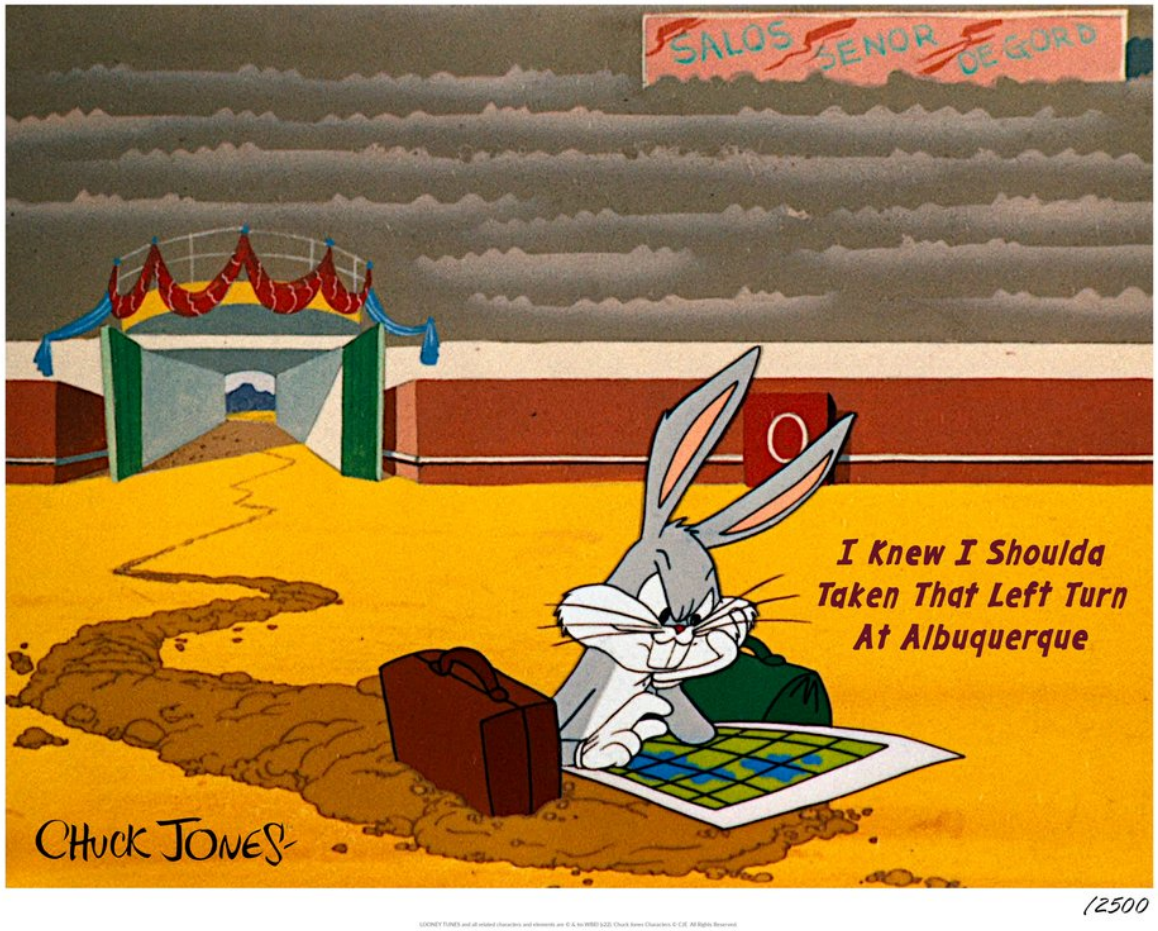
Lecture 10 — Software Architecture

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

July 7, 2025

System Design



Sometimes, 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.

System Design

Example: if we need to do lots of network calls to get the data we need, it will be slow compared to locally. What if we can only get items one at a time?

System design interviews are another popular screening method for candidates in industry.

A lot of system design problems are quite open-ended (that's good).

Can I Change That?

Whether you can change things is very situation-dependent.

You probably can’t convince your boss to rewrite the whole codebase.

It’s not even that your argument is necessarily wrong!

Maybe you could change data layout in the database?
Or maybe that works for a different use case that’s more important.

Another Course?

Software architecture courses cover this sort of topic in much more depth!
Let's think about the implications of architecture on code performance.

High level decisions are rarely made with performance in mind (or a wild guess).

Quick catch-up about monoliths vs microservices...

Architecture Patterns

Down a level, let's talk about design patterns!

- Producer-Consumer?
- Blackboard?
- Message-passing vs. Shared Memory?

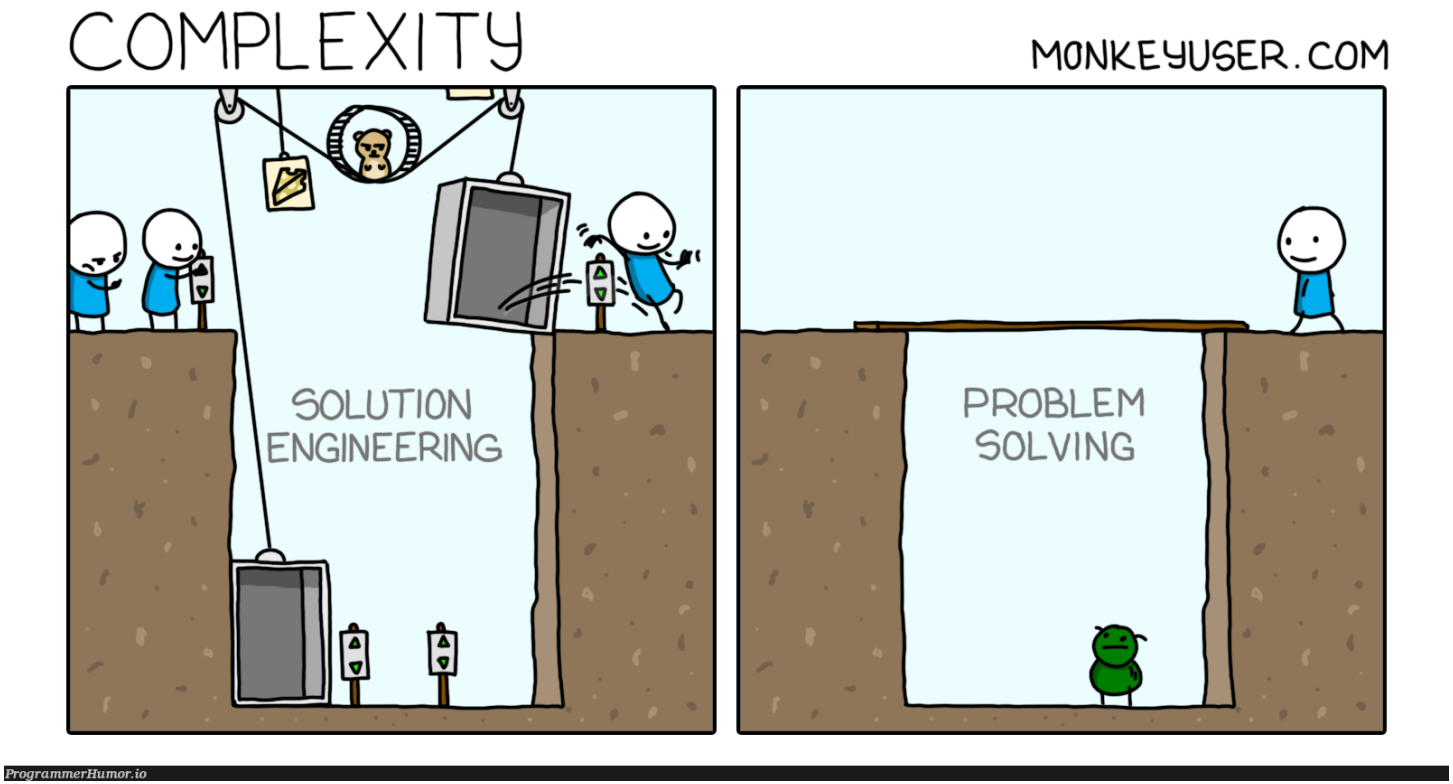
Often: boil it down to a real-life analogy. Assembly line?

Implementation

Zoom in more and we get more decisions to make. Threads? Framework?

Can we choose configuration parameters maybe?

Complexity is the Enemy



Complexity is often the result of looking at big companies and thinking their choices are best practices and “silver bullets”.

Essential and Non-Essential Complexity

Some complexity is essential: tax software is complex because tax law is complicated and constantly changing!

Other things are our own fault: `VisitationArrangingOfficer`?!

`https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition`

What about Abstraction?

Abstraction is usually good in software architecture, right?

“We can solve any problem by introducing an extra level of indirection.”
– David J. Wheeler

Wikipedia even calls it the Fundamental Theorem of Software Engineering.

Abstraction

Every layer of indirection can easily add additional overhead and result in duplication of work because the right information is not available.

It also slows down the speed of development while you have to just write more stuff to move data around and to keep track of what should be on what level.

They What Now?

Ancient problem, hopefully died out: Architecture Astronauts.

They don't do much directly, but encourage over-engineering and overly-complex solutions. Yikes.

How does that make it slow? More moving data, longer lines of communication!

Pitfalls

Leading Doctor Reveals the No. 1 Worst Carb You Are Eating
Mediconews

The $$$ Moneymaking Secret that Banks Don't Want You To Know
Bankfacts

These 12 Impossible Pet Rescue Stories Will Melt Your Heart!
Cutepups Inc

Four huge architecture mistakes to avoid! Number two will shock you!

Pitfall 1: Excessive Network Calls

Reduce the number of network calls to reduce time to completion of requests.

Every call has overhead: establish connection, authenticate, unpack request...

A lot of small requests may be slower than one larger request!

Pitfall 1: Excessive Network Calls

Minimize the transfer of data, too.

Is this a drawback of microservices? Yes, but rarely a dealbreaker.

Common variant: N+1 query problem.

N+1 Query Problem

It happens when you want to fetch a list of something, and then for each of those, fetch a list of related entities.

Example: send e-mails to all customers who haven’t paid their invoices.

Get unpaid invoices, then find the client for each one, and send them an e-mail.

N+1 Query Problem

Problem: If there are 500 unpaid invoices, this approach takes 501 queries.

You might say this is easy to avoid:

You know about `JOIN` or `WHERE... IN` queries!

ORMs Though

Use an ORM (Object-Relational Mapping)? The framework turns code into some database query statements that you didn't write yourself.

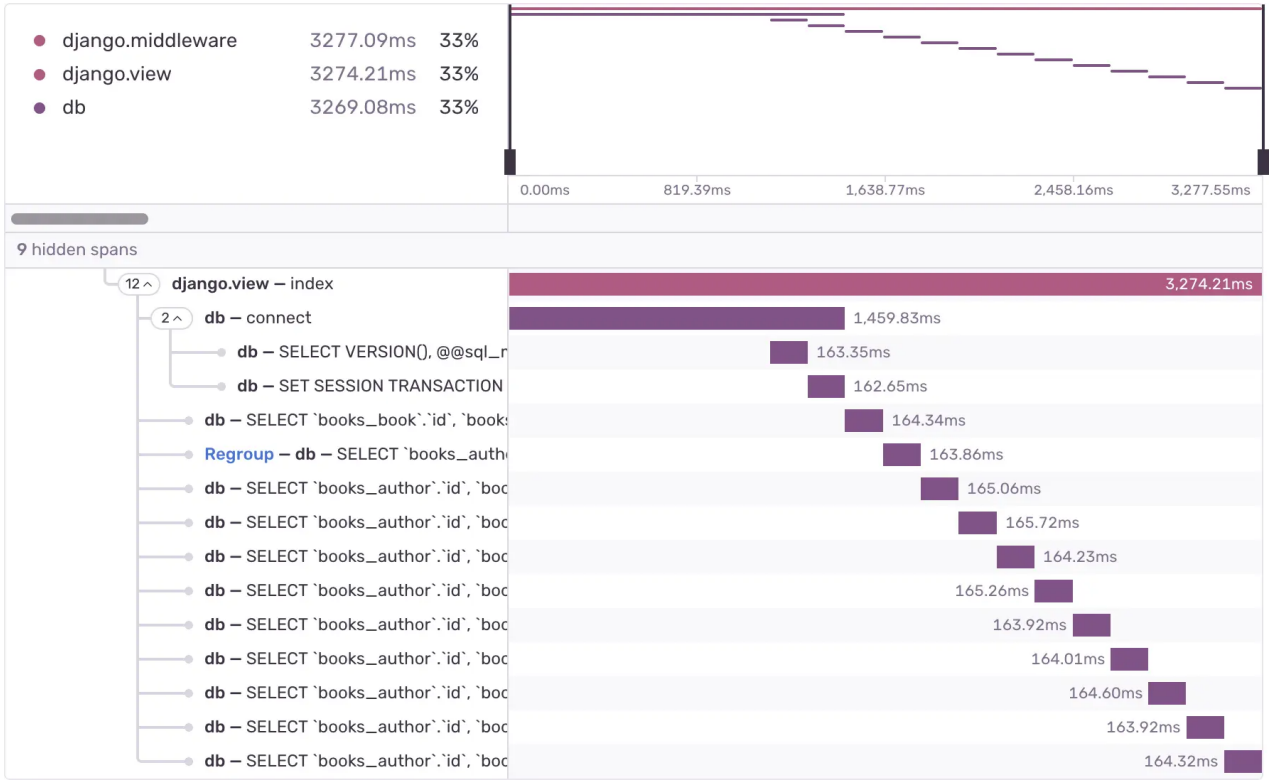
These may unintentionally generate the N+1 variant of your query.

It's not on purpose, but you might want to verify.

Solutions: write your own, or enable eager loading.

N+1 Example

Here’s an example from the Sentry Docs:



N+1 Code Example

The documentation for the Rust crate `juniper_eager_loading` provides a great example of how an N+1 query can result from a GraphQL schema.

```
schema {
    query: Query
}

type Query {
    allUsers: [User!]!
}

type User {
    id: Int!
    country: Country!
}

type Country {
    id: Int!
    name: String!
}
```

N+1 Code Example

And a caller executes the query:

```
query SomeQuery {
    allUsers {
        country {
            name
        }
    }
}
```

N+1 Code Example

The naive implementation will run the following queries:

```
select * from users
select * from countries where id = ?
select * from countries where id = ?
select * from countries where id = ?
select * from countries where id = ?
...
```

But we actually would prefer...

```
select * from users
select * from countries where id in (?, ?, ?, ?)
```

How to solve this?

N+1 Code Example

The core idea is changing how the data structures work to load information if needed but avoid it if it's not.

How do you imagine a user is represented in the system?

```
struct User {
    id: i32,
    country_id: i32
}
```

The only way to get the country name is to load the country...

N+1 Code Example

Alternative that separates the GraphQL models from the Rust structures:

```
mod models {
    pub struct User {
        id: i32,
        country_id: i32
    }

    pub struct Country {
        id: i32,
        name: String
    }
}

struct User {
    user: models::User,
    country: HasOne<Country>,
}

struct Country {
    country: models::Country
}

enum HasOne<T> {
    Loaded(T),
    NotLoaded,
}
```

Finishing N+1 Example

First, load the users, get a list of country IDs, then query the countries in that set of IDs, and match them up.

That means replacing the `User.country` value of `HasOne::NotLoaded` with `HasOne::Loaded(matching_country)`.

Then we can just return the set of names after the second query.

Much better than the naive approach.

Excessive Network Calls

Finally, another strategy that reduces the number of network calls is caching.

We've discussed it at great length earlier in the course. No need to repeat it.

Pitfall 2: Bottlenecks (Chokepoints)

Example: authorization service invoked on every API call.

Solution: `jwt` – JSON web token that you can validate.
It is kind of like a passport.

Generalization: can you distribute work?

Pitfall 3: Over-Taking Responsibility

Searching and filtering in the application is over-taking responsibility.

Why do the job of another component of the system?

Let the database do what the database does best.

Another example: doing rendering work in the backend.

But remember to validate the frontend input!

Pitfall 3: Over-Taking Responsibility

That’s often caused by organizational or technical constraints.

Examples: gatekeeping or the remote server being another company’s code.

Sometimes: trade efficient execution for efficient delivery.

Pitfall 3: Over-Taking Responsibility

Over-taking responsibility also tends to result in painful change management.

Also: high levels of tech debt if people are afraid of touching a critical system.

No software-architecture decisions can solve organizational problems.

Pitfall 4: Too Much Waiting

We've already covered asynchronous I/O!

Moving to an asynchronous model is also valuable in reducing *perceived* waits even if the actual execution time is no faster.

Remember *response time*?

Pitfall 4: Too Much Waiting

Another example: every thread or process waiting to acquire a lock around some shared data.

Imagine there is one `Arc<Mutex<Vector>>`.

Solution: establish a channel and send the data to one thread.