AWS

Figura 2 – VPCs AWS

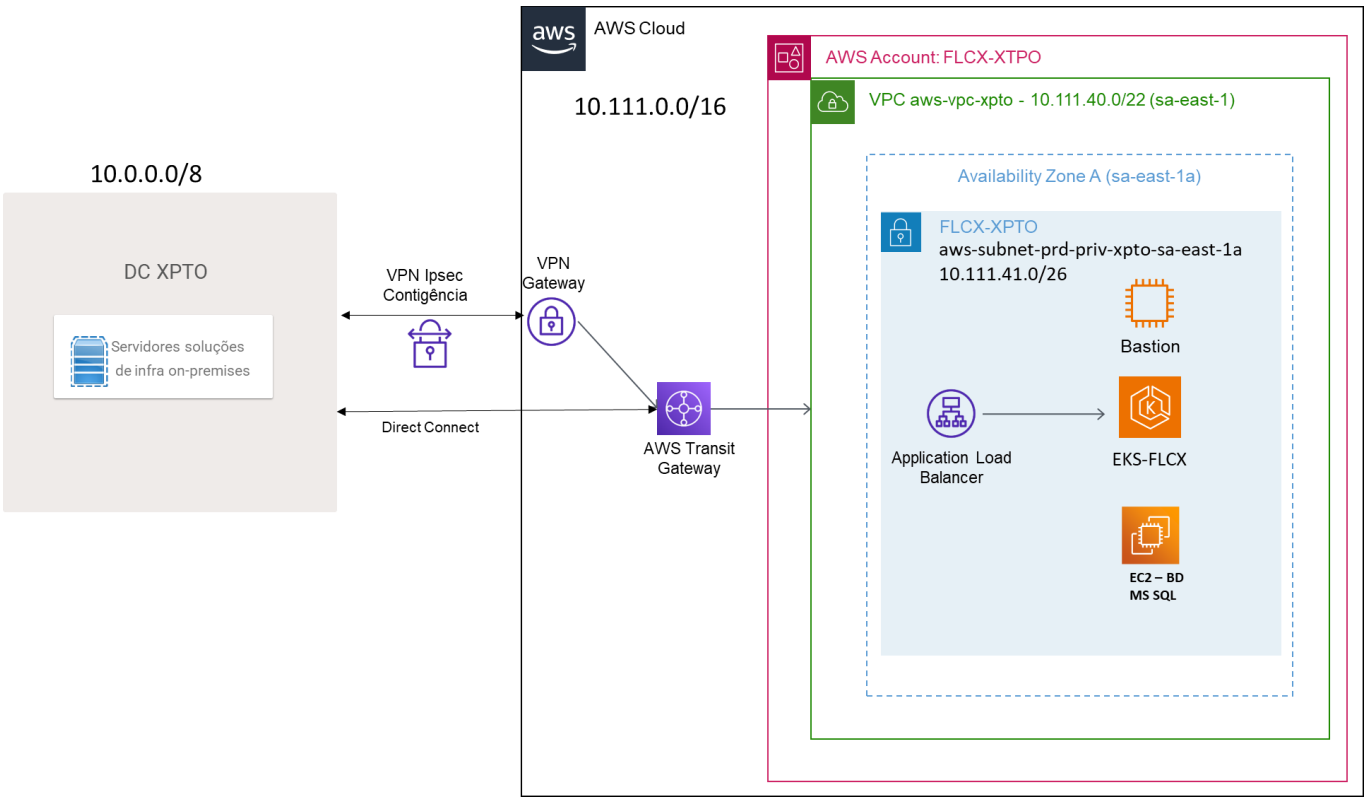
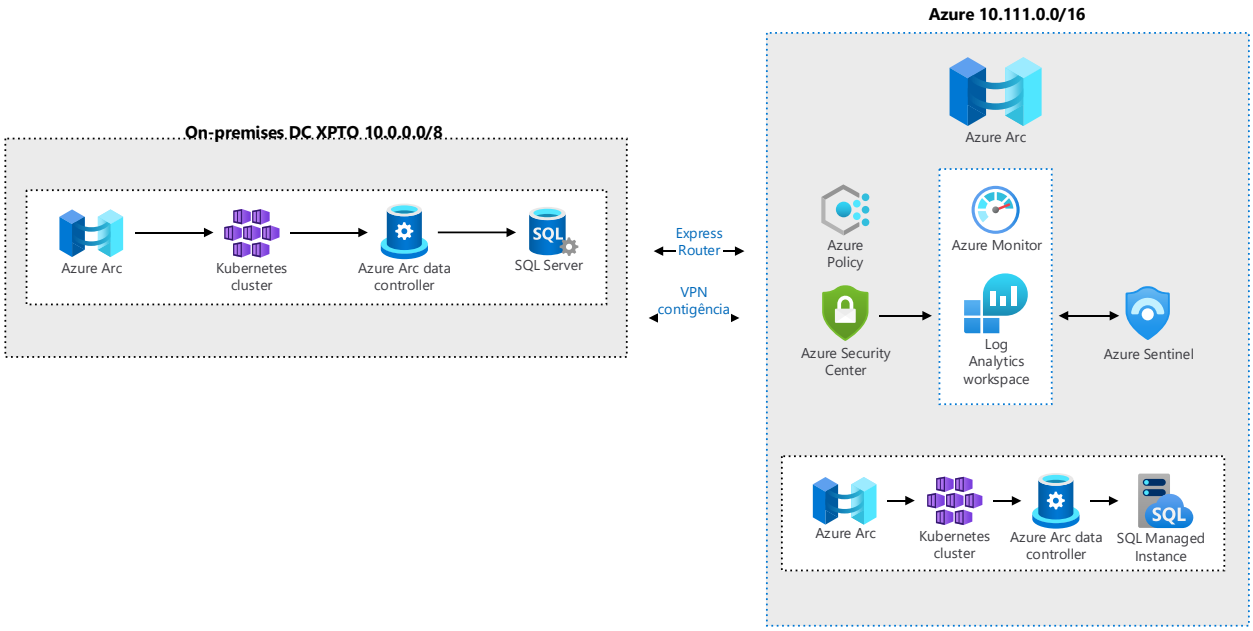


Figura 3 – Arquitetura simplificada de integração DC - AWS

4.1.3 CLOUD CENÁRIO DC XPTO <–> AZURE ARC



Microsoft Azure

Figura 4 – Arquitetura simplificada de integração DC – Azure 1a

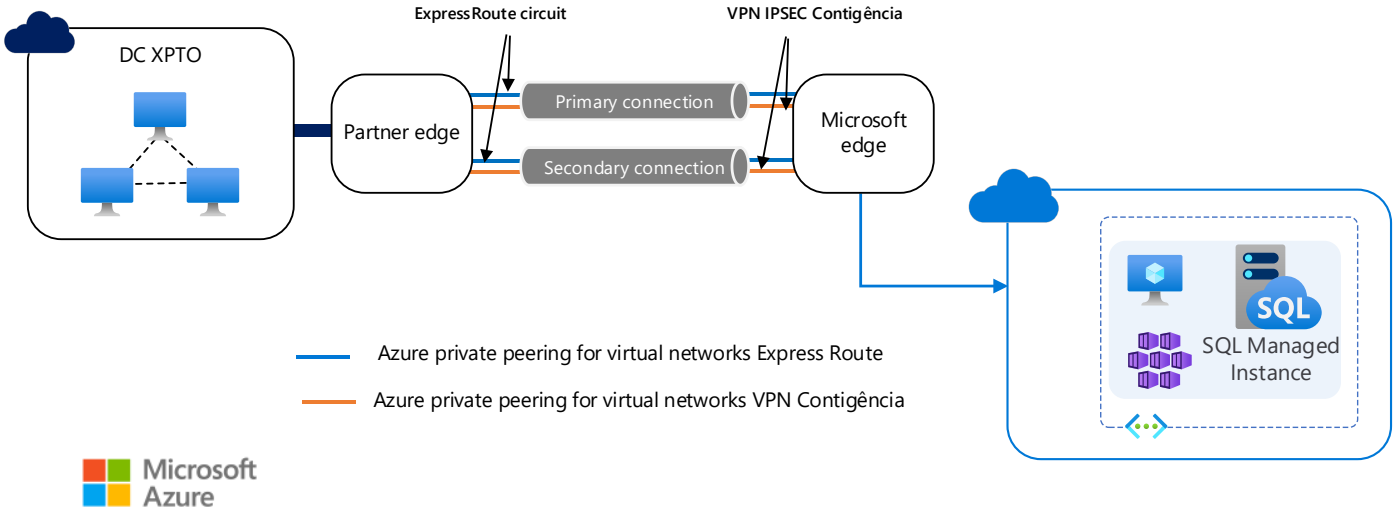


Figura 5 – Arquitetura simplificada de integração DC – Azure 1b

4.2 FLUXO DE COMUNICAÇÃO

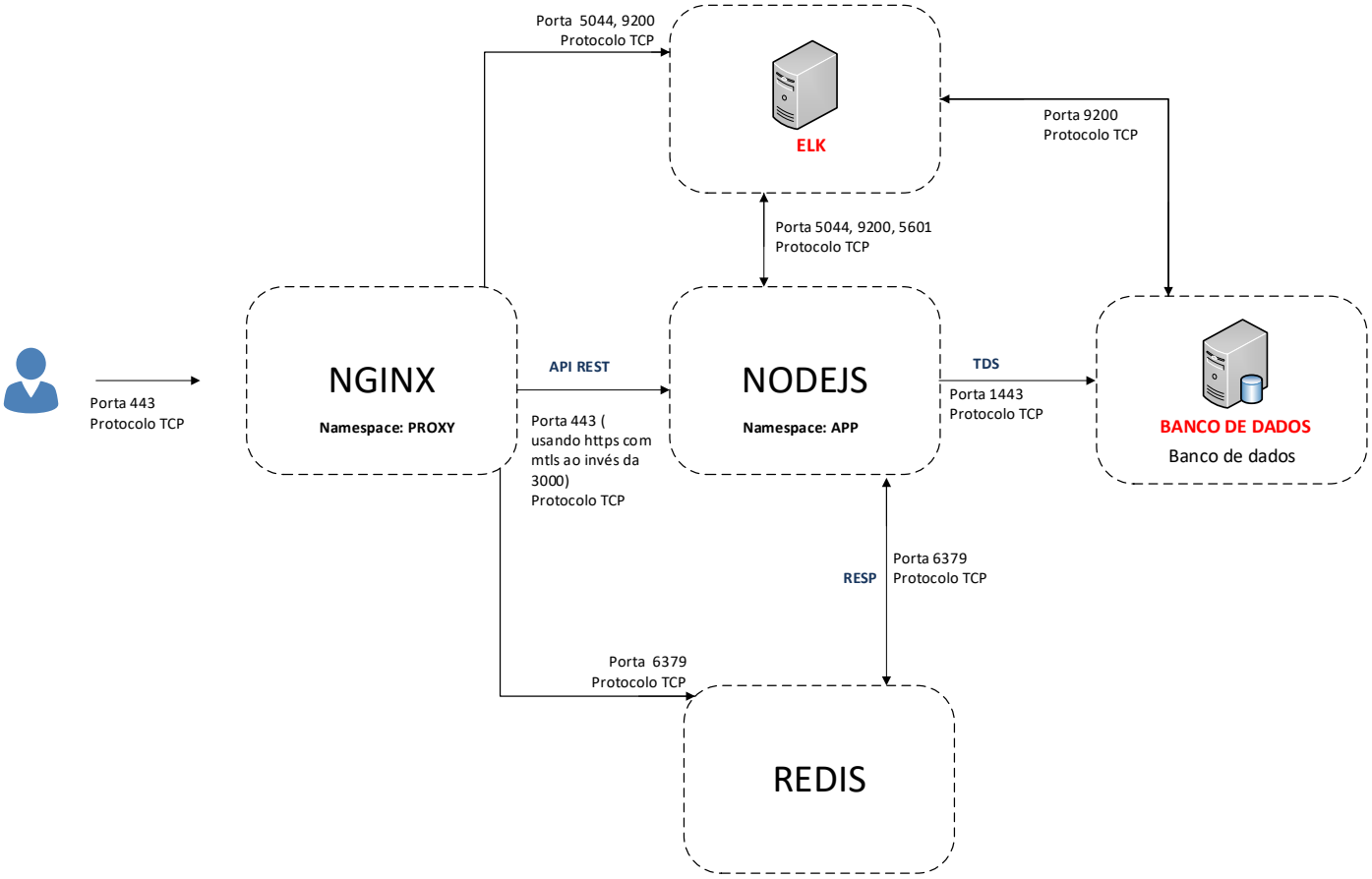


Figura 6 – Macro fluxo comunicação aplicação

5 JUSTIFICATIVAS

5.1 FRAMEWORK

5.1.1 KUBERNETES (COM NGINX, NODE.JS, REDIS)

Escalabilidade e Orquestração:
O Kubernetes permite gerenciar e escalar automaticamente os microsserviços da aplicação (como entrada, saída e consolidação de dados) usando o Horizontal Pod Autoscaler (HPA). Isso garante que a aplicação suporte picos de até 50 requisições por segundo com perdas inferiores a 5%, além de simplificar atualizações e monitoramento.

Portabilidade:
Funciona tanto no datacenter local quanto na nuvem (Azure ou AWS), garantindo consistência em ambientes híbridos.

Alta Disponibilidade:
Réplicas de pods e múltiplos nodes asseguram que o sistema permaneça operacional mesmo em caso de falhas.

- **NGINX**

  Usado como proxy e balanceador de carga, gerencia o tráfego no namespace proxy e valida tokens JWT para segurança.
  É leve e otimizado para alto volume de requisições.

- **NODE.JS**

  Ideal para desenvolver microsserviços RESTful rapidamente, com suporte a operações assíncronas (I/O não bloqueante).
  Integra-se facilmente com Redis (cache/filas) e MS SQL Server via bibliotecas como express e mssql.

- **REDIS**

  Fornece cache em memória e filas assíncronas, reduzindo a carga no banco de dados e garantindo baixa latência em operações frequentes.

5.1.2 NAMESPACES SEGREGADOS (PROXY E APP)

Segurança:
Separa o NGINX (proxy) dos microsserviços e Redis (app), com Network Policies controlando o tráfego (ex.: NGINX acessa Node.js apenas na porta 443).
O RBAC (Role-Based Access Control) restringe permissões por namespace, aplicando o princípio de privilégio mínimo.

Organização:
Permite gerenciar recursos de forma independente, escalando NGINX ou Node.js separadamente conforme a demanda.

5.1.3 BANCO DE DADOS MS SQL SERVER

Confiabilidade:
Suporta transações consistentes e alta disponibilidade com Always On Availability Groups, replicando dados assincronamente entre o master (datacenter) e o slave (nuvem).

Performance:
Otimizado para consultas complexas e relatórios financeiros, atendendo às necessidades do fluxo de caixa.

5.1.4 PROMETHEUS E GRAFANA

Controle de Métricas:
Utiliza um modelo pull-based, obtendo dados de endpoints /metrics expostos por serviços como Node.js, NGINX e Redis.

Alertas:
Permite configurar regras de alerta para notificar a equipe sobre anomalias (ex.: uso excessivo de CPU ou alta latência), possibilitando ações rápidas para manter os serviços operacionais.

Dashboards:

Exibe o estado dos serviços em tempo real, organizados por namespaces (ex.: proxy para NGINX, app para Node.js e Redis).

Análise de Dados:

Facilita a correlação entre métricas e logs, proporcionando uma visão unificada para diagnóstico de problemas.

5.1.5 ELK (ELASTICSEARCH, LOGSTASH, KIBANA)

Monitoramento:
Centraliza logs do Kubernetes (via Fluentd) e do MS SQL Server (via Filebeat) no Elasticsearch, com visualização em dashboards no Kibana.
Ajuda a identificar e resolver problemas rapidamente, oferecendo visibilidade sobre métricas críticas.

5.1.6 WORKLOAD HÍBRIDO (CONSIDERANDO 60% DATACENTER LOCAL, 40% NUVEM)

Custo e Controle:
Manter 60% no datacenter local reduz custos operacionais e dá mais controle sobre dados sensíveis.

Escalabilidade e Resiliência:
Os 40% na nuvem permitem escalar rapidamente em picos de demanda e garantem continuidade em caso de falhas no datacenter.

5.1.6.1 CENÁRIO 1: DATACENTER LOCAL + AZURE ARC COM AZURE SQL MANAGED INSTANCE

Justificativas Específicas

Gerenciamento com Azure Arc:
Integra o cluster Kubernetes e o MS SQL Server master (no datacenter) como recursos Azure, centralizando governança e monitoramento no Azure Portal.

Azure SQL Managed Instance:
Hospeda a réplica do banco na nuvem com alta disponibilidade, backups automáticos e integração nativa com Always On Availability Groups, reduzindo a sobrecarga operacional.

Segurança:
Integração com Azure Active Directory (AAD) para autenticação SSO e aplicação de políticas de segurança unificadas (ex.: Azure Defender for SQL).

Facilidade Operacional:
O Azure Traffic Manager distribui o tráfego entre datacenter (60%) e Azure (40%), com monitoramento simplificado via Azure Arc.

Vantagens:
Ideal para quem busca simplicidade, integração nativa com AAD e alta disponibilidade com menos esforço operacional.

5.1.6.2 Cenário 2: Datacenter Local + AWS com EKS e MS SQL em EC2

Justificativas Específicas

EKS (Elastic Kubernetes Service):
Hospeda os 40% da carga na AWS com auto scaling e integração com serviços como ALB (Application Load Balancer) para balanceamento de tráfego.

MS SQL Server em EC2:
A réplica assíncrona roda em uma instância EC2, oferecendo controle total sobre configurações como Always On Availability Groups.

Flexibilidade:
AWS permite ajustes finos no EKS e em serviços complementares, ideal para equipes experientes.

Performance:
Oferece baixa latência e alta disponibilidade, com opções robustas de rede e armazenamento.

Vantagens:
Melhor para quem precisa de maior controle e já tem expertise no ecossistema AWS.

5.2 Tabela Comparativa

Aspecto	**Cenário 1 (Azure Arc)**	**Cenário 2 (AWS)**
Gerenciamento	Centralizado via Azure Arc	Separado (EKS e EC2)
Banco de Dados	SQL Managed Instance (gerenciado)	MS SQL em EC2 (manual)
Segurança	Políticas unificadas com AAD	Configurações manuais (IAM)
Complexidade	Menor, mais automatizado	Maior, mais configurável

Tabela comparativa - 1

5.3 Quando Escolher Cada Cenário?

Cenário 1:
Priorize se busca integração simples com AAD, gerenciamento unificado e menos esforço operacional.

Cenário 2:
Escolha se prefere flexibilidade, controle total e tem experiência com AWS.

5.4 Conclusão

A stack tecnológica foi adotada por oferecer escalabilidade (Kubernetes, Node.js, Redis), segurança (NGINX com mTLS, namespaces, RBAC), confiabilidade (MS SQL Server), visibilidade (ELK) e custo-benefício (workload híbrido). O Cenário 1 (Azure Arc) simplifica a gestão e melhora a resiliência com SQL Managed Instance, enquanto o Cenário 2 (AWS) dá mais controle com EKS e EC2, mas exige mais configuração. A escolha depende das prioridades da equipe, como facilidade operacional ou flexibilidade técnica.

6 AUTENTICAÇÃO E SEGURANÇA

A autenticação e a segurança são fundamentais para proteger uma aplicação de fluxo de caixa, garantindo que apenas usuários autorizados acessem os serviços e que os dados financeiros permaneçam confidenciais e íntegros. A stack tecnológica utiliza componentes como um provedor de identidade central, NGINX como proxy seguro, **namespaces (proxy, app) para isolamento**, criptografia em todas as camadas e ferramentas de monitoramento para alcançar esses objetivos. A seguir, exploramos como esses elementos se aplicam aos cenários de integração entre um datacenter local e as nuvens Azure e AWS.

6.1 CENÁRIO 1: DATACENTER LOCAL INTEGRADO COM AZURE

Autenticação

- **Provedor de Identidade**: O Azure Active Directory (AAD) é utilizado como provedor central de identidade. Ele suporta padrões como OAuth 2.0, OpenID Connect (OIDC) e SAML, permitindo Single Sign-On (SSO) entre o datacenter local e os serviços na nuvem Azure.
- **Integração**: O AAD se conecta nativamente ao Azure Kubernetes Service (AKS) e ao Azure Arc, que estende a gestão do Azure ao datacenter local. Isso facilita a autenticação federada, onde as credenciais corporativas existentes são reutilizadas.
- **Banco de Dados**: O Azure SQL Managed Instance, usado para armazenar os dados financeiros, suporta logins baseados em AAD, eliminando a necessidade de credenciais separadas.

Segurança

- **Criptografia**: O tráfego externo é protegido com HTTPS (porta 443) via NGINX, enquanto a comunicação interna entre NGINX e Node.js usa mTLS (Mutual TLS). A conexão ao Azure SQL Managed Instance é criptografada com TLS (porta 1433).
- **Controle de Acesso**: O NGINX valida tokens JWT emitidos pelo AAD antes de encaminhar requisições aos microsserviços. Network Policies e RBAC (Role-Based Access Control) em namespaces como proxy e app restringem o tráfego e os privilégios.
- **Gerenciamento Centralizado**: O Azure Arc unifica as políticas de segurança, aplicando ferramentas como o Azure Defender for SQL tanto no datacenter local quanto na nuvem.
- **Monitoramento**: Ferramentas como ELK, Grafana e Prometheus fornecem visibilidade e alertas sobre anomalias, com logs detalhados para auditoria.

Vantagem

A integração nativa com o AAD e o gerenciamento centralizado via Azure Arc simplificam a implementação e a manutenção da segurança.

6.2 Cenário 2: Datacenter Local Integrado com AWS

Autenticação

- **Provedor de Identidade**: Pode-se usar o AAD como provedor externo via SAML ou OIDC, integrando-o ao AWS Identity and Access Management (IAM), ou optar diretamente pelo AWS IAM como provedor central. Isso permite SSO entre o datacenter local e o Amazon Elastic Kubernetes Service (EKS).
- **Integração**: A autenticação federada exige configuração manual para sincronizar identidades entre o datacenter e a AWS, utilizando SAML ou OIDC.
- **Banco de Dados**: O MS SQL Server rodando em uma instância EC2 não possui suporte nativo a logins federados como no Azure, demandando ajustes manuais para integrar com o provedor de identidade.

Segurança

- **Criptografia**: Assim como no Azure, o NGINX usa HTTPS para tráfego externo e mTLS para comunicação interna com Node.js. A conexão ao MS SQL em EC2 é protegida com TLS (porta 1433).
- **Controle de Acesso**: O NGINX valida tokens JWT, enquanto Security Groups e políticas de rede no EKS restringem o tráfego. O AWS IAM define permissões granulares para usuários e serviços.
- **Gerenciamento**: A segurança é configurada manualmente via AWS IAM, Security Groups e VPCs, oferecendo maior controle, mas sem a centralização nativa do Azure Arc.
- **Monitoramento**: ELK, Grafana e Prometheus são igualmente utilizados para detectar anomalias e gerar logs de auditoria.

Vantagem

A flexibilidade do AWS IAM e das configurações manuais permite personalizar a segurança conforme necessidades específicas.

6.3 Tabela Comparativa

Aspecto	**Cenário 1 (Azure Arc)**	**Cenário 2 (AWS)**
Integração de Identidade	Nativa com AAD	Via SAML/OIDC, exige mais configuração
Gerenciamento de Segurança	Centralizado via Azure Arc	Manual via AWS IAM e Security Groups
Autenticação no Banco	Suporte nativo no SQL Managed Instance	Configuração manual no MS SQL em EC2
Facilidade de Implementação	Alta devido à integração nativa	Moderada, mais esforço manual
Flexibilidade	Menor, mas simplificada	Maior, com controle detalhado

Tabela comparativa - 2

6.4 CONCLUSÃO

Ambos os cenários – datacenter local integrado com Azure e com AWS – oferecem autenticação robusta e segurança avançada para a aplicação de fluxo de caixa, utilizando um provedor de identidade central, NGINX como proxy seguro, isolamento via namespaces, criptografia em todas as camadas e monitoramento contínuo. O cenário com **Azure** se destaca pela simplicidade e integração nativa com o AAD e o Azure Arc, sendo ideal para quem busca agilidade na implementação. Já o cenário com **AWS** oferece maior flexibilidade e controle por meio do AWS IAM e configurações manuais, atendendo a quem prefere personalização detalhada. A escolha depende das prioridades: simplicidade (Azure) ou customização (AWS).

7 DR

Disaster Recovery (DR) em cenários de datacenter local integrado com Azure e com AWS podem prover RTO (Recovery Time Objective) e RPO (Recovery Point Objective) adequados para uma aplicação crítica como fluxo de caixa.

- RTO (Recovery Time Objective): Representa o tempo máximo que a aplicação pode ficar indisponível após uma falha antes de causar impactos. Para uma aplicação crítica como fluxo de caixa, que exige alta disponibilidade, o RTO ideal é muito baixo, na casa de **minutos** (ex.: 5 a 15 minutos).
- RPO (Recovery Point Objective): Quantidade máxima de dados perdida, medida pelo tempo entre o último backup ou réplica e a falha. Em um sistema financeiro, onde cada transação é crucial, o RPO deve ser próximo de **zero** (ex.: segundos ou, no máximo, poucos minutos).

7.1 Cenário 1: Datacenter Local + Azure Arc + Azure SQL Managed Instance

- **RTO no Azure**
  No cenário com **Azure Arc** e **Azure SQL Managed Instance (MI)**, a recuperação é altamente otimizada:
  - **Failover Automático:** O SQL Managed Instance suporta grupos de disponibilidade (como Always On Availability Groups) com failover automático para uma réplica na nuvem, reduzindo o tempo de inatividade **a 5-15 minutos.**
  - **Redirecionamento de Tráfego:** O Azure Traffic Manager redireciona o tráfego para o cluster Kubernetes (AKS) na nuvem quase instantaneamente após o failover.
  - **Vantagem:** A automação nativa do Azure minimiza a intervenção humana, garantindo um RTO baixo e confiável.
- **RPO no Azure**
  - **Replicação Assíncrona**: A réplica no SQL MI é atualizada de forma assíncrona com um atraso mínimo (segundos a minutos), resultando em um RPO de até **5 minutos**.
  - **Cache com Redis**: Dados voláteis podem ser mantidos no Redis, mas para um RPO menor, é necessário configurar persistência (ex.: AOF ou RDB) para evitar perdas.
  - **Otimização**: Ajustando a frequência da replicação ou usando backups frequentes, o RPO pode ser reduzido ainda mais.

7.2 Cenário 2: Datacenter Local + AWS + MS SQL em EC2

- **RTO na AWS**
  No cenário com **AWS** e **MS SQL Server rodando em instâncias EC2**, o processo é menos automatizado:
  - **Failover Manual**: A promoção da réplica no EC2 para o papel de master exige intervenção manual, o que aumenta o tempo de recuperação para **30-60 minutos**.
  - **Redirecionamento de Tráfego**: O **AWS Application Load Balancer (ALB)** redireciona o tráfego para o cluster EKS, mas o processo completo é mais lento devido à falta de automação nativa.
  - **Desvantagem**: A dependência de ações manuais eleva o RTO, tornando-o menos ideal para uma aplicação crítica.

RPO na AWS

- **Replicação Assíncrona**: Assim como no Azure, o uso de Always On Availability Groups em replicação assíncrona resulta em um RPO de até **5 minutos**.
- **Cache com Redis**: Configurações de persistência no Redis são igualmente necessárias para minimizar perdas de dados.
- **Risco**: A falta de automação pode introduzir atrasos na sincronização, afetando a consistência do RPO.

Otimizações para uma Aplicação Crítica como Fluxo de Caixa

Para garantir que o RTO e o RPO atendam aos requisitos rigorosos de uma aplicação de fluxo de caixa, algumas melhorias podem ser implementadas em ambos os cenários:

- **Automação Avançada**:
  - **Azure**: Scripts ou políticas de failover automático já são nativos, mas podem ser refinados para reduzir o RTO para menos de 5 minutos.
  - **AWS**: Configurar **AWS Lambda** ou ferramentas de automação para disparar o failover, diminuindo o RTO para algo próximo de 15-20 minutos.
- **Replicação Síncrona**:
  - Em ambos os cenários, adotar replicação síncrona (em vez de assíncrona) pode zerar o RPO, garantindo que nenhum dado seja perdido. Isso exige uma conexão de baixa latência e alta largura de banda entre o datacenter local e a nuvem, mas é viável para sistemas financeiros críticos.
- **Backups Frequentes**:
  - **Azure**: O SQL MI suporta backups automáticos com alta frequência (ex.: a cada 1-5 minutos).
  - **AWS**: Snapshots frequentes no EC2 ou uso do **AWS Backup** podem reduzir o RPO para segundos.
- **Testes Regulares**:
  - Simulações de falhas devem ser realizadas periodicamente para validar os tempos de RTO e RPO e ajustar as configurações conforme necessário.

7.3 TABELA COMPARATIVA

Aspecto	**Cenário 1 (Azure Arc)**	**Cenário 2 (AWS)**
RTO	5-15 minutos (automático)	30-60 minutos (manual)
RPO	Até 5 minutos (assíncrono)	Até 5 minutos (assíncrono)
Automatização	Alta (nativa)	Baixa (exige customização)
Flexibilidade	Gerenciamento simplificado	Maior esforço manual

Tabela comparativa - 3

7.4 CONCLUSÃO

Para uma aplicação crítica como fluxo de caixa, o cenário Datacenter Local + Azure Arc com Azure SQL Managed Instance é superior, oferecendo um RTO de 5-15 minutos e um RPO de até 5 minutos, com automação integrada e menor complexidade operacional. Já o cenário Datacenter Local + AWS com MS SQL em EC2 apresenta um RTO mais alto (30-60 minutos) devido à necessidade de intervenção manual, embora o RPO seja comparável (até 5 minutos).

Para atender aos requisitos mais exigentes (RTO e RPO próximos de zero), recomenda-se implementar replicação síncrona e backups frequentes em ambos os casos, com o Azure se destacando pela facilidade de automação e gerenciamento. Assim, o Azure é a escolha mais robusta e ágil para garantir a continuidade de uma aplicação financeira essencial.

utilizando **Ansible**, **Terraform** (ou **AWS CloudFormation**, se restrito a ferramentas AWS), podemos reduzir significativamente o RTO no cenário descrito para a **AWS**, chegando a **15-30 minutos**. Isso envolve automatizar o failover do MS SQL Server, o redirecionamento de tráfego no EKS e a atualização das conexões dos microsserviços. Embora seja uma melhoria expressiva em relação ao processo manual, o **Azure** ainda seria **mais rápido para aplicações críticas devido à sua automação nativa**.

8 MONITORAÇÃO E OBSERVABILIDADE

8.1 MONITORAÇÃO OSI

O Modelo OSI divide a comunicação de rede em 7 camadas, permitindo uma análise estruturada do tráfego. Vamos mapear como Grafana, Prometheus e outras ferramentas podem monitorar eventos em cada camada relevante para a aplicação de fluxo de caixa.

- **Camada 1 – Física (Conexões Físicas)**
  - Ferramenta: Prometheus com Node Exporter.
  - Monitoramento: O Node Exporter coleta métricas de hardware (ex.: taxa de erros de pacotes, latência de interface de rede) nos nós do Kubernetes (datacenter e nuvem).
  - Exemplo: Um pico de erros na interface de rede pode indicar falhas físicas (ex.: cabos ou switches). Grafana exibe essas métricas em um dashboard, permitindo ação rápida para evitar quedas.
- **Camada 2 – Enlace (Switches, VLANs)**
  - Ferramenta: Prometheus com SNMP Exporter.
  - Monitoramento: Monitora switches e VLANs usados para segmentar o tráfego entre namespaces (proxy e app). Métricas como colisões de pacotes ou erros de frame são coletadas via SNMP.
  - Exemplo: Um aumento de colisões pode indicar problemas de configuração no datacenter local ou na VPC (AWS). Grafana alerta a equipe para corrigir a segmentação.
- Camada 3 – Rede (IP, Roteamento)
  - Ferramenta: Prometheus com Blackbox Exporter.
  - Monitoramento: Verifica a conectividade IP entre componentes (ex.: NGINX → Node.js, Node.js → SQL). Métricas como latência de pacotes, perda de pacotes e tempo de resposta são coletadas.
  - Segurança: Detecta tentativas de acesso não autorizado (ex.: IPs fora do permitido pelas Network Policies).
  - Exemplo: Um aumento na perda de pacotes entre o datacenter e o Azure pode indicar problemas de roteamento, visíveis em um dashboard Grafana, permitindo ajustes no Azure Traffic Manager.
- **Camada 4 – Transporte (TCP/UDP)**
  - Ferramenta: Prometheus com Node Exporter e NGINX Prometheus Exporter.
  - Monitoramento: Métricas como conexões TCP ativas, retransmissões e timeouts são coletadas para portas específicas (ex.: 443 para HTTPS, 6379 para Redis, 1433 para SQL).
  - Eficiência: Retransmissões altas podem indicar congestionamento, afetando o desempenho.
  - Segurança: Um número anormal de conexões TCP pode sugerir um ataque DDoS, que pode ser correlacionado com logs do ELK.
  - Exemplo: Grafana mostra um pico de retransmissões na porta 443 entre NGINX e Node.js, indicando problemas de rede que podem ser mitigados ajustando configurações de TCP.
- **Camada 5 – Sessão (Gerenciamento de Sessões)**
  - Ferramenta: ELK (Elastic Stack).
  - Monitoramento: Logs do NGINX (namespace proxy) registram detalhes de sessões (ex.: duração, falhas de autenticação).
  - Segurança: Detecta tentativas de hijacking de sessão ou falhas repetidas de autenticação (ex.: tokens JWT inválidos).
  - Exemplo: Kibana exibe um aumento de falhas de autenticação, sugerindo um ataque de força bruta, permitindo bloqueio proativo de IPs suspeitos.
- **Camada 6 – Apresentação (Criptografia, Formato)**
  - Ferramenta: Prometheus com Blackbox Exporter e Grafana.
  - Monitoramento: Verifica a integridade do TLS (ex.: certificados expirados, falhas de handshake).
  - Segurança: Garante que o tráfego HTTPS (NGINX → Usuários, NGINX → Node.js com mTLS) e TLS (Node.js → SQL) esteja funcionando corretamente.
  - Exemplo: Um alerta no Grafana indica que um certificado está prestes a expirar, permitindo renovação via cert-manager antes de falhas.

- **Camada 7 – Aplicação (HTTP, APIs)**
    - Ferramenta: Prometheus com NGINX Prometheus Exporter e Grafana.
    - Monitoramento: Métricas de requisições HTTP (ex.: taxas de erro 5xx, latência de APIs REST).
    - Eficiência: Identifica gargalos em endpoints críticos (ex.: /entradas no Node.js).
    - Segurança: Detecta padrões de tráfego anormais (ex.: aumento de requisições suspeitas), correlacionando com logs no ELK para análise detalhada.
    - Exemplo: Grafana mostra um pico de erros 403, indicando falhas de autorização, que podem ser investigadas via logs no Kibana.

8.1.1 Abordagens Complementares

Além do Modelo OSI, podemos adotar abordagens complementares para monitoramento de rede:

- **Análise de Fluxo de Rede (NetFlow/IPFIX):**
    - Ferramenta: ntopng ou AWS VPC Flow Logs (no cenário AWS).
    - Monitoramento: Registra fluxos de tráfego entre componentes (ex.: datacenter → AKS/EKS).
    - Segurança: Identifica tráfego suspeito (ex.: portas não autorizadas).
    - Eficiência: Ajuda a otimizar rotas de tráfego entre o datacenter e a nuvem.
- **Monitoramento de Segurança (WAF e IDS):**
    - Ferramenta: AWS WAF (AWS) ou Azure Application Gateway com WAF (Azure).
    - Monitoramento: Analisa tráfego na camada 7 para detectar ataques (ex.: SQL injection, XSS).
    - Segurança: Bloqueia requisições maliciosas antes que cheguem ao NGINX.

- **Especificidades por Cenário**

8.1.2 Cenário 1: Datacenter Local + Azure Arc + Azure SQL Managed Instance

- **Monitoramento de Rede:**
    - Azure Network Watcher: Complementa o Prometheus/Grafana, fornecendo diagnóstico de rede (ex.: latência entre datacenter e AKS).
    - Azure Monitor: Integra-se com Grafana para exibir métricas de rede do SQL MI (ex.: conexões TCP na porta 1433).
- **Segurança:**
    - Network Watcher pode detectar tráfego não autorizado entre o datacenter e o Azure, correlacionando com logs no ELK.
- **Eficiência:**
    - Dashboards no Grafana mostram latência de rede entre o datacenter e o AKS, permitindo ajustes no Azure Traffic Manager para otimizar o tráfego.

8.1.3 Cenário 2: Datacenter Local + AWS + MS SQL em EC2

- **Monitoramento de Rede:**
    - AWS VPC Flow Logs: Registra fluxos de tráfego entre o datacenter e o EKS, integrando com Prometheus/Grafana via exportadores.
    - Amazon CloudWatch: Fornece métricas de rede (ex.: latência de pacotes), que podem ser visualizadas no Grafana.
- **Segurança:**
    - VPC Flow Logs ajudam a identificar tráfego suspeito (ex.: IPs fora das regras de Security Groups), com alertas configurados no Prometheus.
- **Eficiência:**
    - Grafana exibe métricas de latência entre o datacenter e o EKS, permitindo ajustes no AWS ALB para melhorar o desempenho.

8.2 TABELA COMPARATIVA

Aspecto	**Cenário 1 (Azure Arc)**	**Cenário 2 (AWS)**
Monitoramento de Rede	Azure Network Watcher e Monitor	VPC Flow Logs e CloudWatch
Segurança de Rede	Integração nativa com Azure Security	Configuração manual com Security Groups
Eficiência do Tráfego	Ajustes via Azure Traffic Manager	Ajustes via AWS ALB
Complexidade	Baixa (ferramentas nativas)	Moderada (exige integração de ferramentas)

Tabela comparativa - 4

8.2.1 CONCLUSÃO

Grafana, Prometheus e ferramentas complementares (como Azure Network Watcher e AWS VPC Flow Logs) permitem monitorar eventos de rede em todas as camadas do Modelo OSI, garantindo segurança (detecção de tráfego suspeito, validação de criptografia) e eficiência (otimização de latência e roteamento). No cenário Azure, a integração nativa com ferramentas como Network Watcher simplifica o monitoramento e ajustes, enquanto no cenário AWS, VPC Flow Logs e CloudWatch oferecem flexibilidade, mas exigem mais configuração. Ambas as abordagens protegem a aplicação de fluxo de caixa e otimizam o tráfego, com o Azure se destacando pela facilidade e o AWS pela personalização.

8.3 OBSERVABILIDADE

8.3.1 PILARES DA OBSERVABILIDADE

Observabilidade é sustentada por três pilares principais, que já foram parcialmente implementados na stack (Kubernetes, NGINX, Node.js, Redis, ELK, Grafana, Prometheus). Vamos expandir e detalhar cada pilar:

- **Métricas (Prometheus e Grafana)**
    - **O que já temos**: Prometheus coleta métricas (ex.: latência, erros HTTP, uso de CPU) e Grafana as visualiza em dashboards, cobrindo camadas do Modelo OSI (ex.: latência de rede na Camada 3, erros HTTP na Camada 7).
    - **Aprimoramento**:
        - **Métricas Customizadas nos Microsserviços**: No Node.js, usar bibliotecas como prom-client para criar métricas específicas da aplicação (ex.: tempo de processamento de uma entrada no fluxo de caixa).
        - **Métricas de Banco de Dados**: Usar **Prometheus SQL Exporter** para coletar métricas detalhadas do MS SQL Server (ex.: tempo médio de queries, bloqueios de transação), tanto no datacenter quanto na Azure SQL MI ou EC2.
        - **Dashboards Avançados**: Criar dashboards no Grafana com alertas baseados em thresholds (ex.: latência de API > 500ms), correlacionando métricas de diferentes camadas (rede, aplicação, banco).
- **Logs (ELK)**
    - **O que já temos**: O ELK (Elasticsearch, Logstash, Kibana) centraliza logs de NGINX, Node.js, Redis e MS SQL Server, permitindo análise de eventos de autenticação e erros.
    - **Aprimoramento**:
        - **Logs Estruturados**: Padronizar logs no formato JSON em todos os componentes (ex.: NGINX, Node.js), facilitando a busca e análise no Kibana.
        - **Correlação com Contexto**: Adicionar IDs de transação (correlation IDs) aos logs, permitindo rastrear uma requisição desde o NGINX até o Node.js, Redis e MS SQL Server.
            - Exemplo: Um correlation ID txn-123 é incluído no cabeçalho HTTP pelo NGINX e propagado para logs de todos os serviços, visível no Kibana.
        - **Análise de Segurança**: Usar Kibana para criar visualizações que detectem padrões de ataque (ex.: múltiplas falhas de autenticação por IP), complementando a análise de rede feita com Prometheus.

- **Tracing (OpenTelemetry ou Jaeger)**
    - **O que falta**: Atualmente, a stack não inclui tracing distribuído, essencial para entender o fluxo de requisições em um sistema de microsserviços.
    - **Aprimoramento**:
        - **Implementar OpenTelemetry**: Instrumentar o NGINX, Node.js e conexões com Redis e MS SQL Server para gerar traces.
            - No Node.js, usar a biblioteca @opentelemetry para capturar spans:
        - **Backend de Tracing**: Usar **Jaeger** ou **Elastic APM** (integrado ao ELK) para armazenar e visualizar traces.
        - **Benefício**: Tracing permite identificar gargalos (ex.: uma query lenta no MS SQL Server) e rastrear falhas em requisições distribuídas, como uma entrada que falha ao ser processada.
        - Pilares da Observabilidade

8.3.2 FERRAMENTAS COMPLEMENTARES

Além de Grafana, Prometheus, ELK e OpenTelemetry/Jaeger, podemos adicionar:

- **Loki (para Logs)**:
    - Uma alternativa leve ao ELK, Loki é otimizado para logs em ambientes Kubernetes. Ele se integra ao Grafana, permitindo correlacionar logs e métricas em um único painel.
    - **Exemplo**: Loki coleta logs do NGINX e Node.js, e Grafana exibe um gráfico de latência HTTP ao lado de logs de erros, facilitando a análise.
- **AWS X-Ray (no Cenário AWS)**:
    - No cenário AWS, o X-Ray pode ser usado para tracing distribuído, complementando o OpenTelemetry. Ele rastreia requisições entre o EKS, EC2 e outros serviços AWS.
    - **Exemplo**: X-Ray mostra que uma requisição ao MS SQL em EC2 está lenta devido a bloqueios de transação, visível em um mapa de serviço.
- **Azure Monitor (no Cenário Azure)**:
    - No cenário Azure, o Azure Monitor coleta métricas e logs do AKS e do Azure SQL MI, integrando-se ao Grafana para uma visão unificada.
    - **Exemplo**: Azure Monitor detecta um pico de latência no SQL MI, correlacionado com métricas de rede no Grafana.

8.3.3 FERRAMENTAS COMPLEMENTARES

- **Cenário 1: Datacenter Local + Azure Arc + Azure SQL Managed Instance**
    - **Métricas**: Prometheus e Azure Monitor coletam métricas do AKS e do SQL MI (ex.: tempo de queries, conexões TCP). Grafana exibe essas métricas em dashboards, permitindo correlacionar latência de rede com desempenho do banco.
    - **Logs**: ELK centraliza logs, e o Azure Monitor adiciona logs de diagnóstico do SQL MI, visíveis no Kibana.
    - **Tracing**: OpenTelemetry instrumenta os microsserviços no AKS, e os traces são armazenados no Jaeger, mostrando o fluxo completo de uma requisição (ex.: NGINX → Node.js → SQL MI).
    - **Vantagem**: A integração nativa com Azure Monitor e a facilidade de configuração de tracing no AKS tornam a observabilidade mais simples e centralizada.
- **Cenário 2: Datacenter Local + AWS + MS SQL em EC2**
    - **Métricas**: Prometheus coleta métricas do EKS e do EC2 (via SQL Exporter), complementado por métricas do **Amazon CloudWatch**. Grafana exibe tudo em dashboards unificados.
    - **Logs**: ELK centraliza logs, e o **AWS CloudWatch Logs** adiciona logs do EC2, que podem ser exportados para o Elasticsearch.
    - **Tracing**: OpenTelemetry com AWS X-Ray rastreia requisições no EKS, mostrando gargalos (ex.: latência entre Node.js e EC2).
    - **Vantagem**: AWS X-Ray oferece uma visão detalhada do tráfego entre serviços AWS, mas a configuração de tracing e exportação de logs exige mais esforço.

8.3.4 BENEFÍCIOS PARA SEGURANÇA E EFICIÊNCIA

- **Segurança**:
    - Tracing (OpenTelemetry/Jaeger) ajuda a identificar requisições suspeitas (ex.: tempos de resposta anormais que podem indicar ataques).
    - Logs (ELK/Loki) correlacionados com métricas (Prometheus) detectam padrões de ataque (ex.: aumento de erros 403 com IPs suspeitos).
- **Eficiência**:
    - Métricas e traces identificam gargalos (ex.: latência alta no Redis ou no SQL), permitindo ajustes (ex.: aumentar réplicas do Redis).
    - Dashboards unificados no Grafana mostram o impacto de mudanças (ex.: latência reduzida após otimizar Network Policies).

8.4 TABELA COMPARATIVA

Aspecto	**Cenário 1 (Azure Arc)**	**Cenário 2 (AWS)**
Métricas	Prometheus + Azure Monitor	Prometheus + CloudWatch
Logs	ELK + Azure Monitor Logs	ELK + CloudWatch Logs
Tracing	OpenTelemetry + Jaeger	OpenTelemetry + AWS X-Ray
Integração	Nativa e simplificada	Exige configuração adicional
Complexidade	Baixa	Moderada

Tabela comparativa - 5

Conclusão

A observabilidade é ampliada com **métricas** (Prometheus/Grafana), **logs** (ELK/Loki) e **tracing** (OpenTelemetry/Jaeger ou AWS X-Ray), proporcionando uma visão completa do sistema em ambos os cenários. No **Azure**, a integração nativa com Azure Monitor facilita a implementação, enquanto no **AWS**, ferramentas como X-Ray e CloudWatch oferecem flexibilidade, mas com maior esforço de configuração. Essa abordagem garante **segurança** (detecção de anomalias), **eficiência** (otimização de desempenho) e **resiliência** (diagnóstico rápido de falhas), essencial para uma aplicação crítica como fluxo de caixa em um ambiente híbrido.

9 AUTOMAÇÃO PARA GANHO DE PRODUTIVIDADE

Vamos discutir como implementar **automação** para aumentar a produtividade nos ambientes **Datacenter Local**, **Azure Arc com Azure SQL Managed Instance**, e **AWS com MS SQL em EC2**, considerando a aplicação de fluxo de caixa. Focaremos no uso de **Ansible**, **Terraform** (e, para AWS, opcionalmente **AWS CloudFormation**), detalhando como essas ferramentas podem melhorar a **eficiência**, **reduzir erros**, **aumentar produtividade** e **diminuir o RTO (Recovery Time Objective)**. Também exploraremos ferramentas complementares que podem potencializar a automação.

Objetivos da Automação

A automação visa:

- **Eficiência**: Reduzir o tempo necessário para tarefas repetitivas (ex.: provisionamento, configuração).
- **Redução de Erros**: Eliminar falhas humanas em configurações manuais.
- **Produtividade**: Permitir que a equipe foque em tarefas estratégicas, não operacionais.
- **Redução de RTO**: Acelerar a recuperação em cenários de desastre (DR).

9.1 AUTOMAÇÃO COM ANSIBLE, TERRAFORM E FERRAMENTAS COMPLEMENTARES

Automação no Datacenter Local

- **Provisionamento de Infraestrutura (Terraform)**
    - **O que Automatizar**: Configuração do cluster Kubernetes, nós, redes (VLANs) e storage para o MS SQL Server master.
    - **Como**: Usar Terraform para definir a infraestrutura como código (IaC).
        - Exemplo: Criar nós do Kubernetes e configurar redes no datacenter local (hcl):

```
resource "vsphere_virtual_machine" "k8s_node" {
 name           = "k8s-node-${count.index}"
 count          = 3
 resource_pool_id = data.vsphere_resource_pool.pool.id
 datastore_id     = data.vsphere_datastore.datastore.id
 num_cpus       = 4
 memory         = 8192
 guest_id       = "ubuntuLinuxGuest"
 network_interface {
  network_id = data.vsphere_network.network.id
 }
 disk {
  label = "disk0"
  size  = 50
 }
}
```

    - **Benefício**: Provisionamento rápido e consistente, reduzindo erros manuais.

- **Configuração e Gerenciamento (Ansible)**
  - **O que Automatizar**: Instalação do Kubernetes, configuração de namespaces (proxy, app), Network Policies, e instalação do Fluentd para logs no ELK.
  - **Como**: Criar playbooks Ansible para configurar os nós e serviços.
    - Exemplo: Playbook para instalar o Kubernetes e configurar namespaces (yml):

```
- name: Configurar cluster Kubernetes
  hosts: k8s_nodes
  tasks:
    - name: Instalar kubeadm, kubelet e kubectl
      apt:
        name: "{{ packages }}"
        state: present
      vars:
        packages:
          - kubeadm
          - kubelet
          - kubectl
    - name: Criar namespaces
      kubernetes.core.k8s:
        state: present
        definition:
          apiVersion: v1
          kind: Namespace
          metadata:
            name: "{{ item }}"
      loop:
        - proxy
        - app
```

  - **Benefício**: Configurações repetíveis e livres de erros, aumentando a produtividade da equipe.
- **Monitoramento e Alertas (Prometheus/Grafana)**
  - Automatizar a implantação do Prometheus e Grafana via Ansible, configurando dashboards pré-definidos para monitorar o cluster Kubernetes e o MS SQL Server.
  - **Benefício**: Reduz o tempo de setup inicial e garante visibilidade imediata.

9.2 Cenário: Datacenter Local + Azure Arc com Azure SQL Managed Instance

- **Provisionamento de Infraestrutura (Terraform)**
  - **O que Automatizar**: Criação do AKS, configuração do Azure SQL MI, e integração com Azure Arc para gerenciar o datacenter local.
  - **Como**: Usar Terraform para provisionar recursos na Azure.
    - Exemplo: Criar um cluster AKS e conectar o datacenter ao Azure Arc (hcl):

```
resource "azurerm_kubernetes_cluster" "aks" {
  name                = "aks-fluxo-de-caixa"
  location            = "East US"
  resource_group_name = azurerm_resource_group.rg.name
  dns_prefix          = "aksfluxo"
  default_node_pool {
    name       = "default"
    node_count = 2
```

```
    vm_size    = "Standard_D2_v2"
  }
  identity {
    type = "SystemAssigned"
  }
}

resource "azurerm_arc_kubernetes_cluster" "local_cluster" {
  name                = "dc-local-cluster"
  resource_group_name = azurerm_resource_group.rg.name
  location            = "East US"
  cluster_id          = "dc-local-cluster-id"
}
```

- **Benefício**: Provisionamento rápido e consistente, com integração nativa ao Azure Arc.

- **Configuração e Gerenciamento (Ansible)**

  - **O que Automatizar**: Configuração do AKS (ex.: implantação de NGINX, Node.js, Redis), instalação do Fluentd, e configuração do Always On Availability Groups no SQL MI.

  - **Como**: Playbooks Ansible para gerenciar o AKS e o SQL MI.

    - Exemplo: Configurar Always On no SQL MI (ymal):

```
- name: Configurar Always On no Azure SQL MI
  hosts: localhost
  tasks:
    - name: Habilitar Always On Availability Group
      azure.azcollection.azure_rm_sqlmanagedinstance:
        resource_group: "rg-fluxo-de-caixa"
        name: "sqlmi-fluxo-de-caixa"
        state: present
        availability_group_name: "ag-fluxo-de-caixa"
        primary_endpoint: "sqlmi-fluxo-de-caixa.database.windows.net"
```

  - **Benefício**: Reduz erros na configuração e acelera a implantação.

- **Redução de RTO**:

  - Ansible pode automatizar o failover do SQL MI (embora já seja nativo), ajustando conexões no AKS. Scripts via **Azure Functions** podem ser usados para disparar ações de failover, mantendo o RTO em **5-15 minutos**.

- **Ferramentas Extras**:

  - **Azure DevOps**: Para pipelines de CI/CD que automatizam deploy de aplicações no AKS.

  - **cert-manager**: Automatiza a renovação de certificados TLS/mTLS, garantindo segurança sem intervenção manual.

9.3 Cenário: Datacenter Local + AWS com MS SQL em EC2

- **Provisionamento de Infraestrutura (Terraform ou AWS CloudFormation)**
    - **O que Automatizar**: Criação do EKS, instâncias EC2 para o MS SQL Server, e configuração de VPCs e Security Groups.
    - **Como**: Usar Terraform (ou CloudFormation, se restrito a ferramentas nativas AWS).
        - Exemplo com Terraform (hcl):

```
resource "aws_eks_cluster" "eks" {
 name     = "eks-fluxo-de-caixa"
 role_arn = aws_iam_role.eks_role.arn
 vpc_config {
  subnet_ids = aws_subnet.subnets[*].id
 }
}

resource "aws_instance" "mssql_ec2" {
 ami           = "ami-0c55b159cbfafe1f0" # AMI com MS SQL Server
 instance_type = "m5.large"
 subnet_id     = aws_subnet.subnets[0].id
 tags = {
  Name = "mssql-ec2"
 }
}
```

    - **Benefício**: Infraestrutura provisionada rapidamente, com consistência e sem erros manuais.
- **Configuração e Gerenciamento (Ansible)**
    - **O que Automatizar**: Configuração do EKS (implantação de NGINX, Node.js, Redis), instalação do MS SQL Server na EC2, e configuração do Always On Availability Groups.
    - **Como**: Playbooks Ansible para gerenciar o EKS e o EC2.
        - Exemplo: Configurar o MS SQL Server e o Always On:

```
- name: Configurar MS SQL Server no EC2
 hosts: mssql_ec2
 tasks:
  - name: Habilitar Always On Availability Group
    win_shell: |
     Invoke-Sqlcmd -Query "ALTER AVAILABILITY GROUP [ag-fluxo-de-caixa] JOIN;"
    args:
     executable: powershell.exe
```

    - **Benefício**: Configuração consistente e rápida, reduzindo erros e aumentando produtividade.

- **Redução de RTO**:
    - Como discutido anteriormente, Ansible e Terraform automatizam o failover, ajustando o ALB e promovendo a réplica na EC2. Isso reduz o RTO de **30-60 minutos** (manual) para **15-30 minutos**.
    - **AWS Lambda** pode ser usado para disparar scripts Ansible automaticamente em resposta a eventos do CloudWatch, aproximando o RTO de 10-15 minutos.
- **Ferramentas Extras**:
    - **AWS Systems Manager (SSM)**: Automatiza tarefas de manutenção no EC2 (ex.: aplicação de patches no MS SQL Server).
    - **AWS CodePipeline**: Para pipelines de CI/CD que automatizam deploy no EKS.
    - **cert-manager**: Similar ao Azure, automatiza certificados TLS/mTLS.

9.4 Benefícios Gerais da Automação

- **Eficiência**: Terraform provisiona ambientes em minutos, e Ansible aplica configurações de forma consistente, eliminando processos manuais demorados.
- **Redução de Erros**: Configurações como código (IaC) e playbooks Ansible garantem consistência, evitando erros humanos.
- **Produtividade**: A equipe foca em tarefas estratégicas (ex.: melhorias na aplicação) enquanto tarefas operacionais são automatizadas.
- **Redução de RTO**: Scripts automatizam o failover, reduzindo o tempo de recuperação em ambos os cenários (Azure: 5-15 minutos; AWS: 15-30 minutos com automação).

9.5 Ferramentas Extras para Potencializar a Automação

- **Jenkins ou GitHub Actions**:
    - Automatizam pipelines de CI/CD para deploy contínuo de microsserviços (Node.js, NGINX) no AKS/EKS e no datacenter local.
    - **Exemplo**: Um commit no repositório dispara a construção de uma nova imagem Docker, que é implantada automaticamente no Kubernetes.
    - **Benefício**: Reduz o tempo de deploy e garante consistência.
- **Kubeadm ou Kubespray (Ansible-based)**:
    - Automatizam a implantação e atualização do Kubernetes no datacenter local, simplificando a gestão do cluster.
    - **Benefício**: Acelera a configuração inicial e upgrades.

- **ArgoCD**:
    - Ferramenta de GitOps para gerenciar configurações do Kubernetes declarativamente.
    - **Exemplo**: Um repositório Git contém os manifests do Kubernetes (ex.: NGINX, Node.js), e o ArgoCD os aplica automaticamente.
    - **Benefício**: Garante que o estado desejado do cluster seja mantido, reduzindo erros.

9.6 TABELA COMPARATIVA

Ferramentas Extras	**Ferramenta 1**	**Ferramenta 2**	**Ferramenta 3**
Kubeadm, ArgoCD	Kubeadm	ArgoCD	
Azure DevOps, cert-manager	Azure DevOps	cert-manager	
AWS SSM, CodePipeline, cert-manager	AWS SSM	CodePipeline	cert-manager

Tabela comparativa - 6

9.7 CONCLUSÃO

A automação com **Terraform** e **Ansible** aumenta a **eficiência**, **reduz erros**, **melhora a produtividade** e **diminui o RTO** em todos os ambientes. No **Datacenter Local**, provisiona e configura o cluster Kubernetes rapidamente. No **Azure**, a automação complementa a integração nativa do Azure Arc, mantendo um RTO baixo (5-15 minutos). No **AWS**, Ansible e Terraform (ou CloudFormation) reduzem o RTO para 15-30 minutos, superando o processo manual. Ferramentas como **ArgoCD**, **Azure DevOps**, e **AWS CodePipeline** potencializam a automação, garantindo consistência e permitindo que a equipe foque em inovação, enquanto a aplicação de fluxo de caixa opera com alta disponibilidade e resiliência.

10 DevSecOps

O **DevSecOps** incorpora práticas de segurança em todas as etapas do ciclo de vida de desenvolvimento e operações, garantindo que a segurança seja tratada como um componente essencial, e não como um acréscimo tardio. Para a aplicação de fluxo de caixa, isso significa integrar ferramentas de varredura de vulnerabilidades (CVEs), automação de segurança, e verificações contínuas no pipeline CI/CD, protegendo os ambientes híbridos (datacenter local, Azure, AWS) enquanto se mantém eficiência, produtividade e baixa latência de recuperação (RTO/RPO).

10.1 DevSecOps no Datacenter Local

- **Ferramenta de Varredura de Imagem: Trivy**
    - **Integração com CI/CD**: O Trivy é usado em pipelines CI/CD (ex.: Jenkins ou GitLab CI) para escanear imagens Docker antes do deploy no cluster Kubernetes.
        - **Exemplo**: Configuração em um pipeline Jenkins:

```
yaml
RecolherEncapsularCopiar
pipeline {
 agent any
 stages {
  stage('Scan Image') {
   steps {
    sh 'trivy image --severity HIGH,CRITICAL --exit-code 1 fluxo-de-caixa-entrada:latest'
   }
  }
  stage('Deploy to Kubernetes') {
   when { expression { currentBuild.result == null || currentBuild.result == 'SUCCESS' } }
   steps {
    sh 'kubectl apply -f deployment.yaml'
   }
  }
 }
}
```

    - **Automação**:
        - **Ansible**: Automatiza a execução do Trivy em pipelines CI/CD e bloqueia deploys se vulnerabilidades críticas forem detectadas.

```
yaml
RecolherEncapsularCopiar
- name: Escanear imagem com Trivy
 command: trivy image --severity HIGH,CRITICAL --exit-code 1 fluxo-de-caixa-entrada:latest
 register: trivy_result
 ignore_errors: true
- name: Falhar se vulnerabilidades críticas forem encontradas
 fail:
  msg: "Vulnerabilidades críticas encontradas na imagem!"
 when: trivy_result.rc != 0
```

        - **Terraform**: Configura repositórios locais (ex.: Harbor) para armazenar imagens escaneadas.
    - **Segurança**:
        - Garante que imagens (ex.: NGINX, Node.js, Redis) sejam livres de CVEs antes do deploy, complementando autenticação (AAD) e criptografia (HTTPS, mTLS).

- **OPA Gatekeeper**: Pode ser configurado no Kubernetes para bloquear imagens não escaneadas ou vulneráveis.

- **Observabilidade**:
  - Resultados do Trivy podem ser exportados como logs JSON para o ELK, permitindo visualização no Kibana ou integração com Grafana (via métricas).

- **Benefício**: Integra segurança diretamente no pipeline DevSecOps do datacenter local, reduzindo riscos sem dependência de ferramentas de nuvem.

10.2 DevSecOps no Cenário 1: Datacenter Local + Azure Arc com Azure SQL Managed Instance

- **Ferramenta de Varredura de Imagem: Microsoft Defender for Containers**
  - **Integração com CI/CD**: O Defender for Containers escaneia imagens no Azure Container Registry (ACR) e no AKS, com verificação contínua e no push.
    - **Exemplo**: Pipeline no Azure DevOps que escaneia imagens e falha se vulnerabilidades críticas forem detectadas:

```
yaml
RecolherEncapsularCopiar
steps:
 - task: Docker@2
   displayName: Push image to ACR
   inputs:
    command: push
    containerRegistry: acr-fluxo-de-caixa
    repository: fluxo-de-caixa-entrada
    tags: latest
 - task: AzureCLI@2
   displayName: Check Defender for Containers Scan Results
   inputs:
    azureSubscription: 'AzureServiceConnection'
    scriptType: bash
    scriptLocation: inlineScript
    inlineScript: |
     az acr run --cmd "acr check-health -n acrfluxodecaixa --image fluxo-de-caixa-entrada:latest" /dev/null
```

  - **Automação**:
    - **Terraform**: Configura o ACR e habilita o Defender for Containers, como já exemplificado anteriormente.
    - **Ansible**: Garante que pipelines no Azure DevOps falhem se CVEs críticos forem encontrados.
  - **Segurança**:
    - Protege imagens no AKS, complementando autenticação (AAD), criptografia (HTTPS, mTLS), e políticas de segurança via Azure Arc.
    - **Azure Policy**: Pode bloquear deploys de imagens vulneráveis.
  - **Observabilidade**:

- Resultados do Defender for Containers são integrados ao Azure Monitor, visualizados no Grafana junto com métricas de rede (Prometheus) e logs (ELK).

- **Benefício**: A integração nativa com o Azure simplifica a adoção do DevSecOps, garantindo segurança contínua.

10.3 DevSecOps no Cenário 2: Datacenter Local + AWS com MS SQL em EC2

- **Ferramenta de Varredura de Imagem: Amazon Inspector**
  - **Integração com CI/CD**: O Amazon Inspector escaneia imagens no Amazon Elastic Container Registry (ECR) e no EKS, com escaneamento contínuo e no push.
    - **Exemplo**: Configuração no AWS CodePipeline para escanear imagens:

```
yaml
RecolherEncapsularCopiar
stages:
 - name: Source
   actions: [...]
 - name: Scan
   actions:
    - name: InspectorScan
      actionTypeId:
        category: Test
        owner: AWS
        provider: Inspector
        version: "1"
      configuration:
        RepositoryName: "fluxo-de-caixa-repo"
```

  - **Automação**:
    - **Terraform**: Configura o ECR e habilita o Amazon Inspector, como já exemplificado.
    - **Ansible**: Integra o Inspector ao AWS CodePipeline, bloqueando deploys vulneráveis.
  - **Segurança**:
    - Protege imagens no EKS, complementando autenticação (AWS IAM ou AAD via SAML/OIDC), criptografia (HTTPS, mTLS), e políticas de segurança via Security Groups.
    - **AWS Security Hub**: Centraliza resultados do Inspector, permitindo ações automatizadas.
  - **Observabilidade**:
    - Resultados do Inspector são integrados ao CloudWatch, visualizados no Grafana junto com métricas de rede (Prometheus) e logs (ELK).
    - AWS X-Ray pode correlacionar falhas de deploy com vulnerabilidades.
  - **Benefício**: O escaneamento contínuo do Inspector garante segurança robusta no pipeline DevSecOps.

10.4 Ferramentas Extras para DevSecOps em Todos os Ambientes

- **Snyk**:
    - Escaneia imagens, dependências e código-fonte, com integração a pipelines CI/CD (Jenkins, Azure DevOps, AWS CodePipeline).
    - **Automação**: Ansible pode configurar Snyk em pipelines para escanear e sugerir remediações.
    - **Benefício**: Oferece remediações automáticas e é compatível com ambientes híbridos.
- **Aqua Security**:
    - Plataforma de segurança para contêineres, com escaneamento de CVEs e conformidade.
    - **Benefício**: Suporta datacenter local, Azure e AWS, com políticas de segurança personalizáveis.
- **Harbor (Datacenter Local)**:
    - Repositório de imagens com escaneamento integrado (via Trivy ou Clair).
    - **Automação**: Terraform provisiona o Harbor, e Ansible configura pipelines para escanear imagens.
    - **Benefício**: Centraliza a gestão de imagens no datacenter local.

10.5 Impacto nos Quesitos Analisados

- **Autenticação e Segurança**:
    - DevSecOps com Trivy (Datacenter Local), Defender for Containers (Azure), e Amazon Inspector (AWS) garante que imagens sejam livres de CVEs, complementando autenticação (AAD/AWS IAM) e criptografia (HTTPS, mTLS).
- **Automação**:
    - Ansible e Terraform integram ferramentas de varredura em pipelines CI/CD, automatizando verificações e bloqueando deploys vulneráveis, aumentando a produtividade.
- **RTO/RPO**:
    - Imagens seguras reduzem riscos durante failovers, mantendo o RTO (Azure: 5-15 minutos; AWS: 15-30 minutos) e o RPO próximo de zero com replicação síncrona.
- **Observabilidade**:
    - Resultados de varredura são visualizados no Grafana (via Azure Monitor, CloudWatch, ou logs do Trivy no ELK), correlacionando vulnerabilidades com métricas e logs.

10.6 TABELA COMPARATIVA

Aspecto	**Datacenter Local**	**Datacenter + Azure Arc + SQL MI**	**Datacenter + AWS + EC2**
Ferramenta de Varredura	Trivy (open-source)	Microsoft Defender for Containers	Amazon Inspector
Integração com CI/CD	Jenkins, GitLab CI	Azure DevOps	AWS CodePipeline
Automação	Ansible (pipelines), Terraform (repositórios)	Terraform (ACR), Ansible (pipelines)	Terraform (ECR), Ansible (pipelines)
Segurança	Imagens seguras, OPA Gatekeeper	Imagens seguras, Azure Policy	Imagens seguras, AWS Security Hub
Observabilidade	ELK, Grafana (logs/métricas)	Azure Monitor, Grafana (métricas)	CloudWatch, Grafana (métricas)
Complexidade	Moderada (configuração local)	Baixa (integração nativa)	Moderada (configuração necessária)

Tabela comparativa - 7

10.7 CONCLUSÃO

A inclusão do **DevSecOps** como tópico separado fortalece a segurança da aplicação de fluxo de caixa, integrando varredura de imagens (Trivy no datacenter local, Microsoft Defender for Containers no Azure, Amazon Inspector na AWS) ao pipeline CI/CD. Isso garante que vulnerabilidades sejam identificadas e mitigadas antes do deploy, complementando autenticação, criptografia, automação e observabilidade. Ferramentas como **Snyk**, **Aqua Security**, e **Harbor** adicionam flexibilidade e robustez, enquanto Ansible e Terraform automatizam o processo, aumentando a produtividade e reduzindo riscos em todos os ambientes híbridos.

11 GitHub e GitHub Actions

11.1 Integração com GitHub e GitHub Actions

A integração da aplicação de fluxo de caixa com o **GitHub** e o **GitHub Actions** permite a automação do pipeline de CI/CD, garantindo que o ciclo de desenvolvimento seja eficiente, seguro e alinhado às melhores práticas de DevOps. Este capítulo descreve como o código da aplicação será gerenciado no GitHub e como o GitHub Actions será configurado para automatizar testes, construção de imagens Docker, varredura de segurança e implantação no Kubernetes (on-premises e na nuvem).

Repositório no GitHub

O código-fonte da aplicação de fluxo de caixa será hospedado em um repositório público no GitHub, seguindo o requisito do desafio. A estrutura do repositório será organizada para facilitar a colaboração e a automação:

```
fluxo-de-caixa/
├── src/                # Código-fonte da aplicação (Node.js)
├── Dockerfile          # Arquivo para construir a imagem Docker
├── helm/               # Helm Chart para implantação no Kubernetes
├── tests/              # Testes automatizados (ex.: unitários)
├── .github/workflows/  # Workflows do GitHub Actions
|   ├── ci-cd.yaml      # Pipeline de CI/CD
|   └── security-scan.yaml  # Workflow para varredura de segurança
├── README.md           # Documentação do projeto
└── docs/               # Documentação adicional (ex.: este documento)
```

- **Repositório Público**: O repositório será criado no GitHub (ex.: github.com/seu-usuario/fluxo-de-caixa).
- **Branches**:
    - main: Código estável e pronto para produção.
    - develop: Código em desenvolvimento, para testes e integração.
    - Branches de feature (ex.: feature/nova-funcionalidade) para novos desenvolvimentos.

Configuração do GitHub Actions para CI/CD

O **GitHub Actions** será usado para automatizar o pipeline de CI/CD, cobrindo as seguintes etapas:

1. **Testes**: Executar testes unitários da aplicação.
2. **Construção**: Construir uma imagem Docker para o Node.js.
3. **Varredura de Segurança**: Usar Trivy para verificar vulnerabilidades na imagem.

4. **Publicação**: Publicar a imagem no repositório de contêineres (ex.: Docker Hub, Azure Container Registry ou Amazon ECR).

5. **Implantação**: Atualizar o Helm Chart e implantar no Kubernetes (on-premises e na nuvem).

Exemplo de Workflow: .github/workflows/ci-cd.yaml

```yaml
name: CI/CD Pipeline para Fluxo de Caixa

on:
  push:
    branches:
      - main
      - develop
  pull_request:
    branches:
      - main
      - develop

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout do Código
        uses: actions/checkout@v3

      - name: Configurar Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'

      - name: Instalar Dependências
        run: npm install
        working-directory: ./src

      - name: Executar Testes Unitários
        run: npm test
        working-directory: ./src

  build-and-push:
    runs-on: ubuntu-latest
    needs: test
    steps:
      - name: Checkout do Código
        uses: actions/checkout@v3

      - name: Configurar Docker Buildx
        uses: docker/setup-buildx-action@v2

      - name: Login no Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Construir e Publicar Imagem Docker
        uses: docker/build-push-action@v4
        with:
          context: ./src
          file: ./Dockerfile
          push: true
```

```
        tags: |
          ${{ secrets.DOCKER_USERNAME }}/fluxo-de-caixa:${{ github.sha }}
          ${{ secrets.DOCKER_USERNAME }}/fluxo-de-caixa:latest

  security-scan:
    runs-on: ubuntu-latest
    needs: build-and-push
    steps:
      - name: Checkout do Código
        uses: actions/checkout@v3

      - name: Executar Varredura com Trivy
        uses: aquasecurity/trivy-action@master
        with:
          image-ref: ${{ secrets.DOCKER_USERNAME }}/fluxo-de-caixa:${{ github.sha }}
          severity: 'HIGH,CRITICAL'
          exit-code: '1'  # Falha o job se vulnerabilidades forem encontradas

  deploy:
    runs-on: ubuntu-latest
    needs: security-scan
    steps:
      - name: Checkout do Código
        uses: actions/checkout@v3

      - name: Configurar Helm
        uses: azure/setup-helm@v3
        with:
          version: 'v3.9.0'

      - name: Configurar kubectl
        uses: azure/setup-kubectl@v3
        with:
          version: 'v1.24.0'

      - name: Autenticar no Cluster Kubernetes (On-Premises)
        run: |
          echo "${{ secrets.KUBE_CONFIG }}" > kubeconfig
          export KUBECONFIG=kubeconfig
          kubectl config use-context on-premises-cluster

      - name: Implantar no Kubernetes On-Premises
        run: |
          helm upgrade --install fluxo-de-caixa ./helm \
            --namespace app \
            --set nodejs.image=${{ secrets.DOCKER_USERNAME }}/fluxo-de-caixa:${{ github.sha }}

      - name: Autenticar no AKS (Azure)
        uses: azure/login@v1
        with:
          creds: ${{ secrets.AZURE_CREDENTIALS }}

      - name: Configurar Contexto AKS
        run: |
          az aks get-credentials --resource-group fluxo-de-caixa-rg --name aks-fluxo-de-caixa

      - name: Implantar no AKS
        run: |
          helm upgrade --install fluxo-de-caixa ./helm \
            --namespace app \
```

```
--set nodejs.image=${{ secrets.DOCKER_USERNAME }}/fluxo-de-caixa:${{ github.sha }}
```

11.2 Explicação do Workflow

- **Trigger**: O pipeline é acionado em pushes ou pull requests para main e develop.
- **Testes**: Executa testes unitários do Node.js para garantir a qualidade do código.
- **Construção**: Constrói a imagem Docker e a publica no Docker Hub (pode ser substituído por Azure Container Registry ou Amazon ECR).
- **Varredura de Segurança**: Usa o Trivy para identificar vulnerabilidades críticas na imagem.
- **Implantação**: Implanta a aplicação no Kubernetes (on-premises e AKS), atualizando o Helm Chart com a nova imagem.

Segredos no GitHub

Para garantir a segurança do pipeline, os segredos serão configurados no GitHub:

- DOCKER_USERNAME e DOCKER_PASSWORD: Credenciais para o Docker Hub.
- KUBE_CONFIG: Configuração do cluster Kubernetes on-premises.
- AZURE_CREDENTIALS: Credenciais para autenticação no Azure (JSON com clientId, clientSecret, subscriptionId, tenantId).

Esses segredos podem ser adicionados em Settings > Secrets > Actions no repositório GitHub.

11.3 Benefícios da Integração

- **Automação**: O pipeline CI/CD reduz o trabalho manual, garantindo consistência e rapidez nas implantações.
- **Segurança**: A varredura de vulnerabilidades com Trivy assegura que apenas imagens seguras sejam implantadas.
- **Escalabilidade**: A integração com Helm permite implantações uniformes em ambientes on-premises e na nuvem.
- **Rastreabilidade**: O GitHub fornece versionamento e histórico de mudanças, facilitando auditorias.

11.4 Estrutura do Repositório no GitHub

O repositório público já inclui:

- Código-fonte (src/).
- Helm Chart (helm/).
- Documentação em Markdown (README.md, docs/).
- Workflows do GitHub Actions (.github/workflows/).
- **URL do Repositório**: https://github.com/seu-usuario/fluxo-de-caixa
  Observação: Substitua pelo link real do repositório.

11.5 CONCLUSÃO

A adição da integração com GitHub e GitHub Actions completa o projeto, alinhando-o aos requisitos do desafio. A aplicação de fluxo de caixa agora possui um pipeline CI/CD automatizado que garante qualidade, segurança e eficiência nas implantações, tanto no datacenter local quanto na nuvem (Azure e AWS). A solução combina alta disponibilidade, segurança, otimização de custos, resiliência, automação e governança, com o GitHub Actions como peça central para a entrega contínua.

12 Conclusão do Projeto de Desafio

Após quase uma semana de discussões detalhadas, consolidamos uma solução abrangente e estratégica para o projeto de desafio, abordando os tópicos cruciais de **alta disponibilidade e escalabilidade**, **segurança e controle de acesso**, **otimização de custos**, **resiliência e recuperação (DR)**, e **automação e governança**. Abaixo, apresentamos uma conclusão que reflete como cada aspecto foi cuidadosamente considerado e implementado, resultando em uma solução robusta para suportar a aplicação de fluxo de caixa em cenários híbridos (datacenter local, Azure e AWS).

12.1 Alta Disponibilidade e Escalabilidade

A solução foi projetada para garantir que a aplicação esteja sempre acessível e capaz de lidar com variações de demanda.

- **Clusters Kubernetes**: Implantados no datacenter local e na nuvem (Azure com AKS e AWS com EKS), utilizando **autoscaling** (HPA e Cluster Autoscaler) para ajustar recursos dinamicamente com base na carga (ex.: CPU > 70%).
- **Distribuição de Carga**: Balanceamento entre datacenter (60%) e nuvem (40%), assegurando redundância e resposta a picos de tráfego (ex.: 50 req/s).
- **Resultado**: Operação contínua e escalabilidade eficiente, atendendo às necessidades dos usuários sem interrupções.

12.2 Segurança e Controle de Acesso Aprimorados

A segurança foi priorizada com práticas avançadas para proteger dados e restringir acesso a usuários autorizados.

- **Autenticação**: Integração com **Azure Active Directory (AAD)** para autenticação centralizada via **OAuth 2.0**, **OIDC** e **SAML** (SSO).
- **Criptografia**: Uso de **HTTPS**, **mTLS** (comunicação interna) e **TLS** (bancos de dados).
- **Controle no Kubernetes**: Namespaces segregados (proxy, app) com **Network Policies** e **RBAC**.
- **Varredura de Segurança**: Ferramentas como **Trivy**, **Microsoft Defender for Containers** (Azure) e **Amazon Inspector** (AWS) integradas ao pipeline DevSecOps.
- **Resultado**: Dados protegidos, acesso seguro e conformidade com padrões de segurança.

12.3 Otimização dos Custos

A eficiência financeira foi um foco central, equilibrando desempenho e economia.

- **Tagueamento**: Recursos marcados com tags detalhadas (ex.: projeto=fluxo-de-caixa, ambiente=producao) para rastreamento de custos.
- **Contratações Estratégicas**: Uso de **Reserved Instances** para cargas fixas e **Spot Instances** para tarefas não críticas.
- **Monitoramento**: Ferramentas como **Azure Cost Management**, **AWS Cost Explorer**, **Grafana** e **Prometheus** com alertas para anomalias.
- **Resultado**: Custos otimizados, transparência financeira e ajustes proativos para evitar desperdícios.

12.4 Resiliência e Recuperação (DR)

A solução foi desenhada para minimizar o impacto de falhas, garantindo continuidade operacional.

- **Cenário Azure**: **Azure SQL Managed Instance** com failover automático, oferecendo **RTO** de 5-15 minutos e **RPO** próximo de zero (replicação síncrona).
- **Cenário AWS**: **MS SQL em EC2** com automação via **Ansible** e **Terraform**, alcançando **RTO** de 15-30 minutos e **RPO** de até 5 minutos (replicação assíncrona).
- **Resultado**: Recuperação rápida e perda mínima de dados, essencial para uma aplicação financeira crítica.

12.5 Automação e Governança

A automação e a governança foram reforçadas para assegurar consistência e conformidade.

- **Infraestrutura como Código (IaC)**: Uso de **Terraform** para provisionamento e **Ansible** para configuração, criando ambientes replicáveis e auditáveis.
- **Governança**: Políticas implementadas com **Azure Policy**, **AWS Organizations** e práticas de FinOps para controle de custos e segurança.
- **Resultado**: Redução de erros humanos, maior eficiência e conformidade facilitada com diretrizes organizacionais.

12.6 Resumo Final

A solução proposta é **robusta**, **segura**, **economicamente eficiente**, **resiliente** e **bem governada**, atendendo plenamente aos requisitos do projeto. A abordagem híbrida (datacenter local + Azure/AWS) oferece flexibilidade e redundância, enquanto tecnologias como Kubernetes, AAD, Terraform e práticas de FinOps proporcionam uma base moderna e sustentável. Essa infraestrutura suporta as operações financeiras críticas da organização, garantindo alta disponibilidade, proteção de dados, controle de custos, recuperação rápida em falhas e governança eficaz. Estamos preparados para desafios futuros com uma solução escalável, auditável e alinhada às metas estratégicas.

13 Anexos

13.1 Referências de Ferramentas e Tecnologias Utilizadas

Este anexo lista as ferramentas e tecnologias mencionadas ao longo do projeto, com links para suas documentações oficiais, que podem ser consultadas para mais detalhes sobre implementação e uso.

- **Terraform**
  Ferramenta de infraestrutura como código (IaC) usada para provisionamento de recursos.
  Documentação oficial: https://www.terraform.io/docs
- **Ansible**
  Ferramenta de automação utilizada para configuração e orquestração.
  Documentação oficial: https://docs.ansible.com/ansible/latest/index.html
- **Kubernetes**
  Plataforma de orquestração de contêineres usada para alta disponibilidade e escalabilidade.
  Documentação oficial: https://kubernetes.io/docs/home/
- **RKE (Rancher Kubernetes Engine)**
  Ferramenta de gerenciamento de clusters Kubernetes, útil para implantação e operação no datacenter local.
  Documentação oficial: https://rancher.com/docs/rke/latest/en/
- **Azure Active Directory (AAD)**
  Serviço de autenticação e autorização utilizado para controle de acesso.
  Documentação oficial: https://learn.microsoft.com/en-us/azure/active-directory/
- **Azure SQL Managed Instance**
  Banco de dados gerenciado no Azure, utilizado para resiliência e recuperação.
  Documentação oficial: https://learn.microsoft.com/en-us/azure/azure-sql/managed-instance/
- **AWS EC2 e EKS**
  Serviços da AWS para computação e orquestração de contêineres.
  Documentação oficial:
  - EC2: https://docs.aws.amazon.com/ec2/
  - EKS: https://docs.aws.amazon.com/eks/
- **Grafana**
  Ferramenta de visualização para monitoramento e observabilidade.
  Documentação oficial: https://grafana.com/docs/
- **Prometheus**
  Sistema de monitoramento e alertas para observabilidade.
  Documentação oficial: https://prometheus.io/docs/
- **Trivy**
  Ferramenta open-source para varredura de vulnerabilidades em imagens Docker.
  Documentação oficial: https://aquasecurity.github.io/trivy/
- **Microsoft Defender for Containers**
  Solução de segurança para contêineres no Azure.
  Documentação oficial: https://learn.microsoft.com/en-us/azure/defender-for-cloud/defender-for-containers-introduction
- **Amazon Inspector**
  Serviço de segurança para varredura de vulnerabilidades na AWS.
  Documentação oficial: https://docs.aws.amazon.com/inspector/
- **Azure Cost Management**
  Ferramenta para monitoramento e gestão de custos no Azure.
  Documentação oficial: https://learn.microsoft.com/en-us/azure/cost-management-billing/

- **AWS Cost Explorer**
  Ferramenta para análise e gestão de custos na AWS.
  Documentação oficial: https://docs.aws.amazon.com/cost-management/latest/userguide/what-is-costexplorer.html
- **ELK Stack (Elasticsearch, Logstash, Kibana)**
  Conjunto de ferramentas para observabilidade e análise de logs.
  Documentação oficial: https://www.elastic.co/guide/index.html
- **Azure Functions e AWS Lambda**
  Serviços serverless para automação de scripts.
  Documentação oficial:
  - Azure Functions: https://learn.microsoft.com/en-us/azure/azure-functions/
  - AWS Lambda: https://docs.aws.amazon.com/lambda/

13.2 SOBRE PAULO LYRA

Sou um **profissional sênior com 34 anos de experiência em Tecnologia da Informação e Telecomunicações (TIC)**, com uma carreira consolidada em grandes empresas e startups, incluindo nomes de peso como iBest, Brasil Telecom, Oi e V.tal. Nos últimos 22 anos, atuei em ambientes dinâmicos de empresas nacionais e multinacionais, acumulando expertise em projetos complexos e entregas de alto impacto.

Minha Trajetória e Contribuições

Ao longo da minha carreira, ocupei posições estratégicas que moldaram minha capacidade de liderar e transformar desafios em soluções inovadoras. Dentre as principais funções que desempenhei, destaco:

- **Arquiteto de Infraestrutura de TI e Arquiteto Cloud**: Projetei e implementei arquiteturas resilientes e escaláveis em ambientes Cloud (pública, privada, híbrida) e On Premises, utilizando tecnologias como AWS, GCP, OCI e Openshift.
- **Gerente de Projetos de TI e Consultor Técnico**: Lidei projetos de infraestrutura desde a concepção até a entrega, aplicando frameworks ágeis e seguros, otimizando recursos e alinhando soluções às diretrizes de governança e custos.
- **Líder de Equipes e Especialista Sênior**: Gerenciei equipes multidisciplinares, fornecedores e stakeholders, promovendo colaboração e resultados acima das expectativas.

Um marco recente foi minha atuação na V.tal, onde liderei o processo de **foundations (landing zone)** para AWS, GCP e OCI, estabelecendo bases sólidas para operações em nuvem. Essa experiência reflete minha habilidade em entregar soluções que combinam inovação, eficiência e segurança.

Expertise Técnica e Foco em Inovação

Meu conhecimento técnico abrange um espectro amplo e atualizado, permitindo-me atuar como um profissional hands-on e estratégico. Domino tecnologias como:

- **Cloud e MultiCloud**: AWS, GCP, OCI, Openshift, EKS, GKE, OKE.
- **Automação e DEVOPS**: Terraform, Ansible, Git, Docker, Kubernetes.
- **Monitoramento e Otimização**: Grafana, Prometheus, FinOps.
- **Segurança e Resiliência**: Cyber Security, HA (alta disponibilidade), DR (recuperação de desastres).
- **Outras Competências**: Linux, Virtualização (VMware, Proxmox), Big Data (Cloudera), IA, Redes, Scrum, ITIL.

Essa expertise me permite construir soluções robustas, como a apresentada neste projeto, que equilibra alta disponibilidade, escalabilidade, segurança e eficiência financeira.

Habilidades que Me Diferenciam

Sou reconhecido por competências que vão além da técnica, agregando valor estratégico às organizações:

- **Liderança e Visão de Negócio**: Motivo equipes e alinho tecnologia aos objetivos corporativos, com foco em resultados mensuráveis.
- **Resiliência e Adaptabilidade**: Entrego projetos sob pressão, adaptando-me a cenários complexos com criatividade e ética.
- **Comunicação e Colaboração**: Facilito o diálogo entre equipes, stakeholders e fornecedores, além de formar novos profissionais com espírito de equipe.

- **Inovação e Melhoria Contínua**: Proponho soluções disruptivas, como a automação deste projeto com Terraform e Ansible, garantindo consistência e governança.

Compromisso com Resultados

Neste projeto de desafio, demonstrei minha capacidade de integrar alta disponibilidade, segurança, otimização de custos, resiliência e automação em uma solução coesa e prática. Minha experiência de 34 anos, aliada a uma abordagem hands-on e estratégica, me posiciona como um profissional preparado para enfrentar desafios complexos e entregar valor sustentável. Estou pronto para aplicar esse mesmo rigor e visão em novos projetos, contribuindo para o sucesso da organização com inovação, liderança e resultados concretos.

Meus contatos:

Paulo.lyra@gmail.com

Paulo@ibest.com

Fone e ZAP + 55 (61) 98401-1394

LN

https://www.linkedin.com/in/paulolyra/?trk=opento_sprofile_details

https://www.linkedin.com/in/paulolyra?lipi=urn%3Ali%3Apage%3Ad_flagship3_profile_view_base_contact_details%3BT6SrBzjwRxCmOodU3vtInA%3D%3D