

Lecture 10 — Software Architecture

Jeff Zarnett

2025-06-02

System Design

When you look at the situation, you might find the reason that the runtime of some code is what it is results from the design of the larger system in which the data lives. For example, if we have to do a lot of network calls to get the data that's needed, that will increase the time to do the operation compared to getting all the data locally or in only one network call. If we can't get all the data in one shot, we have to loop, and that loop might easily turn what is otherwise linear runtime into another of the accidentally-quadratic examples.

System design interviews are another popular screening method for candidates in industry. I [JZ] like this better than the leetcode interviews in terms of understanding a candidate's ability to do the (typical) work of software development. Unlike most algorithm implementations which are solved problems (that is, there exist one or more optimal answers), a lot of system design problems are quite open-ended and a problem that isn't "read my mind" likely has multiple valid options that you can choose... if you can justify your choice appropriately.

Whether you can change the data layout or the system architecture is very situation-dependent. It is generally unlikely that you will be able to convince your company to split up their five-year old monolith codebase because it would be faster in some scenarios. It's not necessarily that your argument is invalid, it's just that the opportunity cost is huge and such a cost has to be outweighed by sufficient customer value. Similarly, it might be optimal for your use case to change how the data is represented in the database, but that version might be worse for another, more common, use case and so the structures won't change.

Sometimes you will get input, or get to make the choice. Let's talk about that a little bit!

Choosing the Right Architecture

Pitfalls

Whether or not we get to choose the architecture, at least we can discuss some ideas about what's better and worse for a specific part of the program. Maybe mostly worse. Just like the clickbait articles: four huge architecture mistakes to avoid! Number two will shock you!

Excessive Network Calls. As previously referenced, reducing the number of network calls to get the data will be an improvement in the execution time. Network is slow and comes with unpredictable delays for various reasons. Each call has overhead to establish the connection, authenticate or validate the token¹, and unpack the request before any of the actual handling of it can take place. Thus, a lot of small requests is likely slower than one larger request. That may be partially offset by parallelization (like in asynchronous I/O situations), but not always.

The converse of that, though is that how much data is sent is also a factor in the length of time the communication takes. In a silly example, the app requests the whole table from the database server to the application and then searches and filters in memory. If the table is huge, sending all of it would take a long time. And that assumes that there's enough memory in the application and that the network call would not time out for being quite so large...

Worries about too many network calls can be used as an argument against a microservices architecture and favouring monoliths. It's a drawback of this architecture, to be sure, but it will never be the only criteria for deciding.

¹You are including auth in your APIs, right? RIGHT?

Considering honestly the pros and cons of the architecture is normal, and if anyone ever tells you that their preferred solution has no drawbacks whatsoever, you should be very skeptical of that.

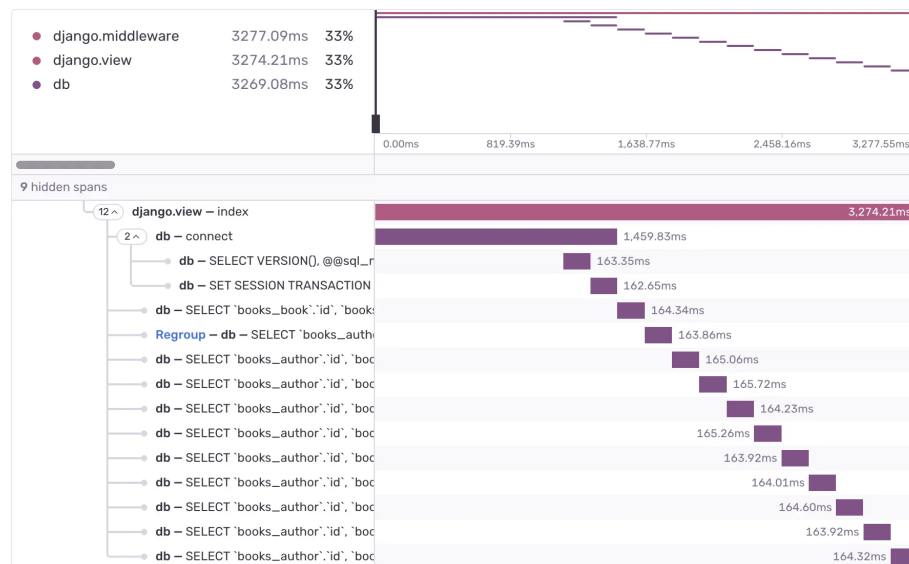
A specific version of this is called the N+1 query problem. In short, it happens when you want to fetch a list of something, and then for each of those, fetch a list of related entities. Imagine you want to send e-mails to all the customers who have not yet paid their invoices. How to do that? Query for the customer ID of each invoices where the status is unpaid. Then, for each of those customer IDs, look up their e-mail address, and send the payment reminder e-mail to them.

At first glance that doesn't sound bad, because this is how you might carry out such an operation if doing it with printed paper invoices and contact info in your phone. But if there are 500 unpaid invoices, this approach results in 501 database queries. One to get the list of customer IDs, and then one each for every invoice to get the customer's e-mail. That's too many!

You might think that such a thing is easy to avoid – and it can be, if writing the SQL directly. We could use a join query to get the list of e-mails directly in one query, or use a `WHERE customer_id IN (...)` type of clause to get it down to two queries. I prefer the one query approach, if possible, but two is certainly better than 500. What did I mean about not writing the SQL directly?

If you're using some sort of ORM (Object-Relational Mapping), then you have some intermediate framework (e.g., Hibernate) that turns some (Rust or other language) code into some database query statements that you didn't write yourself. The ORMs may unintentionally generate for you the N+1 query-problem variant of your query. It's not on purpose, but it is worth checking the generated SQL these produce just to be sure. Some ORMs let you write your own SQL; for others you want to pursue strategies like eager loading (that is, fetching related entities alongside the ones you asked for originally, even if you might not need the related ones) [Oza18].

Here's an example of what an N+1 query looks like when you look at it with profiling tools, from the Sentry Docs:²



Bottlenecks (Chokepoints). If your architecture results

Over-Taking Responsibility. The earlier example about searching and filtering in the application instead of the database also exemplifies another problem: over-taking responsibility. Even if the size of the table is so small that transmitting all of it over the network takes negligible time, why search and filter it in memory of the application? The database server has more information, like how the data is stored on disk or what index it can use to do the search. Therefore the most efficient approach is

²<https://docs.sentry.io/product/issues/issue-details/performance-issues/n-one-queries/>

Too Much Waiting.

References

[Oza18] Brent Oza. Common entity framework problems: $N + 1$, 2018. Online; accessed 2025-06-02. URL: <https://www.brentozar.com/archive/2018/07/common-entity-framework-problems-n-1/>.