Table Of Contents

- Collecting Requirements
  - Functional Requirements
  - Non-Functional Requirements
- The Basics
  - Client / Server model
  - Stateless vs Stateful Services
  - Databases
  - Caching
  - Queues
  - Scaling
- Trade Offs
  - Consistency vs Availability
  - Latency vs Durability
  - Cost vs Performance
  - Monolith vs Microservices
  - Read vs Write Optimization
  - Real-Time vs Eventually Consistent
- Designing Step by Step (Interviews)
- Additional Resources
  - Books
  - Free Stuff Online
  - Paid Stuff Online

The Ultimate System Design Playbook

System design has quickly become what I believe to be the most important part of Software Engineering. Each decision you make in a design has extremely long term implications, and the effects of the choices you make might not be felt until years down the line when a totally new set of engineers runs into a scaling constraint or bottleneck. The rise of Large Language Models (LLMs) has made this even more apparent, as quickly writing code has never been easier, leaving software engineers like us even more time to hone our design skills.

These skills are not only critical on the job, they are also a necessity to landing roles at top tier tech firms (FAANG) all the way down to early stage startups. No one wants to hire an engineer that can't lay out a scalable plan for an application and justify their choices and trade offs made along the way. Of course, no one is expecting you to design a fully functional YouTube or Instagram in 45 minutes, but they do want to see how you approach the problem:

- Can you gather requirements?
- Do you understand trade-offs?
- Do you consider edge cases, scale, and failure?
- Can you explain your thinking clearly?

This is a concise system design guide, tailored to the kinds of questions you'll get in interviews and the kinds of decisions you'll make when designing real large-scale distributed systems. It's not a textbook. It's not a 300-page monologue on distributed systems theory. Instead, in about 40 pages, you'll learn:

- How to approach any system design interview with confidence
- The most common components and design patterns
- Key trade-offs and failure modes to watch for
- What to practice next and where to go deeper
- Additional resources for further growth

While the primary focus of this guide is real-world systems, I've included specific annotations and side notes throughout the guide for interview prep as well. If a tip or strategy is specifically

tailored for interviews, it will be marked as such. Otherwise, you can assume the guidance applies to general system architecture principles.

Collecting Requirements

In both real world system designs and interviews, most people tend to jump into the design far too quickly. The most successful companies in the world are constantly getting user feedback and opinions before they ever release a product. Why should system design be any different? In this section, I'll cover how to gather requirements, what requirements and metrics you should focus on, and what to do when they inevitably change halfway through.

Functional Requirements

Functional requirements define what features the system should have from a user perspective, and they are the first things you should clarify in a system design. In an interview setting, system design questions are often intentionally vague.

Example: Design Instagram

When approaching a service as broad as Instagram, it is simply not possible to design the entire thing in a 30-45 minute interview. As the engineer, we need to get to the core components (**functional requirements)** of the app that the interviewer is actually looking for. For something like Instagram, here are a few of the paths you could go down:

- Designing how users scroll through a personalized feed
- Implementing account creation, authentication, and session handling
- Ensuring the feed never runs out of new posts
- Handling direct messaging and post sharing
- Building a content moderation system
- Detecting duplicate videos or plagiarized content
- Creating a creator dashboard and analytics backend
- Etc …

Any one or combination of these requirements could take weeks or months to design for a full time engineer when you consider how many small decisions could impact performance for an app with Instagram's scale. Let's say we decided to go with the ability for users to scroll through the feed. Our functional requirements may look something like this:

Functional Requirements

- Users can scroll through the feed and never run out of posts

- Posts are recommended based on who the user follows
- Users do not see posts from blocked accounts
- The feed ranking algorithm updates in real-time based on user behavior (e.g., time spent on a post)

Gathering requirements doesn't have to be complicated, but it does require active questioning. Your job is to take a vague prompt and drill down until you have a clear problem to solve. The best way to do this is by asking targeted, open-ended questions that force the interviewer to make decisions with you.

Example: Design Instagram
"Are we focusing on the user-facing feed or the creator-side tools?"
"Should I handle authentication and user accounts, or assume those services already exist?"
"Do we need to support messaging between users as part of this design?"

As you gather information, don't expect everything to be handed to you. It's expected that you will make reasonable assumptions for things that seem out of scope. For example, if you're designing the feed, you might assume the existence of a User Service that handles following/blocking logic. Expanding upon our Instagram example, here is how I might call out the assumption of some Users API that I **did not** design.
"To avoid recommending blocked users, the Feed Service will make a call to the Users API to validate that none of the next 10 posts come from blocked accounts."

Non-Functional Requirements

Non-functional requirements define how well a system performs under specific conditions, focusing on qualities like scalability, availability, latency, security, and reliability rather than user-facing features. These are just as important as functional requirements. If you are designing an application for ten users, it can almost always be a single virtual machine (VM) running a simple web server. If you are designing for millions or billions of users, it will almost always be a distributed system.

Gathering non-functional requirements in an interview is usually straightforward, but the interviewer usually still expects you to probe them to get these to make sure you understand what non-functional requirements are important. Here are the types of questions I like to ask to get this information.

- *“What kind of scale are we targeting? How many users or requests per second?”*
- *“What are the latency expectations for the user?”*
- *“Do we need to design for high availability or is some downtime okay?”*
- *“What’s more important, consistency or availability?”*
- *“Do we need to handle spikes in traffic (e.g. viral posts or launch events)?”*
- *“Are there any compliance, security, or auditability requirements?”*

These questions aim to gather the following type of information:

Requirement Type	What it affects	Example
Scalability	Database, caching, load balancing	System should support 10M DAUs and 1K writes/sec
Latency	API structure, caching, replication	Users should see feed updates in < 200ms
Availability	Multi-region design, replication, failover	99.99% uptime; tolerate AZ failure
Durability	Storage choices, backup strategy	Media uploads must never be lost after confirmation
Consistency	DB model, replication strategy	Feed should show recent posts even under network partitions
Security & Privacy	Encryption, auth, data isolation	Private messages must be end-to-end encrypted
Observability	Logging, metrics, alerting	The system should expose metrics for feed latency, cache hit rates, etc.

Treat non-functional requirements like hard constraints, they'll often have more influence on your architecture than the features themselves. A basic chat app and a WhatsApp-scale chat platform might have identical functionality, but the non-functional requirements will demand completely different designs. Later in the guide, we'll dive a bit more into how to handle some of these requirements, like trading off strong consistency for lower latency on writes.

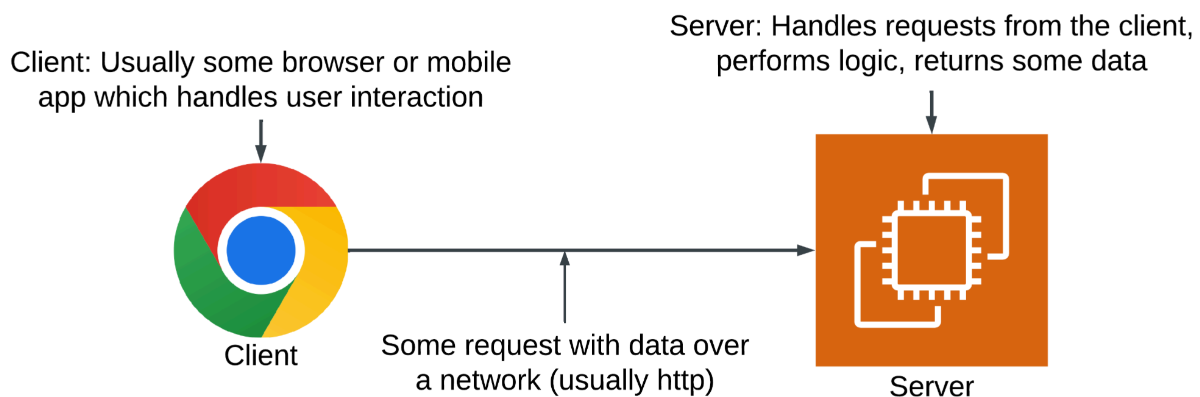
Now that we've gathered both functional and non-functional requirements, it's time to translate them into architectural decisions. To do that effectively, you need a solid understanding of the foundational building blocks of system design, starting with the core components that appear in almost every system.

The Basics

This section introduces the most common components of any large scale system. You can (and should) do more research into each one of these topics, but I will give enough of an overview here to help you get your feet wet and create basic designs which cover most cases.

Client / Server model

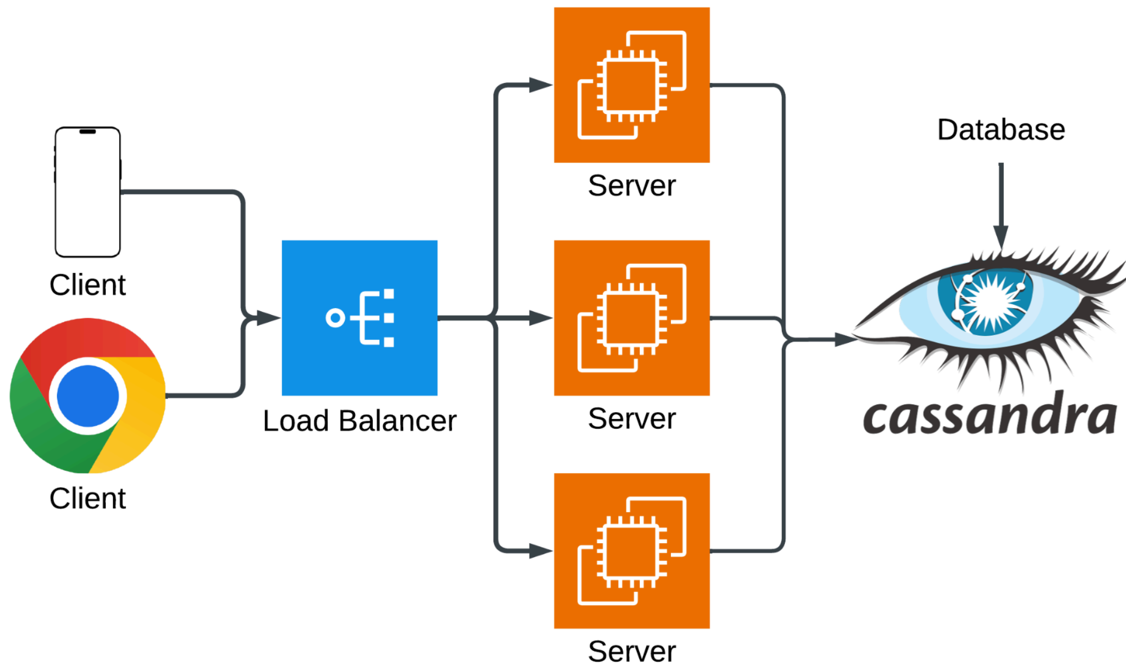
The client server model is the bread and butter of most modern web apps. It is a simple to understand architectural pattern which separates your design into two basic roles.



The client server model

This is the most simple example of a client server architecture, in a large scale distributed system, we may have multiple clients, a bunch of servers running the same code serving those requests, a load balancer distributing traffic with databases and other services getting called on

the backend.



The client server model - distributed

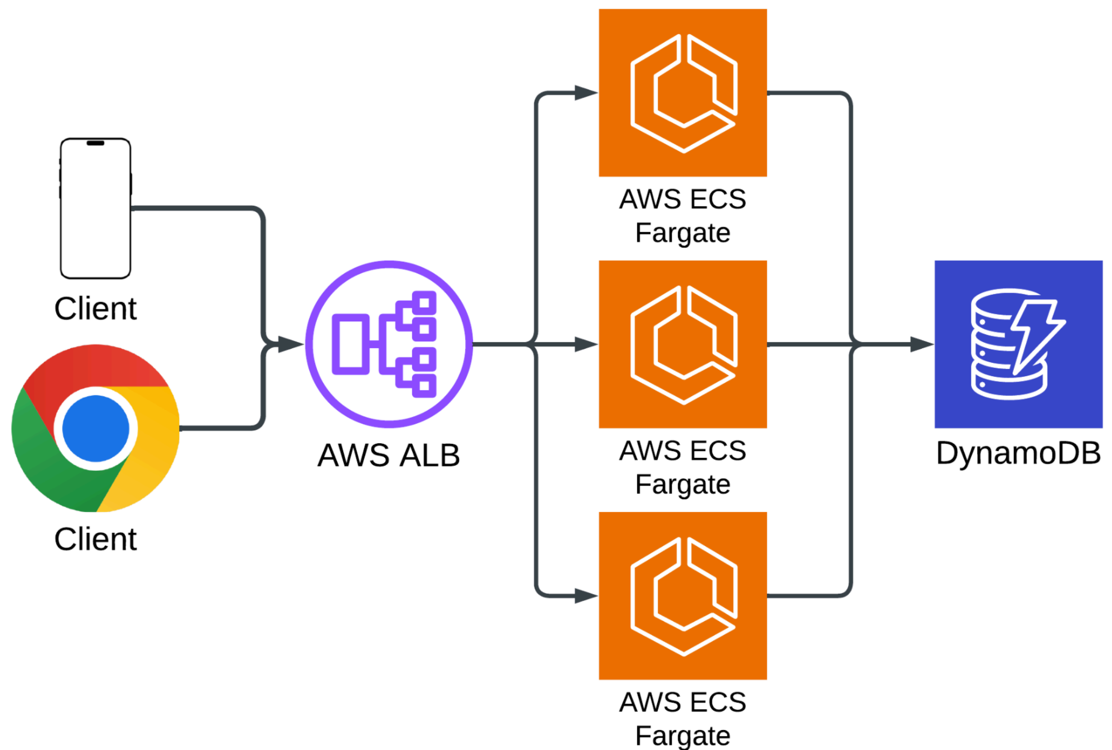
Pretty much every design question you will ever see from a chat app like WhatsApp to a live streaming video platform like Twitch starts with some version of this pattern. Understanding this architecture and interaction is critical for things like:

- Deciding where logic lives (client vs server)
- Planning for performance bottlenecks (e.g. slow APIs, client retries)
- Structuring your authentication and session model
- Thinking about what can be cached on the client vs the server

In a cloud-native system (like an application hosted on AWS):

- Clients send traffic to a load balancer (like AWS ALB or GCP Load Balancer)
- The load balancer routes traffic to stateless compute nodes (e.g. EC2, Fargate, Cloud Run)
- Each server instance should be stateless, so it can be scaled horizontally (this will be covered later in the basics) or replaced without impact

Here is a simple example of the bare components for the same design using AWS services.



The client server model on AWS

If you want to read a little more about this, the MDN docs give a great overview of how this works

Stateless vs Stateful Services

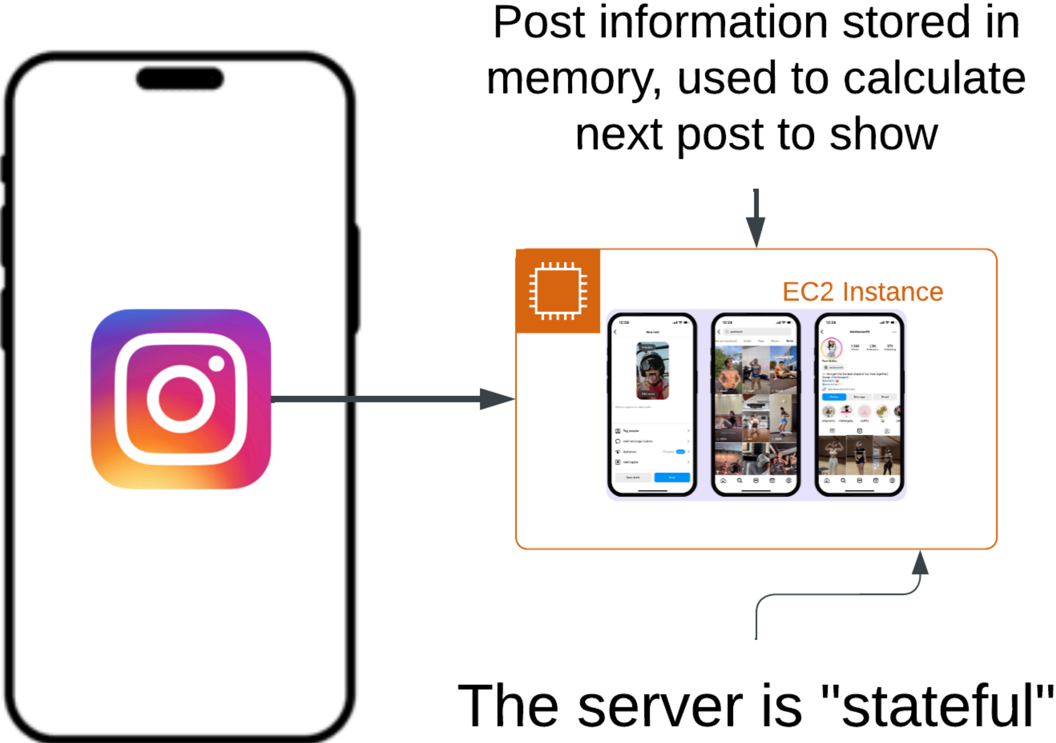
I hate stateful services, and so do most interviewers. That said they have their time and place, so understanding the tradeoffs of each so you can justify a decision either way is critical. Some quick definitions for you:

Stateless services: These do not store any session or user data between requests. Every request is independent and contains all the information the server needs to process it. This makes them easy to scale horizontally, quick to recover from failure, and ideal for modern cloud environments and microservices.

Stateful services: These retain information about past interactions — like sessions, open connections, or cached in-memory data. These are more difficult to scale, and are often

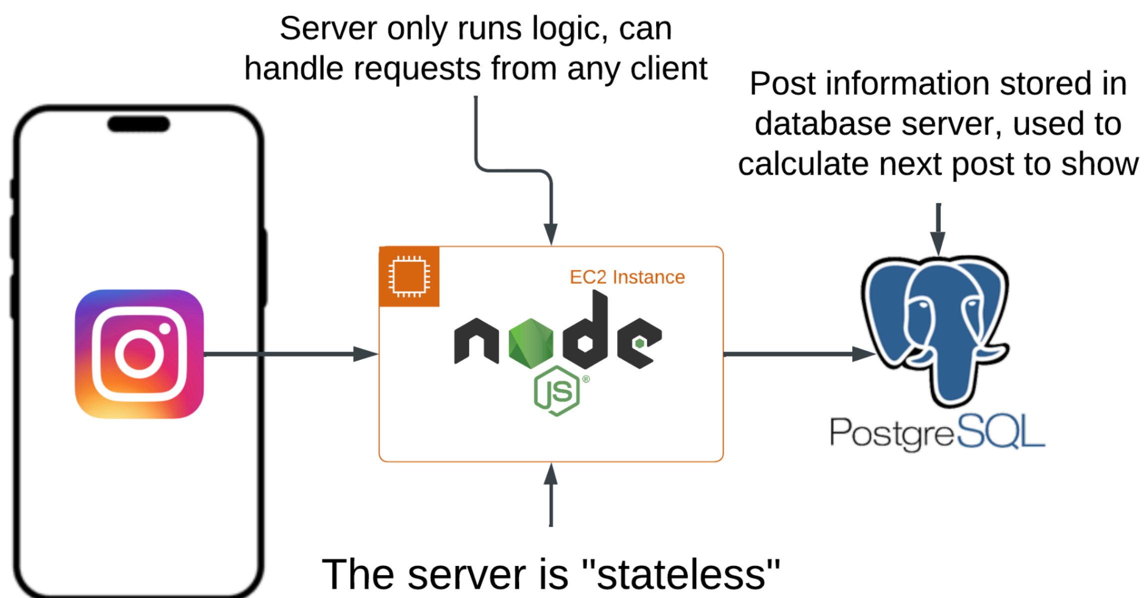
unrecoverable from failures. In exchange we don't need nearly as much information from the client, and we don't have to rely on databases to store all our information as we can just keep it in memory for extremely low latency. This makes stateful systems very common for things like multiplayer games.

In interviews and production cloud systems stateless services are the default unless you have a very specific reason not to. Let's continue with our instagram example. In a stateful design here, the client would make a connection with a server, and pass in some information like which of the past 3 posts in the feed they liked. If the client made another connection with the server, the **same** server would use that information to determine what posts the user might want to see next. The client does not need to re-pass information about recent posts because the server already has it in memory. As you can imagine, we now need to make sure all requests from client A go to server A because only server A has the information in memory.



Stateful server

Now there is a little bit of nuance here that is often misunderstood. While a completely "stateless" system by definition should maintain **no** state of any kind (all information should be passed in by the client), in the real world we do need to keep track of users and their actions. This is done by delegating state to an external system like a database server. So our system still technically has "state" (users are associated with posts and likes in the database) but our server itself is stateless, unlocking a world of benefits for us. If we need to scale, we can add more servers and not worry about always making sure a client is connected to the same server. If one server fails, we can quickly restart it without worrying about what was in memory at the time.



Stateless server

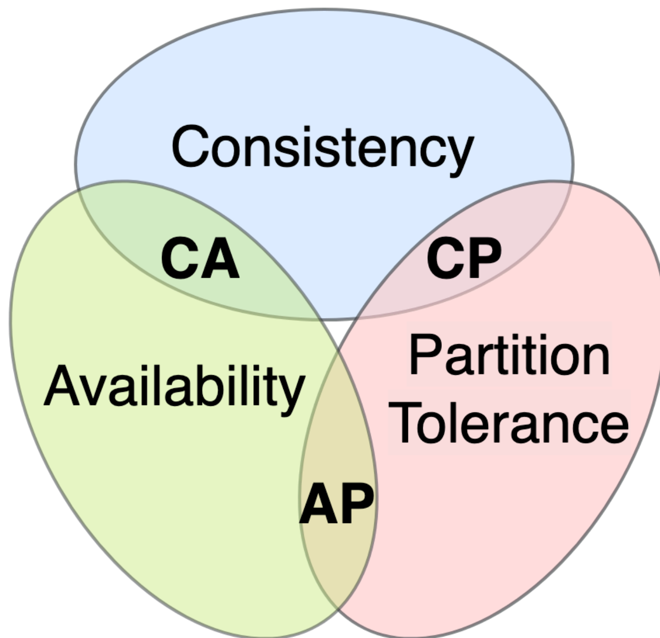
FreeCodeCamp has a [nice article] which explains this very simply if you want to read more. Whether your services are stateless or stateful, almost every system needs to persist data somewhere. This brings us to one of the most critical parts of any design: the database. Choosing the right one can make or break your system.

Databases

It is no understatement to say that picking your database is the most important decision in a system design. It defines how you store, retrieve, and scale your data, and it has huge implications for performance, consistency, availability, and cost. A wrong decision here can cause developers working on your service to face the worst pain imaginable: a database migration. Let's break down the main types of databases and how they behave in the context of the CAP theorem. You can reference this section when we go into selecting a database a little later in this paper.

The CAP theorem states that in any distributed data system, you can only guarantee two of the following three at the same time:
C — Consistency: Every read returns the most recent write
A — Availability: Every request receives a response (even if it's stale)
P — Partition Tolerance: The system continues to operate despite network failures between nodes
You must tolerate partitions in distributed systems (P is a given), so you're often choosing between:
CP systems → Prioritize consistency (but may return errors under network issues)
Ex: A distributed SQL database like CockroachDB ensures all nodes return the most recent data, but during a network partition, it may reject reads/writes to avoid serving stale data.

AP systems → Prioritize availability (but may return stale or eventually consistent data)
Ex: A system like Cassandra remains available during partitions by serving potentially stale data, ensuring the app still responds even if some replicas can't communicate.



The CAP theorem visualized

Keep this in mind when we explore some of the most common types of databases.

Relational Databases (RDBMS)

Relational databases store data in structured tables with rows and columns, enforcing a strict schema and supporting complex relationships through foreign keys and joins. They are ACID-compliant, meaning they guarantee atomicity, consistency, isolation, and durability, making them ideal for systems that require strong data integrity and transactional safety. Common examples include PostgreSQL, MySQL, and SQL Server. These databases are best suited for applications like banking, order processing, or inventory management — places where relational structure and correctness matter most. Under the CAP theorem, relational databases typically lean toward CP (Consistency and Partition Tolerance), meaning they may sacrifice availability during network failures in favor of preserving consistent data.

Key-Value Stores

Key-value stores are simple and incredibly fast, built to retrieve values by a unique key with minimal overhead. There's no rigid schema, and most values are opaque to the system (e.g., strings, JSON blobs). This simplicity makes them easy to scale horizontally, with extremely low latency even under heavy load. Popular examples include DynamoDB, Redis, and Riak. These databases are ideal for caching layers, session management, personalization data, or shopping carts, use cases where speed and scale are more critical than complex querying. From a CAP perspective, key-value stores are usually AP (Availability and Partition Tolerance), favoring high uptime and responsiveness even if that means serving stale or eventually consistent data.

Document Stores

Document databases store data as semi-structured documents, often in formats like JSON or BSON (binary JSON). Unlike relational databases, document stores offer a flexible schema, allowing you to model data more naturally for use cases like content platforms or user profiles. They support nested structures and dynamic fields, making them developer-friendly and fast to iterate with. Examples include MongoDB, Firebase, and Couchbase. These are great for rapidly evolving applications or scenarios where each "record" (document) might look different. In terms of CAP, document stores vary. MongoDB, for example, can operate as either CP (strong consistency) or AP (eventual consistency) depending on how it's configured. This flexibility allows you to trade between performance and consistency based on your needs.

Wide-Column Stores

Wide-column databases organize data into column families like a giant two-dimensional hash map. Each row can have a different number of columns, which makes them efficient for sparse or time-series data. This model is great for high write throughput and large-scale workloads. Systems like Cassandra, HBase, and Google Bigtable fall into this category. These databases are often used for logging systems, telemetry, time-series analytics, or any high-volume write-heavy applications. In CAP terms, wide-column stores typically fall under AP, prioritizing availability and partition tolerance while accepting eventual consistency. For workloads where write availability is more important than up-to-the-second reads, they're often the best choice.

Graph Databases

Graph databases are purpose-built for storing and traversing relationships between entities, using a structure of nodes and edges. They are only useful when your queries are more about how things are connected than what things are. Examples include Neo4j and Amazon Neptune. These databases are ideal for social networks, recommendation engines, fraud detection systems, or any domain where deep, real-time traversal of connections is critical. The CAP behavior of graph databases varies: some are designed to be CP, ensuring consistent graph views even under partition, while others can be tuned for AP characteristics to favor availability.

If you want to read more about databases, I recommend Designing Data Intensive Applications. If you want something a bit lighter, this github has some good resources and prep material. Of course, even the most carefully chosen database can become a bottleneck under heavy load. That's where caching comes in, offering a powerful way to reduce latency and take pressure off your database.

Caching

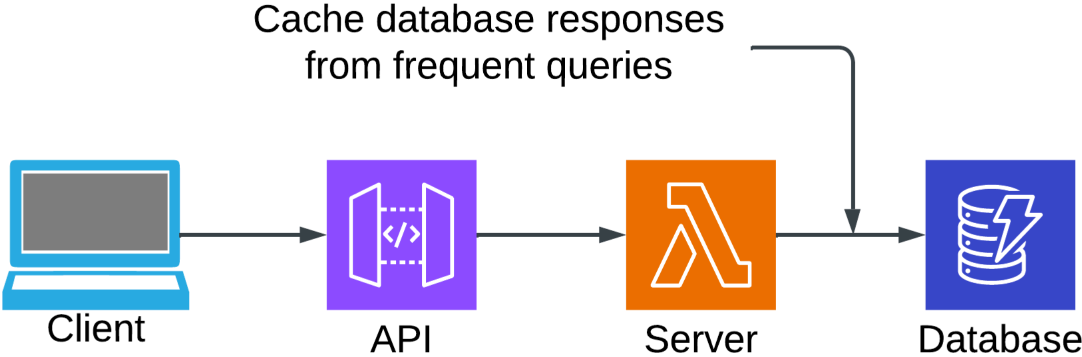
Caching is one of the simplest and most effective ways to speed up your system and reduce load on downstream services. When done well, it can make a slow, expensive operation feel instant, reducing both latency for your clients, and cost on your cloud bill. When done poorly, it can cause stale data, painful bugs, and system-wide inconsistencies.

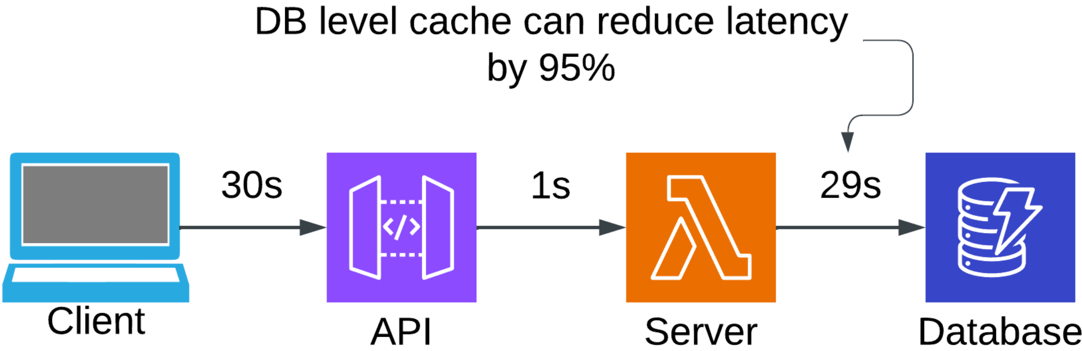
"There are 2 hard problems in computer science: cache invalidation, naming things, and off-by-1 errors."

Cache invalidation: The process of removing or updating stale data in a cache to ensure that future reads return fresh and accurate results.

At its core, caching is about temporarily storing frequently accessed data in a fast, in-memory store so that future requests don't have to hit the slower underlying system. So that begs the question: where should I cache? One of the best / easiest places to cache is between the server and your database. This is useful if running database queries itself is a bottleneck for you. A

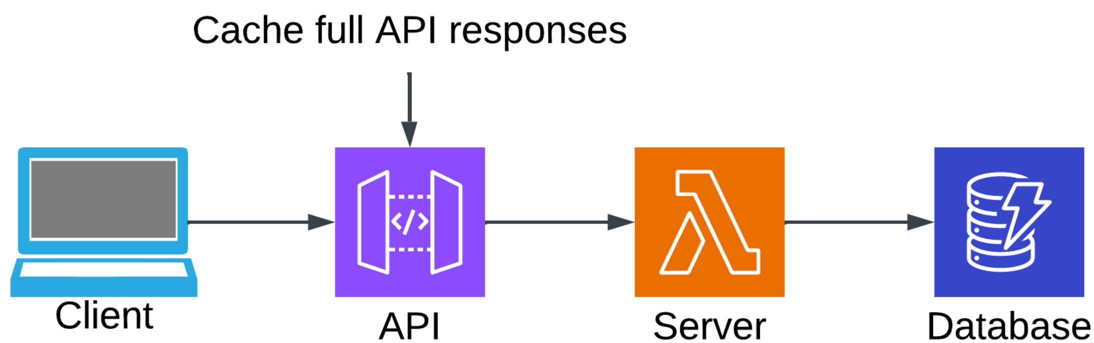
cache at this level is also generally easier to invalidate compared to some of the other places we could cache.

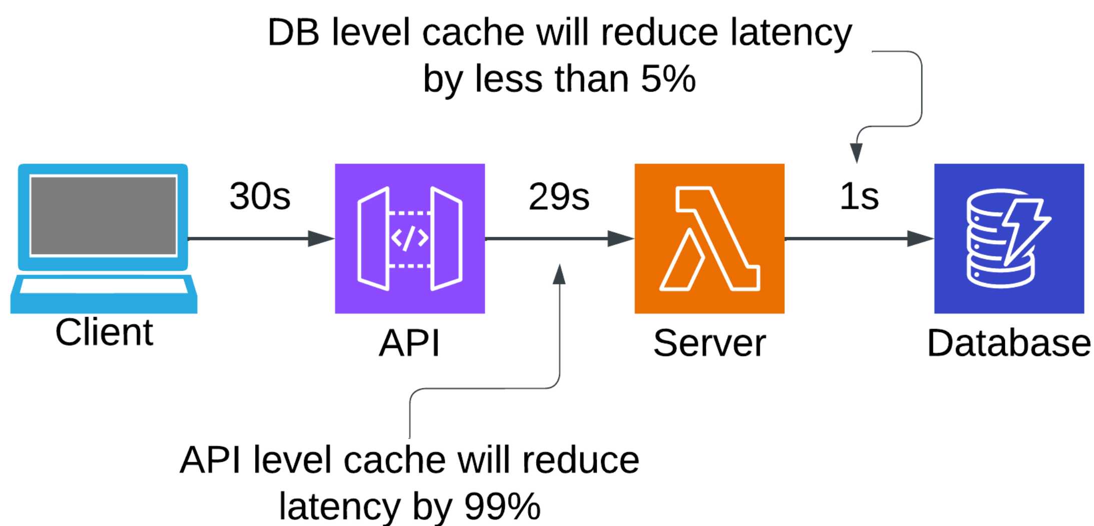




Caching - slow database

If the database isn't necessarily the bottleneck, and most of your latency is coming from processing in the server, adding a cache at the database level isn't very useful. We can instead place a cache at the API level to help with this case. For example if we get 2 identical get requests in a 30 second period, with a cache, we only need to process the request on the server once, which will greatly reduce latency (and reduce compute costs).

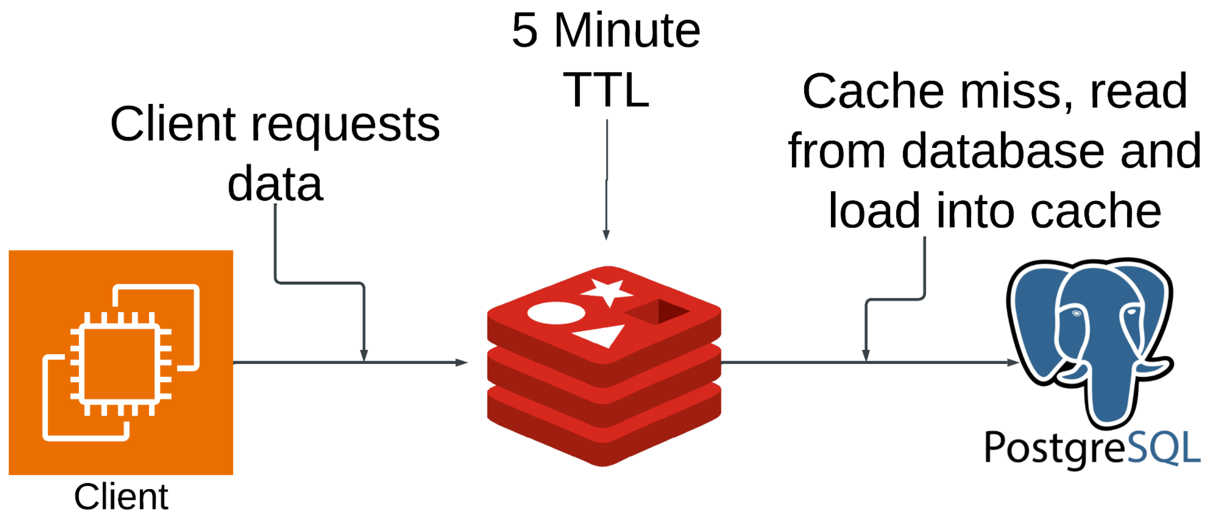




Caching - slow processing

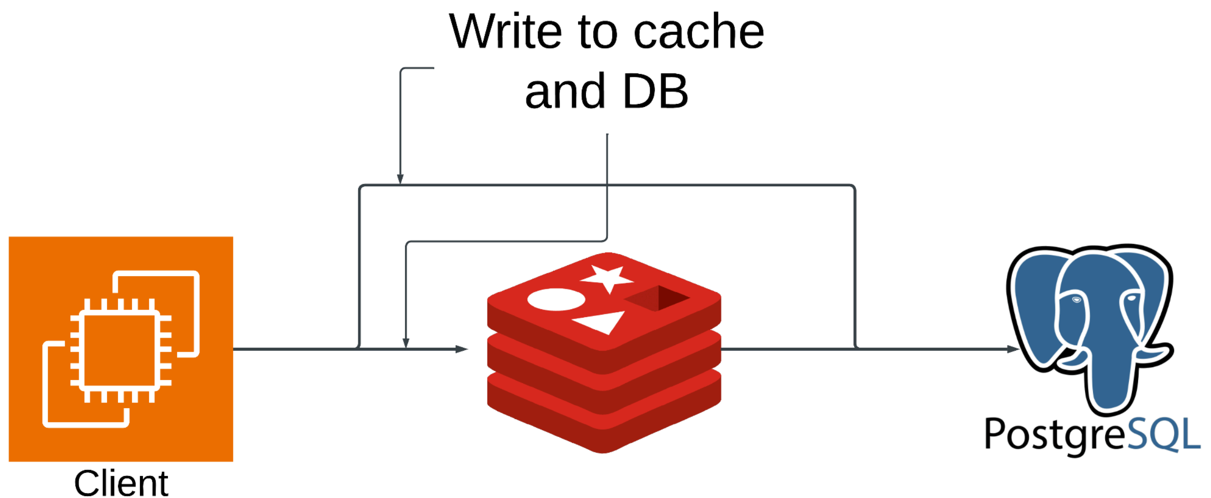
Caching is by no means a silver bullet that will solve all your problems with no trade offs. As soon as we introduce a cache, we now need to decide when to cache data and when to evict it. Let's look at a few eviction policies. The first and easiest to implement and understand is called Time To Live (TTL). If the data is not in the cache when we make a request, bring it into the cache and set a timer. Once that timer is up, evict the data from the cache. The amount of time on the timer is called the time to live. In this example, the data will always be in the cache for

exactly 5 minutes.



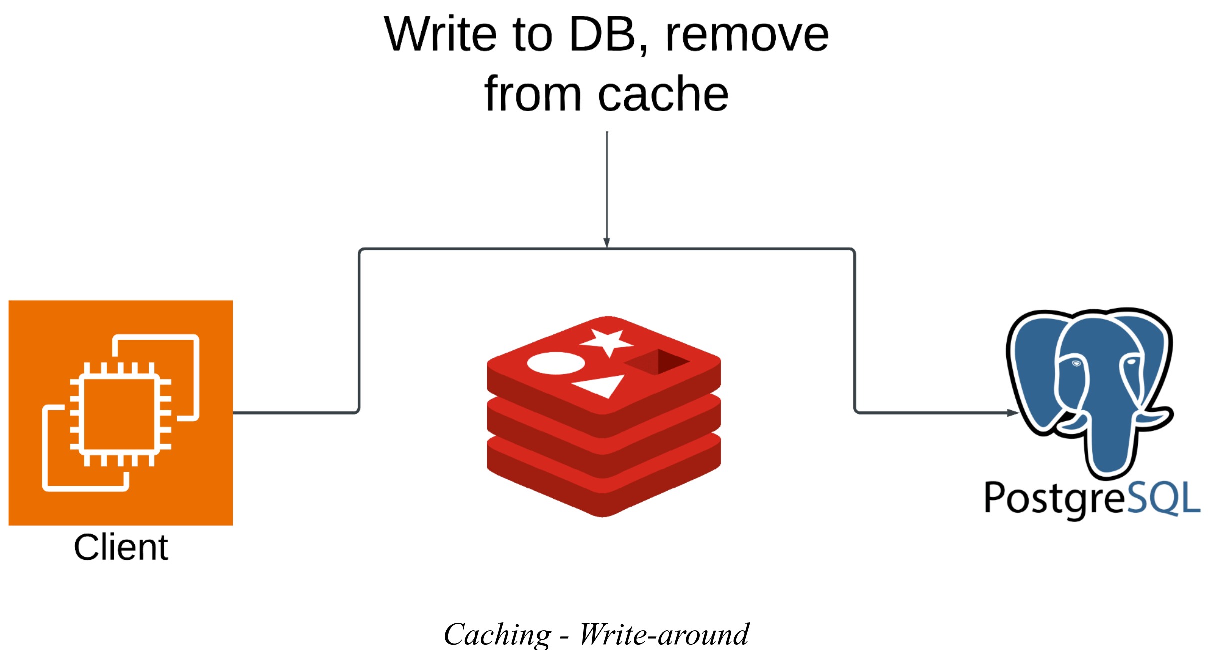
Caching - Time to live

As you can imagine, TTL has some flaws, as someone could easily write to the underlying data store during the 5 minutes your data is cached, and now your users are getting stale data. Of course, there are ways to mitigate this issue. Using a write-through cache, the server will update both the cache and the database any time there is a new write. This mitigates the issue of users getting stale data, but can still lead to some weird issues if one of the writes fails and users now have to deal with the latency of writing to two data stores instead of just one.



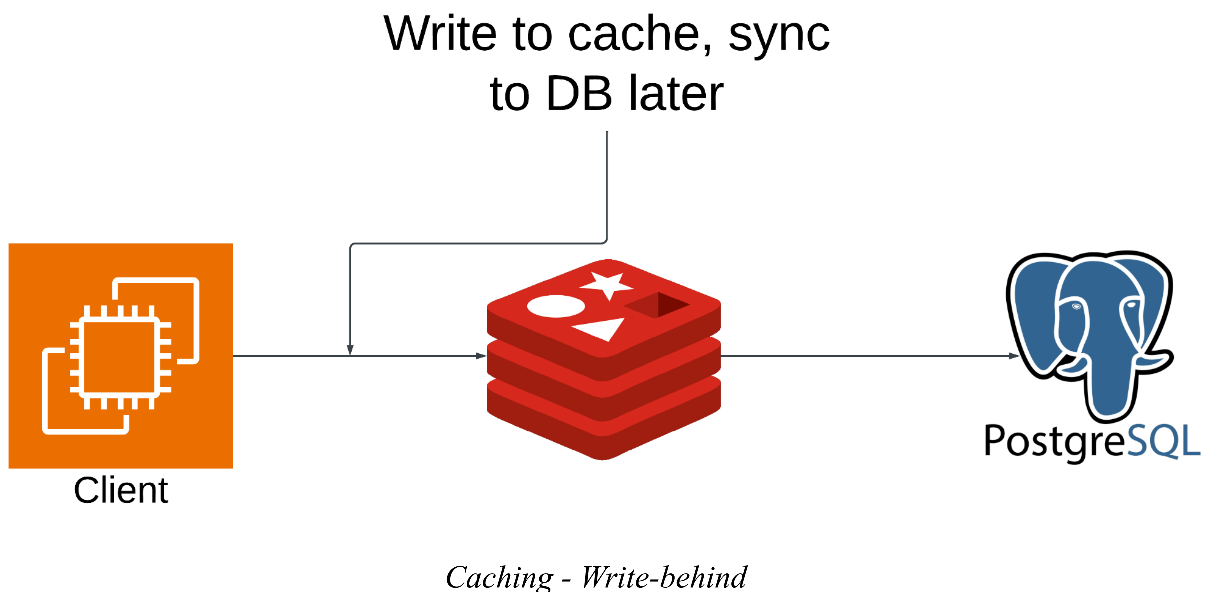
Caching - Write-through

If you don’t want to write to the cache on every update, you'll like the next approach. In a write-around cache, we simply write to the database and remove the old data from the cache. The next time someone tries to get that data, it will be brought back into the cache with the new data you wrote to the database. This is sometimes useful, but I almost always prefer a write through cache.



Caching - Write-around

If your goal is to reduce latency on writes, you probably haven't liked any of the solutions so far. The write-behind cache solves for this. We first write to the cache, which is an extremely low latency operation, then we write to the database. Writing to the database happens asynchronously, separate from the client-facing write acknowledgement, so the client will see a successful write before the data is actually written to the database. While this does make writes super fast, it also introduces some risk. If the cache crashes before we get a chance to update the database, that write is lost. Additionally, if there are other users accessing the database directly, they could see stale data.



Caching - Write-behind

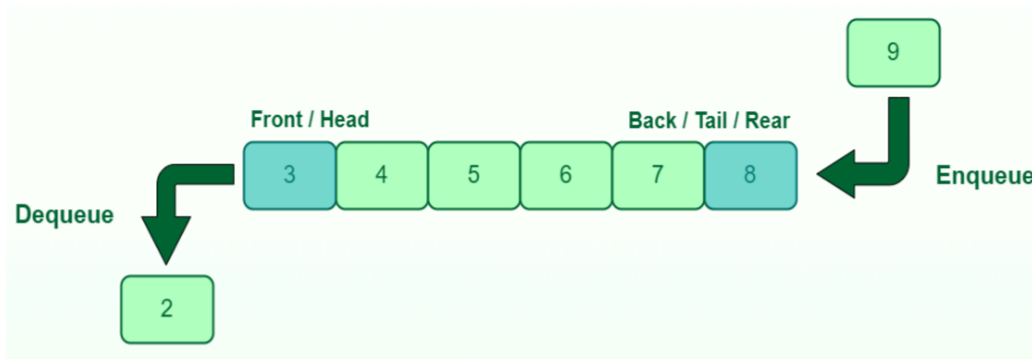
These are not the only cache invalidation strategies that exist, just some of the most common ones you should probably note. It's also worth knowing that all of these policies can be (and often are) combined with TTL. When a user makes a request for data, if it is not in the cache, load it into the cache and give it a TTL. Then you can implement something like a write-through policy to make sure data stays fresh even if users are writing to it.

The same github as the prior section has a nice chapter on caching. While caching helps you serve data faster, it doesn't help when your system is overwhelmed by more requests than it can process in real time. For that, we turn to queues, a key tool for smoothing out traffic and building resilient systems.

Queues

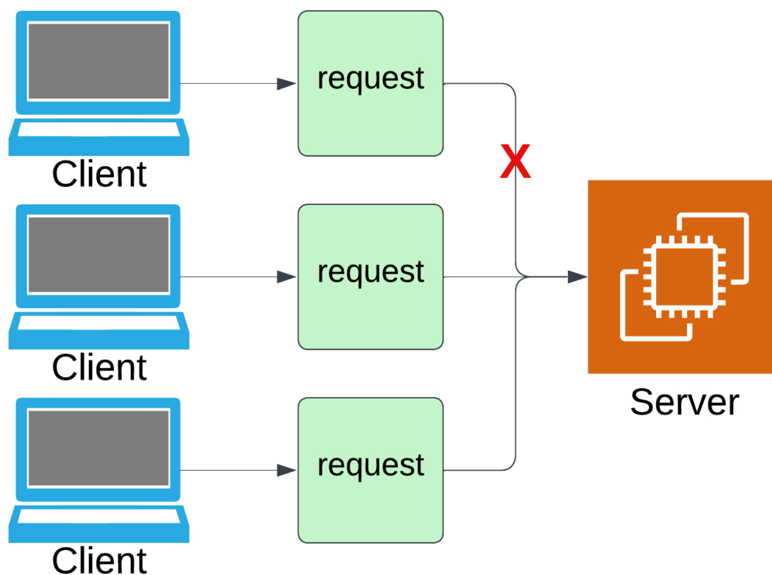
You probably learned about queues in a university data structures and algorithms course, or when you were studying for leetcode interviews. They work exactly the same in a system. When

requests come in, you place them in a queue, and the server processes them as it is able.



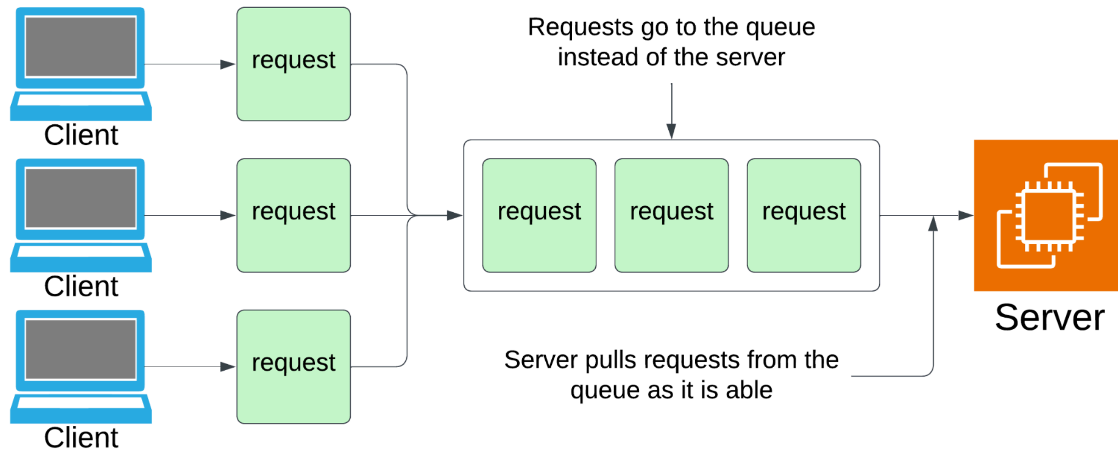
Queue

Let's talk through a few of the major benefits of using a queue in a system design. First and foremost, a queue is your best (and pretty much only) tool for gracefully handling massive spikes in traffic without over-scaling your service. Imagine we have a very bad web server that can only handle 2 requests at a time. The third incoming request at the same time is dropped (500 or 429 error) because the server was busy serving the other 2 requests.



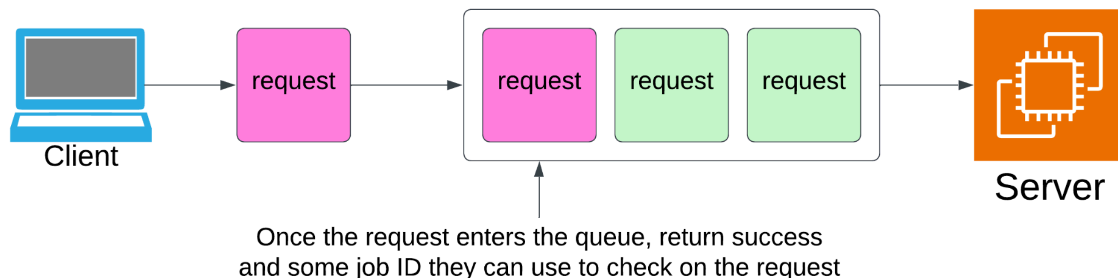
Server with no queue

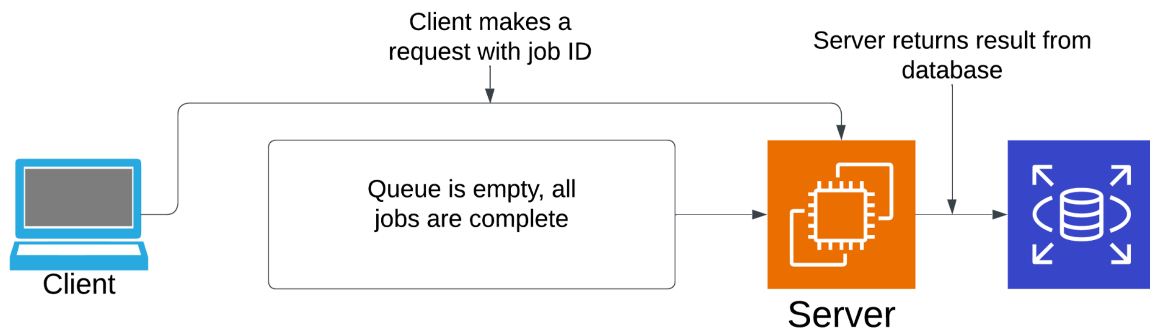
Let's introduce a queue to store these requests before they are sent to the server.



Server with simple queue

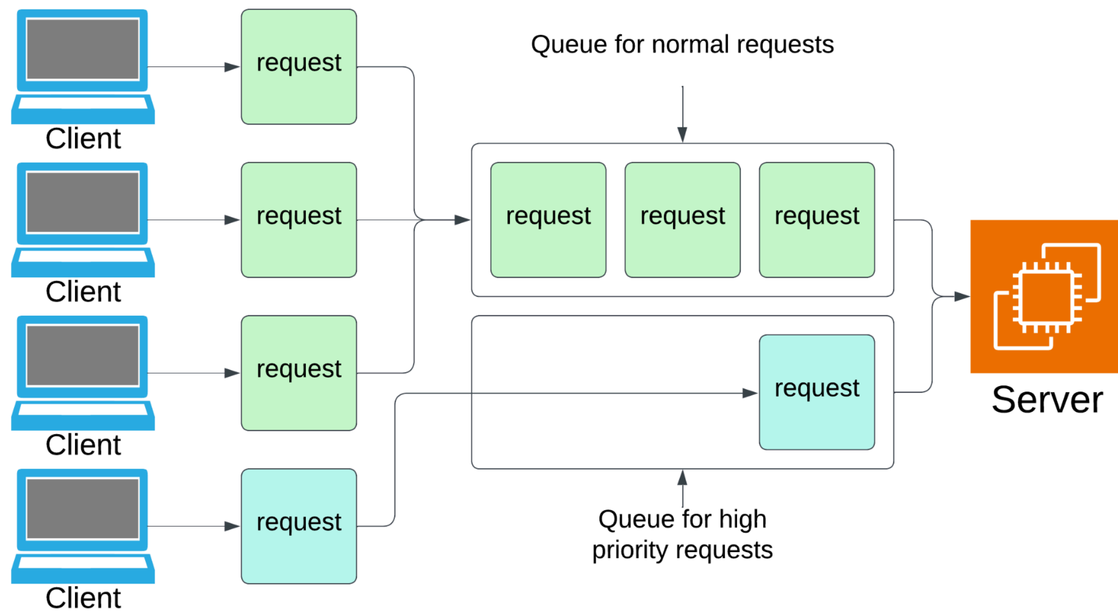
As you can see, even though our server can only process 2 requests at a time, it can now serve all 3 of the requests without dropping any of them. Hopefully you are already wondering about the trade off we are making here (remember, nothing is ever free). By introducing the queue, we are no longer instantly returning a result to the client. Normally we are returning some response indicating that we have accepted their request, and that we are processing it with some ID they can use to make another request to check the status. When they make a request in the future with that job ID, we can return the status of the job, or the finished product if it is complete.





Queues for async jobs

Another nice benefit we get when using queues is that services can scale independently. A producer application (one that adds requests to a queue) can handle as much traffic as it wants at once, and add requests to the queue of a consumer application without worrying about overloading that service, or synchronous responses. This type of architecture is very common for compute heavy things like building applications or video processing. I really like [this website](), it has some great visualizations of using queues and how they help smooth things out.
While I love queues and think they are actually very underutilized in both real world systems and interviews, they do come with a few giant pitfalls you have to watch out for. The first of which is the ever-expanding queue. Imagine you have a server which can process 50 transactions per second (tps), but requests are getting enqueued at a steady rate of 100tps. Your queue will keep getting longer and longer, and customers will be facing massive latency, and eventually never even get a response from your server. There are few things you can do to help mitigate this. Firstly, ADD MORE QUEUES!! We can introduce additional queues for high priority requests, or faster requests so that your server does not get stuck processing one huge job, and ignoring tons of small requests that may take a fraction of the time to complete.



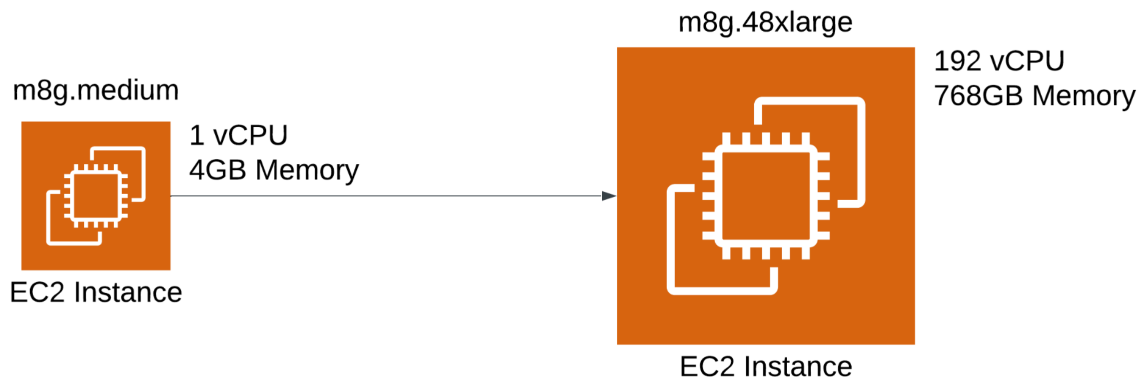
Priority queue

Secondly, you can scale up your servers as your queue gets longer. The nice part about this is your system has time to scale (servers choose when they get requests from the queue) instead of getting bombarded with requests, then having to scale as a reaction to requests getting dropped.

Scaling

At some point, every system outgrows its initial capacity. When that happens, you have 2 choices: scale up (vertical scaling) or scale out (horizontal scaling)

Vertical Scaling

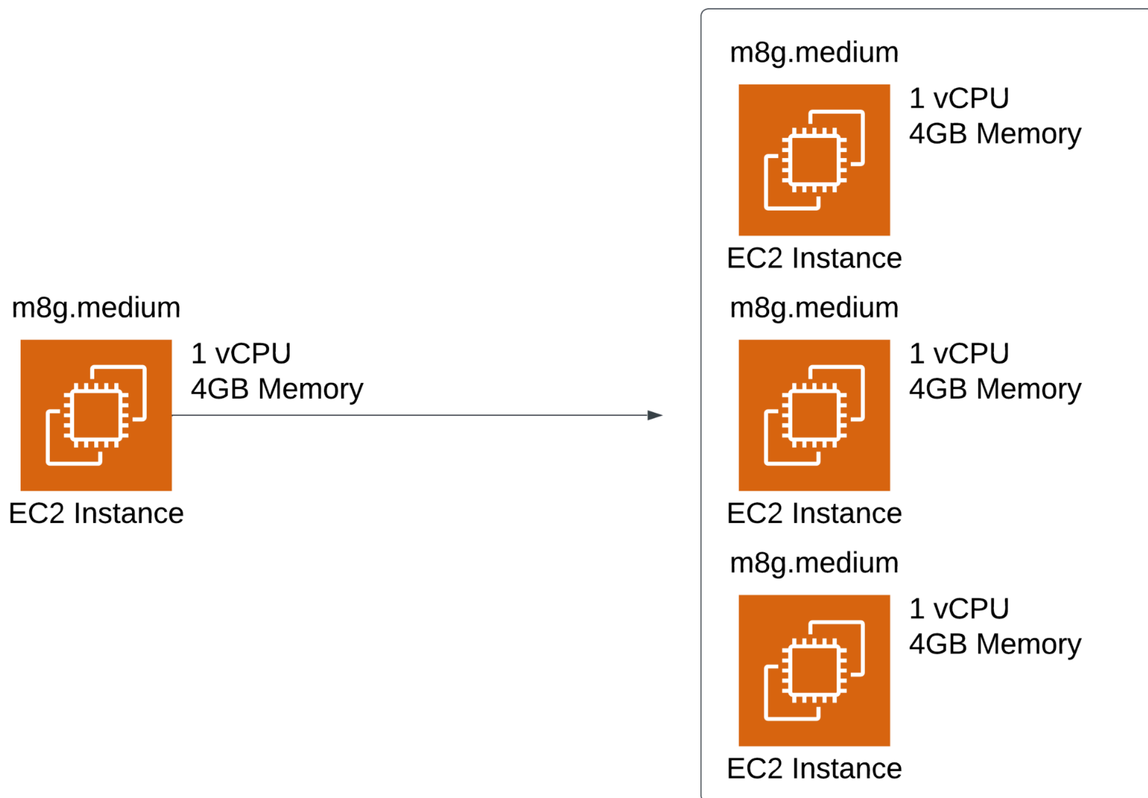


Vertical Scaling

Vertical scaling (scaling up) means increasing the power of a single machine. You're giving the box more CPU, RAM, or disk. On the cloud, this is as simple as paying more for a more powerful EC2 instance that has more of whatever the current bottleneck is in your system. This works well for monoliths and applications where you never intend to serve a huge amount of traffic, like a personal blog. Vertical scaling quickly hits some pretty unsurmountable limitations.

1. Hardware only gets so good. At some point, you can no longer buy a better GPU, or your motherboard runs out of slots for more RAM.
2. Hardware has diminishing returns to price. The best hardware is much more expensive than mediocre hardware. It is usually more cost effective to just get 2 mediocre CPUs instead of spending all your money to buy the best and newest.
3. Single point of failure. If your entire application is running on one machine and that machine crashes, your application is dead. There is no second instance running to pick up that traffic.

Horizontal Scaling



Horizontal Scaling

Horizontal Scaling (scaling out) means adding more machines to handle load. Instead of scaling one beefy server, you run more copies behind a load balancer. This should be your default in most cases. Here are some of the advantages of horizontal scaling

1. You can scale almost infinitely. If you need to serve more requests, just add more machines. You are only limited by money, and the physical amount of compute resources that exist in the world.
2. It improves reliability. You have multiple machines running your app already. If one crashes, you can take whatever traffic it was serving, and move it to another machine seamlessly.
3. It works great on the cloud. Cloud computing has made this a hell of a lot easier. Services like AWS, GCP, or Azure already have a ton of computers sitting around for you to use. You can easily use more or less machines as your traffic fluctuates, and you only need to pay for what you are currently using.

As I have repeated many times, nothing is without trade offs. With a single machine, we know all traffic is served by the same instance, so if we need to store session data, it is quite easy. Now that we are trying to distribute compute across multiple machines, we need to either make sure all servers are stateless (you remember this from above, right??), or handle sticky sessions with a load balancer. We also add the added complexity of managing a fleet of servers instead of just a single server. Something needs to monitor our servers, which are healthy, which have too much traffic, which aren't getting any, etc. This alone costs money and can be extremely complex.

Whether you scale vertically or horizontally, every architecture decision involves compromise. Understanding trade-offs is the heart of good system design, and why the next section is arguably the most important.

Trade Offs

"There is no such thing as a free lunch"

Every single decision you make in your design will come with trade offs. These could be small trade offs, like sacrificing a few ms of processing time for cheaper servers, or huge trade offs, like deciding on using an eventually consistent database to achieve drastically faster writes. Understanding and justifying these trade offs is the most important skill that you are tested on in a system design interview. In this section, I will walk through some of the most common ones, and when you should make them.

Consistency vs Availability

This is the classic trade-off in distributed systems. The CAP theorem tells us you can't have Consistency, Availability, and Partition Tolerance all at once you must choose two. Since partition tolerance is non-negotiable (networks fail all the time), you're usually choosing between Consistency (C) and Availability (A) during a network partition.

Consistency: Consistency means that all nodes in the system see the same data at the same time. If I write a new value to the system, and you read from a different server immediately after, you should see my change. If consistency is guaranteed, then stale reads are never allowed. This is often implemented with mechanisms like distributed locking, write-quorums, or leader election (not covered in this guide). If we choose Consistency over Availability, it means users **will not be able to access our system** during a partition.

Here are some classic cases where you should probably choose consistency. These are not comprehensive, but a good way to calibrate your sense for what things require strongly consistent data.

1. Financial transactions (e.g., bank account balances)
2. User permissions or security settings
3. Inventory systems (e.g., don't sell out-of-stock items)

Imagine a situation where you just transferred $500 from checking to savings. A partition occurs between two data centers: one has the updated balance, one doesn't. If the system prioritizes consistency, it will block access to your account until it confirms both data centers are in sync. This means no service during the partition, but you'll never see the wrong balance in your accounts. If the system prioritizes availability, it will let you access your account, even if one replica shows an old balance, and you get uninterrupted service, but data might be inconsistent for a short time.

Availability: Availability means that the system will always respond to a request, even if it can't guarantee the data is the most up-to-date. If a server can't get the latest data, it still returns the last known value from a given node. The system is designed to keep functioning even during network splits or partial outages.

Availability is usually a good choice when user experience and uninterrupted access are the most important thing. This applies to applications where users expect a continuous, uninterrupted experience, even if it means minor inconsistencies in data for a short period. A few cases where you would normally choose availability:

1. News feeds (e.g., social media timelines)
2. Caches (better to serve old data than no data)
3. Logging/analytics systems

Can You Have Both?

Sometimes, a system can prioritize both consistency and availability, just in different parts of the architecture. Imagine you're scrolling through an e-commerce site. The homepage shows products near you that you might be interested in, and it's one of the most heavily trafficked pages on the site. For a page like this, the system would likely prioritize availability. It needs to be fast and responsive, even if that means showing slightly outdated prices or inventory counts. That's okay, because this page is just for browsing. Once you click into a product to check out, the trade-off shifts, and the checkout service needs to prioritize consistency. We need to be absolutely sure the item is still in stock before we confirm your purchase or promise a delivery

date. Serving stale data here could result in overselling or broken customer trust, which is much more costly than stale data on the browsing page.

Latency vs Durability

This trade-off is all about what happens when you write data: do you want it to be fast, or do you want it to be safe? You often can't have both. Here is some guidance on why this trade off happens, and when to pick what.

Latency refers to how quickly your system responds to a request. Users love fast systems. Fast writes mean your UI feels snappy, users don't have to wait, and throughput goes up.
Here are a few common strategies for reducing latency on writes:

1. Acknowledge the write before it's safely persisted (e.g., write-behind cache which we covered in the caching section)
2. Buffer writes and flush them to disk later
3. Only write to memory (volatile) instead of disk (durable)

These may seem quite risky, but there are a few cases where prioritizing write latency is very useful, i.e. when UX responsiveness is more important than occasional data loss, when writes are low-risk or reversible (e.g., likes, views, logs), or when your system is behind a reliable queue or retry mechanism.

Durability means that once a write is acknowledged, it won't be lost, even if the system crashes immediately afterward.
Here are some strategies systems can use to improve durability:

1. Write synchronously to disk
2. Replicate to multiple nodes before acknowledging
3. Use commit logs, WALs (Write Ahead Logs), or consensus protocols

You probably want to prioritize durability when a lost write is unacceptable (e.g., payments, orders, password change), and a system needs to be 100% crash safe.

Durability takes time, disk is slower than memory, replication introduces network hops, consensus requires multiple nodes to agree. TLDR: every durability mechanism adds latency. If you want fast writes, you often have to acknowledge before the data is truly safe. If you want guaranteed durability, you have to wait, even if that adds 50–200ms to your request.

Cost vs Performance

Every performance boost comes at a cost. Whether it's faster hardware, more memory, stronger SLAs, or more replication, you're paying either AWS or NVIDIA. This trade-off is about finding the sweet spot between making your system fast and not blowing the budget. I don't personally find this one to be super important for interviews in big tech, but it is certainly a consideration at smaller companies, and 100% a consideration for every real world system design.

Cost refers to how much you spend to run and maintain your system.
This includes both costs for energy, water, compute (AWS bills, etc), AND engineering costs. DO NOT forget to calculate engineering costs. If you want to do everything with your own hardware and skip the cloud costs, fine, but now you have to calculate the cost of building and maintaining all that. Oftentimes it can actually be cheaper to just use the cloud offerings, especially at a small scale. Once you have billions of users, be my guest and build your own data center.

Some simple measures you can take to reduce cost are using smaller instances or fewer replicas, introducing batching or rate limiting, or reducing retention periods on logs/data. Logging is always mind-blowingly expensive.

Performance usually means faster response times, lower latency, or higher throughput.
You can prioritize performance if it **directly** increases revenue. Would anyone buy from Apple if their website took 5s to load and constantly showed out of date prices for their products?

Unfortunately, this is often extremely difficult to calculate. Just keep it in mind when you make your design or get an unreasonable requirements doc from a PM and have to inform them you'll be spending $1,000,000 on AWS to meet their expected SLAs.

Monolith vs Microservices

When I started as an engineer in big tech, I loved the idea of microservices. On the surface it makes a lot of sense. Development teams can own their own products, scale independently, and not have to worry about other teams breaking their services. Unfortunately over time, I started to see the reality of microservice hell. New microservices popping up for every single minute use case, most of which could probably be served by an excel spreadsheet or a new api endpoint in an existing service. This isn't to say you should never use microservices, but just be extremely careful where you draw the line between different components in your system.

A **monolith** is a single, unified application. All your logic, APIs, background jobs, and maybe even frontend rendering live in the same codebase and are deployed together. This model is incredibly simple to get started with. It's easier to reason about, easier to debug, and requires far less tooling to manage. 99% of startups and side projects should start with a monolith. You move faster, and it's much less complex to manage and scale.

Microservices break your system into separate services. Each one handles a specific concern (**clear separation of concerns)** user service, payment service, feed service, and so on. These services can be developed, deployed, and scaled independently. If your feed service needs ten times the resources as your user service, that's easy to handle with microservices. They also enable larger teams to work without stepping on each other's toes. The trade-off is complexity. Now you're dealing with network calls between services, version mismatches, retries, authentication between services, and distributed monitoring. It's a lot to handle, especially if your team is small or doesn't have good DevOps support.

In an interview setting, the biggest factor I look at is how tightly coupled the different parts of the application are, and whether they need to scale together. If most requests touch nearly every part of the system and each component relies heavily on data from the others, a monolith makes

more sense. You avoid the complexity and failure modes that come with network communication, and everything scales as a single unit. On the other hand, if different parts of the system serve distinct use cases, are often used independently, and only occasionally interact with each other, microservices are usually the better fit.

Read vs Write Optimization

Almost every system leans toward being either read-heavy (most are read heavy) or write-heavy, and your design should reflect that. The way you model your data, choose your indexes, design your cache, and structure your services will all depend on which matters more for your use case.

If your system is **read-optimized,** that means the most critical bottleneck is how fast and efficiently you can serve data. These systems are often query-heavy, with users constantly fetching information. Think dashboards, feeds, search results, or analytics platforms. In these cases, you'll want to denormalize data, cache aggressively, and structure your databases to minimize joins or slow scans. These designs often cause writes to become more complex and costly, because updating one piece of data may require writing to multiple places.

Write-optimized systems focus on getting data in quickly and reliably, often at very high volumes. Logging pipelines, telemetry systems, payment gateways, or sensor data ingestion are good examples. You'll often write in bulk, append-only, or with minimal transformations. Reads might be slow or require post-processing.

Here is a common example: in an Instagram-style app, writing a new post might involve duplicating data across multiple tables or document stores (author info, tags, feed entries). That's fine, because each of those reads needs to be fast, and writes are infrequent by comparison. But if you were designing the backend for a financial transaction engine, you'd likely go the opposite direction: normalize everything, make writes easy and auditable, and accept a bit more cost when serving data.

If your system has very different requirements for reads and writes, i.e. different volumes, latencies, consistency needs, or even different data shapes, it might make sense to separate the two paths entirely. This is a known and semi-common strategy called CQRS

CQRS stands for Command Query Responsibility Segregation. The idea is simple: split your system into two parts. One part handles commands (writes), and the other handles queries (reads). This pattern gives you the flexibility to optimize each side independently. You can structure your write path to be normalized, transactional, and durable, while designing your read path to be denormalized, fast, and tailored to the needs of the UI. Let's go back to the Instagram example. When a user creates a post, the write service might store that post in a normalized database: one table for users, one for posts, another for tags. Later, a background job can fan that post out to denormalized, precomputed feed tables which are sorted and cached for fast access. That read model might live in a separate store entirely, like Redis or ElasticSearch. The result is fast writes, fast reads, and decoupling between the two. CQRS can add a lot of moving parts: syncing the read and write models, handling eventual consistency, and debugging issues across systems, so it's not always wise to implement unless you really need both and don't mind the extra cost and development work.

Real-Time vs Eventually Consistent

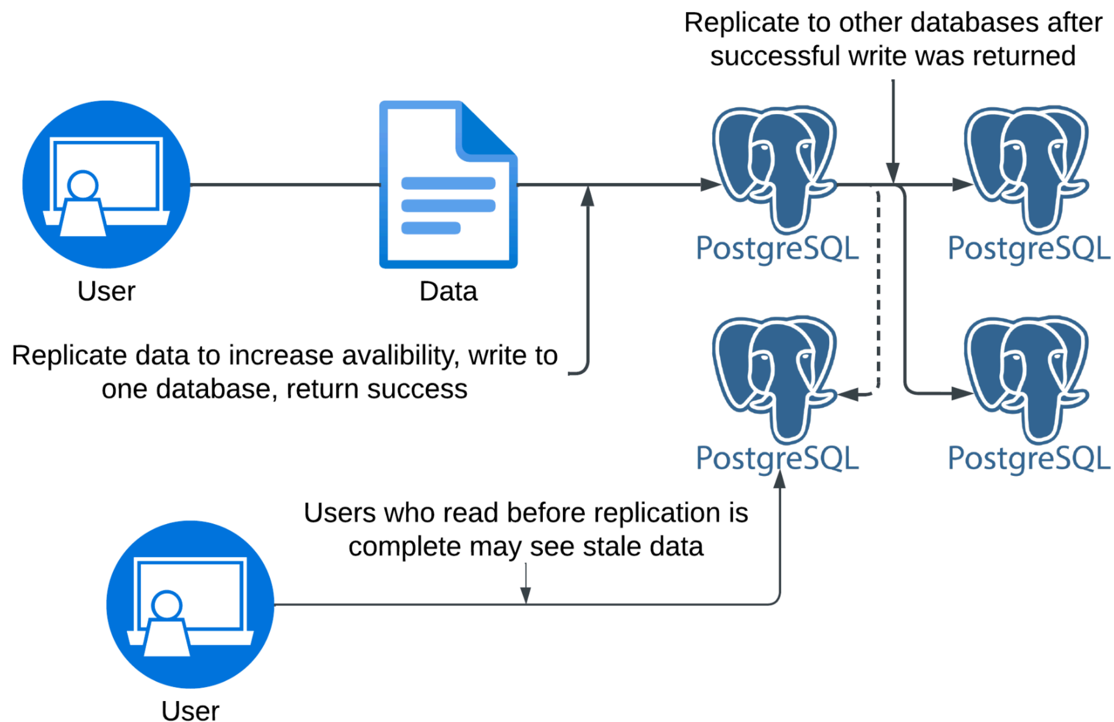
Strongly vs eventually consistent data might just be the bane of my existence. For some reason, customers & PMs can never grasp the fact that data is not going to be immediately up to date and ready within 10ms.

Real-time (consistent, strongly consistent) means data is always fresh and reflects the latest write. To make this happen in a distributed database or system, you need distributed locking, consensus, synchronous replication, and possibly a bunch of failover logic. It adds write latency, it's harder to scale, and it becomes a nightmare under load.

Eventually consistent systems accept that, for a brief period, different parts of the system might see different versions of the data. But eventually (once replication catches up) everything settles

into the correct state. This is often good enough and for most systems it's the only practical choice.

Example (**eventual consistency**) Let's say someone updates their profile picture. Does it need to change across all services and devices immediately? Not really. If one tab takes 10 seconds to catch up, no one really cares.

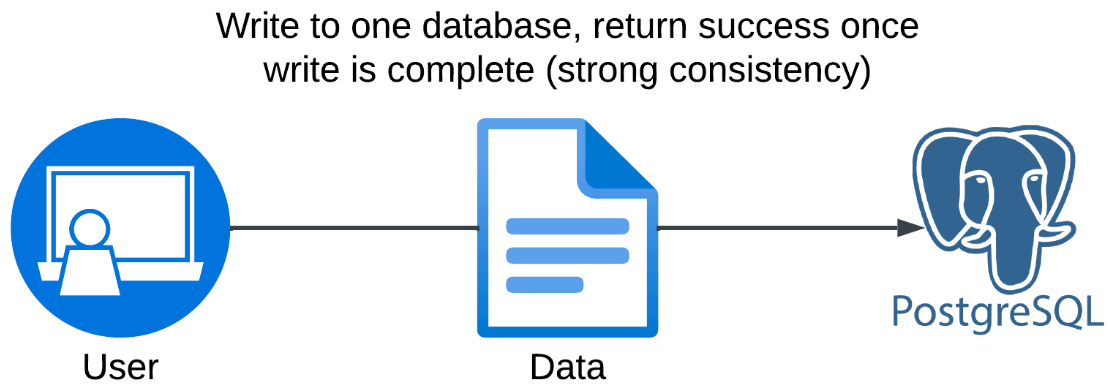


Eventual Consistency

Example (**strong consistency**) Imagine you're resetting the password on your email account. Once you've confirmed the change, you expect all devices and services to immediately require the new password, especially for security-sensitive actions like signing in or sending email.

If one replica of the authentication service is out-of-date and still accepts the old password, an attacker could gain unauthorized access. In this case, we must guarantee that the password

update is propagated across all services before acknowledging success to the user. This is a classic case where strong consistency is non-negotiable.



Strong Consistency

In practice, most real-world systems are a mix. Most user-facing data like likes, comments, or feeds are eventually consistent. Things like transactions, account states, or permission changes are real-time or strongly consistent.

[Designing Data Intensive Applications] has a great chapter on this and when to use what. You now understand the components and trade-offs that make up a real system, but knowing the pieces isn't enough. In interviews (and real life), your success depends on how you bring them together. That's where a clear, methodical design process comes in.

Designing Step by Step (**Interviews)**

It's tempting to jump straight into the design, or picking your database as soon as you get into the interview. This is probably the worst mistake you can make. Here is exactly what you should do instead.

1. Clarify the prompt and restate it in your own words (1 minute)

I always start with this because it instantly gives you a couple seconds to think, and it clarifies that there are no misunderstandings between you and the interviewer.

2. Gather functional requirements (3-5 minutes)

Always gather functional requirements first. This is a natural second step from restating the prompt. If there are certain parts of the design or service you are trying to build you feel more comfortable with, try to make those components part of the required / initial functional requirements.

3. Gather non-functional requirements (2-3 minutes)

This step should be straightforward. Remember to get the non-functional requirements that will affect your server design, but also those that will affect your database. You can always gather more requirements late in the design, but demonstrating the ability and foresight to know what you need up from is always a good look for you.

4. Make estimates (3-5 minutes)

While this is quite similar to gathering non-functional requirements, some companies will throw out your application if you skip this step, so I figured I should explicitly call it out. Here we need to take the non-functional requirements, and translate them to how we need to scale our application. Let's say we have gathered the following requirement for our design:

Support 1,000 daily average users

Let's stick with the Instagram example, and assume each user scrolls for 10 minutes per day. Each minute they scroll, they view 10 different posts. Each post is a request to our server. Based on this, each user makes roughly 100 requests per day: 10 minutes × 10 requests/minute = 100 requests/day. Given 100,000 users, that gives us 100,000 users × 10 requests/user = 10,000,000 requests/day. We also know there are 86,400 seconds in a day, so assuming traffic is distributed equally, we can calculate TPS: TPS = 10,000,000 requests / 86,400 seconds ≈ 116 TPS. You could stop here, but in reality, traffic is probably not distributed equally. If most of your users are in America, you probably have large fluxes in traffic, maybe during lunch, right after work, and before folks go to bed. You can use this to infer a peak TPS of ~1,000. Based on these requirements, you can define how many servers you need. In this case a single VM would handle this level of traffic.

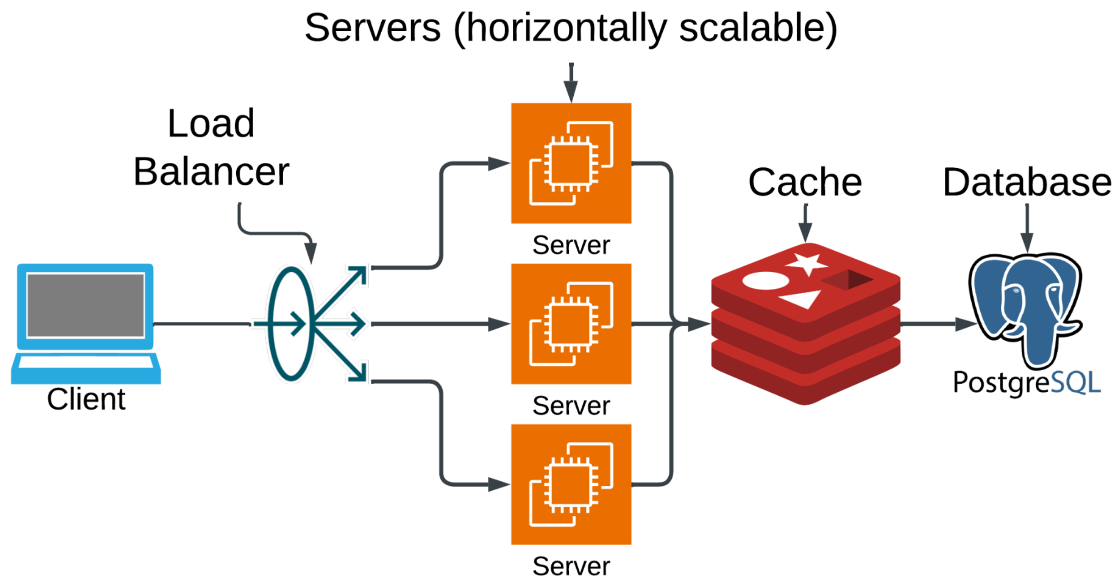
5. Define the API contracts (3-6 min)

You don't always need to do this, but it is nice especially if you are interviewing for a software engineering position (some other positions like TPM, PM, SDM also require system design interviews). If you are designing a single service or API, this is usually pretty quick, and can be a good way to demonstrate knowledge of things like RESTful API principles (not covered in this guide).

6. Design the high level architecture (8-12 min)

This is where you get to put on your artist hat and draw the boxes and lines that represent a system. You've seen a few of these on this document, and if you watch my short form content,

you have seen hundreds. A simple scaffolding that works for most designs:



Simple design scaffolding

If you like, you can also throw in monitoring, queues, references to 3rd party APIs, etc, but this simple framework is usually a good start.

7. Go deep on a single component (8-12 min)

So you can do a high level design and draw a picture. Great. Now go deep into one component of the design. Most of the time this should be the database. Explain why you picked the database you did. Lay out the design for your tables if you are using SQL, or the indexes and partitions if you are using NoSQL. This is a great chance for you to demonstrate your knowledge / expertise in any given area. Use this chance to show you understand tradeoffs, and when to do things like choose strong vs eventual consistency, or synchronous vs asynchronous processing if you go deep outside the data layer.

8. Plan for scale and failure (3-6 min)

It is not hard to design a system that works when things are good. As a software engineer, (especially in big tech) we are often hired to get a system from working 99.9% of the time to 99.999% of the time. For something like AWS, that simple change can result in billions of dollars in value and customer trust. For this section, I ask myself questions like

> *"What happens if traffic spikes 10x overnight?"*
> *"What if a region goes down?"*
> *"What if the cache crashes?"*

It's nice to think about these questions and handle them before you are forced to by the interviewer. Not every system needs multi region failover and caching for lower latency, but if you do, best to design for it up front. Some things to consider here are rate limiting, circuit breakers, multi-region failover, and built-in retries for failed requests.

9. Wrap it up with a high level summary (1 min)

I finish with a summary for the same reason I start with restating the question. It helps me understand exactly what I did, buys a couple seconds, and oftentimes helps you see gaps in your design you might have missed during the process. It is a lot like talking to the rubber duck on your desk when you are writing code. Highlight any key trade offs you made, and what future extensions might look like if you had more time, or needed to support more features.

At the end of the day, there is no one correct design for any given question. As long as you are able to explain why you made the choices you made, and highlight the trade offs you thought were worth making, you can put together a great system design.

What's Next?

After reading this guide, the best next step is not to disappear for three months and try to memorize every distributed systems textbook ever written. The best next step is consistent practice. That's why I built **The Daily Dev**, my system design app and private community for software engineers who want to actually get better at this stuff. Every day, you get one curated system design question, a breakdown of the answer, resources to go deeper, and access to a community of other developers (and myself) practicing alongside you. System design is not something you learn by cramming once before an interview. You build the skill by repeatedly thinking through tradeoffs, constraints, failure modes, and architectural decisions until they become second nature. So if you want to keep going from here, join The Daily Dev and start building your system design judgment one question at a time. See you inside!

https://thedailydevweb.arjaythedev.com/