October 19, 2026 | software design, ai

What requirements don't you have?

When designing software, the requirements you don’t have are as important as the requirements you do. Identifying what you can compromise on — what can be slow, or stale, or inflexible — is usually the key to an elegant design.

Elegant systems drop some requirements

For instance, take preforking web servers like Unicorn. Because they lack the feature of shared mutable state between server instances, they can run each server in its own process, which provides isolation for free. Because Unicorn specifically ignores the problem of slow client handling[1], it can queue requests directly on Unix sockets, drastically simplifying the implementation.

Because S3 does not support atomic multi-key updates — you cannot update multiple blobs in a single transaction — it can store data at the blob level however it chooses. That makes it much faster, since reads and writes don’t have to cross storage partitions, and much easier to scale, since less coordination means fewer sources of truth.

StatsD uses UDP by default, which means it doesn’t guarantee that every piece of data you send makes it into storage. In exchange for dropping that requirement, it incurs much less network overhead, which is a big advantage for a stats aggregation service that’s constantly shoveling data over the wire.

The general design approach of fanning out data to an edge cache buys a lot of speed and scaling by dropping the requirement that data must always be up-to-date. SQLite buys operational simplicity — it can run out of a single file! — by dropping the requirement of concurrent writes. By accepting high tail latency[2], Cursor can scale their distributed git backend more easily. Well-designed systems turn their weaknesses into strengths.

Why dropping requirements is hard

Anyone who's spent time designing real-world software (or who's studied distributed systems) knows that system design is a process of compromise. If you attempt to satisfy every single potential requirement — speed, concurrency, up-to-date data, a simple interface, supporting complex user behaviors — you will fail to design a working system at all.

Figuring out what requirements you can relax is difficult because **it combines both business and technical expertise**. It is technically challenging to identify precisely the level of staleness that a particular design change will allow; it is likewise challenging from a business perspective to determine whether that level of staleness is acceptable to users.

In most cases, an engineer who understands the technical side must try and explain it to a product manager, who can make a judgement on the business side. But this is a very difficult partnership to get right. Engineers and PMs have different incentives and little ability to check each other's work, which makes it hard to build trust. Without trust, the product manager is liable to avoid giving any ground at all[3], which leaves the design in an impossible state.

This is an underappreciated reason why solo developers or startups can sometimes run rings around larger companies. Because the technical and business decisions are united in a single person, they can make design tradeoffs that would be impossible for a large company, enabling them to build systems that are far more elegant and powerful.

Working with LLMs

Knowing what requirements you don't have becomes far more important when working with coding agents. Frontier models are very capable, but they have all gone through the purifying flames of RLVR: like high-achieving schoolchildren, they are obsessed with satisfying the Grader. On top of that, they have very little business context compared to an experienced human. That often manifests as them catching correctness or consistency "bugs" that are really just hypothetical unsatisfied requirements.

For instance, it's not uncommon for an LLM to casually introduce a mutex or other concurrency mechanism to ensure that two pieces of semi-related data are kept in sync, even if not told to do so. I've seen LLMs add weird checks in the middle of request handling to ensure that some already-checked piece of data hasn't changed mid-flight, even if that wouldn't successfully prevent the feared TOCTOU bug[4]. LLMs are obsessed with edge cases like "what if the current user loses their admin status in the middle of us handling one of their admin requests?"[5], or "what if the cache in storage partition A is updated a nanosecond before the one in storage partition B,

leading some users to see different data?“. Particularly if run in a loop or told to “make no mistakes”, they will pile on cases like these until they’ve transformed your four-line change into a thousand-line monster.

This is why I’m confused by some people’s claims that code review is dead. About half the time I review a large AI-generated change, I notice some unnecessary requirement that makes the diff more complex. When I dig deeper, the LLM invariably introduced this requirement on its own.

When working with a cache, I will often spell out in my prompt that I don’t mind the data sometimes being stale or briefly inconsistent between cache instances. When reviewing Golang code, the first thing I do is scan for added mutexes or layers of indirection that don’t look necessary to me.

If you allow requirements like these to propagate unchecked through a codebase, they will petrify the system over time. The whole thing will be such a mess of checking and re-checking and holding locks and enforcing consistency that it cannot be scaled at all. Keeping systems elegant requires a ruthless focus: on accepting stale data or occasional tail latency, on avoiding all but critical instances of coordination, and in general on figuring out what requirements you *don’t* have to satisfy.

---

1. The problem: if an HTTP request sent by someone with slow internet trickles in one packet per second, does that have to tie up a server process for the whole time? Unicorn’s answer is ‘yes, so put nginx or HAProxy in front of me’.

   ↩

2. When a read request hits a stale storage node, that node must update itself before responding.

   ↩

3. I’ve also seen this happen in reverse, where the product manager desperately tries to convince an engineer that some level of stale data is acceptable, while the engineer digs their heels in and insists on the “perfect” system. The consequences are similarly dire.

   ↩

4. Of course, simply doing a separate check shortens the TOCTOU window instead of removing it.

   ↩

5. In every single system I've worked on, it would be fine to simply carry out (or fail) the admin request: if somebody was allowed to perform some operation 10ms ago, it's likely not a disaster to let them perform it now.

   ↩

If you liked this post, consider subscribing to email updates about my new posts, or sharing it on Hacker News.

Here's a preview of a related post that shares tags with this one.

> **Build agents, not pipelines**
>
> There are only two ways to use LLMs in a computer program: as part of a pipeline, or as an agent. In other words, either you express the control flow of the program in code, or you give an LLM tools and allow it to manage the control flow itself.
>
> Here's how you might structure a trivial "summarize a bunch of information and email it to me" program as a pipeline:
> Continue reading...

Search posts

Search