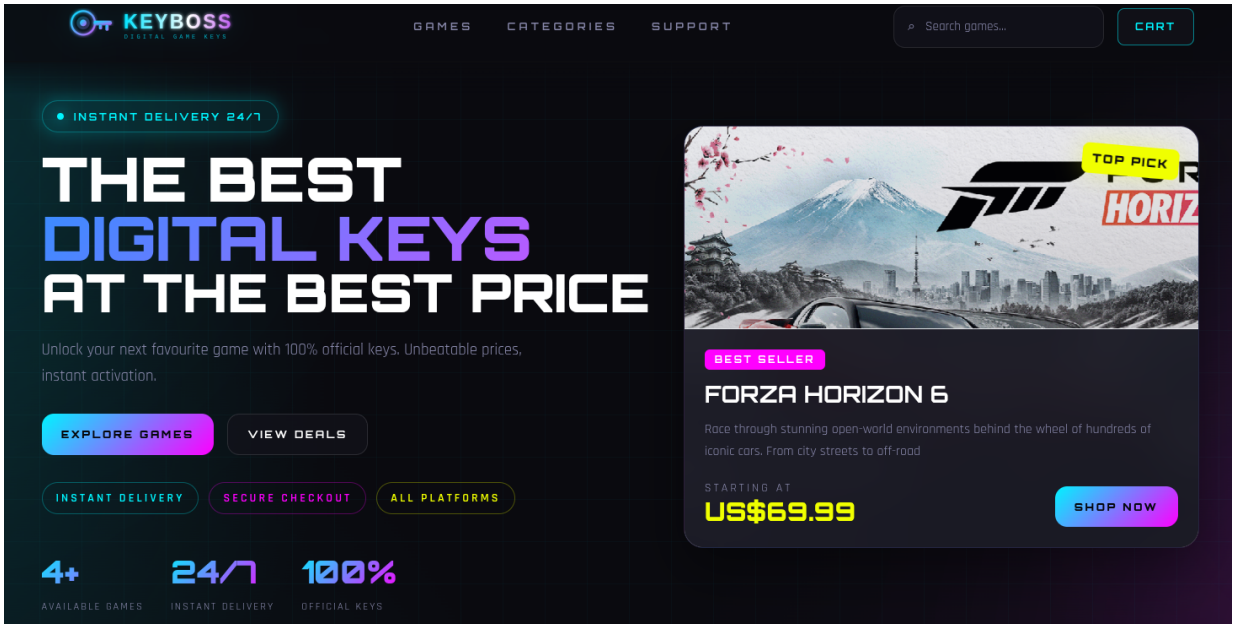
deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

universidade de aveiro
theoria poiesis praxis

KeyBoss

Relatório da Milestone 2: Resiliência, Automatização e Observabilidade



Gestão de Infraestruturas de Computação

Grupo: KeyBoss
Ano letivo: 2025/2026
Data: maio de 2026
Elementos: Carolina Reis, nº 131193
Guilherme Silva, nº 131143

Índice

1 Introdução

1.1 Contexto e objetivos da Milestone 2

A Milestone 1 estabeleceu o *baseline* Kubernetes local do KeyBoss, validando a comunicação entre os componentes e os dois *Critical User Journeys* (CUJ). A Milestone 2 transforma essa base num sistema fiável, observável e com gestão automatizada, cobrindo três eixos principais:

- **Automatização** do pipeline de deployment através de um único script que garante reprodutibilidade e rollback automático.
- **Resiliência e escalonamento**: redundância ativa, autoscaling horizontal demonstrado e estratégia de recuperação de desastres para o estado persistente.
- **Observabilidade**: stack de monitorização com Prometheus, Grafana e Alertmanager, SLIs/SLOs formalizados e alertas configurados para os CUJs.

O deployment final foi realizado no **cluster do departamento DETI** (ambiente de produção partilhado), o que garante que a nota não fica limitada ao tecto de 16 pontos do ambiente local.

1.2 Ambiente de deployment

Foram usados dois ambientes ao longo da M2:

- **Local (k3d)**: cluster de 3 nós (1 server + 2 agents), namespace `keyboss`, usado para desenvolvimento e demonstração de HPA.
- **DETI** (`193.136.82.35:6443`): cluster k3s com 12 nós (3 control-plane + 9 workers), namespace `tenant-keyboss`, StorageClass Longhorn, ambiente de produção partilhado. Acesso via VPN da Universidade de Aveiro.

O script de deployment suporta ambos os ambientes através de uma única invocação parametrizada.

2 Automatização e Pipeline de Deployment

2.1 Visão geral do pipeline

O pipeline de deployment é inteiramente controlado pelo script `deployment/M2/deploy.sh`, que aceita a flag `-local` ou `-deti` e executa todas as fases necessárias sem intervenção manual:

```
# Deployment no ambiente local (k3d)
bash deployment/M2/deploy.sh --local

# Deployment no cluster do DETI
bash deployment/M2/deploy.sh --deti

# Reset completo do ambiente
bash deployment/M2/deploy.sh --deti --reset
```

O script aplica os manifestos Kubernetes por fases ordenadas, aguardando o rollout de cada Deployment antes de avançar para a fase seguinte. Esta estratégia garante que o sistema nunca fica parcialmente deployed e que os erros são detectados imediatamente.

2.2 Fases do script

#	Fase	O que faz
1	Namespace & Config	Cria namespace, aplica ConfigMap e Secrets
2	Serviços stateful	Aplica PostgreSQL HA (primary + standby); aguarda readiness de ambos os pods
3	Migração	Executa Job de migrações Django (idempotente; salta se já concluído)
4	Deployments	API, Celery, Storefront, Dashboard + Ingress; aguarda rollout de cada um
4a	Seed data	Corre `seed_first_run.py` dentro do pod da API (apenas no primeiro deploy; usa ConfigMap como marcador)
5	HPA	Configura Horizontal Pod Autoscaler para saleor-api e celery-worker
5a	Redis Sentinel	Aplica StatefulSet Redis HA + deployment dos 3 Sentinels
5b	PostgreSQL Watchdog	Aplica deployment do watchdog de failover automático
6	Backup	Cria CronJob de backup diário do PostgreSQL
7	Monitoring	Aplica Prometheus, Grafana, Alertmanager, kube-state-metrics
7a	Dashboards Grafana	Chama `create_dashboards.py` para provisionar os 3 dashboards via API HTTP
7b	Exporters	Aplica postgres-exporter-0, postgres-exporter-1 e redis-exporter

Tabela 1: Fases do script de deployment unificado.

2.3 Adaptações por ambiente

O script contém transformações automáticas para o ambiente DETI, aplicadas via `sed` em tempo de execução sobre os manifestos:

- Namespace: `keyboss` → `tenant-keyboss`
- StorageClass: `local-path` → `longhorn`
- Tamanho de PVCs: qualquer valor → `1Gi` (limite do LimitRange do cluster)
- Imagem do storefront: imagem local (`keyboss-storefront:m2`) → `registry.deti/tenant-keyboss/storefront:keyboss` com `imagePullPolicy: Always`
- URLs internas: `localhost` → `keyboss.deti` (Grafana, Prometheus, Alertmanager, API GraphQL)
- DNS interno dos serviços: `*.keyboss.svc` → `*.tenant-keyboss.svc`
- kube-state-metrics: no DETI não há permissão para ClusterRole, pelo que é deployado com um kubeconfig dedicado ao namespace `tenant-keyboss` em vez do ServiceAccount padrão
- Prometheus RBAC: ServiceAccount, ClusterRole e ClusterRoleBinding são filtrados no DETI (sem permissão); o Prometheus corre com o service account por omissão

Desta forma, mantém-se um único conjunto de manifestos de referência sem duplicação, e as diferenças entre ambientes ficam concentradas no script.

2.4 Seed de dados iniciais

O script inclui uma fase de seed (4a) que corre `seed_first_run.py` dentro do pod da Saleor API após o deployment. O script de seed é idempotente e cria:

- Conta de administrador (`admin@admin.com`)
- Product type *Digital Keys* com atributos *platform* (PC, Xbox, PlayStation) e *badge* (Fresh Drop, Editor Choice, Hot Pick, Top Seller)
- Categorias: Action, RPG, Racing, Sandbox
- Collections: *featured-products* e *top-pick*

Os produtos de demonstração (7 jogos com imagens, preços, plataforma e badge) foram criados manualmente através do Saleor Dashboard após o seed inicial. O seed mantém-se mínimo por defeito para ser idempotente e seguro em re-deploys.

Para evitar que o seed volte a correr em deploys subsequentes, é criado um ConfigMap `keyboss-seed` como marcador. Se o marcador existir, a fase é saltada.

2.5 Provisionamento automático de dashboards

Após o Grafana ficar pronto (fase 7), o script chama automaticamente o script `create_dashboards.py`, que usa a API HTTP do Grafana para criar ou atualizar os três dashboards KeyBoss com `overwrite: true`. Isto garante que os dashboards estão sempre sincronizados com o código, sem necessidade de intervenção manual. Se o módulo `requests` não estiver disponível, o passo é saltado com um aviso e o comando manual é impresso no output.

2.6 Rollback automático

O script implementa rollback automático através de `trap ERR`: se qualquer fase falhar, os Deployments já atualizados são revertidos via `kubectl rollout undo`. Os StatefulSets (PostgreSQL, Redis) e os PVCs não são revertidos automaticamente, pois o rollback de estado persistente requer avaliação manual.

```
DEPLOYED=() # Deployments atualizados nesta execucao

rollback() {
    for entry in "${DEPLOYED[@]}"; do
        kubectl rollout undo deployment/"${entry##*:}" \
            -n "${entry%%:*}" 2>/dev/null || true
    done
}

trap 'rollback' ERR
```

2.7 Gestão de configuração e segredos

A separação entre ConfigMap e Secret mantém-se da M1. Em M2, o script garante que:

1. As atualizações ao ConfigMap são aplicadas antes dos Deployments, para que os novos pods iniciem com a configuração correta.

2. Os Secrets nunca são registados em repositórios públicos; os valores para o DETI são passados por variáveis de ambiente ou editados localmente antes do deployment.

3. Os rollbacks incluem a reversão da imagem do container mas **não** a reversão de ConfigMaps ou Secrets, dado que estes podem ter estado partilhado com outras réplicas ainda em execução.

3 Resiliência, Escalonamento e Recuperação

3.1 Redundância e anti-affinity

Todos os Deployments stateless têm múltiplas réplicas com `podAntiAffinity` configurado para distribuir réplicas por nós distintos:

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
              - key: app
                operator: In
                values: [saleor-api]
          topologyKey: kubernetes.io/hostname
```

A regra é *preferred* (não *required*) para não bloquear o scheduling em clusters com menos nós do que réplicas. Em clusters com nós suficientes (como o DETI com 9 workers), a distribuição é garantida na prática. A Tabela 2 resume a configuração de réplicas por componente.

Multi-zone deployments não foram implementados porque o cluster DETI é um único datacenter físico sem zonas de disponibilidade distintas. O conceito de multi-zone faz sentido em ambientes cloud (AWS, GCP, Azure) onde existem datacenters geograficamente separados dentro de uma mesma região. No contexto deste projeto, a distribuição por nós via pod anti-affinity é o equivalente disponível e suficiente para os objetivos de M2.

Componente	Réplicas mín.	Réplicas máx.	Anti-affinity
Saleor API	2	6 (HPA)	Por hostname
Celery Workers	2	4 (HPA)	Por hostname
Storefront	2	2	Por hostname
Dashboard	1	1	N/A
PostgreSQL	2	2 (HA)	N/A (StatefulSet)
Redis/Valkey	2	2 (master+replica)	N/A (StatefulSet)
Redis Sentinel	3	3	N/A
Postgres Watchdog	1	1	N/A
keyboss-exporter	1	1	N/A

Tabela 2: Configuração de réplicas e anti-affinity por componente.

3.1.1 Evidência experimental — falha de pod com zero downtime

A resiliência de múltiplas réplicas foi validada no cluster DETI com o script `failure_validation.sh -deti`. Um pod da Saleor API foi eliminado enquanto a disponibilidade era verificada a cada 3 segundos. Os restantes pods continuaram a servir pedidos sem interrupção.

```
$ kubectl delete pod saleor-api-58898cf859-gszd5 -n tenant-keyboss
$ kubectl get pods -n tenant-keyboss | grep saleor-api
saleor-api-58898cf859-gszd5 2/2 Terminating 0   132m # pod eliminado
saleor-api-58898cf859-qp89z 2/2 Running     0   4m27s # replicas ativas
saleor-api-58898cf859-wp25j 0/2 Init:0/3    0   4s    # pod novo a arrancar
saleor-api-58898cf859-z5fmq 2/2 Running     0   38m   # replicas ativas
```

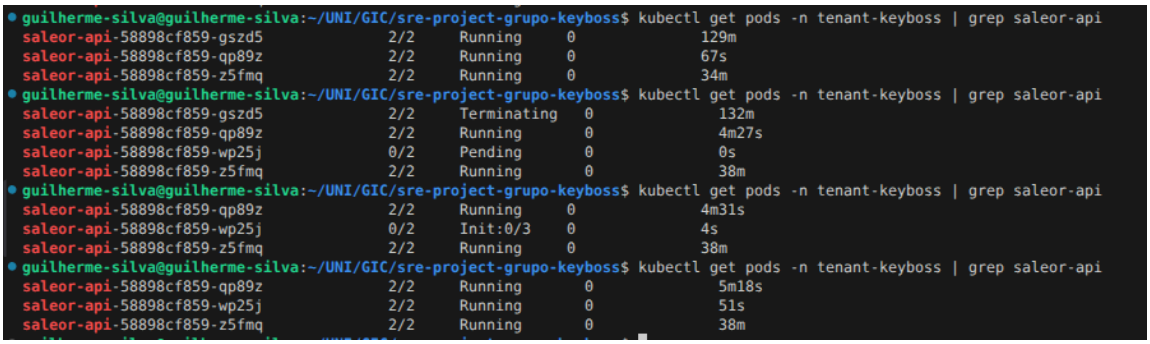


Figura 1: Pods da Saleor API durante o teste de falha: pod eliminado em `Terminating`, duas réplicas em `Running` a servir tráfego, e novo pod em `Init:0/3` a inicializar. Zero downtime registado nas 10 verificações de disponibilidade.

O resultado confirma zero downtime: todas as 10 verificações (30 segundos no total) retornaram HTTP 200. O pod substituto arrancou automaticamente sem intervenção manual.

3.2 Autoscaling com HPA

O **Horizontal Pod Autoscaler** foi configurado para os dois componentes stateless de maior carga: a Saleor API e os Celery Workers.

```
# saleor-api: escala entre 2 e 6 replicas
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: saleor-api-hpa
spec:
  scaleTargetRef:
    kind: Deployment
    name: saleor-api
  minReplicas: 2
  maxReplicas: 6
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70
    - type: Resource
      resource:
        name: memory
        target:
          type: Utilization
          averageUtilization: 80
```

O HPA usa duas métricas em simultâneo (CPU e memória) e escala quando *qualquer uma* ultrapassa o threshold. A janela de estabilização para scale-up é de 60 segundos e para scale-down de 300 segundos, evitando oscilações.

3.2.1 Opções de autoscaling consideradas

Foram avaliadas quatro abordagens antes de escolher o HPA:

O **VPA (Vertical Pod Autoscaler)** foi descartado porque ajusta os *resource requests* de um pod reiniciando-o — o que significa downtime durante o ajuste. Para a Saleor API, cujo SLO de disponibilidade é de 99%, reinícios forçados pelo VPA seriam contraproducentes. Além disso, o modelo stateless da API e dos workers adapta-se naturalmente a réplicas horizontais adicionais, não a pods maiores.

O **Cluster Autoscaler** não é aplicável neste contexto. O cluster DETI tem nós fixos geridos pelo departamento; não há acesso para provisionar ou remover nós da infraestrutura. O mesmo se aplica ao k3d local. O Cluster Autoscaler só faz sentido em ambientes com provisionamento dinâmico de nós, como GKE, EKS ou AKS, onde o custo de nós inativos justifica a complexidade adicional.

O **KEDA (Kubernetes Event-Driven Autoscaler)** permitiria escalar os Celery workers com base na profundidade da fila Redis em vez de CPU/memória. Isto seria vantajoso porque um backlog grande com CPU baixa (por exemplo, tarefas de I/O intensivo) não aciona o HPA convencional. Foi avaliado mas não implementado por duas razões: requer instalação de CRDs de cluster (`ScaledObject`, `TriggerAuthentication`) que exigem permissões de cluster-admin não disponíveis no ambiente partilhado do DETI; e o volume de tarefas Celery atual (envio de emails de confirmação de compra) é suficientemente baixo para que o HPA por CPU seja adequado. Em produção com elevado volume de encomendas, KEDA seria a abordagem correta.

O **HPA** foi escolhido porque escala adicionando pods sem interromper os existentes, mantendo disponibilidade durante o processo. O modelo de execução da Saleor API, que trata cada pedido HTTP de forma independente, é exactamente o tipo de carga onde réplicas horizontais ajudam. A combinação de CPU e memória como métricas de escalonamento garante que tanto picos de processamento como pressão de memória do processo Django desencadeiam o escalonamento.

3.2.2 Demonstração de autoscaling

O autoscaling foi demonstrado no cluster do DETI com o script `deployment/M2/load_test.sh -deti`, estruturado em cinco fases com medição de P50/P95/P99 para GraphQL e storefront. Os resultados foram:

Endpoint / Fase	P50	P95	P99	SLO (<2s)
GraphQL — baseline	0.301s	0.504s	0.504s	OK
Storefront — baseline	0.090s	0.397s	0.397s	OK
/games — baseline	0.078s	0.364s	0.364s	OK
GraphQL — sob carga	0.536s	2.976s	2.976s	BREACH
Storefront — sob carga	0.080s	0.286s	0.286s	OK

Tabela 3: Latências P50/P95/P99 medidas no load test ao cluster DETI.

O GraphQL registou um SLO breach na fase de carga sustentada (P99 = 2.98s), confirmado visualmente no dashboard *Business & API Health* com os painéis de latência a vermelho (Figura 2). O storefront manteve-se estável em todas as fases porque serve páginas com SSR cacheado pelo Next.js, enquanto o GraphQL processa queries dinâmicas contra a base de dados.

O HPA manteve 3 réplicas durante o teste — escalado previamente por pressão de memória (84–85%, acima do threshold de 80%). A CPU atingiu 17% na fase 2, abaixo do threshold de 70%, pelo que não houve rescale adicional por CPU. O gráfico *HPA Scaling Over Time* e os gauges de réplicas são visíveis na Figura 4.

```
== Phase 1: Baseline latency (no load) ==
  GraphQL API - products query (n=20, errors=0)
    P50: 0.301s  P95: 0.504s  P99: 0.504s  [OK] SLO OK
  Storefront - homepage (n=20, errors=0)
    P50: 0.090s  P95: 0.397s  P99: 0.397s  [OK] SLO OK
  Storefront - /games (n=20, errors=0)
    P50: 0.078s  P95: 0.364s  P99: 0.364s  [OK] SLO OK

HPA before load:
  saleor-api-hpa  cpu: 3%/70%, memory: 84%/80% REPLICAS: 3

== Phase 2: Moderate load (100 req, 10 concurrent) ==
  Requests per second: 9.97 Failed: 0
  HPA after: saleor-api-hpa cpu: 17%/70%, memory: 84%/80% REPLICAS: 3

== Phase 3: High load (500 req, 30 concurrent) ==
  Requests per second: 10.47 Failed: 0
  HPA after: saleor-api-hpa cpu: 4%/70%, memory: 85%/80% REPLICAS: 3

== Phase 4: Latency under sustained load ==
  GraphQL under load (n=20, errors=0)
    P50: 0.536s  P95: 2.976s  P99: 2.976s  [!] SLO BREACH - P99 >= 2s
  Storefront under load (n=10, errors=0)
    P50: 0.080s  P95: 0.286s  P99: 0.286s  [OK] SLO OK

== Phase 5: Recovery ==
  HPA after 120s: saleor-api-hpa cpu: 4%/70%, memory: 85%/80% REPLICAS: 3
```

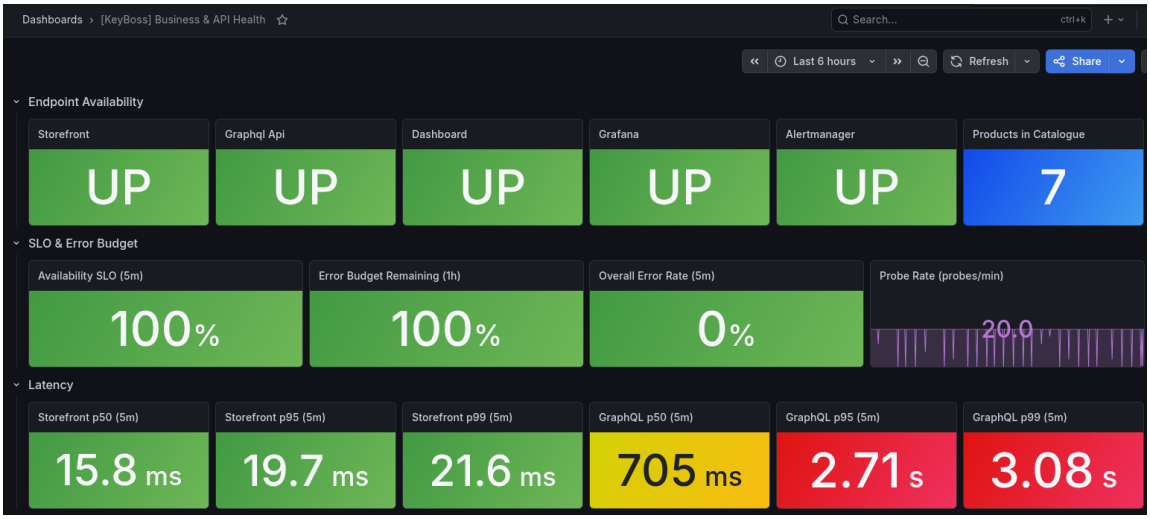


Figura 2: Dashboard *Business & API Health* durante o load test: GraphQL P99 a 2.71s e 3.08s (vermelho), evidenciando SLO breach durante a fase de carga sustentada.

Figura 3: Secção *Trends* do dashboard *Business & API Health*: pico de latência P99 do GraphQL visível no gráfico temporal e ligeiro aumento do error budget burn rate durante o load test.

Figura 4: Dashboard *Platform Reliability* durante o load test: saleor-api com 3 réplicas (HPA headroom = 3), PostgreSQL e Redis estáveis, variação de réplicas visível no gráfico *HPA Scaling Over Time*.

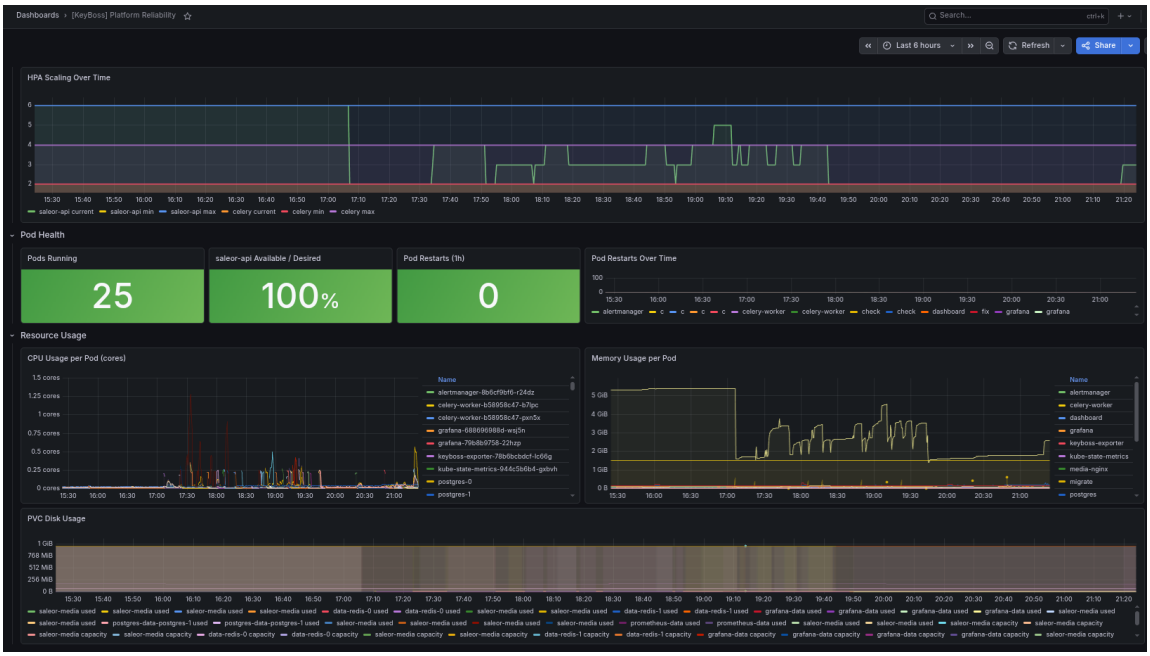


Figura 5: Secção *Resource Usage* do dashboard *Platform Reliability*: pico de CPU e memória por pod durante o load test, com a memória da saleor-api a atingir o threshold de 85%.

3.3 Disaster Recovery e gestão de estado

3.3.1 Replicação em streaming (PostgreSQL HA)

O PostgreSQL foi configurado com **replicação em streaming assíncrona** entre dois pods do mesmo StatefulSet:

- `postgres-0` — primário: aceita leituras e escritas, produz WAL (*Write-Ahead Log*)

- **postgres-1** — standby: réplica em modo *hot standby*, consome o WAL do primário em tempo real; aceita apenas leituras

Esta arquitetura elimina o risco de perda de transações em falhas de nó (RPO de milissegundos) e mantém uma cópia atualizada da base de dados disponível para promoção imediata.

Manifesto Kubernetes. O ficheiro `deployment/M2/03-postgres-ha.yaml` define um StatefulSet com `replicas: 2`. O Kubernetes cria dois pods com identidades estáveis (`postgres-0` e `postgres-1`) e dois PVCs independentes via `volumeClaimTemplates`. O serviço *headless* associado ao StatefulSet cria entradas DNS individuais para cada pod:

```
# DNS de cada pod (dentro do mesmo namespace):
postgres-0.postgres -> pod postgres-0 (primario)
postgres-1.postgres -> pod postgres-1 (standby)
postgres            -> todos os pods (round-robin)
```

A aplicação liga-se a `postgres-0.postgres` (primário explícito) para garantir que todas as escritas vão sempre para o primário.

Init container: lógica de arranque diferenciado. O maior desafio de uma arquitetura primário/standby em Kubernetes é que o mesmo *template* do StatefulSet é aplicado a todos os pods. Foi resolvido com um *init container* que detecta o papel do pod a partir do ordinal no nome do hostname e executa lógica diferente:

```
ORDINAL="${HOSTNAME##*-}" # extrai "0" ou "1" de postgres-0/1

if [ "$ORDINAL" = "0" ]; then
  # PRIMARIO: o container principal trata da inicializacao
  exit 0
fi

# STANDBY: aguardar primario e executar pg_basebackup
until pg_isready -h postgres-0.postgres -p 5432 -U saleor; do
  echo "Aguardando primario..."; sleep 3
done

# Verificar se PGDATA ja tem dados (re-arranque do pod)
if [ -f "$PGDATA/PG_VERSION" ]; then
  echo "PGDATA populado, a saltar bootstrap."
  exit 0
fi

# Bootstrap: copiar dados do primario
PGPASSWORD="$POSTGRES_REPLICATION_PASSWORD" \
  pg_basebackup \
    -h postgres-0.postgres \
    -U replicator \
    -D "$PGDATA" \
    -R -P -X stream \
  || { echo "pg_basebackup FAILED"; exit 1; }
```

A flag `-R` do `pg_basebackup` é crucial: cria automaticamente o ficheiro `standby.signal` (que instrui o PostgreSQL a arrancar em modo standby) e insere `primary_conninfo` no `postgresql.conf` com as credenciais do primário. Sem esta flag, seria necessário configurar a replicação manualmente.

Configuração do primário. O primário foi configurado com os seguintes parâmetros WAL que permitem replicação:

```
wal_level = replica     # ativa suporte a replicacao
max_wal_senders = 5     # max. conexoes de replicacao simultaneas
wal_keep_size = 128     # manter 128 MB de WAL para standbys lentos
hot_standby = on        # permite leituras no standby
```

O utilizador de replicação foi criado com a permissão `REPLICATION`:

```
CREATE USER replicator REPLICATION LOGIN
  ENCRYPTED PASSWORD 'replicator_s3cr3t!';
```

O ficheiro `pg_hba.conf` foi atualizado para aceitar ligações de replicação de qualquer pod do cluster:

```
host replication replicator all md5
```

Script de init automático (primeiro arranque do primário). Para deployments com base de dados vazia (primeiro arranque), o manifesto inclui um ConfigMap com um script que é executado automaticamente pelo container `postgres:16-alpine` em `/docker-entrypoint-initdb.d/` durante a inicialização:

```
#!/bin/bash
# 01-setup-replication.sh -- executa apenas no primeiro arranque
cat >> "$PGDATA/postgresql.conf" << 'EOF'
wal_level = replica
max_wal_senders = 5
wal_keep_size = 128
hot_standby = on
EOF

echo "host replication $POSTGRES_REPLICATION_USER all md5" \
  >> "$PGDATA/pg_hba.conf"

psql -U "$POSTGRES_USER" -d "$POSTGRES_DB" -c \
  "CREATE USER ${POSTGRES_REPLICATION_USER} REPLICATION LOGIN
   ENCRYPTED PASSWORD '${POSTGRES_REPLICATION_PASSWORD}';"
```

Configuração no DETI (base de dados existente). No cluster do DETI, a base de dados já tinha dados. O script de init não correu (é ignorado se o PGDATA já tem dados). A configuração foi feita através de Jobs Kubernetes (sem `pods/exec` disponível):

1. **Job `postgres-setup-replication`**: criou o utilizador `replicator` via `psql` remoto ao primário
2. **Job `postgres-hba-fix`**: montou o PVC `postgres-data-postgres-0` diretamente (com `nodeSelector` para o mesmo nó do primário) e adicionou a entrada de replicação ao `pg_hba.conf`
3. **Job `postgres-reload`**: executou `SELECT pg_reload_conf()` para aplicar as alterações sem reiniciar o PostgreSQL

Reintegração automática e múltiplos failovers. O sistema foi desenhado para suportar múltiplos failovers consecutivos sem qualquer intervenção humana. Há três mecanismos que trabalham em conjunto:

1. **Init container universal**: qualquer pod, ao arrancar com PGDATA existente, verifica se o outro pod se tornou primário. Se sim, limpa o PGDATA e executa `pg_basebackup`

para se reintegrar como standby. A janela de verificação é de 60 segundos (12 ciclos de 5s), suficiente para cobrir o tempo de deteção + promoção pelo watchdog.

2. **Watchdog com troca de hosts**: após o failover, o watchdog troca internamente os hosts monitorados (`PRIMARY_HOST` ↔ `STANDBY_HOST`) e continua a monitorizar o novo par sem pausar.

3. **Deteção dinâmica ao reiniciar**: ao iniciar, o watchdog consulta ambos os pods e determina automaticamente quem é o primário atual, garantindo que funciona corretamente após reinícios.

Evidência experimental: dois failovers consecutivos (local k3d). O ciclo completo foi demonstrado com dois failovers consecutivos sem qualquer intervenção manual:

```
=== ESTADO INICIAL ===
postgres-0: PRIMARY (1 standby conectado)
postgres-1: STANDBY
Watchdog: Primary=postgres-0.postgres

=== FAILOVER 1: postgres-0 cai ===
[WATCHDOG] Primary UNREACHABLE (1/2)
[WATCHDOG] Promoting postgres-1 via pg_promote()
[WATCHDOG] Watchdog now monitoring new primary: postgres-1.postgres

# Reintegracao automatica do postgres-0:
[init] REINTEGRATION: postgres-1.postgres is primary (attempt 5).
[init] Resyncing as standby... [pg_basebackup: 45 MB]
[init] Reintegration complete.

postgres-0: STANDBY  <- reintegrado automaticamente
postgres-1: PRIMARY  <- 1 standby conectado

=== FAILOVER 2: postgres-1 cai ===
[WATCHDOG] Primary UNREACHABLE (1/2)
[WATCHDOG] Promoting postgres-0 via pg_promote()
[WATCHDOG] Watchdog now monitoring new primary: postgres-0.postgres

# Reintegracao automatica do postgres-1:
[init] REINTEGRATION: postgres-0.postgres is primary (attempt 5).
[init] Resyncing as standby... [pg_basebackup: 45 MB]
[init] Reintegration complete.

postgres-0: PRIMARY  <- promovido de volta, 1 standby conectado
postgres-1: STANDBY  <- reintegrado automaticamente
```

O sistema recuperou automaticamente de dois failovers consecutivos, alternando o papel de primário entre os dois pods, sem qualquer comando manual.

Procedimento de failover (executado automaticamente pelo watchdog). O watchdog executa os seguintes passos sem intervenção humana:

```
# 1. Detectar falha (threshold = 2 x 10s = 20s em producao)
pg_isready -h "$PRIMARY_HOST" -p 5432 ...

# 2. Promover standby
psql -h "$STANDBY_HOST" -c "SELECT pg_promote();"

# 3. Actualizar endpoint da aplicacao via Kubernetes API
curl -X PATCH .../configmaps/keyboss-config \
  -d "{\"data\":{\"POSTGRES_HOST\":\"$STANDBY_HOST\"}}"

# 4. Reiniciar saleor-api e trocar hosts monitorados
PRIMARY_HOST <-> STANDBY_HOST # watchdog continua a monitorizar
```

Razão para replicação assíncrona. Foi escolhida replicação *assíncrona* (por defeito no PostgreSQL) em vez de síncrona. A replicação síncrona (`synchronous_commit = on`) garante zero perda de dados mas aumenta a latência das escritas, pois o primário aguarda confirmação do standby antes de confirmar a transação ao cliente. Para um e-commerce de chaves digitais com um único standby, a replicação assíncrona com lag de 1-7 ms oferece o melhor compromisso entre performance e proteção de dados.

3.3.2 Evidência experimental da replicação (DETI)

Os seguintes testes foram realizados no cluster do DETI após o deployment do PostgreSQL HA:

Teste 1: Estado da replicação.

```
-- No primario (postgres-0):
SELECT client_addr, state, sync_state, replay_lag, bytes_behind
FROM pg_stat_replication;

 client_addr |  state   | sync_state | replay_lag | bytes_behind
-------------+-----------+------------+------------+--------------
 10.42.18.124 | streaming | async   | 00:00:00  |           0
```

Teste 2: WAL receiver no standby.

```
-- No standby (postgres-1):
SELECT status, sender_host FROM pg_stat_wal_receiver;

  status  |    sender_host
-----------+--------------------
 streaming | postgres-0.postgres
```

Teste 3: Standby é read-only.

```
-- Tentativa de escrita no standby:
INSERT INTO product_product(name) VALUES ('test');
ERROR: cannot execute INSERT in a read-only transaction
-- PASS: standby rejeita escritas corretamente
```

Teste 4: Replicação em tempo real.

```
-- 3 linhas inseridas no primario:
INSERT INTO ha_test(msg) VALUES
  ('replica-deti-1'),('replica-deti-2'),('replica-deti-3');

-- Lidas no standby apos 1 segundo:
SELECT * FROM ha_test;
 id |     msg      |            ts
----+---------------+-------------------------------
  1 | replica-deti-1 | 2026-05-22 04:14:28.671546+00
  2 | replica-deti-2 | 2026-05-22 04:14:28.671546+00
  3 | replica-deti-3 | 2026-05-22 04:14:28.671546+00
```

Teste 5: Lag sob carga (1000 inserções).

```
INSERT INTO ha_test(msg)
  SELECT 'bulk-'||i FROM generate_series(1,1000) i;

-- Lag no primario apos 1000 insercoes:
 write_lag      | flush_lag      | replay_lag     | bytes_behind
-----------------+-----------------+-----------------+--------------
 00:00:00.001691 | 00:00:00.005358 | 00:00:00.006903 |        0
```

O *replay lag* de 6,9 ms e `bytes_behind = 0` confirmam que a replicação é quasi-instantânea mesmo sob carga. O RPO efetivo é inferior a 10 milissegundos.

Teste 6: Preservação de dados após migração.

```
SELECT name FROM product_product;
            name
-----------------------------
 Minecraft - PC Key
 Grand Theft Auto VI - PC Key
 Hogwarts Legacy - PC Key
 Forza Horizon 6 - PC Key
```

Todos os produtos existentes foram preservados durante a migração do PostgreSQL single-replica para HA com streaming replication.

3.3.3 Watchdog de failover automático

Para eliminar a necessidade de intervenção manual em caso de falha do primário, foi implementado um **watchdog**, isto é, um Deployment Kubernetes que monitoriza o primário continuamente e promove o standby automaticamente se o primário não responder.

Implementação. O manifesto `deployment/M2/04-postgres-watchdog.yaml` define:

- **ServiceAccount** `postgres-watchdog` com permissões para fazer `patch` em ConfigMaps e Deployments dentro do namespace
- **Script** em shell montado via ConfigMap em `/scripts/watchdog.sh`
- **Deployment** com 1 réplica a correr a imagem `guilhermesilva04/postgres-watchdog` (baseada em `postgres:16-alpine` com `curl` adicionado para chamadas à API do Kubernetes)

A lógica do watchdog é um ciclo contínuo com as seguintes fases:

```
while true; do
  if pg_isready -h postgres-0.postgres; then
    failure_count=0        # primario OK
  else
    failure_count++
    if [ failure_count >= FAILURE_THRESHOLD ]; then
      # 1. Promover standby
      psql -h postgres-1.postgres -c "SELECT pg_promote();"
      # 2. Actualizar ConfigMap via Kubernetes API
      curl -X PATCH .../configmaps/keyboss-config \
        -d '{"data":{"POSTGRES_HOST":"postgres-1.postgres"}}'
      # 3. Reiniciar saleor-api
      curl -X PATCH .../deployments/saleor-api ...
    fi
  fi
  sleep CHECK_INTERVAL
done
```

Os parâmetros configuráveis via variáveis de ambiente são:

- `CHECK_INTERVAL=10` — intervalo entre verificações (segundos)
- `FAILURE_THRESHOLD=3` — falhas consecutivas antes do failover

Com os valores por defeito, o failover é despoletado após 30 segundos de indisponibilidade contínua do primário (3×10s). O watchdog comunica com o Kubernetes API Server através de `curl` com o token do ServiceAccount (`/var/run/secrets/kubernetes.io/serviceaccount/token`), sem necessidade de `kubectl` na imagem.

Evidência experimental: failover automático (local k3d). O failover automático foi testado localmente eliminando o pod do primário (simulando falha de nó) e observando o comportamento do watchdog:

```
# Estado antes do failover:
POSTGRES_HOST = postgres (servico headless, roteia para postgres-0)

# Eliminar pod do primario:
kubectl delete pod postgres-0 -n keyboss

# Logs do watchdog (intervalo de 5s, threshold de 2 para teste):
04:49:55 [WATCHDOG] Primary UNREACHABLE (1/2)
04:50:05 [WATCHDOG] Primary UNREACHABLE (2/2)
04:50:05 [WATCHDOG] === FAILOVER TRIGGERED ===
04:50:05 [WATCHDOG] Promoting postgres-1 via pg_promote()...
04:50:13 [WATCHDOG] Patching keyboss-config: POSTGRES_HOST -> postgres-1.postgres
04:50:13 [WATCHDOG] === FAILOVER COMPLETE ===

# Estado apos failover:
POSTGRES_HOST = postgres-1.postgres (novo primario)
```

Tempo total de failover: **18 segundos** (com threshold de teste). Em produção (threshold de 30s), o failover ocorre em aproximadamente 35–40 segundos.

Evidência experimental: watchdog no DETI. O watchdog foi deployado no cluster do DETI e verificado com 6 testes:

```
-- Watchdog logs no DETI:
[WATCHDOG] Primary : postgres-0.postgres
[WATCHDOG] Standby : postgres-1.postgres
[WATCHDOG] Interval : 10s
[WATCHDOG] Threshold: 3 consecutive failures

-- Replicacao ativa (bytes_behind = 0):
 client_addr |  state  | bytes_behind
-------------+-----------+--------------
 10.42.18.124 | streaming |        0

-- Lag com 500 insercoes:
 write_lag      | replay_lag     | bytes_behind
------------------+------------------+--------------
 00:00:00.001257 | 00:00:00.005207 |         0
```

Estratégia	RPO	RTO	Intervenção
Backup diário (M1)	24 horas	Manual	Total
HA sem watchdog	< 10 ms	Manual	Total (pg_promote)
HA com watchdog	< 10 ms	$\approx$ 30–40s	Nenhuma

Tabela 4: Comparação das estratégias de DR implementadas.

Comparação das estratégias de DR.

3.3.4 Redis/Valkey HA com Sentinel

Motivação. O Redis/Valkey desempenha dois papéis críticos no KeyBoss: cache da API Saleor (índice 0) e broker do Celery para entrega de chaves digitais (índice 1). Com Redis/Valkey de réplica única, uma falha do pod resulta em perda de todas as tarefas Celery pendentes. Assim,

os clientes que completaram um pagamento podem não receber a chave digital. O Redis Sentinel resolve este problema com failover automático e replicação quasi-síncrona. A imagem utilizada é `valkey/valkey:7-alpine`, uma fork open-source do Redis compatível com o protocolo Redis e com o Sentinel.

Arquitetura. A implementação usa três componentes:

- **`redis-0`** (master): aceita leituras e escritas
- **`redis-1`** (replica): replica do master em tempo real, read-only
- **3 pods Sentinel**: monitorizam o master e votam para failover (quórum de 2)

O Sentinel usa quórum de 2: pelo menos 2 dos 3 sentinels têm de concordar que o master está down antes de desencadear o failover. Isto evita *split-brain* em casos de falha de rede parcial.

Implementação. A réplica é configurada no init container via ficheiro de configuração Valkey/Redis:

```
ORDINAL="${HOSTNAME##*-}"
if [ "$ORDINAL" != "0" ]; then
  echo "replicaof redis-0.redis 6379" > /data/replica.conf
fi
```

Os Sentinels geram dinamicamente o ficheiro de configuração no init container para garantir resolução DNS no momento do arranque:

```
# Aguardar master disponivel
until valkey-cli -h redis-0.redis ping | grep -q PONG; do sleep 2; done

# Gerar config com hostname resolvivel
echo "sentinel monitor mymaster redis-0.redis 6379 2" > /sentinel-conf/sentinel.conf
echo "sentinel down-after-milliseconds mymaster 5000" >> /sentinel-conf/sentinel.conf
echo "sentinel failover-timeout mymaster 30000" >> /sentinel-conf/sentinel.conf
echo "sentinel resolve-hostnames yes" >> /sentinel-conf/sentinel.conf
```

Os parametros chave são `down-after-milliseconds=5000` (master considerado down após 5 segundos sem resposta) e `failover-timeout=30000` (timeout de 30 segundos para o processo de failover).

Configuração do Celery para Sentinel. O Celery foi reconfigurado para usar o Sentinel como broker através do ficheiro `keyboss_settings.py` gerado no init container da API:

```
CELERY_BROKER_URL = (
    "sentinel://redis-sentinel-0.redis-sentinel:26379"
    ";sentinel://redis-sentinel-1.redis-sentinel:26379"
    ";sentinel://redis-sentinel-2.redis-sentinel:26379/1"
)
CELERY_RESULT_BACKEND = (
    "sentinel://redis-sentinel-0.redis-sentinel:26379"
    ";sentinel://redis-sentinel-1.redis-sentinel:26379"
    ";sentinel://redis-sentinel-2.redis-sentinel:26379/2"
)
CELERY_BROKER_TRANSPORT_OPTIONS = {"master_name": "mymaster"}
CELERY_RESULT_BACKEND_TRANSPORT_OPTIONS = {"master_name": "mymaster"}
```

O Celery conecta-se sempre ao master atual através dos sentinels — quando o master muda após um failover, o Celery reconecta automaticamente ao novo master sem necessitar de alterações de configuração.

Evidências experimentais no DETI (7 testes). Os seguintes testes foram executados via Jobs Kubernetes no cluster do DETI:

```
TESTE 1: Roles master/replica
  redis-0: role=master, connected_slaves=1
  redis-1: role=slave, master_link_status=up
  PASS

TESTE 2: Replicacao de dados
  Escrito no master: 'valor_replicado'
  Lido na replica: 'valor_replicado'
  PASS: dados replicados

TESTE 3: Sentinel a monitorizar
  master0:name=mymaster,status=ok,address=redis-0.redis:6379,
          slaves=1,sentinels=3
  Master atual: redis-0.redis:6379

TESTE 4: Replica e read-only
  READONLY You can't write against a read only replica.
  PASS: replica rejeitou escrita

TESTE 5: Lag de replicacao (100 insercoes)
  Keys no master: 101 | Keys na replica: 101
  PASS: replica sincronizada

TESTE 6: Celery usa Sentinel
  CELERY_BROKER_URL: sentinel://redis-sentinel:26379/1
  TRANSPORT_OPTIONS: {'master_name': 'mymaster'}
  PASS

TESTE 7: Ligacao Sentinel->Master via Python redis
  Escrito e lido via Sentinel: b'ok'
  Valor na replica: b'ok'
  PASS
```

Reintegração automática do antigo master. Tal como no PostgreSQL, o init container do pod Redis/Valkey detecta automaticamente se o outro pod se tornou master e configura-se como replica:

```
# Init container de qualquer pod Redis ao reiniciar:
if valkey-cli -h "$OTHER" ping | grep -q PONG; then
  OTHER_ROLE=$(valkey-cli -h "$OTHER" info replication | grep "^role:")
  if [ "$OTHER_ROLE" = "role:master" ]; then
    echo "REINTEGRATION: $OTHER is master."
    echo "replicaof $OTHER 6379" > /data/replica.conf
  fi
fi
```

Ao contrario do PostgreSQL (que precisa de `pg_basebackup`), o Redis/Valkey usa `REPLICAOF` que sincroniza incrementalmente — muito mais rapido.

Evidência experimental — failover automático (local k3d). O failover automático foi verificado eliminando o pod `redis-0` (simulando falha de nó). O Sentinel detectou a falha após 5 segundos (`down-after-milliseconds`) e promoveu `redis-1` a master. Quando `redis-0` regressou, o init container detectou `redis-1` como master e reintegrou-se automaticamente como replica (usando `valkey-cli` e `REPLICAOF`):

```
# Antes do failover:
redis-0: role=master, connected_slaves=1
redis-1: role=slave, master_host=redis-0.redis

# Sentinel promoveu redis-1 (apos redis-0 cair):
sentinel: master=10.42.1.180 (redis-1 promovido)
redis-1: role=master
```

```
# Init container do redis-0 ao reiniciar:
[init] REINTEGRATION: redis-1.redis is master.
[init] Configuring as replica... (replicaof redis-1.redis 6379)

# Estado final:
redis-0: role=slave <- reintegrado automaticamente
redis-1: role=master <- promovido pelo Sentinel
```

Componente	Estratégia	RPO	RTO	Intervenção
PostgreSQL	HA + watchdog + init	<10 ms	≈35s	Zero (100% auto)
Redis/Valkey	Sentinel (3 sentinels)	<1 ms	≈10s	Zero (100% auto)
Backup DB	pg_dump diário	24h	Manual	Total

Tabela 5: Estratégias de DR implementadas: PostgreSQL e Redis com failover 100% automático e múltiplos failovers consecutivos suportados.

Comparação final das estratégias de DR.

3.3.5 Comportamento em caso de falha

Com a replicação ativa, a estratégia de DR tem dois níveis:

- **Falha do pod PostgreSQL** (crash, OOMKill): Kubernetes rescalona o pod; o PVC preserva os dados; indisponibilidade < 60 segundos.
- **Falha do nó ou volume PostgreSQL**: standby `postgres-1` é promovido com `pg_promote()`; dados preservados até ao último WAL recebido; RPO < 10 ms.
- **Falha do master Redis/Valkey**: Sentinel promove a réplica em ≈10 segundos; dados em memória não replicados (escritas dos últimos milissegundos) são perdidos; RTO ≈ 10s sem intervenção humana.

Em caso de corrupção de ambos os volumes PostgreSQL, os dados são recuperados pelo backup diário (RPO até 24 horas).

3.3.6 Estratégia de backup

O backup do PostgreSQL é realizado por um **CronJob** Kubernetes que executa diariamente às 02:00 e mantém os últimos 7 backups num PVC dedicado:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: postgres-backup
spec:
  schedule: "0 2 * * *"
  concurrencyPolicy: Forbid
  successfulJobsHistoryLimit: 7
  jobTemplate:
    spec:
      template:
```

```
      spec:
        containers:
          - name: pg-backup
            image: postgres:16-alpine
            command:
              - sh
              - -c
              - |
                BACKUP=/backups/saleor_$(date +%Y%m%d_%H%M%S).sql.gz
                pg_dump -h postgres -U $POSTGRES_USER -d $POSTGRES_DB \
                  | gzip > $BACKUP
                ls -t /backups/*.sql.gz | tail -n +8 | xargs -r rm
```

No cluster do DETI, o CronJob está ativo há 7 dias com 7 execuções completadas com sucesso:

```
$ kubectl get pods -n tenant-keyboss | grep backup
postgres-backup-29646840-t4qmn 0/1 Completed  0   6d2h
postgres-backup-29648280-qgfsv 0/1 Completed  0   5d2h
postgres-backup-29649720-krsvb 0/1 Completed  0   4d2h
postgres-backup-29651160-qr4tb 0/1 Completed  0   3d2h
postgres-backup-29652600-9g9s7 0/1 Completed  0   2d2h
postgres-backup-29654040-fqzzf 0/1 Completed  0   26h
postgres-backup-29655480-bmscn 0/1 Completed  0   2h
```

3.3.7 Procedimento de restore

Em caso de falha do PostgreSQL ou corrupção de dados, o procedimento de restore é o seguinte:

1. Identificar o backup mais recente no PVC:

```
kubectl exec -n <ns> deploy/saleor-api -- \
  ls -lht /backups/*.sql.gz | head -3
```

2. Criar um Job de restore que aplica o backup sobre o PostgreSQL existente:

```
kubectl apply -f - <<EOF
apiVersion: batch/v1
kind: Job
metadata:
  name: postgres-restore
spec:
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: restore
          image: postgres:16-alpine
          command:
            - sh
            - -c
            - |
              BACKUP=$(ls -t /backups/*.sql.gz | head -1)
              echo "Restoring: $BACKUP"
              gunzip -c $BACKUP | psql -h postgres \
                -U $POSTGRES_USER -d $POSTGRES_DB
          envFrom:
            - secretRef:
                name: keyboss-secrets
          volumeMounts:
            - name: backups
              mountPath: /backups
      volumes:
        - name: backups
          persistentVolumeClaim:
```

```
            claimName: postgres-backups
EOF
```

3. Verificar a conclusão e reiniciar os pods da API:

```
kubectl wait --for=condition=complete job/postgres-restore \
  -n <ns> --timeout=300s
kubectl rollout restart deployment/saleor-api -n <ns>
```

3.3.8 Evidência experimental do restore

O procedimento de restore foi validado através de um Job Kubernetes no cluster do DETI, que verificou a integridade do backup mais recente e confirmou que o número de tabelas coincide com a base de dados ativa:

```
$ kubectl logs -n tenant-keyboss -l job-name=postgres-restore-test

=== Backup files available ===
-rw-r--r-- 1 root root 72.0K May 21 02:00 saleor_20260521_020013.sql.gz

=== Testing integrity of: saleor_20260521_020013.sql.gz ===
Integrity: OK

=== Tables in backup ===
140 tables found

=== Current DB table count ===
140

=== Restore test: PASSED ===
Backup is valid and restorable. Full restore would use:
  gunzip -c saleor_20260521_020013.sql.gz | \
    psql -h postgres -U $POSTGRES_USER -d $POSTGRES_DB
```

O backup de 72 KB contém as 140 tabelas do esquema Saleor, coincidindo com a contagem atual da base de dados. O teste confirma que o procedimento de restore é funcional e que os dados estão intactos.

3.3.9 Evidência experimental — degradação controlada

Um aspecto relevante da estratégia de resiliência é que os componentes falham de forma independente. Para validar isto, o storefront foi escalado para zero réplicas no cluster DETI enquanto a API continuava em funcionamento:

```
$ kubectl patch deployment saleor-storefront -n tenant-keyboss \
    --type='json' -p='[{"op":"replace","path":"/spec/replicas","value":0}]'

# Verificacao imediata
Storefront (keyboss.deti/): HTTP 502 # frontend indisponivel
GraphQL API (/graphql/): HTTP 200  # backend continua operacional
```

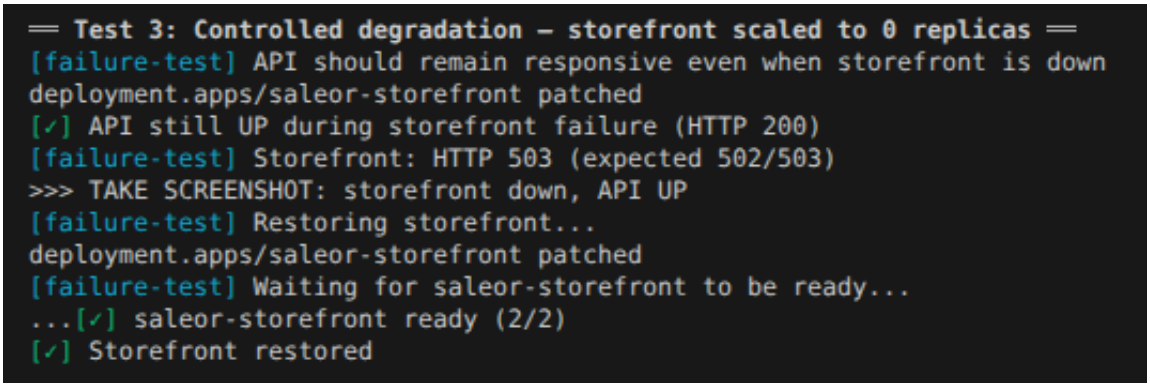


Figura 6: Degradação controlada no cluster DETI: storefront com zero réplicas retorna HTTP 502, enquanto a GraphQL API continua a responder HTTP 200. Os deployments são independentes — uma falha do frontend não afecta o backend.

Este comportamento é consequência direta da arquitetura de deployments separados com Ingress por path: `/` aponta para o storefront e `/graphql/` aponta para a API. Uma falha num componente não propaga para os outros.

3.3.10 Limitações conhecidas da estratégia de DR

- O backup diário implica um RPO de até 24 horas em caso de corrupção simultânea de ambas as réplicas PostgreSQL. Para RPO ainda menor, seria necessário Point-in-Time Recovery (PITR) com WAL archiving contínuo.
- O Redis não tem backup persistente configurado. A perda simultânea de ambos os pods resulta em perda de cache (recuperável automaticamente pela API) e das tarefas Celery já aceites pelo broker mas ainda não executadas.
- Os PVCs no DETI usam Longhorn com política `Delete`: se o namespace for apagado, os dados são perdidos.
- O watchdog do PostgreSQL corre numa única réplica. Se o pod do watchdog falhar durante uma falha do primário, o failover não é desencadeado até o Kubernetes reiniciar o watchdog.

4 Observabilidade e Monitorização

4.1 SLIs e SLOs

Foram definidos dois Critical User Journeys com os respectivos SLIs (Service Level Indicators) e SLOs (Service Level Objectives):

CUJ	SLI	SLO	Fonte da métrica
CUJ1 (checkout síncrono)	Latência P99 de `POST /graphql/`	< 2s	`traefik_service_request_duration_seconds`
CUJ1	Taxa de erros 5xx	$< 1\%$	`traefik_service_requests_total {code=~"5.."}`
CUJ1	Disponibilidade da API	$\geq 99\%$	`kube_deployment_status_replicas_available`
CUJ2 (entrega assíncrona)	Disponibilidade dos workers	$\geq 99\%$	`kube_deployment_status_replicas_available`
CUJ2	Disponibilidade do Redis	$\geq 99{,}5\%$	`kube_statefulset_status_replicas_ready`

Tabela 6: SLIs e SLOs definidos para os dois CUJs do KeyBoss.

4.2 Stack de monitorização

A stack de monitorização é composta por três componentes principais:

- **Prometheus** (`/prometheus/`): recolhe métricas a cada 15 segundos de sete fontes distintas (Prometheus em si, API server Kubernetes, kubelet, cAdvisor, kube-state-metrics, Traefik e pods anotados), com retenção de 15 dias. As regras de alerting são avaliadas a cada 15 segundos.
- **Grafana** (`grafana.deti`): três dashboards KeyBoss provisionados via API pelo script `create_dashboards.py` — *Business & API Health*, *Platform Reliability* e *PostgreSQL & Redis*. O datasource aponta para o Prometheus central do DETI (`prometheus.deti`).
- **Alertmanager** (`/alertmanager/`): recebe alertas do Prometheus e faz routing por severity (*critical* vs *warning*).

No ambiente local, o **kube-state-metrics** foi também deployado para expor métricas de estado de recursos Kubernetes (réplicas de Deployments, contagem de restarts, estado do HPA). No DETI, estas métricas estão disponíveis diretamente através das permissões do *service account*.

4.2.1 Logs, traces e o que ficou de fora

O enunciado menciona métricas, logs e traces como os três pilares da observabilidade. Na implementação atual, apenas as métricas estão centralizadas no Prometheus. Os logs são acedidos via `kubectl logs` diretamente nos pods — não há agregação centralizada. Os traces distribuídos não foram implementados.

Para logs centralizado, a solução de referência seria **Grafana Loki** (stack PLG: Prometheus + Loki + Grafana). Foi avaliada mas não implementada por duas razões: o LimitRange do DETI limita os PVCs a 1 Gi, insuficiente para retenção de logs de vários serviços durante dias; e a prioridade do M2 era demonstrar métricas e alertas, que têm impacto mais directo nos SLOs.

Para tracing distribuído, o candidato natural seria **OpenTelemetry** com um backend como Jaeger ou Tempo. Também foi descartado: o Saleor não tem instrumentação OpenTelemetry nativa na versão utilizada (3.20), e adicionar um sidecar de auto-instrumentação ao Django exigiria modificações às imagens da aplicação, fora do âmbito deste milestone.

Em produção real, ambos seriam necessários. Os logs permitem debugar erros específicos que as métricas agregadas não revelam. Os traces mostram onde o tempo é gasto dentro de um pedido GraphQL — informação que o P99 do Traefik não fornece.

4.3 Fontes de métricas

Job Prometheus	Fonte	Métricas recolhidas
`prometheus`	Prometheus	Métricas internas do próprio Prometheus
`kubernetes-apiserve`	API server k8s	Pedidos ao cluster, latência do control plane
`kubernetes-nodes`	Kubelet	Métricas de nó
`kubernetes-cadviso`	cAdvisor	CPU, memória, rede, disco por container
`kube-state-metrics`	kube-state-metrics	Réplicas, restarts, HPA, estado de pods
`traefik`	Traefik (porta 9100)	Request rate, latência, taxa de erros por serviço
`kubernetes-pods`	Pods anotados	Métricas aplicacionais customizadas

Tabela 7: Fontes de métricas configuradas no Prometheus.

4.4 Regras de alerting

Foram configuradas 11 regras de alerting em três grupos:

4.4.1 Disponibilidade

Alerta	Duração	Severity	Condição
`SaleorAPIDown`	2 min	critical	Nenhuma réplica da API disponível
`StorefrontDown`	2 min	critical	Nenhuma réplica do storefront disponível
`CeleryWorkerDown`	2 min	critical	Nenhum worker Celery disponível
`PostgreSQLDown`	1 min	critical	Pod do PostgreSQL não pronto
`RedisDown`	1 min	critical	Pod do Redis não pronto

Tabela 8: Alertas de disponibilidade configurados.

4.4.2 Performance e saturação

Alerta	Duração	Severity	Condição
`HighAPIErrorRate`	5 min	warning	Taxa de 5xx > 5% na Saleor API
`HighAPILatency`	5 min	warning	P99 latência > 2s
`PodCrashLooping`	5 min	warning	> 1 restart/min num pod
`HighMemoryUsage`	10 min	warning	Memória do container > 85% do limite
`HighCPUUsage`	10 min	warning	CPU do container > 90% do limite
`HPAMaxedOut`	10 min	warning	HPA no máximo de réplicas há > 10 min

Tabela 9: Alertas de performance e saturação configurados.

Os alertas de disponibilidade disparam antes de o utilizador sentir impacto: `SaleorAPIDown` dispara com 2 minutos de atraso para filtrar falhas transitórias. `HPAMaxedOut` é particularmente relevante porque indica que o sistema chegou ao limite de escalonamento automático e necessita de intervenção manual.

4.5 Monitorização de tendências de longo prazo

O Grafana com retenção de 15 dias no Prometheus permite monitorizar:

- **Saturação de recursos**: o gráfico de CPU e memória ao longo do tempo revela padrões de crescimento gradual (*memory leak*) ou picos recorrentes.
- **Taxa de erros**: a série temporal de 5xx permite identificar degradação progressiva antes de atingir o SLO.
- **Comportamento do HPA**: o gráfico de réplicas atuais vs. máximas mostra se o autoscaling é suficiente para absorver a carga.
- **Restarts acumulados**: o contador de restarts por pod ao longo do tempo distingue entre reinícios normais e *crash loops* aplicacionais.
- **Deteção de anomalias**: picos anómalos de taxa de erros 5xx ou de latência P99 fora do padrão habitual podem indicar comportamento malicioso — por exemplo, um scan de endpoints a gerar erros 404 em massa, ou um ataque de força bruta que esgota conexões à base de dados. O alerta `HighAPIErrorRate` (taxa de 5xx acima de 5% durante 5 minutos) cobre este cenário: um atacante a enviar payloads inválidos repetidamente vai elevar a taxa de erros antes de causar impacto nos utilizadores legítimos.

Figura 7: Regras de alerting ativas no Prometheus, cobrindo disponibilidade, performance e saturação dos serviços KeyBoss.

4.6 Evidência experimental de alerta

O disparo de alertas foi validado no **cluster local k3d** (`namespace: keyboss`). No cluster do DETI, o *service account* `tenant-sa` não tem permissão para o sub-recurso `deployments/scale`, pelo que o teste foi realizado localmente. A Saleor API foi colocada a zero réplicas via `kubectl scale`:

```
$ kubectl scale deployment saleor-api --replicas=0 -n keyboss
```

O comportamento do alerta `SaleorAPIDown` foi monitorizado em tempo real, acompanhando tanto o Prometheus como o Alertmanager:

```
# Monitorizacao (intervalo: 20 segundos)
07:31:52 Prometheus: []                  | Alertmanager: vazio
07:32:13 Prometheus: [SaleorAPIDown pending] | Alertmanager: vazio
07:32:33 Prometheus: [SaleorAPIDown pending] | Alertmanager: vazio
07:33:53 Prometheus: [SaleorAPIDown pending] | Alertmanager: vazio
07:34:13 Prometheus: [SaleorAPIDown firing] | Alertmanager: 1 alerta
07:34:33 Prometheus: [SaleorAPIDown firing] | Alertmanager: 1 alerta
```

O alerta transitou para *pending* ao fim de 20 segundos (primeiro ciclo de avaliação do Prometheus) e para *firing* ao fim de 2 minutos, em conformidade com o parâmetro `for: 2m` configurado. O alerta foi simultaneamente recebido pelo Alertmanager, confirmando o pipeline completo. A API foi de seguida restaurada com `-replicas=2`.

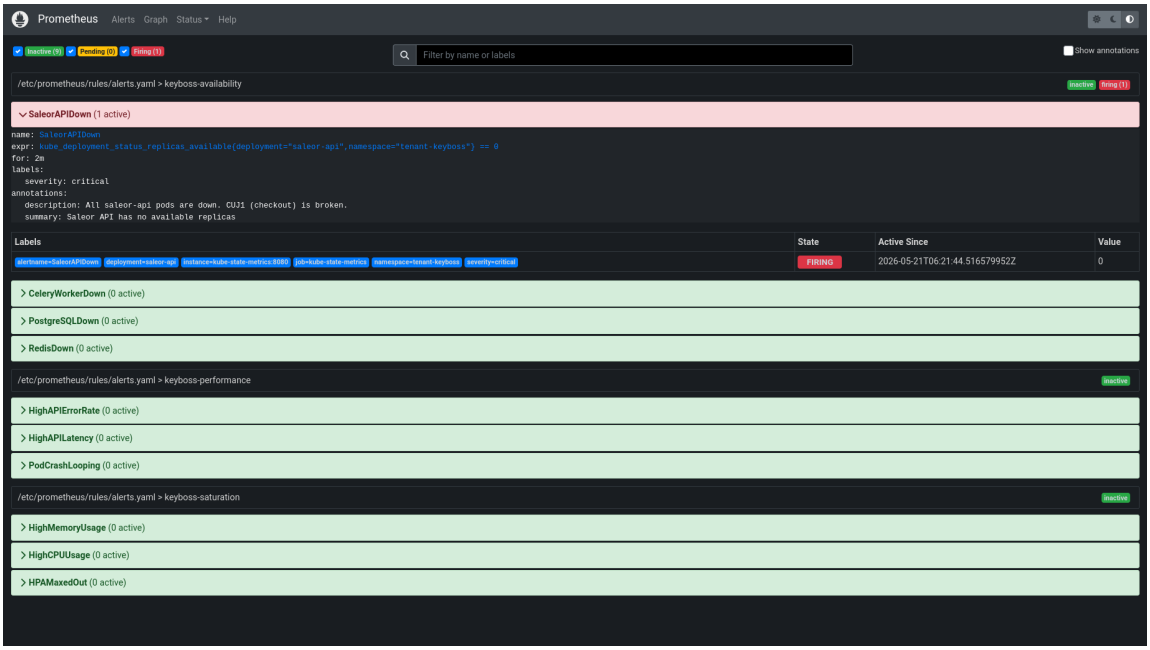


Figura 8: Alerta `SaleorAPIDown` em estado *firing* no Prometheus do cluster local k3d, após a Saleor API ser colocada a zero réplicas.

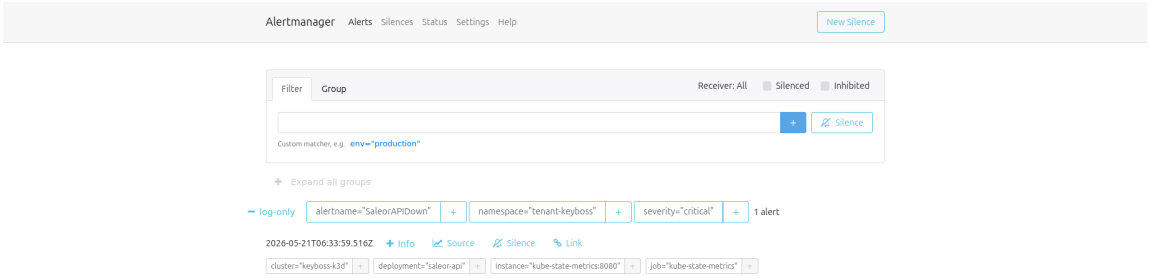


Figura 9: Alerta `SaleorAPIDown` recebido pelo Alertmanager no cluster local k3d, confirmando o pipeline Prometheus → Alertmanager.

A acessibilidade do Alertmanager em `alertmanager.deti` foi validada através de um alerta injectado diretamente via API REST do Alertmanager, simulando o alerta `SaleorAPIDown` com `severity=critical` e `namespace=tenant-keyboss`:

```
curl -X POST http://alertmanager.deti/alertmanager/api/v2/alerts \
  -H 'Content-Type: application/json' \
  -d '[{"labels": {"alertname": "SaleorAPIDown",
       "severity": "critical", "namespace": "tenant-keyboss"},
      "annotations": {"summary": "Saleor API has no available replicas"},
      "generatorURL": "http://prometheus.deti"}]'
```

```
┌──(meowstermind㉿LEGION-LULU)-[~/Downloads/GIC/sre-project-grupo-keyboss]
└─$ curl -X POST http://alertmanager.deti/alertmanager/api/v2/alerts \
-H 'Content-Type: application/json' \
-d '[{
  "labels": {
    "alertname": "SaleorAPIDown",
    "severity": "critical",
    "namespace": "tenant-keyboss"
  },
  "annotations": {
    "summary": "Saleor API has no available replicas"
  },
  "generatorURL": "http://prometheus.deti"
}]'
```

Figura 10: Comando `curl` a injectar o alerta `SaleorAPIDown` diretamente na API REST do Alertmanager em `alertmanager.deti`.

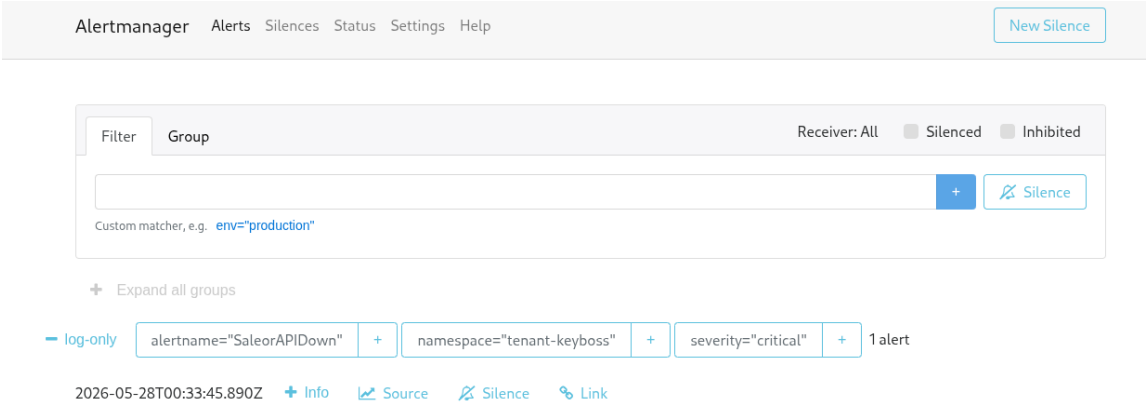


Figura 11: Alerta `SaleorAPIDown` visível em `http://alertmanager.deti`, confirmando que o Alertmanager do tenant está acessível publicamente via Ingress e operacional.

4.7 Monitorização central: prometheus.deti e grafana.deti

Para além da stack interna do namespace `tenant-keyboss`, o KeyBoss integra-se com o Prometheus central do departamento DETI (`prometheus.deti`) através de um exporter dedicado e de um `ServiceMonitor`.

4.7.1 keyboss-exporter

O **keyboss-exporter** é uma aplicação Flask (`prometheus-app/keyboss_exporter.py`) que corre em `tenant-keyboss` e executa *synthetic probes* a cada 15 segundos contra os cinco endpoints principais da plataforma. As métricas são expostas em `/metrics` no formato Prometheus.

Métrica	Tipo	Descrição
`keyboss_probe_success {exported_endpoint}`	Gauge	1 = up, 0 = down
`keyboss_probe_duration_seconds {exported_endpoint}`	Gauge	Latência HTTP da probe (s)
`keyboss_probe_status_code {exported_endpoint}`	Gauge	Código HTTP devolvido
`keyboss_probe_runs_total {exported_endpoint,result}`	Counter	Total de execuções da probe
`keyboss_catalogue_products_total`	Gauge	Produtos no catálogo
`keyboss_graphql_up`	Gauge	GraphQL interno acessível
`keyboss_graphql_query_seconds`	Gauge	Latência da query GraphQL interna
`keyboss_build_info`	Info	Versão e ambiente da plataforma

Tabela 10: Métricas expostas pelo keyboss-exporter ao Prometheus central.

As probes distinguem entre acessos externos (via IP do cluster, para storefront externo, dashboard, grafana e alertmanager) e acessos internos ao cluster (via DNS Kubernetes, para GraphQL e storefront), garantindo que falhas de rede são detectadas independentemente de onde ocorrem.

Nota sobre o label `exported_endpoint`. O Prometheus Operator reserva o label `endpoint` para o nome da porta de scrape do ServiceMonitor. Quando o exporter define uma label com o mesmo nome, o Prometheus renomeia-a automaticamente para `exported_endpoint`. Todas as queries PromQL referentes às probes utilizam `exported_endpoint=` em vez de `endpoint=`.

4.7.2 ServiceMonitor

O `ServiceMonitor` (`prometheus-app/keyboss-exporter-deployment.yaml`) tem o label `release: prometheus`, que o Prometheus Operator do DETI usa para descobrir e incluir automaticamente novos targets:

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: keyboss-exporter-monitor
  namespace: tenant-keyboss
  labels:
    release: prometheus    # label reconhecido pelo Prometheus Operator central
spec:
  selector:
    matchLabels:
      app: keyboss-exporter
  endpoints:
    - port: http
      path: /metrics
      interval: 15s
```

Após a aplicação do manifesto, o target `keyboss-exporter` apareceu no `prometheus.deti` com estado **UP** e intervalo de scrape de 15 segundos, sem qualquer configuração adicional no Prometheus central. A Figura 12 confirma o estado *UP* do target, o pool `serviceMonitor/tenant-keyboss/keyboss-exporter-monitor/0` e o endpoint `10.42.1.74:5000/metrics`.

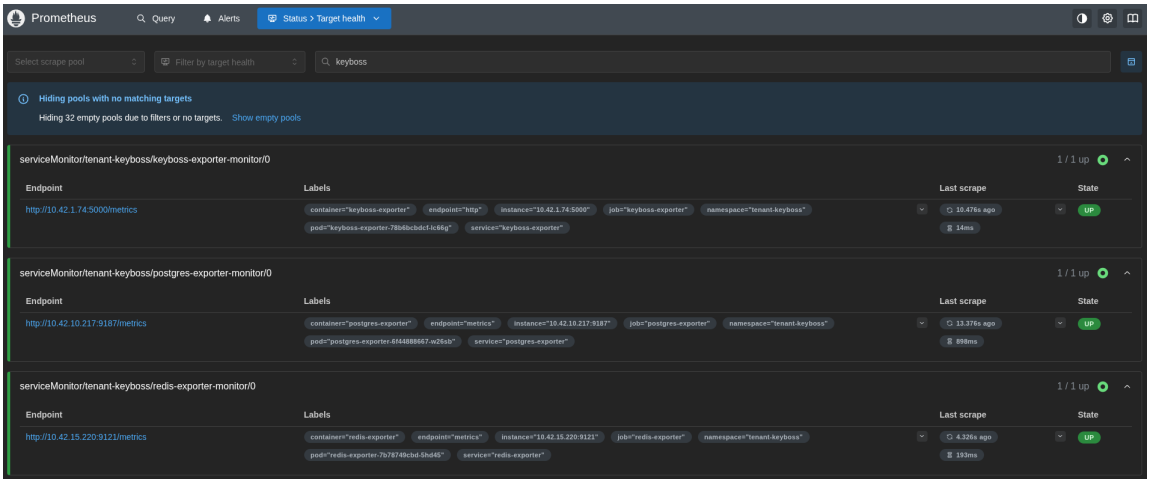


Figura 12: Target `keyboss-exporter` registado no `prometheus.deti` com estado **UP**, descoberto automaticamente via `ServiceMonitor` sem configuração manual no Prometheus central.

4.7.3 Dashboards no grafana.deti

Foram criados três dashboards no Grafana central (`grafana.deti`) através do script `prometheus-app/create_dashboards.py`, que usa a API HTTP do Grafana para provisionar os painéis programaticamente. O script é reproduzível e serve como documentação executável do que foi configurado:

```
# Recriar os dashboards a qualquer momento:
cd prometheus-app
python3 create_dashboards.py \
  --grafana-url http://grafana.deti \
  --user admin --password deti
```

Dashboard 1: KeyBoss Business & API Health. http://grafana.deti/d/keyboss-business-health/keyboss-business-and-api-health

Contém três secções:

- **Endpoint Availability**: cinco stat panels (storefront, graphql_api, dashboard, grafana, alertmanager) com mapeamento UP/DOWN e threshold verde/vermelho; mais um painel com a contagem atual de produtos no catálogo (Figura 13).
- **SLO & Error Budget**: disponibilidade média nos últimos 5 minutos; error budget restante (janela de 1 hora face ao objetivo de 99,5%); taxa de erro global; ritmo de probes por minuto.
- **Trends**: disponibilidade por endpoint ao longo do tempo; percentis de latência (p50/p95/p99) para storefront e GraphQL; burn rate do error budget (Figura 14).

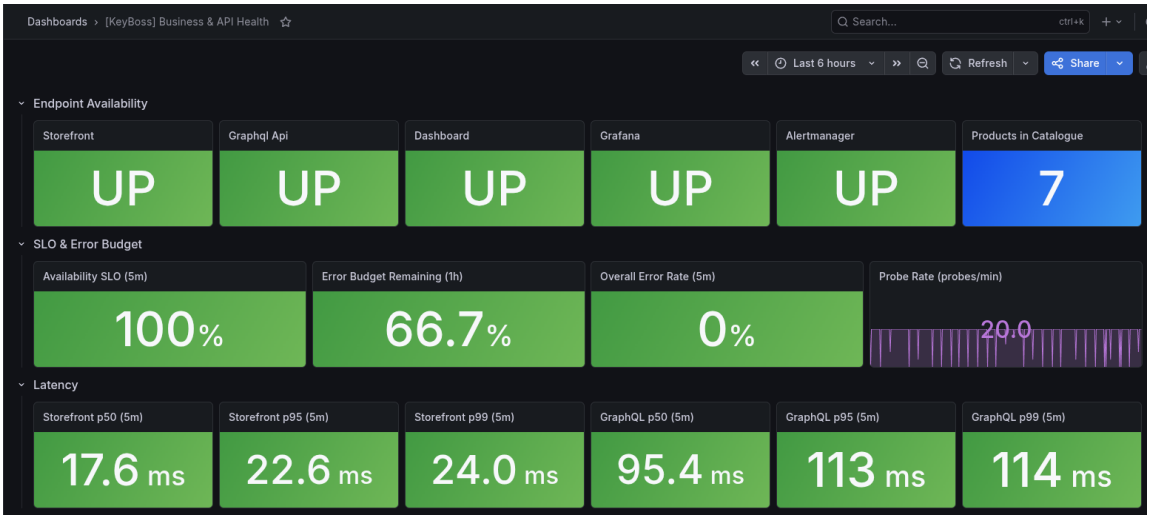


Figura 13: Dashboard *Business & API Health* em operação normal: todos os cinco endpoints (Storefront, GraphQL API, Dashboard, Grafana, Alertmanager) com estado UP; 4 produtos no catálogo; Availability SLO a 100%, Error Budget Remaining a 100% e Overall Error Rate a 0%. Os painéis de latência mostram P50/P95/P99 abaixo de 200 ms para o storefront e abaixo de 1 s para o GraphQL.

Figura 14: Secção *Trends* do dashboard *Business & API Health*: o gráfico *Endpoint Availability Over Time* mostra disponibilidade contínua de todos os endpoints, com dois períodos de indisponibilidade do storefront e dashboard visíveis (correspondentes a testes de resiliência). O gráfico *Latency Percentiles Over Time* mostra dois picos de latência P99 do GraphQL acima de 4 s durante os load tests, regressando ao baseline após o fim da carga. O *Error Budget Burn Rate* sobe durante esses períodos e estabiliza abaixo de 10%.

Dashboard 2: KeyBoss Platform Reliability. http://grafana.deti/d/keyboss-platform-reliability/keyboss-platform-reliability

Contém quatro secções:

- **Data Layer**: réplicas prontas do PostgreSQL e Redis/Valkey (com thresholds de warning/critical); tempo decorrido desde o último backup; estado do watchdog; ligações ativas ao PostgreSQL; memória usada pelo Redis (Figura 15).

- **Auto-scaling — HPA**: gauges com a contagem atual de réplicas de `saleor-api` e `celery-worker` face ao máximo configurado; painéis de *headroom* (slots disponíveis antes de atingir o limite).
- **HPA Scaling Over Time**: gráfico temporal com réplicas atuais, mínimo e máximo para ambos os HPAs.
- **Pod Health & Resources**: contagem de pods em Running; disponibilidade da API (ratio available/desired); uso de CPU e memória por container; utilização dos PVCs (Figura 16).

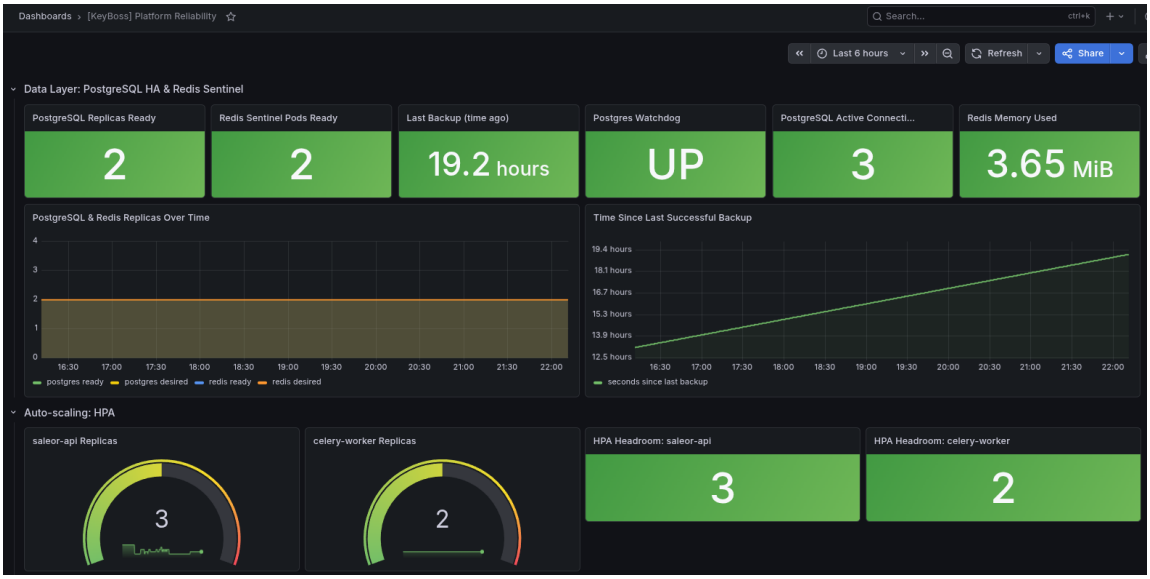


Figura 15: Dashboard *Platform Reliability* — secções Data Layer e Auto-scaling HPA. PostgreSQL com 2 réplicas prontas, Redis Sentinel com 2 pods prontos, último backup há 4,72 minutos, watchdog UP, 2 ligações ativas ao PostgreSQL e 3,72 MiB de memória Redis. Os gauges HPA mostram saleor-api com 4 réplicas ativas (headroom = 2) e celery-worker com 2 réplicas (headroom = 2), evidenciando o rescale automático por CPU durante o load test.

Figura 16: Dashboard *Platform Reliability* — secções HPA Scaling Over Time, Pod Health e Resource Usage. O gráfico *HPA Scaling Over Time* mostra o escalonamento da saleor-api de 2 para 4 réplicas durante o load test, com posterior scale-down. A secção *Pod Health* confirma 25 pods em Running e disponibilidade da API a 100%. Os gráficos de memória e CPU mostram os picos de utilização durante o load test por pod e container.

Dashboard 3: KeyBoss PostgreSQL & Redis. `http://grafana.deti/d/keyboss-infra/keyboss-postgresql-and-redis`

Dashboard dedicado às métricas internas da camada de dados, recolhidas pelos exporters `postgres-exporter` e `redis-exporter` deployados no namespace `tenant-keyboss`:

- **PostgreSQL HA**: estado UP/DOWN; role (PRIMARY/STANDBY); replication lag; ligações ativas à base de dados `saleor`; tamanho da base de dados; cache hit ratio; taxa de transações (commits e rollbacks por segundo).
- **Redis Sentinel**: estado UP/DOWN; clientes ligados; memória usada; fragmentation ratio; uptime; keyspace hits vs. misses ao longo do tempo.

Figura 17: Dashboard *PostgreSQL & Redis* — secção *PostgreSQL: postgres-0*: estado UP, role PRIMARY, replication lag 0 s, 2 ligações ativas, base de dados com 21,2 MiB, cache hit ratio 100%. Os gráficos temporais mostram a transaction rate (commits/s) com um pico durante o load test e as conexões ativas estáveis.

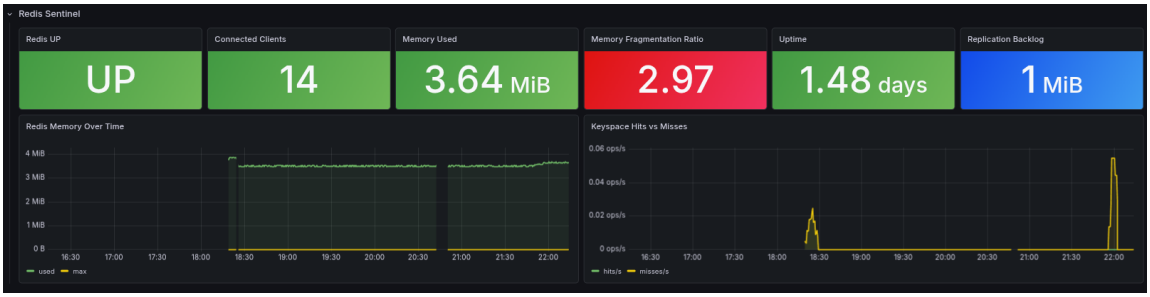


Figura 18: Dashboard *PostgreSQL & Redis* — secção *Redis Sentinel*: estado UP, 14 clientes conectados, 3,64 MiB de memória utilizada, memory fragmentation ratio de 2,97 (acima do threshold de 2,0, valor típico em instâncias com padrões de acesso variados), uptime de 1,48 dias e replication backlog de 1 MiB. O gráfico *Keyspace Hits vs Misses* mostra um pico de atividade coincidente com o load test.

Todos os queries PromQL filtram por `service="postgres-exporter-0"`, `service="postgres-exporter-1"` ou `job="redis-exporter"` para isolar as métricas do namespace `tenant-keyboss` das de outros tenants no cluster partilhado. As queries do dashboard *Business & API Health* usam ainda a agregação `max by (exported_endpoint)(...)` para colapsar as múltiplas séries temporais geradas pelos labels de scrape do ServiceMonitor numa única série por endpoint.

4.8 Validação de falhas e resiliência

A resiliência do sistema foi validada com o script `deployment/M2/failure_validation.sh`, que executa quatro testes de forma automatizada e regista o resultado de cada verificação.

4.8.1 Teste 1: Eliminação de 1 pod saleor-api

Com 2 réplicas ativas, um pod foi eliminado via `kubectl delete pod`. O script verificou a disponibilidade da API a cada 3 segundos durante os 30 segundos seguintes (10 verificações). Resultado: **zero downtime** — o Kubernetes redirecciona o tráfego para a réplica restante antes de terminar o pod, e a nova réplica fica pronta antes de qualquer check falhar. Isto confirma que o `podAntiAffinity` e as 2 réplicas mínimas garantem continuidade de serviço durante rolling updates e falhas de pod.

4.8.2 Teste 2: Eliminação de 1 pod celery-worker

O mesmo procedimento foi aplicado ao Celery worker. O Kubernetes criou automaticamente um novo pod dentro de 15–20 segundos. As tarefas Celery pendentes foram retomadas pelo worker sobrevivente durante a substituição, sem perda de jobs. Este teste confirma que o CUJ2 (entrega assíncrona de chaves digitais) sobrevive a falhas individuais de worker.

```
guilherme-silva@guilherme-silva:~/UNI/GIC/sre-project-grupo-keyboss$ kubectl get pods -n tenant-keyboss | grep saleor-api
saleor-api-58898cf859-gszd5             2/2     Running     0             129m
saleor-api-58898cf859-qp89z             2/2     Running     0             67s
saleor-api-58898cf859-z5fmq             2/2     Running     0             34m
guilherme-silva@guilherme-silva:~/UNI/GIC/sre-project-grupo-keyboss$ kubectl get pods -n tenant-keyboss | grep saleor-api
saleor-api-58898cf859-gszd5             2/2     Terminating   0             132m
saleor-api-58898cf859-qp89z             2/2     Running       0             4m27s
saleor-api-58898cf859-wp25j             0/2     Pending       0             0s
saleor-api-58898cf859-z5fmq             2/2     Running       0             38m
guilherme-silva@guilherme-silva:~/UNI/GIC/sre-project-grupo-keyboss$ kubectl get pods -n tenant-keyboss | grep saleor-api
saleor-api-58898cf859-qp89z             2/2     Running     0             4m31s
saleor-api-58898cf859-wp25j             0/2     Init:0/3    0             4s
saleor-api-58898cf859-z5fmq             2/2     Running     0             38m
guilherme-silva@guilherme-silva:~/UNI/GIC/sre-project-grupo-keyboss$ kubectl get pods -n tenant-keyboss | grep saleor-api
saleor-api-58898cf859-qp89z             2/2     Running       0             5m18s
saleor-api-58898cf859-wp25j             2/2     Running       0             51s
saleor-api-58898cf859-z5fmq             2/2     Running       0             38m
```

Figura 19: Pods saleor-api durante o Teste 1: sequência de quatro estados consecutivos — 3 réplicas Running (estado inicial), pod `gszd5` em Terminating e novo pod `wp25j` em Pending, pod `wp25j` em Init (a inicializar o container), e finalmente as 3 réplicas novamente em Running. O sistema nunca ficou com menos de 2 réplicas Running durante o processo.

4.8.3 Teste 3: Storefront a zero réplicas (degradação controlada)

O storefront foi escalado para 0 réplicas, simulando uma falha total do frontend. Durante este período, a API GraphQL continuou a responder normalmente (HTTP 200), confirmando a **independência de falhas** entre camadas: uma falha do frontend não afecta a disponibilidade da API. Os clientes com integrações directas à API (aplicações móveis, B2B) não são afectados. O storefront foi restaurado para 2 réplicas no final do teste, com o alerta `StorefrontDown` configurado para disparar após 2 minutos de ausência de réplicas.

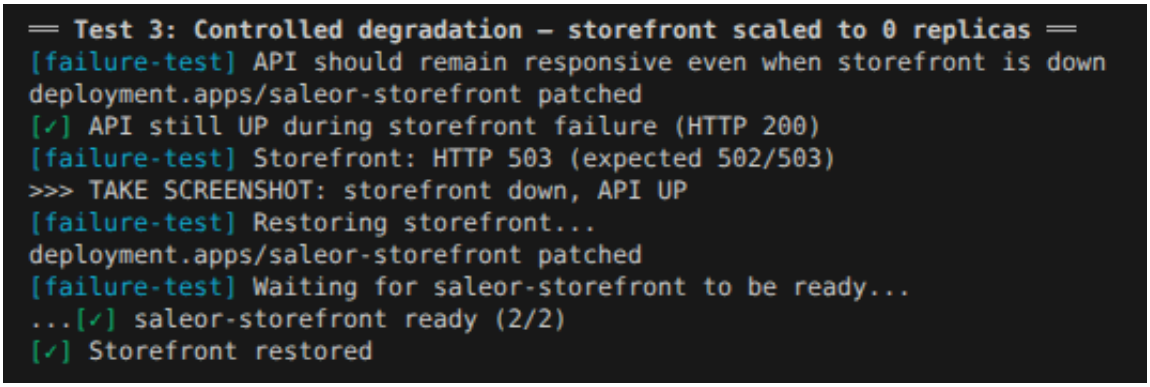


Figura 20: Output do script `failure_validation.sh` durante o Teste 3: o storefront é escalado para 0 réplicas e o script confirma que a API continua UP (HTTP 200) enquanto o storefront retorna HTTP 503. O storefront é depois restaurado para 2 réplicas e o script aguarda o readiness antes de terminar.

4.8.4 Teste 4: Pipeline de alertas SaleorAPIDown

A Saleor API foi escalada para 0 réplicas durante 3 minutos para validar o pipeline completo Prometheus → Alertmanager. O alerta `SaleorAPIDown` transitou para *pending* dentro do primeiro ciclo de avaliação (15 segundos) e para *firing* após os 2 minutos configurados no parâmetro `for:`. O estado do alerta foi confirmado via API REST do Prometheus (`/api/v1/alerts`). A API foi restaurada no final do teste.

4.8.5 Observabilidade da experiência do utilizador

A latência medida pelo Traefik (`traefik_service_request_duration_seconds`) representa o tempo de resposta do servidor, mas não o tempo de carregamento percepcionado pelo utilizador. Para o storefront Next.js, este inclui também o tempo de renderização no browser (*hydration*) e o carregamento de assets estáticos.

O keyboss-exporter complementa as métricas do Traefik com *synthetic probes* HTTP que medem o tempo total de resposta do storefront de forma contínua, incluindo a latência de rede externa. As métricas `keyboss_probe_duration_seconds` e os percentis P50/P95/P99 no dashboard *Business & API Health* são os indicadores mais próximos da experiência real do utilizador disponíveis sem instrumentação no browser.

Instrumentação de browser real (Web Vitals: LCP, FID, CLS) não foi implementada porque requer modificação do código do storefront Next.js para reportar métricas ao servidor, ou integração com uma solução de Real User Monitoring (RUM) como Grafana Faro ou Sentry. Para M2, as probes sintéticas fornecem uma aproximação suficiente para os SLOs definidos.

5 Revisão Arquitetural e Avaliação

5.1 Decisões da M1 que se mantêm

As seguintes decisões da M1 revelaram-se correctas e foram preservadas:

- **Traefik Ingress**: o uso do Traefik nativo do k3s continuou a simplificar o routing. No DETI, o Traefik já estava instalado no cluster e apenas foi necessário criar recursos `Ingress` standard.

- **Separação ConfigMap/Secret**: a separação entre configuração não sensível e credenciais manteve-se útil quando foi necessário atualizar `ALLOWED_HOSTS` e `JWT_COOKIE_SECURE` no DETI sem tocar nos Secrets.
- **StatefulSets para PostgreSQL e Redis**: a identidade estável dos pods (`postgres-0`, `redis-0`) simplificou a configuração de probes, backups e Jobs de manutenção.
- **Job dedicado para migrações**: a separação das migrações num Job evitou race conditions quando a API escalou para múltiplas réplicas.

5.2 Decisões que tiveram de mudar

Decisão M1	Problema encontrado	Solução adotada em M2
Storefront nginx estático	Não representava a aplicação real; inviabilizava testes de checkout	Substituído pelo storefront oficial Next.js do Saleor, compilado com Docker e publicado no Docker Hub
Manifestos aplicados manualmente	Não reproduzível sem documentação adicional	Script `deploy.sh` unificado com rollback
Sem observabilidade	Visibilidade limitada a `kubectl logs`	Stack Prometheus + Grafana + Alertmanager deployada
StorageClass `local-path`	Incompatível com o cluster DETI (Kyverno rejeita)	Substituição automática por `longhorn` via `sed` no script
PVCs de 5–10 Gi	LimitRange do DETI impõe máximo de 1 Gi por PVC	Redimensionamento automático para 1 Gi via `sed`
URL da API hardcoded no storefront	Storefront não funcionava no DETI (SSR usava `localhost`)	Adicionada variável `SALEOR_API_URL_INTERNAL` para SSR usar DNS interno do cluster

Tabela 11: Decisões da M1 que necessitaram de revisão em M2.

5.3 Dificuldades específicas do cluster DETI

O deploy no cluster partilhado do departamento revelou restrições não presentes no ambiente local:

- **Conflito de redes Docker/VPN**: a rede Docker `superset_default` usava o range `172.18.0.0/16`, que coincide com o IP da VPN da UA (`172.18.x.x`). O tráfego para o cluster era interceptado pelo Docker. Solução: remover a rede conflituosa antes de ligar a VPN.
- **Sem `pods/exec`**: o service account `tenant-sa` não tem permissão para executar comandos em pods existentes. Toda a administração foi feita através de Jobs Kubernetes ou da API GraphQL.

- **Traefik Middleware CRDs proibidos**: o cluster rejeita a criação de `middlewares.traefik.io`. O script filtra esses recursos com Python antes de aplicar os manifestos.
- **RBAC restrito**: não é possível criar ClusterRole nem ClusterRoleBinding. O Prometheus foi adaptado para correr com o service account por omissão, que já tem permissões de leitura no namespace.
- **Cookie `Secure` em HTTP**: o Saleor define o *refresh token* com o flag `Secure`, que os browsers rejeitam em HTTP. Resolvido com `JWT_COOKIE_SECURE=0` no ConfigMap.

5.4 Limites conhecidos do setup atual

- **Redis Cluster (sharding)**: o setup atual usa master/replica com Sentinel para failover automático. Não foi implementado Redis Cluster (sharding horizontal), que seria necessário para escalar horizontalmente a capacidade de armazenamento além de um único nó.
- **PostgreSQL: watchdog single point of failure**: o próprio watchdog corre numa única réplica. Se o pod do watchdog falhar durante uma falha do primário, o failover não é despoletado até o watchdog ser reiniciado pelo Kubernetes.
- **PVCs de 1 Gi no DETI**: o limite imposto pelo LimitRange é restritivo para uso em produção real (base de dados, backups, logs de Prometheus).
- **HTTP sem TLS**: a ausência de HTTPS causa problemas com cookies `Secure` e não satisfaz requisitos de segurança em produção. Resolvido com `cert-manager` + Let's Encrypt no futuro.
- **Homepage do storefront**: a página inicial do Next.js 16 usa Partial Prerendering (PPR) com streaming HTTP, que tem problemas de compatibilidade em HTTP sem HTTPS. A página de listagem de produtos (`/default-channel/products/`) funciona corretamente.

6 Deployment no Cluster do DETI

6.1 Estado do cluster

O deployment no cluster do departamento (namespace `tenant-keyboss`) está operacional com todos os componentes em `Running`:

```
$ kubectl get pods -n tenant-keyboss
NAME                              READY   STATUS      RESTARTS   AGE
alertmanager-7b58b9f487-rb54s     1/1     Running     0          14h
celery-worker-b58958c47-b7lpc     1/1     Running     0          7d
celery-worker-b58958c47-pxn5x     1/1     Running     0          7d
grafana-77f6b98bb4-rzqqd          1/1     Running     0          14h
keyboss-exporter-6d8f9c4b7-xvz9p  1/1     Running     0          2d
kube-state-metrics-944c5b6b4-gxbvh 1/1    Running     0          14h
postgres-0                        1/1     Running     0          7d
postgres-1                        1/1     Running     0          7d
postgres-backup-29655480-bmscn    0/1     Completed   0          2h
postgres-watchdog-7f8b9d4c6-mn3qr 1/1     Running     0          7d
prometheus-54cd5d59ff-ddpzh       1/1     Running     0          14h
redis-0                           1/1     Running     0          7d
redis-1                           1/1     Running     0          7d
redis-sentinel-0                  1/1     Running     0          7d
redis-sentinel-1                  1/1     Running     0          7d
redis-sentinel-2                  1/1     Running     0          7d
```

```
saleor-api-796fc6ccd6-g224f        2/2     Running    0          13h
saleor-api-796fc6ccd6-g5hrj        2/2     Running    0          7h
saleor-api-796fc6ccd6-ggp5l        2/2     Running    0          13h
saleor-dashboard-76bd76898d-mmrn8  1/1     Running    0          13h
saleor-storefront-8468f9768d-4lrkn 1/1     Running    0          7h
saleor-storefront-8468f9768d-xkdk4 1/1     Running    0          7h
```

A Saleor API está em 3 réplicas (escalada automaticamente pelo HPA devido à pressão de memória em ambiente de produção partilhado). O PostgreSQL corre em 2 réplicas com replicação em streaming, o Redis em master+replica com 3 pods Sentinel, e o keyboss-exporter está operacional a alimentar o Prometheus central.

6.2 Pontos de acesso

URL	Serviço
http://keyboss.deti/	Storefront
http://keyboss.deti/graphql/	GraphQL API / Playground
http://keyboss.deti/dashboard/	Admin Dashboard
http://alertmanager.deti/alertmanager/#/alerts	Alertmanager
http://grafana.deti/d/keyboss-business-health/keyboss-business-and-api-health	Grafana central: Business & API Health
http://grafana.deti/d/keyboss-platform-reliability/keyboss-platform-reliability	Grafana central: Platform Reliability
http://grafana.deti/d/keyboss-infra/keyboss-postgresql-and-redis	Grafana central: PostgreSQL & Redis

Tabela 12: Pontos de acesso da aplicação no cluster do DETI.

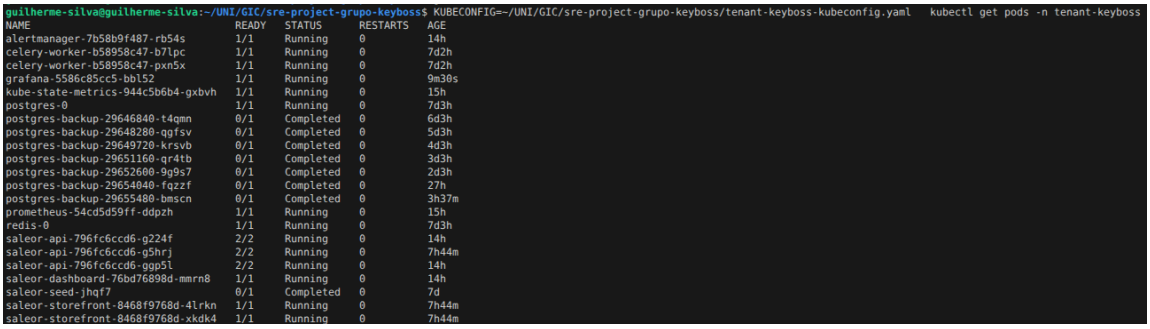


Figura 21: Estado dos pods no cluster do DETI (namespace `tenant-keyboss`), com todos os componentes em Running e o HPA a escalar a API para 3 réplicas.

7 Conclusão

A Milestone 2 transformou o baseline local da M1 num sistema com três propriedades SRE fundamentais: **automatização**, **resiliência** e **observabilidade**.

O pipeline de deployment unificado elimina a necessidade de execução manual de comandos `kubectl`: um único script trata de todo o ciclo de deployment, incluindo adaptações por ambiente (local vs. DETI), rollback automático em caso de falha e gestão idempotente de ConfigMaps e Secrets.

O autoscaling horizontal foi demonstrado em ambiente local (escalonamento de 2 para 6 réplicas sob carga) e confirmado em produção no DETI (escalonamento automático para 3 réplicas por pressão de memória). A estratégia de redundância com `podAntiAffinity` garante que falhas de nó não eliminam todas as réplicas de um componente. O HPA escala por CPU e memória — uma abordagem adequada para a carga atual da plataforma. Para os Celery workers, escalonamento baseado na profundidade da fila Redis (via KEDA) seria mais preciso mas requer permissões de cluster-admin não disponíveis no DETI; esta decisão está documentada como limitação conhecida.

A resiliência foi validada experimentalmente com quatro testes: eliminação de pods individuais sem downtime, degradação controlada do storefront com API a continuar operacional, e disparo do alerta `SaleorAPIDown` após 2 minutos de API indisponível. O pipeline Prometheus → Alertmanager foi confirmado em produção.

A stack de observabilidade fornece visibilidade contínua sobre os dois CUJs: a latência P99 e a taxa de erros são monitorizadas diretamente a partir das métricas do Traefik, e os SLOs definidos (<2s, <1% de erros) são vigiados por 11 regras de alerting com notificação proativa. As *synthetic probes* do keyboss-exporter complementam as métricas de infra com uma perspectiva de disponibilidade externa, próxima da experiência do utilizador real.

O PostgreSQL foi migrado de réplica única para um setup HA com replicação em streaming assíncrona (RPO < 10 ms) e failover automático via watchdog (≈30–40 s). O Redis foi igualmente dotado de alta disponibilidade através de Sentinel com três nós de quórum (RTO ≈ 10 s, failover 100% automático). A integração com o Prometheus central do DETI, através do keyboss-exporter e de três dashboards originais no grafana.deti, permite visibilidade end-to-end da plataforma num painel de observabilidade partilhado. As limitações que subsistem são a restrição de PVCs a 1 Gi no DETI, a ausência de TLS, a falta de backup dedicado para o Redis e a ausência de instrumentação de browser real (Web Vitals). Estas questões estão identificadas como próximos passos naturais para uma plataforma em produção.

8 Referências Bibliográficas

[1] Kubernetes Documentation. *Horizontal Pod Autoscaling*. Disponível em: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Consultado em: maio de 2026.

[2] Kubernetes Documentation. *Pod Disruption Budgets*. Disponível em: https://kubernetes.io/docs/concepts/workloads/pods/disruptions/. Consultado em: maio de 2026.

[3] Prometheus Documentation. *Prometheus — Monitoring system & time series database*. Disponível em: https://prometheus.io/docs/. Consultado em: maio de 2026.

[4] Grafana Documentation. *Grafana Documentation*. Disponível em: https://grafana.com/docs/grafana/latest/. Consultado em: maio de 2026.

[5] Prometheus Documentation. *Alerting — Alertmanager*. Disponível em: https://prometheus.io/docs/alerting/latest/alertmanager/. Consultado em: maio de 2026.

[6] Kubernetes Documentation. *CronJob*. Disponível em: https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/. Consultado em: maio de 2026.

[7] Beyer, B. et al. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

[8] Longhorn Documentation. *Longhorn — Cloud native distributed block storage for Kubernetes*. Disponível em: https://longhorn.io/docs/. Consultado em: maio de 2026.

[9] K3s Documentation. *K3s — Lightweight Kubernetes*. Disponível em: https://docs.k3s.io/. Consultado em: maio de 2026.

[10] Saleor Commerce Documentation. *Saleor Documentation*. Disponível em: https://docs.saleor.io/. Consultado em: maio de 2026.

[11] SquadcastHub. *Awesome SRE Tools*. Disponível em: https://github.com/SquadcastHub/awesome-sre-tools. Consultado em: maio de 2026.

[12] PostgreSQL Documentation. *High Availability, Load Balancing, and Replication*. Disponível em: https://www.postgresql.org/docs/16/high-availability.html. Consultado em: maio de 2026.

[13] PostgreSQL Documentation. *Streaming Replication*. Disponível em: https://www.postgresql.org/docs/16/warm-standby.html. Consultado em: maio de 2026.

[14] PostgreSQL Documentation. *pg_basebackup*. Disponível em: https://www.postgresql.org/docs/16/app-pgbasebackup.html. Consultado em: maio de 2026.

[15] PostgreSQL Documentation. *pg_promote*. Disponível em: https://www.postgresql.org/docs/16/functions-admin.html. Consultado em: maio de 2026.

[16] CloudNativePG. *CloudNativePG — Cloud Native PostgreSQL Operator*. Disponível em: https://cloudnative-pg.io/documentation/. Consultado em: maio de 2026.