ARTIGO TRADUZIDO PARA O PORTUGUÊS

50 Conceitos de System Design

Cinquenta dos conceitos mais importantes de system design, cada um explicado de forma simples e clara: o que faz, por que existe e quando usar.

CONCEITOS	SEÇÕES	EDIÇÃO	FONTE
50	5	2026	DesignGurus

POR QUE IMPORTA

Entender em vez de apenas reconhecer

Aquele tipo específico de constrangimento que atinge engenheiros em reviews de design e entrevistas: alguém menciona um conceito que ele reconhece, mas não consegue definir direito; em vez de perguntar, ele concorda com a cabeça e torce para a conversa seguir adiante antes que peçam para explicar. Todo engenheiro já passou por isso. A palavra é familiar, mas o entendimento por trás dela é raso. System design tem mais desses momentos do que quase qualquer outra área técnica: o vocabulário é rico, os conceitos são abstratos e quase nunca são explicados do zero, porque se assume que todos já os conhecem. O resultado é uma área cheia de engenheiros que usam palavras como idempotência, contrapressão e hash consistente corretamente em frases, mas que teriam dificuldade de explicar, a partir dos primeiros princípios, por que essas coisas existem e que problema resolvem. Este guia cobre 50 conceitos de system design, cada um explicado de forma simples o bastante para você realmente entender — e não apenas reconhecer. Não são definições de dicionário: são explicações reais que mostram o que o conceito é, por que existe, que problema resolve e onde mora o trade-off. A edição 2026 cobre os conceitos fundamentais que já são importantes há anos, junto com os mais novos — especialmente os de sistemas de IA e infraestrutura moderna — que passaram a fazer parte do vocabulário padrão de system design nos últimos anos.

CORE INFRASTRUCTURE

Conceitos Centrais de Infraestrutura

1. **Escalabilidade**
A capacidade de um sistema crescer sem quebrar. Crescimento pode significar mais usuários, mais dados, mais requisições ou maior alcance geográfico. Um sistema escalável expande sua capacidade mantendo o desempenho aceitável. Importante: escalabilidade não é propriedade de um único componente — é do sistema inteiro, que só é tão escalável quanto sua parte menos escalável. Você pode escalar seus servidores de aplicação para aguentar dez vezes o tráfego, e o gargalo simplesmente se move para o banco de dados. Escalabilidade real exige que todas as camadas consigam crescer.

2. **Escala Vertical**
Significa tornar uma única máquina mais poderosa. Quando o sistema precisa de mais capacidade, você faz upgrade do servidor com mais CPU, mais memória ou armazenamento mais rápido. A aplicação e a arquitetura não mudam — você só obtém uma máquina maior. É simples e funciona bem... até deixar de funcionar. O problema: máquinas individuais têm um tamanho máximo, e uma única máquina é um ponto único de falha. Não há redundância: quando ela cai, tudo cai junto.

3. **Escala Horizontal**
Significa adicionar mais máquinas em vez de deixar uma maior. Quando o sistema precisa de mais capacidade, você roda mais cópias dele em mais servidores. Sem teto, e a falha de uma máquina reduz a capacidade ligeiramente, em vez de causar uma interrupção total. Exige que os servidores sejam stateless (sem estado) — que não guardem informações entre requisições de um usuário específico. Se um servidor guarda estado, as requisições precisam voltar ao mesmo servidor, o que quebra a flexibilidade que torna a escala horizontal possível.

4. **Balanceador de Carga (Load Balancer)**
Um componente na frente de um grupo de servidores que distribui as requisições de entrada entre eles. Toda requisição chega primeiro ao balanceador, que escolhe um servidor, encaminha e devolve a resposta. Por fora parece um único servidor; por dentro, muitos compartilham o trabalho. Também verifica a saúde dos servidores e para de enviar tráfego aos que estão falhando. Essa detecção e redundância automáticas é o que torna uma frota de servidores resiliente a falhas individuais.

5. **Latência**
O tempo entre enviar uma requisição e receber a resposta — o atraso experienciado por um único usuário. Baixa latência parece instantânea; alta latência parece lenta. Tem várias fontes que se somam: o tempo de tráfego na rede, o tempo do servidor processando e o tempo do banco respondendo à consulta. Reduzir latência significa identificar qual dessas fontes domina e otimizar exatamente essa parte.

6. **Throughput (Vazão)**
É quanto trabalho um sistema completa em um período, normalmente medido em requisições por segundo. Descreve a capacidade total de trabalho, e não a velocidade de uma requisição individual. Latência e throughput se relacionam, mas diferem: um sistema pode ser rápido por requisição e lento no total se só processa uma por vez; outro pode ser lento por requisição e de alto throughput se processa muitas em paralelo. Projetar para um não dá automaticamente o outro.

7. **CDN (Content Delivery Network)**
Uma rede de servidores distribuídos globalmente que guarda cópias em cache de conteúdo estático e as serve a partir da localização mais próxima do usuário. Em vez de cada usuário buscar uma imagem de um servidor em outro continente, ele busca de um servidor na própria cidade. Reduzem a latência de conteúdo que não muda por usuário (imagens, vídeos, scripts) e desafogam enorme tráfego do servidor de origem. O trade-off: o conteúdo em cache pode ficar desatualizado se mudar e o cache não for invalidado rápido.

8

DNS (Domain Name System)

Traduz nomes legíveis, como example.com, em endereços IP que os computadores usam para se encontrar. É o serviço de diretório da internet. Importa em system design porque pode ser usado para roteamento de tráfego, direcionando usuários para servidores diferentes conforme a localização geográfica ou a saúde dos servidores. Simples, mas lento para reagir a mudanças, pois os registros DNS ficam em cache nos clientes e nos resolvers intermediários.

9

API Gateway

Um ponto único de entrada na frente de vários serviços de backend que trata preocupações transversais como autenticação, rate limiting, terminação SSL e roteamento de requisições em um só lugar. Sem um gateway, cada serviço precisa implementar autenticação e rate limiting de forma independente. Com um, isso é centralizado e todos os serviços atrás dele ganham essa capacidade de graça. O trade-off: o gateway vira um componente crítico, que precisa ser altamente disponível e não pode virar gargalo.

10

Proxy Reverso

Recebe requisições em nome de um ou mais servidores e as encaminha. Fica entre o cliente e o servidor, e o cliente não sabe — nem precisa saber — qual servidor real está respondendo. Lida com cache, compressão, SSL e balanceamento de carga. É próximo do load balancer, porém mais amplo em propósito: o load balancer é um tipo de proxy reverso especializado em distribuir tráfego; o proxy reverso pode fazer muitas outras coisas também.

DATA AND STORAGE

Dados e Armazenamento

11 **Banco de Dados**

Um sistema para armazenar, organizar e recuperar dados que persistem além da vida de uma única requisição ou processo. Sem um banco, qualquer dado criado durante uma requisição desaparece quando ela termina. Existem muitos tipos, otimizados para diferentes formatos e padrões de acesso. A escolha do banco é uma das decisões mais consequentes em system design, porque é difícil mudar depois e todo o resto do sistema depende dela.

12 **Banco SQL**

Também chamado relacional, armazena dados em tabelas com linhas e colunas e impõe relações entre elas com chaves estrangeiras. Usa linguagem de consulta estruturada e garante ACID: atomicidade, consistência, isolamento e durabilidade. É a escolha certa quando os dados são estruturados e relacionais, quando o sistema precisa de transações e quando os padrões de consulta são variados ou não totalmente conhecidos de antemão. A principal limitação: escalar escritas além de um servidor exige sharding, o que adiciona complexidade significativa.

13 **Banco NoSQL**

Uma categoria ampla de bancos que não usam o modelo relacional de tabelas: incluem key-value, document, wide-column e grafos. Cada um troca a flexibilidade de consulta e as garantias de consistência do SQL por vantagens específicas em escala, flexibilidade ou otimização de acesso. O erro mais comum é escolher NoSQL porque soa moderno, em vez de porque o modelo de dados realmente se encaixa. São os padrões de acesso que determinam a escolha certa — não a tendência tecnológica.

14 **ACID**

Sigla para Atomicidade, Consistência, Isolamento e Durabilidade — as garantias que uma transação de banco oferece. Atomicidade: todas as operações de uma transação têm sucesso juntas, ou falham juntas. Consistência: o banco passa de um estado válido a outro estado válido. Isolamento: transações concorrentes não interferem entre si. Durabilidade: dados gravados sobrevivem a uma queda. Essas garantias tornam bancos relacionais a escolha certa para transações financeiras e operações onde a conclusão parcial é pior que a falha total.

15 **Índice (Index)**

Uma estrutura de dados separada que o banco mantém para tornar certas consultas mais rápidas. Sem índice, encontrar linhas que correspondem a uma condição exige escanear todas as linhas da tabela. Com índice, o banco pula direto para as linhas correspondentes. O trade-off: índices aceleram leituras, mas desaceleram escritas — cada insert, update ou delete precisa atualizar todos os índices da tabela afetada. Boa indexação significa indexar as consultas que você realmente executa, não toda coluna.

16 **Sharding**

Divide os dados entre vários bancos, de modo que cada um guarde apenas uma parte. Uma chave de shard (shard key) determina qual banco armazena cada dado. Escala tanto a taxa de escrita quanto a capacidade de armazenamento, porque a carga fica dividida entre muitas máquinas. A parte mais difícil é escolher a chave: uma boa chave distribui dados e tráfego uniformemente; uma ruim cria partições quentes, com um shard recebendo a maior parte do tráfego enquanto outros ficam ociosos. Consultas entre shards são caras: rodam em vários e montam os resultados.

17 **Replicação**

Mantém cópias dos dados em várias máquinas. O padrão mais comum é líder-seguidor: um nó primário lida com todas as escritas, e os nós réplica guardam cópias que atendem leituras. Isso escala a capacidade de leitura e oferece redundância. A sutileza importante é o lag de replicação — o atraso entre uma escrita no primário e seu aparecimento nas réplicas. Nessa janela, leituras nas réplicas retornam dados desatualizados. Costuma ser aceitável, mas precisa ser tratado deliberadamente quando uma leitura deve refletir uma escrita recente.

18 **Cache**

Uma camada de armazenamento rápida que guarda cópias de dados acessados com frequência para evitar operações lentas repetidas. O uso mais comum é um cache em memória na frente do banco, atendendo leituras da memória e só indo ao banco em caso de miss. O trade-off é a obsolescência: o cache guarda uma cópia que pode ficar desatualizada quando a fonte muda. Projetar o cache é decidir o quão velho cada dado pode ficar e por quanto tempo, o que define a política de expiração e invalidação.

19 **Cache-Aside**

O padrão de cache mais comum. A aplicação consulta o cache primeiro. Em acerto (hit), retorna o valor em cache. Em erro (miss), lê do banco, guarda o resultado no cache e o retorna. O cache é populado de forma preguiçosa, apenas conforme os dados são solicitados. É o padrão certo por padrão: é simples, preenche naturalmente o cache com os dados realmente acessados e degrada graciosamente quando o cache está indisponível, pois a aplicação volta ao banco.

20 **Write-Through**

Padrão de cache em que toda escrita vai para o cache e para o banco simultaneamente. O cache está sempre consistente com o banco, porque toda atualização atinge ambos. O trade-off é a latência de escrita: cada escrita precisa ser concluída no cache e no banco antes de retornar ao chamador. Faz sentido quando a consistência leitura-após-escrita é crítica e o custo de latência é aceitável.

21 **Write-Behind**

Padrão de cache em que as escritas vão para o cache imediatamente e são liberadas ao banco de forma assíncrona. As escritas parecem rápidas, pois concluem assim que o cache as confirma. O risco é a perda de dados: se o cache falhar antes de liberar as escritas ao banco, elas se perdem. É apropriado para cargas de alto volume de escrita com alguma perda aceitável, como ingestão de eventos de analytics.

22 **Hash Consistente (Consistent Hashing)**

Técnica para distribuir dados entre nós que minimiza a reordenação quando nós são adicionados ou removidos. Em um anel de hash, cada nó é responsável por uma faixa de valores de hash. Quando um nó é adicionado ou removido, apenas as chaves da faixa adjacente precisam se mover. Sem isso, adicionar um nó a um cluster de dez pode exigir mover 90% dos dados; com hash consistente, mover cerca de 10%. Por isso é a técnica padrão de caches e bancos distribuídos cujo tamanho de cluster muda com o tempo.

23 **Object Storage (Armazenamento de Objetos)**

Um sistema para armazenar grandes arquivos não estruturados como objetos identificados por uma chave. É infinitamente escalável, barato por gigabyte e altamente durável. O exemplo mais comum é o Amazon S3. É o lugar certo para imagens, vídeos, documentos e backups. Não é um banco de dados e não deve ser usado como tal: não suporta transações, consultas complexas nem acesso aleatório de baixa latência a pequenas partes de um arquivo.

24 **Particionamento de Dados**

A prática geral de dividir dados em partes separadas para armazenamento ou processamento. O sharding é uma forma de particionamento; o particionamento por tempo, em que dados de cada mês vão para uma tabela separada, é outra. O particionamento melhora o desempenho de consultas ao escanear menos dados, gerencia o ciclo de vida arquivando ou excluindo partições antigas e distribui a carga entre máquinas. A chave de partição determina qual partição guarda cada dado e deve ser escolhida com base no padrão de consulta dominante.

25 **Event Sourcing**

Armazena o estado de um sistema não como os valores atuais, mas como a sequência de eventos que produziu esses valores. Em vez de guardar que um saldo é R$ 100, o sistema guarda as transações que levaram a esse saldo: deposita 200, saca 50, saca 50. O estado atual é derivado reproduzindo os eventos. Oferece trilha de auditoria completa, facilita reconstruir modelos derivados e permite consultas de viagem no tempo — qual era o estado em qualquer ponto do passado. O trade-off: consultar o estado atual exige reproduzir muitos eventos, a menos que se mantenha um snapshot.

DISTRIBUTED SYSTEMS

Sistemas Distribuídos

26 **Sistema Distribuído**

Um sistema em que componentes rodam em várias máquinas que se comunicam por uma rede para alcançar um objetivo comum. São mais capazes que sistemas de uma única máquina, mas introduzem uma classe de problemas que não existe nela: falhas de rede, falhas parciais e o desafio de manter várias cópias de dados consistentes. Entender sistemas distribuídos é aceitar que a rede é não confiável, que relógios de máquinas diferentes discordam e que qualquer componente pode falhar a qualquer momento.

27 **Teorema CAP**

Afirma que um sistema distribuído pode garantir no máximo duas de três propriedades: consistência, disponibilidade e tolerância a partição. Como partições de rede são inevitáveis em qualquer sistema distribuído real, a tolerância a partição é obrigatória; a escolha real durante uma partição é entre consistência e disponibilidade. Um sistema que escolhe consistência rejeitará requisições em vez de servir dados desatualizados; um que escolhe disponibilidade continua atendendo, mas pode devolver dados desatualizados. A maioria faz essa escolha por tipo de dado: registros financeiros favorecem consistência; contadores de redes sociais favorecem disponibilidade.

28 **Consistência Forte**

Cada leitura reflete a escrita mais recente. Não importa qual nó atende a leitura: ele vê os dados mais recentes. Isso exige coordenação entre nós em cada escrita, o que adiciona latência. É a escolha certa quando servir dados desatualizados causaria problema: saldos bancários, níveis de estoque e assentos em eventos de alta demanda. O custo é latência maior e disponibilidade reduzida durante falhas.

29 **Consistência Eventual**

As réplicas convergem para o mesmo estado com o tempo, mas podem diferir brevemente após uma escrita. Uma leitura imediatamente após uma escrita pode retornar o valor antigo se for para uma réplica que ainda não recebeu a atualização. É mais barata e disponível que a consistência forte. É a escolha certa quando uma pequena defasagem é aceitável: curtidas, seguidores e scores de recomendação. O essencial é ser deliberado sobre quais dados toleram defasagem e quais não.

30 **Consenso**

O problema de fazer vários nós de um sistema distribuído concordarem sobre um único valor apesar de falhas. É a base da eleição de líder, de locks distribuídos e de logs replicados. Algoritmos como Raft e Paxos resolvem o consenso: garantem que uma maioria de nós concorde antes de um valor ser confirmado, o que impede cenários split-brain em que dois nós acreditam ser o líder. Consenso é caro, pois exige múltiplas idas e vindas de rede; por isso é usado apenas onde a concordância deve realmente ser garantida.

31 **Eleição de Líder**

Processo pelo qual nós distribuídos concordam sobre qual deles é responsável por coordenar o trabalho. O líder lida com escritas, coordena operações distribuídas ou gerencia recursos que não devem ser duplicados. Exige consenso para evitar que dois nós acreditem ser o líder e aceitem escritas conflitantes. Quando o líder falha, uma nova eleição acontece entre os nós restantes. A janela entre a falha do líder e a chegada de um novo é um período de capacidade reduzida, em que o sistema não executa operações dependentes do líder.

32 **Idempotência**

Uma operação é idempotente se executá-la várias vezes produz o mesmo resultado que executá-la uma vez. HTTP GET é naturalmente idempotente: ler duas vezes o mesmo recurso não o altera. HTTP POST não é: criar um recurso duas vezes cria dois recursos. Idempotência importa porque sistemas distribuídos não podem garantir que uma requisição seja entregue exatamente uma vez — ela pode se perder, a resposta pode se perder, ou o chamador pode repetir sem saber se a primeira tentativa teve sucesso. Operações idempotentes são seguras de repetir; as não idempotentes, quando repetidas, podem causar duplicatas.

33 **Chave de Idempotência**

Um identificador único que um cliente anexa a uma requisição para que o servidor detecte e ignore duplicatas. O servidor guarda quais chaves já processou e retorna o resultado original se a mesma chave chegar de novo, sem re-executar a operação. É o mecanismo padrão para tornar operações não idempotentes seguras de repetir. APIs de pagamento as usam para evitar cobrança dupla quando uma falha de rede causa uma nova tentativa. A chave precisa ser gerada pelo cliente antes da primeira tentativa, para ser enviada a cada repetição.

34 **Two-Phase Commit (2PC)**

Um protocolo para tornar uma transação atômica em vários bancos ou serviços. Na fase de preparação, o coordenador pergunta aos participantes se podem confirmar. Na fase de confirmação, se todos disserem sim, o coordenador manda todos confirmarem; se algum disser não, todos abortam. O problema: os participantes seguram locks entre as duas fases. Se o coordenador cair nessa janela, os participantes ficam bloqueados aguardando indefinidamente. Isso torna o 2PC frágil em sistemas distribuídos grandes, e é por isso que o padrão Saga é mais comum.

35 **Padrão Saga**

Uma alternativa a transações distribuídas para operações que atravessam vários serviços. Divide a operação em uma sequência de transações locais, cada uma com uma ação compensatória que a desfaz se um passo posterior falhar. Se uma colocação de pedido envolve cobrar pagamento, reservar estoque e criar envio, a saga roda cada passo localmente. Se a criação do envio falha, a saga roda as ações compensatórias: libera o estoque e estorna o pagamento. O sistema chega à consistência eventual por compensação, e não por atomicidade.

36 **Anel de Hash Consistente**

Uma técnica que organiza valores de hash em um círculo e atribui a cada nó uma faixa de valores. Os dados são armazenados no primeiro nó no sentido horário a partir de seu valor de hash. Quando um nó é adicionado ou removido, apenas as chaves da faixa adjacente mudam de nó. Nós virtuais — em que cada nó físico ocupa várias posições no anel — melhoram a uniformidade da distribuição e permitem mudanças graduais de capacidade. É a técnica padrão por trás de caches distribuídos como o Redis Cluster.

37 **Clock Skew (Dessincronização de Relógio)**

A diferença de tempo entre relógios de máquinas diferentes. Mesmo com protocolos de sincronização, relógios de máquinas diferentes se afastam com o tempo e podem diferir por milissegundos ou mais. Quebra qualquer lógica que dependa de timestamps de máquinas diferentes para estabelecer ordem de eventos: dois eventos com timestamps a cinco milissegundos de diferença podem ter ocorrido na ordem oposta. Por isso, sistemas distribuídos usam relógios lógicos e vetoriais em vez de tempo de parede para estabelecer ordem causal sem depender de relógios sincronizados.

38 **Relógio Vetorial (Vector Clock)**

Um mecanismo para rastrear a ordem causal de eventos entre nós distribuídos sem depender de relógios sincronizados. Cada nó mantém um contador, e o relógio vetorial captura quais eventos precedem causalmente quais outros. Quando uma mensagem é enviada, o remetente inclui seu relógio vetorial atual; o receptor atualiza o seu tomando o máximo de cada posição. Isso permite ao sistema determinar não apenas quando eventos aconteceram, mas se um pode ter causado outro — a informação necessária para resolver conflitos em sistemas de consistência eventual.

MESSAGING

Mensageria e Comunicação

39 **Message Queue (Fila de Mensagens)**
Um componente que segura mensagens de produtores até que consumidores estejam prontos para processá-las. O produtor envia uma mensagem e segue em frente, sem esperar o consumidor terminar; o consumidor processa no seu próprio ritmo. Filas desacoplam produtores e consumidores no tempo, absorvendo picos de tráfego e permitindo que os dois lados escalem de forma independente. O trade-off: os resultados não ficam imediatamente disponíveis, pois o processamento é assíncrono.

40 **Pub/Sub**
Um padrão de mensageria em que produtores publicam mensagens em um tópico e qualquer número de consumidores assina para receber cópias. Diferente de uma fila, em que cada mensagem vai a um consumidor, o pub/sub transmite cada mensagem a todos os assinantes. É usado quando várias partes independentes do sistema precisam reagir ao mesmo evento. Um evento 'pedido criado' pode disparar reserva de estoque, envio de notificação e registro de analytics simultaneamente — e o pub/sub deixa cada um acontecer sem o serviço de pedido se importar com quem está ouvindo.

41 **Dead-Letter Queue (Fila de Mensagens Mortas)**
Guarda mensagens que falharam no processamento após um número definido de tentativas. Em vez de deixar uma mensagem ruim bloquear a fila principal repetindo indefinidamente, a fila a move para o lado após atingir o limite de tentativas. Sem uma fila morta, uma única mensagem malformada pode parar o processamento de tudo o que vem atrás. Com uma, a mensagem ruim é estacionada para inspeção enquanto o resto da fila continua fluindo. Monitorar a fila morta é essencial para operar qualquer sistema orientado a mensagens.

42 **Contrapressão (Backpressure)**
Um mecanismo em que um consumidor sinaliza ao produtor para desacelerar quando não consegue acompanhar o ritmo. Sem contrapressão, um produtor rápido e um consumidor lento levam a uma fila sem limites que cresce até esgotar a memória. A contrapressão torna a sobrecarga visível e gerenciável, em vez de deixá-la acumular silenciosamente. É implementada com fila limitada que rejeita novas mensagens quando cheia, forçando o produtor a esperar, ou com sinais explícitos de controle de fluxo do consumidor ao produtor.

43 **WebSocket**
Um protocolo que estabelece uma conexão bidirecional persistente entre cliente e servidor. Uma vez estabelecida, qualquer lado pode enviar dados a qualquer momento, sem o custo de abrir uma nova conexão para cada mensagem. É a escolha certa para recursos em tempo real de baixa latência e comunicação bidirecional: chat, colaboração ao vivo, jogos multiplayer e negociação ao vivo. O desafio em escala: cada conexão é stateful e duradoura, o que exige, para escalar horizontalmente, um registro de conexões e um backbone pub/sub para rotear mensagens ao servidor certo.

44 **Server-Sent Events (SSE)**
Um protocolo para transmitir atualizações unidirecionais do servidor para o cliente em uma única conexão HTTP que permanece aberta. O servidor pode empurrar eventos ao cliente a qualquer momento, mas o cliente não pode enviar dados de volta pela mesma conexão. SSE é mais simples que WebSockets e inclui reconexão integrada, o que o torna a escolha certa quando apenas o servidor precisa empurrar dados: notificações ao vivo, preços de ações, streaming de resposta de IA e atualizações de dashboards. A falta de bidirecionalidade é uma característica, não uma limitação, para esses casos.

RELIABILITY AND MODERN

Confiabilidade, Desempenho e Conceitos Modernos

45 **Circuit Breaker (Disjuntor)**

Monitora chamadas a uma dependência e para de fazê-las quando a taxa de falhas excede um limite. Tem três estados: fechado (operação normal), aberto (dependência falhando, todas as chamadas rejeitadas imediatamente) e meio-aberto (testando a recuperação com poucas chamadas de teste). Impede que uma dependência com falhas arraste para baixo o serviço que dela depende. Sem um, um banco lento ou com falha faz os servidores de aplicação acumularem requisições esperando, até esgotarem recursos e falharem também. Com um, a aplicação falha rápido quando o banco não está saudável, liberando recursos e dando ao banco espaço para se recuperar.

46 **Rate Limiting (Limitação de Taxa)**

Limita quantas requisições um cliente pode fazer em um período. Quando o limite é excedido, novas requisições são rejeitadas com um código de status específico até o limite reiniciar. Protege serviços de sobrecarga e abuso e garante uso justo entre todos os clientes. O algoritmo token bucket é o mais comum: cada cliente tem um balde que enche com tokens a uma taxa fixa e esvazia conforme as requisições são feitas. Isso permite rajadas curtas enquanto impõe uma taxa média no longo prazo.

47 **Load Shedding (Descarga de Carga)**

A rejeição deliberada de requisições em excesso quando um sistema está sobrecarregado. Em vez de tentar atender todas as requisições mal e desmoronar, o sistema atende bem o que consegue e recusa o restante com um sinal claro para tentar mais tarde. É o princípio de que disponibilidade parcial é melhor que falha total: um sistema que atende 80% das requisições com sucesso é muito mais útil que um que tenta todas e falha todas. O shedding deve ser priorizado: descartar trabalho de menor prioridade primeiro, para preservar capacidade para as operações mais críticas.

48 **Bloom Filter (Filtro de Bloom)**

Uma estrutura de dados probabilística eficiente em espaço que responde consultas de pertencimento: este item está no conjunto? Pode dizer definitivamente que um item NÃO está no conjunto, mas só pode dizer que um item PODE estar, com uma taxa de falsos positivos ajustável. É usado quando o custo de um falso negativo (perder um membro real) é alto, mas uma pequena taxa de falsos positivos é aceitável, e quando a memória é limitada demais para guardar o conjunto completo. Bancos os usam para evitar buscas caras em disco por chaves que definitivamente não existem; crawlers da web, para rastrear URLs já visitadas.

49 **Embedding**

Uma representação numérica de dados — geralmente texto ou imagens — como um vetor de números de ponto flutuante que captura o significado semântico dos dados. Itens com significado parecido têm vetores próximos no espaço de alta dimensão. São a base da busca e recuperação modernas baseadas em IA: ao converter texto em vetores, um sistema pode encontrar documentos semanticamente semelhantes a uma consulta mesmo sem compartilhar palavras. É o que move busca semântica, sistemas de recomendação e geração aumentada por recuperação (RAG).

50 **RAG (Retrieval-Augmented Generation)**

Uma arquitetura que melhora as respostas de um modelo de linguagem ao recuperar informações relevantes de uma base de conhecimento externa e fornecê-las como contexto. Em vez de depender só do que o modelo aprendeu no treinamento, o sistema busca informações atualizadas ou específicas do domínio no momento da consulta. O pipeline tem duas fases: na ingestão, documentos são divididos em blocos (chunked), convertidos em embeddings e armazenados em um banco vetorial; na consulta, a pergunta é embedded e usada para buscar no banco vetorial os blocos relevantes, que são incluídos no prompt enviado ao modelo. O modelo gera uma resposta ancorada no contexto recuperado, em vez de alucinar a partir do conhecimento geral de treinamento.

RESUMO

Principais conclusões

✅ O que levar deste guia

- **Infraestrutura** — Conceitos de infraestrutura como escalabilidade, balanceamento de carga, CDN e API gateway descrevem como o tráfego flui por um sistema, como é distribuído e como o sistema cresce para lidar com mais.
- **Dados** — Conceitos de dados e armazenamento como sharding, replicação, indexação, padrões de cache e hash consistente determinam como os dados são armazenados, acessados e mantidos consistentes à medida que o sistema escala.
- **Distribuídos** — Conceitos de sistemas distribuídos como o teorema CAP, consenso, idempotência, sagas e dessincronização de relógio descrevem os desafios fundamentais de rodar um sistema em várias máquinas e os padrões que os resolvem.
- **Mensageria** — Conceitos de mensageria como filas, pub/sub, contrapressão e WebSockets descrevem como componentes se comunicam de forma assíncrona e como recursos em tempo real são construídos em escala.
- **Confiabilidade** — Conceitos de confiabilidade e modernos como circuit breakers, rate limiting, load shedding, filtros de bloom, embeddings e RAG cobrem como os sistemas permanecem de pé sob estresse e como capacidades de IA são integradas a arquiteturas de produção.
- **Conclusão** — Nenhum conceito existe isolado. Entender por que cada um existe, que problema resolve e quanto custa é o que transforma uma lista de vocabulário na capacidade de projetar sistemas. A edição 2026 reflete a realidade de que conceitos de infraestrutura de IA, como embeddings e RAG, agora fazem parte do vocabulário padrão de system design — tão comuns em entrevistas quanto cache e sharding eram há cinco anos. Esses 50 conceitos cobrem a grande maioria das entrevistas, discussões e reviews de system design: aprenda o que cada um faz, por que existe e quanto custa, e as conversas que pareciam impenetráveis viram conversas em que você tem algo real a dizer.

Traduzido de “50 System Design Concepts Explained Simply (2026 Edition)” — Arslan Ahmad · DesignGurus (Substack) · Jul 2026 · Material de estudo em português. Conteúdo original: designgurus.substack.com