PB paulobarbosa.dev

SEIS ARQUITETURAS.

Como eu enxergo o que sustenta Twitter, Instagram, YouTube, WhatsApp, Spotify e LinkedIn — e o que dá pra roubar de cada um para sistemas bem menores.

O que tem aqui

01 **Do Twitter (X)**

Uma timeline não é uma consulta. É um resultado calculado antes de alguém pedir, e é essa inversão que sustenta o produto inteiro.

02 **Do Instagram**

O sistema mais impressionante da série é o mais entediante. E o identificador deles carrega a resposta pra um problema que quase todo mundo resolve errado.

03 **Do YouTube**

Upload não é uma requisição, é um pipeline. E a recomendação não escolhe o melhor vídeo: ela elimina quase todos antes de escolher.

04 **Do WhatsApp**

Criptografia ponta a ponta não é uma funcionalidade. É uma restrição que decide o que o servidor pode fazer, e ela simplifica mais do que complica.

05 **Do Spotify**

Dois sistemas escondidos num só: um orçamento de latência medido em milissegundos, e um log de eventos que também é a contabilidade.

06 **Do LinkedIn**

A empresa que resolveu o problema de integração entre sistemas transformando o log num produto. É por isso que o Kafka existe.

Uma nota sobre método: tudo aqui é reconstruído a partir de material público — blog de engenharia, palestra, paper e o que os times contaram em conferência. Não é conhecimento interno, e sistema dessa idade muda por baixo o tempo todo. O que eu trago é a leitura das decisões, não a planta.

CAPÍTULO 01

Como eu enxergo a arquitetura do Twitter (X)

Uma timeline não é uma consulta. É um resultado calculado antes de alguém pedir, e é essa inversão que sustenta o produto inteiro.

Este texto abre uma série em que eu leio a arquitetura de seis produtos que todo mundo usa. Vale dizer de onde vem o material: blog de engenharia, palestra, paper publicado e o que os times contaram em conferência. Não é conhecimento interno, e sistema dessa idade muda por baixo o tempo todo. O que eu trago é a leitura, não a planta.

E é isso que me interessa: não o inventário de tecnologia, e sim a decisão que organiza o resto.

A pergunta que define tudo

Quando você abre o Twitter, precisa ver os tweets de quem você segue, em ordem, agora.

A forma óbvia de resolver isso é uma consulta:

```
select * from tweets
where author_id in (select followed_id from follows where follower_id = ?)
order by created_at desc
limit 50;
```

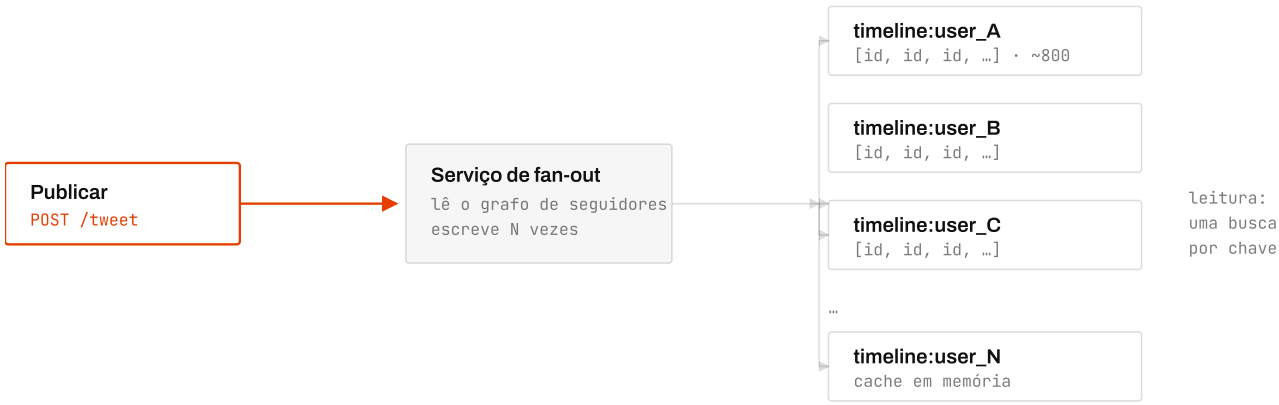
Funciona no seu projeto. Não funciona aqui, e a razão não é o tamanho da tabela: é a assimetria entre leitura e escrita.

Gente escreve pouco e lê muito. A proporção entre abrir a timeline e publicar algo é de ordens de grandeza. Uma arquitetura que faz trabalho pesado na leitura está fazendo trabalho pesado no lado que acontece mil vezes mais.

A inversão é o que eu considero a decisão central do produto: **calcule na escrita, não na leitura**.

Fan-out na escrita

Quando alguém publica, o sistema não guarda o tweet e vai embora. Ele empurra a referência daquele tweet pra dentro da timeline pré-montada de cada seguidor.



FAN-OUT NA ESCRITA — O TRABALHO ACONTECE UMA VEZ, NA PUBLICAÇÃO

A timeline vira uma lista de identificadores num cache em memória, com algumas centenas de posições. Ler passa a ser buscar uma chave e hidratar os tweets. É a operação mais barata que existe.

O que eu acho elegante nisso é o reconhecimento de que a timeline não é uma consulta. É um **resultado materializado**, com todas as consequências que isso carrega: ela pode estar desatualizada por um instante, ela ocupa espaço multiplicado pelo número de usuários, e ela precisa ser reconstruível quando o cache se perde.

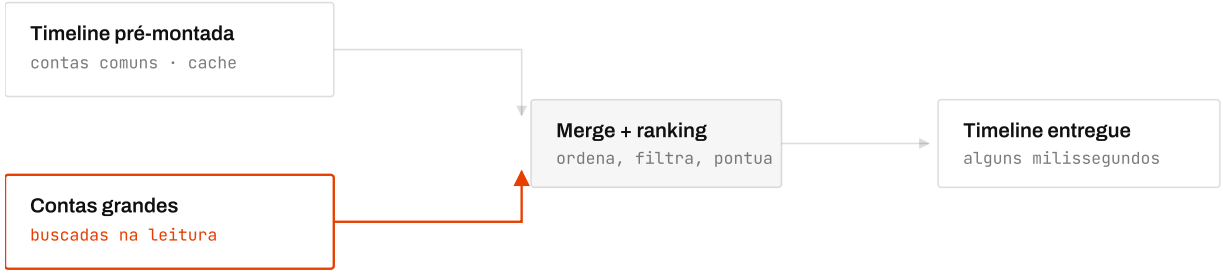
Que é exatamente a mesma discussão de saldo calculado ou materializado, num contexto onde o custo de errar é bem menor.

Onde o modelo quebra

Alguém com trinta milhões de seguidores publica. O fan-out precisa escrever trinta milhões de vezes.

Isso não é um pico: é um pico correlacionado, porque contas grandes publicam em horário de audiência, e várias fazem isso ao mesmo tempo. A fila explode, e a timeline de todo mundo atrasa por causa de um punhado de contas.

A saída, que virou material de estudo em system design, é **híbrida**: contas acima de um certo número de seguidores são excluídas do fan-out. Os tweets delas ficam só na linha do autor, e são buscados e mesclados na hora da leitura.



O MODELO HÍBRIDO — O CASO RARO E CARO SAI DO CAMINHO COMUM

Essa é a decisão que eu mais admiro nesse desenho, e é a mais transferível. Não é uma solução geral bonita: é o reconhecimento de que **a distribuição é desigual e a arquitetura deveria refletir isso**. A cauda longa vai pro caminho barato, o punhado extremo vai pro caminho caro, e ninguém paga o custo do outro.

Quase todo sistema que eu vi sofrer com performance tinha o problema oposto: um caminho único, dimensionado pelo caso médio, sendo destruído pelo caso extremo. Foi exatamente isso no cenário do índice faltando.

O identificador que carrega o tempo

Um detalhe pequeno com efeito enorme: o identificador do tweet não é aleatório nem sequencial global. É um número de 64 bits que embute o instante da criação, o identificador da máquina que gerou e uma sequência.

Isso resolve três coisas de uma vez. Dá pra gerar em paralelo, em várias máquinas, sem coordenação. É ordenável por tempo, então ordenar a timeline é ordenar números. E cabe num inteiro, que indexa e trafega barato.

Escrevi sobre essa família de identificadores em UUIDv7, ULID ou Snowflake. Ver o padrão nascer de uma necessidade concreta é bem mais instrutivo que ler a comparação.

O que a timeline algorítmica muda

Quando a ordem era estritamente cronológica, a timeline pré-montada era quase o produto final.

Com ranking, ela vira **candidata**. O sistema monta um conjunto de algumas centenas de itens possíveis e depois pontua cada um com um modelo, considerando afinidade, engajamento provável, recência e uma pilha de sinais.

Isso divide a arquitetura em duas etapas que passaram a aparecer em todo produto com feed: geração de candidatos, que é barata e ampla, e ranking, que é caro e estreito. Você nunca pontua tudo. Você reduz primeiro, pontua depois.

Vou voltar nesse padrão no texto sobre o YouTube, onde ele é ainda mais explícito.

O que eu levo pra sistema pequeno

Três coisas, e nenhuma delas exige escala.

Descubra a assimetria. Em quase todo sistema existe uma operação que acontece mil vezes mais que outra. Se o trabalho pesado está no lado frequente, mova pro lado raro. Materializar o resultado da consulta cara no momento da escrita é uma técnica disponível pra qualquer um.

Trate o caso extremo por fora. Se 0,1% dos registros tem mil vezes mais volume que a mediana, eles não deveriam passar pelo mesmo caminho. Um `if` explícito pra esse grupo é mais honesto que uma solução genérica que sofre nos dois casos.

Assuma que o cache vai sumir. Timeline materializada é cache, e a pergunta que decide a qualidade do desenho é como reconstruir. Se a resposta for "não sei", você não tem cache: tem uma fonte de verdade

frágil.

O que me impressiona nesse sistema não é a escala. É que a decisão central — inverter o momento do trabalho — foi tomada cedo, e o resto é consequência.

CAPÍTULO 02

Como eu enxergo a arquitetura do Instagram

O sistema mais impressionante da série é o mais entediante. E o identificador deles carrega a resposta pra um problema que quase todo mundo resolve errado.

Segundo texto da série em que eu leio a arquitetura de produtos que todo mundo usa, a partir de material público de engenharia. Não é a planta: é a minha leitura do que organiza o resto.

E, de todos os seis, esse é o que eu acho mais instrutivo — justamente por não ter nada de espetacular.

A tese: tecnologia entediante como estratégia

Quando o Instagram cresceu pra dezenas de milhões de usuários, ele rodava com um time minúsculo, em Python com Django, em cima de PostgreSQL e memcached.

Nada exótico. Nenhum banco novo, nenhuma linguagem nova, nenhum sistema distribuído artesanal.

A leitura preguiçosa é que eles tiveram sorte. A leitura que eu faço é que foi decisão: **cada tecnologia nova é um sistema a mais que alguém precisa entender às três da manhã**, e um time pequeno não tem esse orçamento.

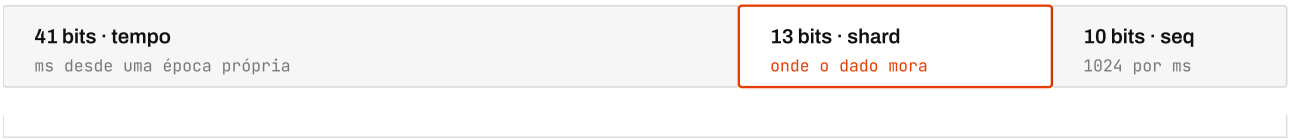
Eles gastaram a inovação onde o problema era realmente deles, e usaram o chato e testado no resto. Isso é uma escolha de engenharia, e é a mais subestimada que existe.

Onde eles inovaram: o identificador

O problema concreto: o Postgres não escala verticalmente pra sempre. Em algum momento você precisa dividir o dado em várias máquinas.

E aí aparece a pergunta que quebra a maioria dos projetos: como gerar identificador único, ordenável por tempo, sem um ponto central de coordenação, num sistema fragmentado?

A resposta deles foi embutir o endereço dentro do próprio identificador.



ordenável por tempo · gerado sem coordenação · diz sozinho em qual shard buscar

O IDENTIFICADOR COMO ENDEREÇO — A ROTA VEM EMBUTIDA NO DADO

O que isso resolve de uma vez só:

Ordenação. Como o tempo está nos bits mais significativos, ordenar por identificador é ordenar por criação. Feed, paginação e índice ficam baratos, pelo mesmo motivo que descrevi em UUIDv7, ULID ou Snowflake.

Roteamento. Dado um identificador de foto, você sabe em qual fragmento ela está sem consultar nenhuma tabela de mapeamento. O roteador é aritmética.

Geração distribuída. Cada fragmento gera o seu, sem falar com ninguém. Não existe serviço central de identificador pra cair.

Tamanho. Cabe num `bigint`. Índice menor, junção mais rápida, menos bytes no fio.

Eu acho essa a decisão mais elegante dos seis textos dessa série. Não porque é complexa: porque ela **elimina** três subsistemas que a maioria dos projetos acaba construindo.

O truque do fragmento lógico

A segunda decisão é a que resolve o problema que aparece depois: você fragmentou em oito máquinas e agora precisa de dezesseis.

Se o mapeamento for direto, `shard = id % 8`, mudar o número de máquinas rehasheia tudo e você tem uma migração monstruosa.

A saída é uma camada de indireção: milhares de fragmentos **lógicos**, distribuídos sobre poucas máquinas **físicas**.

INDIREÇÃO LÓGICA — A MÁQUINA MUDA, O ENDEREÇO DO DADO NÃO

O identificador aponta pro fragmento lógico, que nunca muda. Uma tabela pequena diz em qual máquina aquele fragmento está hoje.

Crescer vira uma operação de infraestrutura: replique o fragmento pra máquina nova, aponte o mapa, apague o antigo. Nenhuma linha muda de identificador. Nenhum código muda.

Essa é a diferença entre uma migração de um dia e uma migração de um trimestre, e ela custa uma tabela de mapeamento decidida no primeiro mês.

O feed, e por que ele é diferente do Twitter

O Instagram enfrenta o mesmo problema de fan-out que descrevi em como eu enxergo a arquitetura do Twitter, com uma diferença que muda a conta: o conteúdo é pesado.

Um tweet é texto. Uma publicação é uma imagem ou vídeo, com várias resoluções, servida por rede de distribuição, com processamento no meio.

Isso separa o sistema em dois caminhos com características opostas.

O caminho dos metadados é pequeno, transacional, ordenado, e vive no banco fragmentado. O caminho da mídia é enorme, imutável depois de escrito, e vive em armazenamento de objeto com cache na borda.

Mídia imutável é uma vantagem que eu acho pouco explorada em sistema comum: se o arquivo nunca muda, o cache nunca precisa ser invalidado, o identificador pode ser o hash do conteúdo, e a distribuição vira um problema resolvido.

O que eu levo pra sistema pequeno

O identificador é uma decisão de arquitetura, não um detalhe. Ele carrega ordenação, roteamento e capacidade de geração distribuída. Escolher `uuid_generate_v4()` por hábito é abrir mão dos três de graça.

Coloque indireção entre o dado e a máquina antes de precisar. Fragmento lógico com mapeamento é barato quando você tem uma máquina só, e é o que torna a segunda máquina um evento operacional em vez de um projeto.

Prefira o chato. Um time pequeno que roda Postgres, um cache e uma fila entende o próprio sistema. O mesmo time com seis bancos especializados passa o dia aprendendo a operar, e o tempo de aprender sai do tempo de construir.

A parte que mais me marca nessa história é essa última. A arquitetura mais impressionante da série não é a que tem a peça mais engenhosa. É a que conseguiu não precisar delas.

CAPÍTULO 03

Como eu enxergo a arquitetura do YouTube

Upload não é uma requisição, é um pipeline. E a recomendação não escolhe o melhor vídeo: ela elimina quase todos antes de escolher.

Terceiro texto da série em que eu leio arquiteturas conhecidas a partir de material público de engenharia. Não é a planta interna: é a leitura do que organiza o resto.

O YouTube tem duas coisas que eu considero de escola: o tratamento do upload e o formato da recomendação. Os dois carregam padrões que dá pra usar em sistema muito menor.

Upload não é uma requisição

A intuição errada é tratar upload como um `POST` que termina quando o arquivo chega.

Um vídeo enviado precisa virar dezenas de arquivos diferentes: várias resoluções, vários formatos, vários perfis de compressão, faixas de áudio separadas, miniaturas, legendas. Isso é minutos ou horas de processamento pesado.

Se isso acontecer dentro da requisição, você tem uma conexão pendurada, um timeout garantido e nenhum jeito de retomar quando falhar.

O desenho é um pipeline com estados.

UPLOAD COMO PIPELINE — O TRABALHO É FATIADO PARA PODER SER PARALELO E RETOMÁVEL

Duas decisões dentro disso me interessam mais que o resto.

Fatiar antes de processar. Transcodificar um vídeo de duas horas como uma unidade é um trabalho de horas que, se falhar aos 90%, recomeça do zero. Fatiado em pedaços curtos, o trabalho vira paralelo em

centenas de máquinas e a falha custa um pedaço, não o vídeo.

Qualidade progressiva. Não é preciso ter todas as resoluções prontas pra publicar. A menor fica pronta primeiro e o vídeo já pode ser assistido, enquanto as outras chegam. O usuário percebe "já está no ar" bem antes de o trabalho terminar.

Esse padrão de entregar o resultado parcial cedo é transferível pra qualquer processamento demorado, e quase ninguém usa.

O streaming é o cliente decidindo

O que sai do empacotamento não é um arquivo de vídeo. É um manifesto que descreve as qualidades disponíveis e os pedaços de cada uma.

O reprodutor baixa o manifesto, escolhe uma qualidade, baixa alguns segundos, mede quanto tempo levou e **decide sozinho** se sobe ou desce de qualidade no próximo segmento.

Isso inverte a responsabilidade de um jeito que eu acho elegante: o servidor não sabe nada sobre a conexão de quem assiste, e é justamente quem assiste que tem essa informação. Em vez de tentar adivinhar do lado errado, o desenho entrega as opções e deixa a decisão onde o dado está.

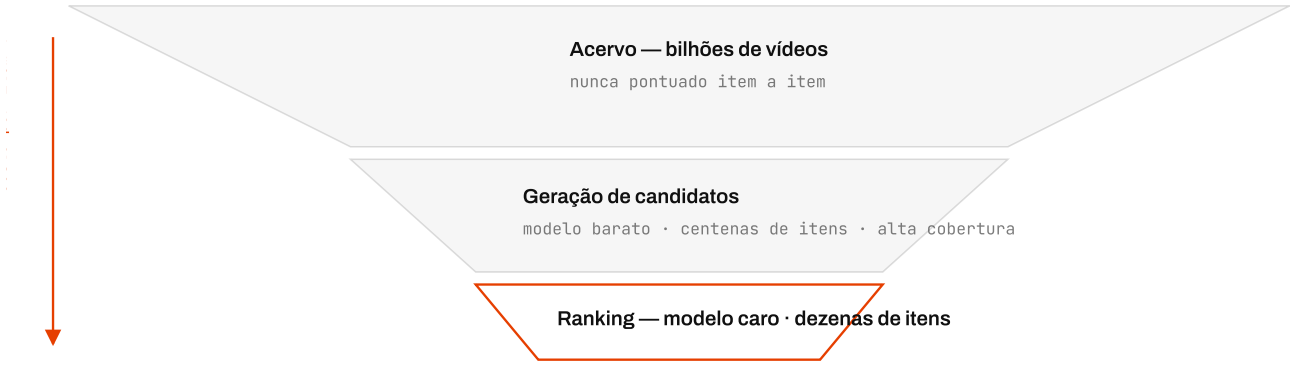
Como consequência, o servidor só entrega arquivo estático. Nada de estado de sessão, nada de transcodificação sob demanda no caminho quente. Tudo cacheável na borda, porque nada é personalizado.

A recomendação é um funil, não uma escolha

Aqui está o padrão que eu mais uso de referência.

O problema parece ser "escolher os melhores vídeos pra essa pessoa" num acervo de bilhões. Pontuar bilhões de itens por usuário, em tempo de requisição, é impossível.

A solução publicada é dividir em duas etapas com naturezas opostas.



O FUNIL — REDUZA BARATO, DEPOIS PONTUE CARO

A geração de candidatos é **barata e generosa**: reduz bilhões pra algumas centenas, com modelos simples e muita heurística. Ela pode errar pra mais, e o custo de um candidato ruim aqui é baixo.

O ranking é **caro e exigente**: pontua algumas centenas com um modelo pesado, considerando muito mais sinal. Ele só existe porque a etapa anterior tornou o volume tratável.

A lição, que vale pra qualquer busca ou recomendação: você nunca aplica a lógica cara no conjunto todo. Você reduz primeiro com algo barato, e a qualidade final depende muito mais da etapa de redução do que do modelo sofisticado. Um item que não vira candidato nunca vai ser recomendado, por melhor que o ranking seja.

Vi o mesmo padrão no Twitter, com outro nome.

A contagem de visualizações que não bate

Um detalhe pequeno que vale como aula de sistemas distribuídos: por muito tempo, contagem de visualização em vídeo viral ficava travada num número enquanto o resto subia.

A razão é que contar de verdade é caro. Cada visualização é um evento em algum lugar do mundo, e somar isso com precisão em tempo real, num sistema distribuído, custa mais do que o número vale.

A escolha foi: mostre um valor aproximado rápido, e reconcilie depois com o número exato, que é apurado com calma, com filtro antifraude.

O que me interessa aqui não é o truque, é o reconhecimento explícito de que **nem todo número precisa ser exato agora**. A contagem de visualização pode ser aproximada. O número que paga o criador não pode.

Sistemas que tratam os dois com o mesmo rigor gastam demais no primeiro, ou de menos no segundo. É a mesma distinção que aparece em saldo calculado ou materializado: descubra qual número é opinião e qual é verdade.

O que eu levo pra sistema pequeno

Todo processamento longo é um pipeline com estado, não uma requisição. Se leva mais de alguns segundos, precisa de fila, de estados persistidos e de retomada por etapa.

Fatie o trabalho até que a falha seja barata. A unidade de reprocessamento deveria ser pequena o suficiente pra que refazer não doa.

Entregue o resultado parcial. A versão de baixa qualidade pronta primeiro vale mais que a versão perfeita dez minutos depois.

Reduza antes de pontuar. Qualquer coisa que ordene por relevância deveria ter uma etapa barata de corte antes da etapa cara. Isso vale pra recomendação, pra busca e pra qualquer relatório que ordene muita coisa.

CAPÍTULO 04

Como eu enxergo a arquitetura do WhatsApp

Criptografia ponta a ponta não é uma funcionalidade. É uma restrição que decide o que o servidor pode fazer, e ela simplifica mais do que complica.

Quarto texto da série em que eu leio arquiteturas conhecidas a partir de material público. Não é a planta: é a leitura do que organiza o resto.

Esse é o caso que mais me interessa dos seis, porque ele inverte uma intuição comum: a restrição de segurança mais dura do conjunto é também o que deixa o sistema mais simples.

O servidor não pode ler nada

Com criptografia ponta a ponta, a mensagem é cifrada no aparelho de quem envia e só é decifrada no de quem recebe. O servidor transporta bytes que não consegue interpretar.

Isso apaga, de uma vez, uma lista de coisas que outros produtos fazem no servidor:

Busca de mensagem no lado do servidor. Impossível. A busca vive no aparelho, sobre o histórico local.

Prévia de conteúdo, moderação automática do texto, recomendação baseada no que foi dito, anúncio por assunto da conversa. Nada disso existe.

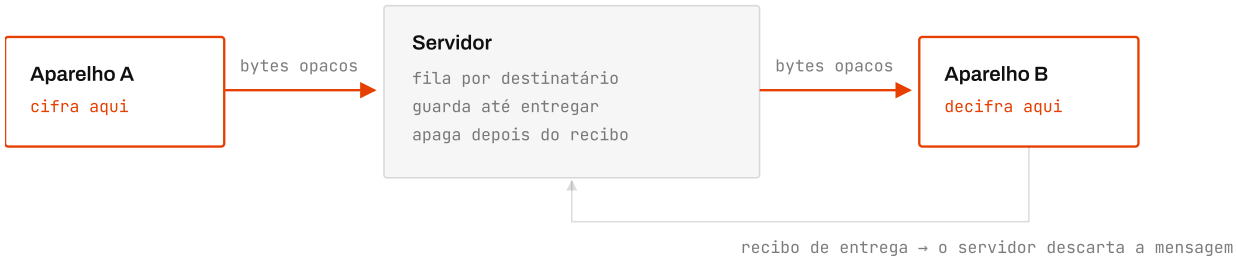
Recuperar histórico ao trocar de aparelho. O servidor não tem o histórico em claro, então isso vira um problema de transferência entre dispositivos ou de backup cifrado com chave que o usuário controla.

Parece uma lista de limitações. Eu leio como um enxugamento: **o servidor tem uma responsabilidade só**, que é entregar bytes para o destinatário certo.

Store and forward, e depois esqueça

O modelo de entrega é o mais simples possível.

O ESTADO PERMANENTE VIVE NOS APARELHOS. O SERVIDOR É UM CORREDOR.



STORE AND FORWARD — GUARDAR É EXCEÇÃO TEMPORÁRIA, NÃO FUNÇÃO DO SISTEMA

O servidor guarda a mensagem cifrada enquanto o destinatário está offline, e descarta assim que a entrega é confirmada.

Isso tem uma consequência de custo que eu acho impressionante: **o armazenamento não cresce com o uso**. Um sistema que guarda todo o histórico de todos os usuários para sempre tem um custo que sobe monotonamente. Um que guarda só o que ainda não foi entregue tem um custo proporcional a quantas pessoas estão offline agora, que é um número quase constante.

A base de dados permanente é o aparelho de cada um. O servidor é infraestrutura de trânsito.

O problema real: conexões, não requisições

Mensageria não é tráfego HTTP comum. Cada aparelho mantém uma conexão aberta, esperando, por horas.

Isso muda o gargalo. Não é CPU por requisição: é **quantas conexões ociosas uma máquina aguenta**, cada uma consumindo um pouco de memória e um descritor de arquivo.

É por isso que a escolha de Erlang aparece em toda conversa sobre esse sistema, e ela não é folclore. A plataforma foi construída para telecomunicações: processos leves aos milhões, isolamento entre eles, supervisão que reinicia o que quebrou sem derrubar o resto.

O modelo mental encaixa quase perfeitamente. Cada conexão é um processo minúsculo e independente. Um erro em uma conversa não contamina as outras. E o número de usuários por servidor fica em uma ordem de grandeza incomum, o que explica a relação famosa entre o tamanho do time e o tamanho da base.

Eu tiro uma lição disso que não é sobre Erlang: **a natureza do gargalo deveria escolher a tecnologia**. Um sistema dominado por conexões ociosas tem restrições diferentes de um dominado por processamento. Escolher a linguagem por gosto e depois lutar contra o gargalo é o caminho comum, e é o mais caro.

Grupo, e o problema de N dispositivos

Aqui a criptografia cobra um preço.

Sem cifra, mandar mensagem pra um grupo de cem pessoas é uma escrita e um fan-out no servidor. Com cifra ponta a ponta, cada destinatário tem uma chave diferente, então o conteúdo precisaria ser cifrado uma vez para cada.

Pior: cada pessoa tem vários aparelhos, e cada aparelho é um destinatário criptográfico independente. Cem pessoas com três aparelhos são trezentas cifragens, feitas no celular de quem enviou.

A saída conhecida é uma chave de remetente para o grupo: quem envia cifra o conteúdo uma vez com essa chave, e distribui a chave individualmente para cada participante. A distribuição da chave é O(N), a mensagem é O(1). Enquanto ninguém entrar nem sair, as mensagens seguintes saem baratas.

Isso explica por que entrar e sair de grupo é caro e por que grupo tem limite de tamanho. O limite não é de banco de dados: é do custo de redistribuir chave.

Escrevi sobre a diferença entre garantir conteúdo e garantir origem em assinatura digital e hash — é a mesma família de decisões, aqui aplicada no caminho quente do produto.

O que a restrição obriga a fazer bem

Um sistema que não pode reler a mensagem depois precisa acertar a entrega na primeira vez.

Isso força identificador de mensagem estável gerado no cliente, para deduplicar reenvio. Força recibo em camadas, com estados distintos para enviado, entregue e lido. Força ordenação que não dependa do relógio do servidor, porque o servidor não é a fonte da verdade do conteúdo.

Tudo isso é idempotência e ordenação de eventos, os mesmos problemas de qualquer fila, só que com uma restrição a mais: quem coordena não pode inspecionar o que está coordenando.

O que eu levo pra sistema pequeno

Restrição forte simplifica. Decidir que o servidor não lê o conteúdo elimina dezenas de funcionalidades possíveis e, com elas, dezenas de subsistemas. Vale perguntar, em qualquer projeto, o que a gente ganharia se decidisse não guardar algo.

Estado permanente no cliente é uma opção legítima. Nem todo dado precisa viver no servidor. Quando o cliente é a fonte, o custo de armazenamento para de crescer com o tempo.

Deixe o gargalo escolher a tecnologia. Conexões ociosas, processamento pesado e escrita intensa são problemas diferentes, e as ferramentas boas em um costumam ser medianas nos outros.

Não guarde o que já foi entregue. É a política de retenção mais simples que existe, e quase ninguém considera, porque guardar parece sempre a opção segura. Guardar é responsabilidade, é custo, e é a primeira coisa que vaza.

CAPÍTULO 05

Como eu enxergo a arquitetura do Spotify

Dois sistemas escondidos num só: um orçamento de latência medido em milissegundos, e um log de eventos que também é a contabilidade.

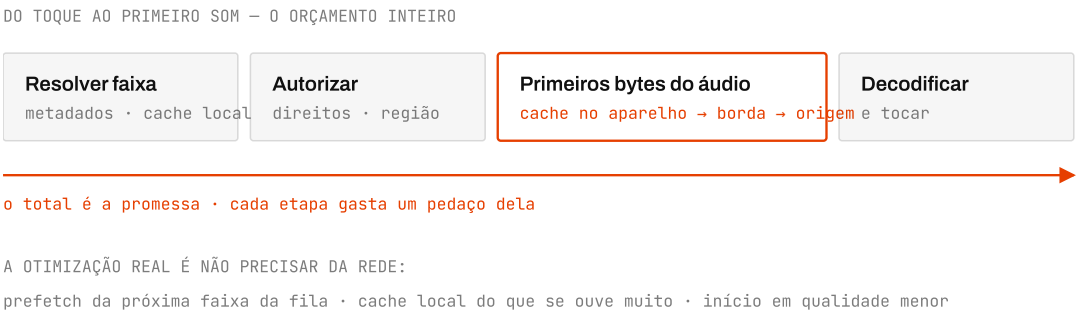
Quinto texto da série em que eu leio arquiteturas conhecidas a partir de material público de engenharia. Não é a planta: é a leitura do que organiza o resto.

O Spotify é o que eu acho mais subestimado do conjunto, porque as duas coisas difíceis dele são invisíveis pra quem usa.

A primeira: o orçamento de latência

O produto tem uma promessa implícita: você aperta o play e a música começa. Não em três segundos. Agora.

Isso vira um orçamento, e orçamento é uma ferramenta de projeto que eu queria ver mais gente usar. Você define o total aceitável e distribui entre as etapas, sabendo que a soma não pode estourar.



ORÇAMENTO DE LATÊNCIA — A ETAPA MAIS CARA É A QUE VOCÊ CONSEGUE EVITAR

E aí vem a parte que eu acho a mais inteligente: a maior otimização não é deixar a rede mais rápida, é **não usar a rede**.

O aparelho guarda o que você ouve com frequência. Ele busca a próxima faixa da fila antes de você chegar nela. Ele começa a tocar com um trecho de qualidade menor enquanto o resto chega.

Nenhuma dessas três é sofisticada. Todas dependem de uma coisa só: alguém decidiu, no começo, que o número que importa é o tempo até o primeiro som, e desenhou pra trás a partir dele.

Vale registrar também a fase em que a distribuição era feita entre os próprios usuários, num modelo par a par, antes de as redes de distribuição ficarem baratas o bastante para substituir aquilo. É um bom lembrete de que

arquitetura é datada: a decisão certa em 2010 deixou de ser em 2015, e desmontar um sistema que funciona porque o mundo mudou é uma decisão de engenharia legítima.

A segunda: o log de eventos é dinheiro

Cada reprodução gera um evento. Multiplicado por centenas de milhões de pessoas, isso é um dos maiores fluxos de eventos que existem.

E é aqui que o sistema fica interessante pra mim, porque **o mesmo fluxo alimenta dois consumidores com exigências opostas**.

O MESMO LOG, DOIS CONTRATOS — UM APROXIMADO, UM EXATO

A personalização pode perder evento. Se 0,1% das reproduções não entrar no cálculo, a recomendação da semana continua boa. Ela tolera atraso, tolera duplicata e trabalha com estimativa.

O pagamento de direitos não pode. Ali cada reprodução é uma fração de centavo devida a alguém, e o total precisa fechar. Perder evento é não pagar. Duplicar é pagar duas vezes.

É o mesmo dado, com dois contratos de qualidade diferentes. E a consequência de arquitetura é que o caminho que alimenta o financeiro precisa de tudo que eu descrevi em o que é um core ledger: identificador estável por evento, deduplicação por constraint, e uma verificação periódica que confere se o total bate.

Tratar os dois caminhos com o mesmo rigor é caro demais de um lado e frouxo demais do outro. Separar os contratos é o que permite otimizar cada um pelo que ele realmente precisa. Escrevi sobre a impossibilidade de entrega exatamente uma vez em exactly-once é mentira — esse é o caso onde a distinção deixa de ser acadêmica e vira dinheiro.

A recomendação, e por que ela funciona

A parte de personalização combina abordagens que sozinhas seriam fracas.

Tem o sinal de comportamento coletivo: quem ouve isso também ouve aquilo. É poderoso e tem um ponto cego famoso, o começo frio — música nova, sem histórico, não é recomendada por ninguém, então nunca ganha histórico.

Tem o sinal de texto: o que se escreve sobre aquele artista em resenha, blog e playlist. Isso dá contexto cultural que o comportamento não dá.

E tem o sinal do próprio áudio: características extraídas do som, que funcionam mesmo para uma faixa que ninguém ouviu ainda. É a peça que resolve o começo frio.

O que eu levo daqui não é o algoritmo. É que cada sinal cobre a fraqueza do outro, e que a combinação foi desenhada a partir das falhas conhecidas de cada um. Isso é engenharia de sistema, não de modelo.

O que eu levo pra sistema pequeno

Escreva o orçamento antes da arquitetura. "Essa tela precisa responder em 300ms" é uma restrição que decide dezenas de escolhas depois. Sem o número, cada decisão é tomada isoladamente e a soma estoura.

A melhor otimização é remover a etapa. Cache local, prefetch e resultado parcial batem qualquer ajuste fino no que já existe.

Um fluxo de eventos, vários contratos. Se o mesmo dado alimenta análise e financeiro, eles não têm o mesmo requisito. Escrever isso explicitamente evita gastar demais num lado e de menos no outro.

Arquitetura tem prazo de validade. A decisão certa há cinco anos pode estar errada hoje porque o custo relativo das coisas mudou. Reavaliar não é retrabalho.

CAPÍTULO 06

Como eu enxergo a arquitetura do LinkedIn

A empresa que resolveu o problema de integração entre sistemas transformando o log num produto. É por isso que o Kafka existe.

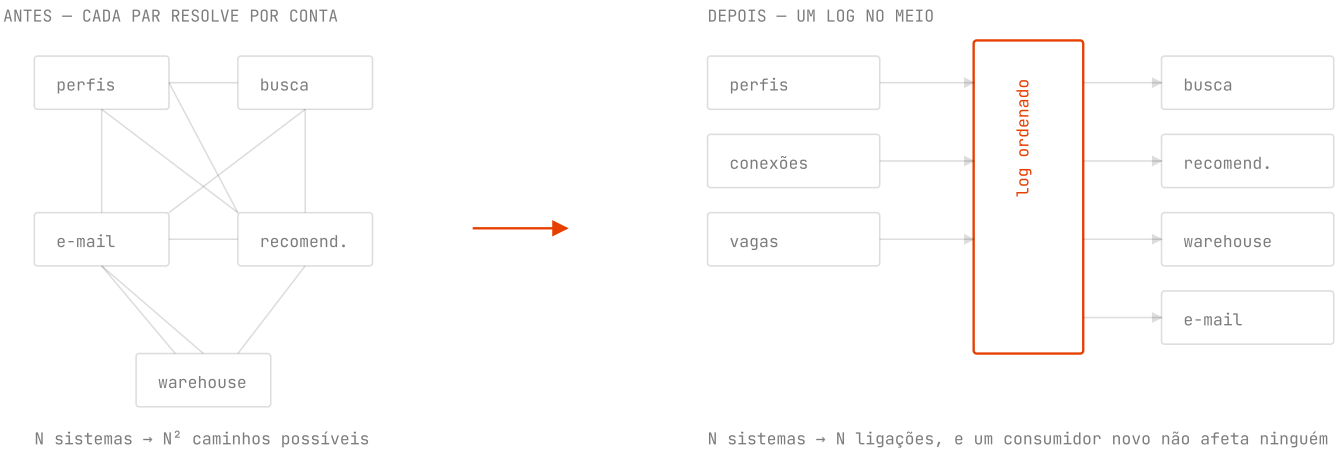
Último texto da série em que eu leio arquiteturas conhecidas a partir de material público de engenharia. Não é a planta: é a leitura do que organiza o resto.

E esse fecha bem, porque a contribuição do LinkedIn não foi um produto: foi uma abstração que hoje está em quase toda empresa de tecnologia.

O problema que quase toda empresa tem

Você tem um sistema de perfis. Depois um de busca, que precisa dos dados do perfil. Depois um de recomendação, que também precisa. Depois um data warehouse, um sistema de e-mail, um de notificação, um de antifraude.

Cada um precisa de uma cópia atualizada de dados que vivem em outro lugar. E cada integração é construída sozinha: um job noturno aqui, uma chamada de API ali, um gatilho de banco acolá, um consumidor de fila específico.



A MUDANÇA DE ABSTRAÇÃO — DE INTEGRAÇÃO PONTO A PONTO PARA UM LOG COMPARTILHADO

Com dez sistemas, isso vira uma teia que ninguém consegue desenhar num quadro. Cada mudança de esquema quebra três integrações que ninguém lembrava que existiam.

A ideia que saiu do LinkedIn e virou Kafka é enganosamente simples: **em vez de cada sistema falar com cada sistema, todo mundo publica num log ordenado e todo mundo lê dele**.

Por que o log é a abstração certa

Eu demorei pra entender por que isso é mais que uma fila com outro nome. A diferença está em três propriedades.

Ordem. O log tem sequência. Cada consumidor sabe exatamente onde parou, e esse marcador é dele, não do produtor. Dois consumidores no mesmo fluxo, em velocidades diferentes, não interferem um no outro.

Retenção. A mensagem não some ao ser consumida. Ela fica pelo período configurado. Isso permite uma coisa que fila tradicional não permite: **reprocessar**. Um consumidor novo lê o log inteiro desde o começo e reconstrói o estado dele.

Essa é a propriedade que muda o custo de errar. Bug no cálculo do índice de busca? Corrige o código, volta o marcador, reprocessa. Sem exportação, sem migração, sem script de backfill.

Desacoplamento real. O produtor não sabe quem consome. Adicionar o oitavo consumidor não exige mudar nada nos produtores. Em integração ponto a ponto, cada consumidor novo é uma negociação.

É a mesma ideia que descrevi em log append-only, levada ao nível da empresa inteira em vez do nível da tabela.

Captura de mudança: o log sem pedir licença

Tem um detalhe operacional que decide se isso funciona na prática.

Pedir para todos os times publicarem eventos é uma migração organizacional demorada. E dupla escrita, gravar no banco e publicar no log, é a receita de inconsistência que eu descrevi em outbox pattern: as duas operações podem falhar separadamente.

A saída é ler o próprio log de transação do banco e transformar cada mudança em evento. O sistema de origem não muda nada, não sabe que está sendo observado, e o fluxo fica garantidamente consistente com o que foi commitado.

Isso é o que hoje se chama captura de dados de mudança, e é a peça que torna a adoção viável sem reescrever tudo.

Os graus de separação

O outro problema interessante desse produto é de grafo, não de log.

Quando você vê um perfil, o sistema diz se a pessoa é conexão de primeiro, segundo ou terceiro grau. Isso parece um detalhe de interface e é uma consulta cara.

Segundo grau é a união das conexões de todas as suas conexões. Alguém com 500 conexões que também têm 500 cada tem, em ordem de grandeza, centenas de milhares de perfis de segundo grau. Terceiro grau explode pra dezenas de milhões.

Fazer isso com junção em banco relacional, em tempo de requisição, para cada visita de perfil, não fecha.

O que se faz é um serviço dedicado, com o grafo em memória, particionado, respondendo consultas de distância limitada com estruturas específicas pra isso. E, para o caso mais frequente, o resultado é pré-calculado e cacheado.

O padrão aqui é o mesmo que apareceu em toda a série: **quando a consulta é cara e frequente, ela vira um sistema próprio**. Não uma query melhor, não um índice novo: um serviço com estrutura de dados desenhada pra aquela pergunta.

O feed, com um problema a mais

O feed enfrenta o mesmo dilema de fan-out que descrevi no Twitter, com um agravante: aqui o conteúdo relevante não é só de quem você segue. Ele vem de conexões de segundo grau, de empresas, de grupos e de anúncios.

Isso amplia demais o conjunto de candidatos e reforça a estrutura de funil que descrevi no YouTube: gerar candidatos barato, pontuar caro.

Fico com a impressão de que, nos seis textos, essa foi a estrutura mais recorrente. Reduzir antes de decidir aparece em todos.

O que eu levo pra sistema pequeno

Um log central resolve integração antes de você ter dez sistemas. Não precisa de Kafka. Uma tabela de eventos com marcador por consumidor já dá ordem, retenção e desacoplamento. O ganho aparece no terceiro consumidor.

Poder reprocessar muda o custo de errar. Se corrigir um bug de cálculo significa voltar o marcador e reler, o medo de mexer some. Sem isso, cada erro vira um script de correção escrito sob pressão.

Não peça a todos os times que publiquem eventos. Leia a mudança na origem. A adoção é o problema difícil, não a tecnologia.

Consulta cara e frequente merece um sistema próprio. Quando otimizar a query já não resolve, a resposta costuma ser uma estrutura de dados diferente, mantida por fora, alimentada pelo log.

Fecho a série com a impressão que ficou mais forte: nenhum desses sistemas é admirável pela tecnologia que usa. Eles são admiráveis por uma decisão tomada cedo, quase sempre sobre **onde o trabalho acontece** — antes ou depois, no cliente ou no servidor, na escrita ou na leitura. O resto é consequência.

Fim

Se algo aqui foi útil, os textos completos e o resto do arquivo estão no site.

paulobarbosa.dev/blog