

CASE · INFRAESTRUTURA

Migração AWS → VPS

Menos custo e banco mais rápido: como um estudo de viabilidade e medições em cada etapa guiaram a migração da Triodrop.

~80%

redução estimada no custo mensal de infraestrutura

4,9x

menor latência por consulta ao banco vs. AWS

790x

redução do gargalo medido na transição

>98%

do banco não era dado de negócio

Laravel · MySQL · Redis · Docker · EasyPanel · Blue-Green · AWS

Willian Yuji Suguiyama

Tech Lead na YesDev · setembro de 2026

linkedin.com/in/willianyujy · willianyujy.com.br

Resumo

- Estudo de viabilidade técnica e financeira com **redução estimada de cerca de 80%** no custo mensal de infraestrutura.
- Migração da Triodrop (SaaS multi-tenant em Laravel) da AWS em Ohio para uma VPS em São Paulo, com estratégia **Blue-Green** e downtime mínimo.
- Em relação ao ambiente original na AWS: latência por consulta **4,9x menor** (1,71 ms → 0,35 ms) e 100 consultas sequenciais **2,4x mais rápidas** (217 ms → 91 ms).
- Um gargalo de **277 ms por consulta** apareceu na transição, foi medido, diagnosticado e eliminado (**790x** de redução).
- Mais de **98% do banco** não era dado de negócio. Foram migrados só os dados que importavam.

1. Contexto

A **Triodrop** é uma plataforma SaaS multi-tenant de dropshipping e marketplace, construída em Laravel, com 100 lojas ativas e integrações em tempo real com Shopee, Mercado Livre, Bling e Asaas.

A infraestrutura rodava na AWS, na região de Ohio (`us-east-2`):

- **Aplicação:** EC2
- **Banco de dados:** Amazon RDS (MySQL)
- **Arquivos:** Amazon S3

A empresa mantinha também outros projetos em instâncias EC2 próprias. O objetivo era **reduzir o custo fixo de infraestrutura sem perder performance**, e tomar essa decisão com números, não com intuição.

2. Estudo de viabilidade

Antes de mover qualquer coisa, levantei o custo de cada componente por projeto: computação (EC2), banco (RDS), armazenamento (EBS e S3) e IPs. Depois comparei com servidores VPS de configuração equivalente.

O levantamento trouxe três conclusões:

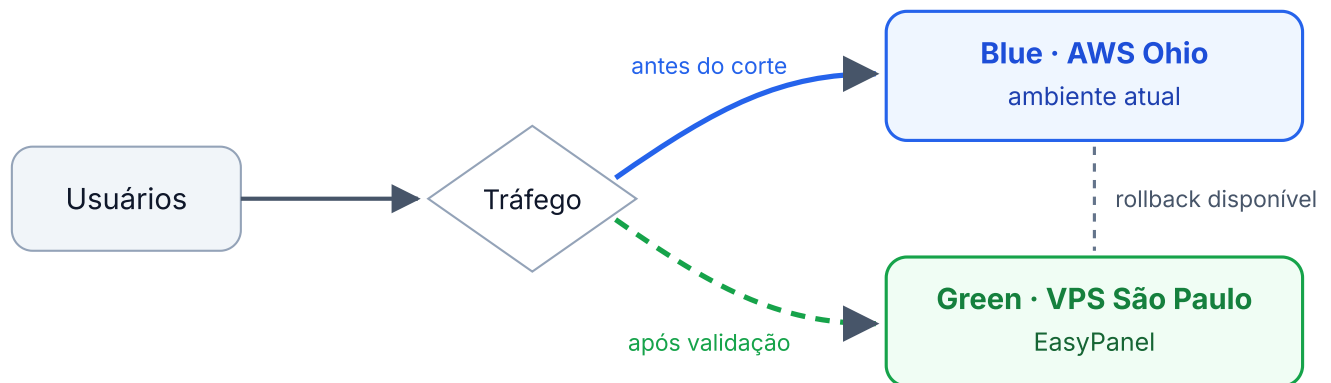
1. **Havia recurso a revisar.** Algumas instâncias existiam sem necessidade clara, e o custo por projeto não era visível até ser separado.
2. **A Triodrop seria a migração mais simples.** Bastava subir um container, porque o plano inicial mantinha banco e arquivos na AWS.
3. **O difícil seriam os outros projetos.** Eles exigiriam reapontar todos os domínios. Manter os IPs na AWS e só redirecionar o tráfego somaria custo e latência, então isso foi descartado.

Decisão: migração gradual. Primeiro os projetos mais simples; a Triodrop depois, com uma janela para terminar de processar jobs, sincronizações e requisições antes do corte.

Resultado do estudo: redução estimada de **cerca de 80%** no custo mensal de infraestrutura.

3. Estratégia de migração: Blue-Green

Para não parar a operação, a migração seguiu o modelo **Blue-Green**:



- O ambiente novo (**green**) sobe em paralelo ao atual (**blue**), sem afetar os usuários.
- O tráfego só é trocado depois que o green é validado.
- O blue fica disponível como rollback até o novo ambiente se provar estável.
- Antes do corte, filas, sincronizações com marketplaces e requisições em andamento terminam de ser processadas.

4. O gargalo que só apareceu na transição

Durante a migração a aplicação passou por três estados:

Estado	Aplicação	Banco	Sessão e cache
A — Original	EC2, Ohio	RDS, Ohio	MySQL
B — Transição	VPS, São Paulo	RDS, Ohio	MySQL, depois Redis
C — Final	VPS, São Paulo	MySQL local (Docker)	Redis local

O estado **B** era o desenho do plano inicial: aplicação no Brasil e banco gerenciado nos EUA. Na prática, **cada consulta SQL levava 277 ms**, e a aplicação ficou praticamente inutilizável.

Por que foi tão ruim

1. A distância multiplica, não soma. A latência de rede é paga em *cada* consulta. Uma página com 40 consultas pagava 40 vezes a viagem entre São Paulo e Ohio:

Consultas por página	A — Original	B — Transição	C — Final
10	17 ms	2.770 ms	3,5 ms
40	68 ms	11.080 ms	14 ms
100	171 ms	27.700 ms	35 ms
250	428 ms	69.250 ms (timeout)	88 ms
500	855 ms	timeout	176 ms

2. A rota era pior que a distância. Pela distância física, o esperado seria algo em torno de 120 ms. Os 277 ms medidos indicavam roteamento ineficiente entre a rede da VPS e a AWS.

3. A conexão também custava caro. Sem conexão persistente, cada requisição abria uma conexão nova com o banco, e isso sozinho levava **1,28 segundo** antes da primeira consulta:

Etapa	Custo estimado
Handshake TCP	415 ms
Handshake TLS	550 a 830 ms
Autenticação MySQL	277 ms
Total	1.284 ms

A decisão

Os números mostraram que manter o banco em outro continente era inviável. Trouxe o **MySQL para dentro da VPS** (Docker) e passei sessão e cache para **Redis local**. Com isso, a latência por consulta caiu de 277 ms para **0,351 ms**, uma redução de **790x**.

Lição: problemas de N+1 conviveram anos com a aplicação sem gerar reclamação. A 1,71 ms por consulta eles eram invisíveis. A migração não criou esses problemas, apenas os deixou visíveis.

5. Como os números foram medidos

- Script PHP executado **dentro do container da aplicação**, usando a camada de banco do próprio Laravel, ou seja, o mesmo caminho que o código de produção percorre.
- A primeira consulta de cada bateria é descartada como aquecimento.
- O round-trip é a média de **50 execuções**, e cada teste comparativo foi repetido **3 vezes**.
- O estado A foi medido a partir de uma máquina em Ohio, na mesma região dos dois bancos, para **isolar o custo de execução do custo de rede**.

6. Resultados

Latência de banco

Métrica	A — AWS Ohio	B — Transição	C — VPS SP
Round-trip por consulta	1,71 ms	277 ms	0,351 ms
Conexão inicial (TCP + TLS + auth)	34,6 ms	1.284 ms	< 10 ms
100 consultas sequenciais	217 ms	27.700 ms	91 ms

Em relação ao ambiente original na AWS: latência por consulta **4,9x menor** e cenário N+1 **2,4x mais rápido**, rodando numa infraestrutura mais barata.

Onde o serviço gerenciado ainda leva vantagem

Isolando a rede, o RDS executa consultas pesadas mais rápido que o MySQL em container. Num `COUNT(*)`, por exemplo, levou 2,95 ms contra 6,45 ms.

O motivo é memória: o RDS tinha 2 GB dedicados ao MySQL, enquanto o container divide a máquina com PHP-FPM, Nginx, Redis e EasyPanel e usava o buffer pool padrão, de 128 MB. Ajustar o `innodb_buffer_pool_size` para 2 GB coloca o banco inteiro em cache e deve eliminar essa diferença.

A diferença de execução é de **milissegundos**; a de rede era de **centenas**. A conclusão da migração não muda, mas parte da vantagem do gerenciado se recupera com configuração.

Tempo estimado de uma requisição completa

Premissas: 40 consultas, leitura e escrita de sessão, 1 chamada ao S3, 1 conexão nova ao banco e 80 ms de PHP.

Componente	A	B	C
Conexão ao banco	35 ms	1.284 ms	5 ms
Sessão	3 ms	554 ms	0,4 ms
40 consultas	68 ms	11.080 ms	14 ms
1 chamada ao S3	10 ms	170 ms	170 ms
Processamento PHP	80 ms	80 ms	80 ms
Total	196 ms	13.168 ms	269 ms

O estado A ainda vence no total por um motivo específico: a aplicação ficava ao lado do S3, em Ohio. **Nos componentes de banco, o estado C é o mais rápido dos três.** Por isso o próximo passo é o S3 (seção 8).

7. Higiene de dados: o banco era quase todo log

O levantamento antes da migração mostrou um banco de **9,4 GB** com esta composição:

Conteúdo	Tamanho	Parte do banco
Logs de integração	8.793 MB	94%
Jobs com falha	446 MB	5%
Dados de negócio	< 160 MB	< 2%

As tabelas de log foram migradas **vazias**, e o banco que chegou à VPS tinha menos de 200 MB. Para que o problema não volte, o plano inclui:

- índice na data de criação dos logs;
- rotina de limpeza em lotes, com retenção de 30 dias;
- limpeza periódica de jobs com falha;
- investigação da causa raiz das falhas;
- avaliação de mover os logs de integração para fora do banco principal.

8. S3: por que não mudar de região

Com o banco em 14 ms, os **170 ms** de uma chamada ao S3 em Ohio passaram a representar **63%** do tempo estimado da requisição. Antes isso ficava escondido atrás dos 12 segundos de espera do banco.

Mover o bucket para São Paulo parece o óbvio, mas:

- **Custa mais.** A região de São Paulo é uma das mais caras da AWS: armazenamento cerca de 1,7x mais caro e tráfego de saída de US\$ 0,15/GB contra US\$ 0,09/GB nos EUA.
- **Resolve só metade.** Aproxima o servidor do bucket, mas não o usuário final. Um usuário em Recife baixando imagens de Ohio continua tendo uma experiência ruim, esteja a VPS onde estiver.

Ordem recomendada:

#	Ação	Custo	Ganho
1	Eliminar chamadas redundantes ao S3 por página	zero	alto
2	CDN na frente do bucket, com domínio próprio	próximo de zero	alto, inclusive para o usuário final
3	Avaliar Cloudflare R2, se o tráfego de saída for relevante	economia	alto
4	Migrar o bucket para São Paulo	mais caro	médio, só em uploads

Uma listagem de 20 produtos que consulta o S3 item a item gasta **3,4 segundos**. Corrigir isso no código custa zero e rende mais do que qualquer mudança de região.

9. Trade-offs assumidos

Sair de um serviço gerenciado não é só economia. O RDS entregava **backup automático com retenção de 7 dias e restauração para qualquer ponto no tempo**. Na VPS isso vira responsabilidade do time: backup agendado para um bucket externo e, principalmente, **teste periódico de restauração**.

Backup que nunca foi restaurado não é backup.

10. Lições

1. **Meça antes de decidir, e durante.** O plano original mantinha o banco na AWS. Foram as medições da transição que mostraram que ele precisava vir junto.
2. **Latência de rede se multiplica pelo número de consultas.** Um N+1 que custa 17 ms perto do banco custa 2,7 segundos longe dele.
3. **Olhe o que o banco realmente guarda.** Mais de 98% não era dado de negócio. Migrar só os dados de negócio deixou a migração mais rápida e barata.
4. **Seja honesto com os números.** O serviço gerenciado ainda vence em execução pura, e o S3 ainda pesa no tempo total. Mostrar isso é o que torna os outros resultados confiáveis.
5. **A solução óbvia nem sempre é a melhor.** Mudar o S3 de região custaria mais e resolveria menos que um CDN e uma revisão de código.

Stack: PHP · Laravel · MySQL · Redis · Docker · EasyPanel · Nginx · AWS (EC2, RDS, S3)

linkedin.com/in/willianyuji · willianyuji.com.br