Solving Common Concurrency Problems

Creating a concurrent API request for searching notes

Beekey Cheung · Follow
Published in Better Programming
9 min read · Dec 8, 2021

Listen Share

Photo by Ganapathy Kumar on Unsplash

Concurrency is a notorious cause of really frustrating bugs. Most software bugs are consistent. If you do X, then Y, then Z, you get Bug A.

You can get race conditions with concurrency though. That's basically a bug where if you do X, then Y, you'll get Bug A maybe 10% of the time. The occurrence of the bug

is intermittent which makes it hard to find the root cause since you can't reproduce it reliably. It also makes it hard to prove you actually fixed the issue. If Bug A only happens 10% of the time, you'll need to try to reproduce the bug a lot to have reasonable confidence you've fixed it.

Dealing with concurrency issues was my bread and butter early on in my career. I loved working with threads and fixing race conditions the senior developers missed. It was a huge confidence boost.

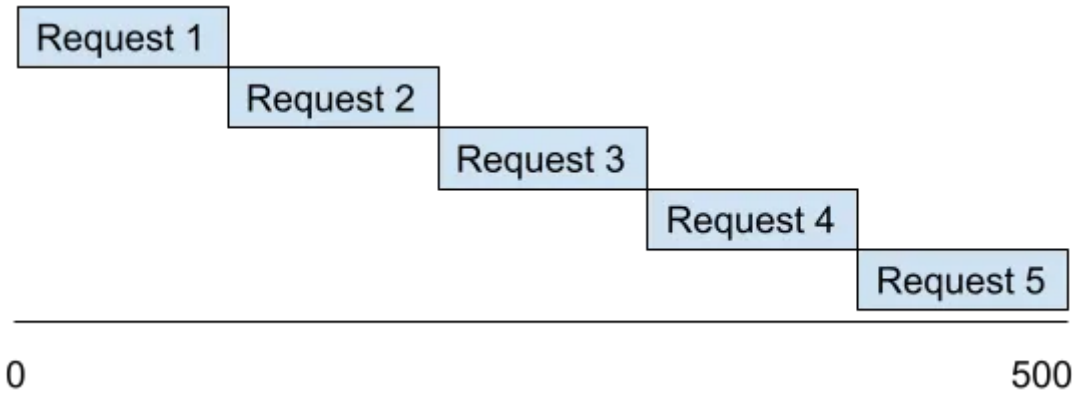
Then I went into an interview and was given a concurrency question. Total. Bomb.

It was then I realized that I was good at a certain type of concurrency problem and that problem happened to be the majority of concurrency issues.

First, let's talk a little bit about what concurrency is. We'll then continue to a simple concurrency problem and then a more gnarly problem.

Concurrency is basically having multiple separate pieces of code to function at the same time. Let's start with a hypothetical and then go into a real situation.

Say I need to make 5 different requests to an API. Each one will take 100ms to complete. If I wait for one to complete before starting the next, I will end up waiting 500ms.



All images by author

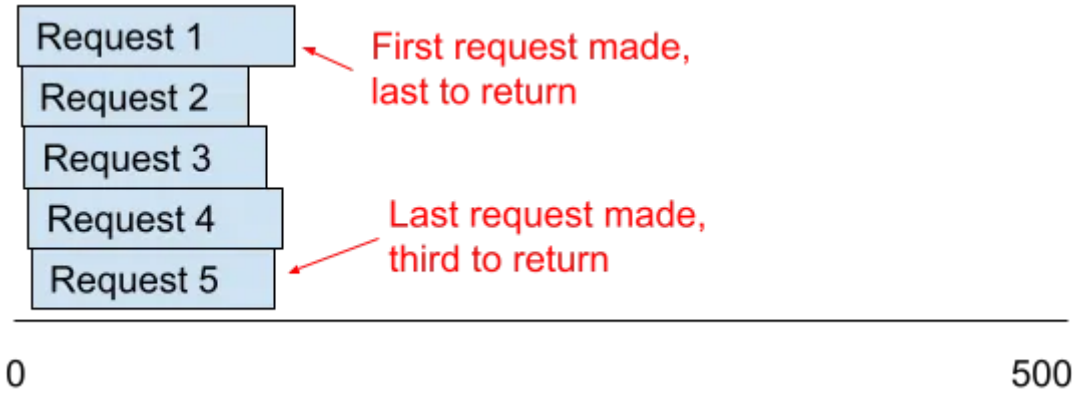
If I execute all those web requests at the same time, I will end up waiting 100ms + some small amount of overhead.

That's a pretty big performance difference. That's usually why concurrency is used.

It sounds like a simple concept, right? That's because it is a simple *concept*.

The problem is the execution. Those API requests take *approximately* 100 ms each, not exactly 100 ms each.

That means you'll make the API requests in order, but the return will be out of order:

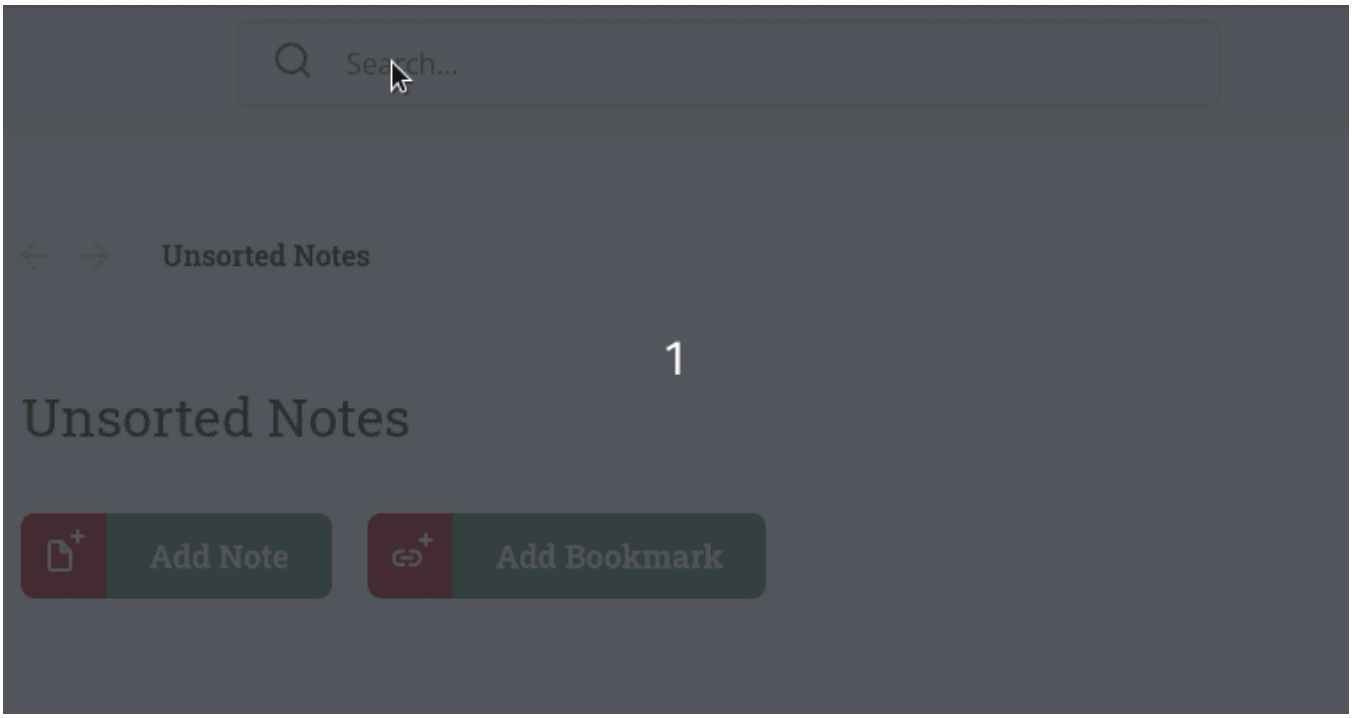


and the return order will be different every time you run this code that executes the API requests.

You get a performance improvement with concurrency, but you give up consistency.

The bugs start popping up if the code handling the response of these API requests uses shared data.

Let's look at a more detailed example of how this can happen. There was a bug in Dynomantle with the suggestions in the search bar.



Here's the problem: an API request is made every time you type a character.

This is to provide you with a smooth experience of seeing suggestions as you type. You type "i" and your notes/bookmarks/emails that start with "i" pop up. You type "in" and the list gets refined to things that start with "in".

How long does it take you to type 5 characters when you know what you want to find? 2 seconds? 1 second? Half a second?

I still need to tune the feature, but right now it takes somewhere between half a second and 1 second to process each API request.

It would be an awful user experience to make someone wait a second between typing each character. So I just make an API request as each character is typed. The problem is that the requests return out of order. The request with 2 characters can return *after* the request with 5 characters.

The suggestions are just stored as a list. Every time a response comes in, the entire list is refreshed. In this situation, the entire list was refreshed with the correct suggestions when the last request returned but then populated with incorrect suggestions when an older request returned.

Fortunately, this is a really easy problem to solve since the requests are made in order.

1) Generate a timestamp or hash every time a request is made. This basically serves as a `requestId`:

```
let requestId = Date.now()
```

2) Set the `requestId` as an additional variable with the suggestion list. Since we make requests in order, this will always be the last request:

```
let requestId = Date.now()
// Datastore is some singleton for
// easy access to these types of variables
datastore.setLastRequestId(requestId)
```

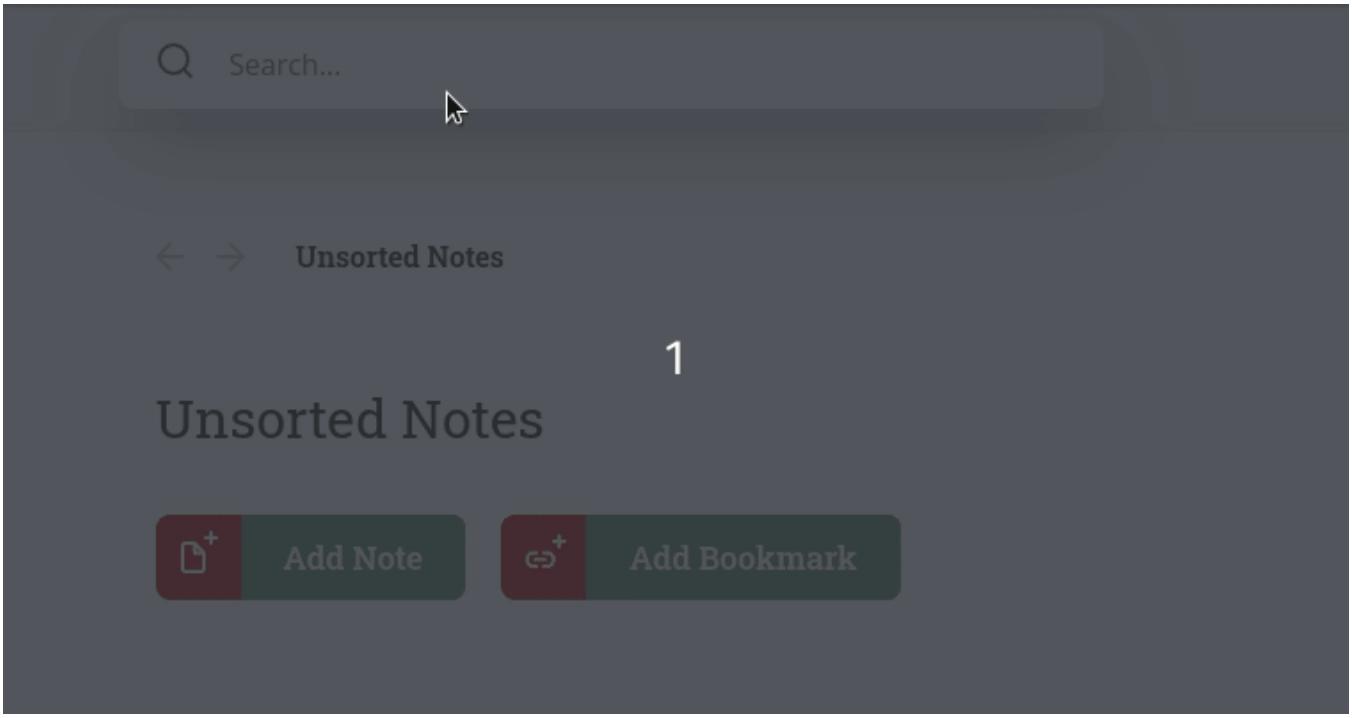
3) Pass the request in the success function of each API call:

```
let requestId = Date.now()
datastore.setLastRequestId(requestId)
$.ajax({
    success: function(json) {
        suggestionsReceived(json, requestId)
    },
})
```

4) Validate when a response comes that it is always for the expected request:

```
suggestionsReceived(
    suggestions: Array,
    requestId: number,
) {
    if(datastore.lastRequestId != requestId) {
        return
    }
    // the rest of the code
}
```

The problem unfortunately is if the user types very fast they could see a delay in seeing any suggestions. Even if they don't use the 2 character suggestion, seeing it populate can provide a sense that the app is doing something versus just waiting.



Solving this requires a small modification to the code above.

We'll stick to using timestamps instead of a hash.

Next, we will store the last `requestId` *received* instead of the last `requestId` made.

```
let requestId = Date.now()
$.ajax({
    success: function(json) {
        suggestionsReceived(json, requestId)
    },
})
```

```
suggestionsReceived(
    suggestions: Array,
    requestId: number,
) {
    datastore.setLastRequestId(requestId)
    // the rest of the code
}
```

Finally, we will only refresh the list if the response has a higher `requestId` than the last one received. Since we use timestamps as the `requestId`, all the requests are in order and bigger ids are more up-to-date requests.

```
suggestionsReceived(
    suggestions: Array,
    requestId: number,
) {
    if(datastore.lastRequestId > requestId) {
        return
    }
    datastore.setLastRequestId(requestId)
    // the rest of the code
}
```

Note: this only works because users are not going to be typing in multiple characters in the same millisecond. If they do, they are pasting content and we only want to make one API request anyway.

Another important note is that this also only works because of how Javascript handles concurrency. It isn't *really* concurrent. Each function executes and completes before another is run.

Try similar code in Java and you will have a bad time because multiple calls to `suggestionsReceived()` can execute at the same time. That means a suggestion response for "in" and "inv" can both pass the check-in the if statement and then execute the rest of the function.

The behavior you will see will be wildly inconsistent depending on how long the rest of the function is and the timing of the two function calls. To get this to work in a truly concurrent language, you'll need to look up how to use locks for that language. Redis also has distributed locks if you're dealing with concurrency across multiple servers.

A lock basically blocks function execution while another function has the lock. If we needed locks in Javascript, it would look something like this:

The risk of course is if we never unlock, then the other functions never execute. If we use multiple locks across multiple functions, we could end up in a situation where two functions are waiting for locks that the other function has locked. Our program now freezes because neither function can progress. That's called a deadlock situation.

The suggestions bug in Dynomantle is a simple concurrency problem because it is in Javascript. Let's go into a more complicated one that happened to be in Java, but one whose lesson should be transferable to many other issues.

My first job out of college was working on a network management application. Example: You are visiting a company and connecting to the guest wifi network. Our application would allow system administrators to configure your access based on login credentials, location in the office, time of day, etc. They could enable or block ports depending on their corporate policy.

The concurrency comes into play by the fact we supported multiple protocols. We supported 802.1x for wifi, but we also supported authentication based on the ethernet port someone connected to, the Kerberos authentication protocol, and a few others.

As soon as you turned your computer on, it would attempt to connect with as many protocols as it was configured to. At. The. Same. Time.

Let's say an admin set a less accessible policy for ethernet port access. You can maybe get ports 80 and 443 (basically just web browsing). If you authenticate with Kerberos you can get wider network access. The problem here is that order is irrelevant. The admin could configure which protocol had priority if a user authenticated with multiple ones.

The code handed to me when I started this project stored the status of authentication in a single database table and each person's MAC address was given one row and one row only:

- `primary_key` — int
- `mac_address` — varchar(255) and a unique key

- `authentication_protocol` — enum
- `status` — enum

(The real table was much more complex, but this was 15 years ago so bear with me)

The `authentication_protocol` column stored the protocol that took the highest priority and was successful. It would also store the protocol of the highest priority if all authentication attempts failed.

I've oversimplified the problem, but we needed thousands of lines of code trying to coordinate all the different protocols that came in, figuring out which had the highest priority, dealing with some of those protocols having multiple authentication steps, dealing with various locks throughout, and also accounting for some quirks in the firmware for various switch and router manufacturers. Customers were pretty unhappy because it rarely worked right and users were constantly getting the wrong network policy assigned to them.

I spent most of the first few months of my career fixing this one bug and then dealing with the follow-up bugs that popped up. Eventually, I realized that the problem was not that our user needs were complex. The problem was we built a bad data model and it made the code way more complex than it needed to be.

The solution was pretty simple. Take that same database table above. Now add a row for each MAC address and protocol. Instead of having one row and trying to coordinate which protocol to display in that row, just add a row for every protocol.

Concurrency is still happening, but you remove the need to coordinate what data to actually save from that concurrency. Each thread/process gets its own data that they exclusively modify. When determining network access for a user, just look up all the rows for that user and select the relevant one.

No locks. No shared data to modify.

The code ends up being simpler because you can ignore concurrency for the most part. Developers are happy. The code works properly. Customers are happy.

Realistic scenarios only had administrators set up 2–3 policies that would apply to a single person so I did end up increasing the table size by 2–3x. However, that's linear growth. Databases can handle linear growth easily. Going from 1000 rows to 3000

rows is irrelevant. Going from 1 million rows to 3 million rows is also irrelevant with modern hardware.

Going from 1 billion rows to 3 billion rows is probably relevant. However, you should have started scaling the database to support 3 billion rows well before you hit 1 billion anyway.

All of that was a long-winded way of saying: increasing the size of a table by 3x is well worth the price of not having to worry about concurrency.

This sort of problem is a common concurrency problem. A lot of data seems like it needs to be accessed and modified by different concurrent processes at once. Most of the time that isn't true. Small tweaks to a data model and taking advantage of the fact that storage is cheap can save your team a ton of work.

I hope this post was helpful to you! Feel free to reach out if you have any questions or would like some advice on your own concurrency problems.

Want to Connect With the Author?

Subscribers to my newsletter also get some bonus content describing some other concurrency problems. You can subscribe here.

This article was originally published at https://blog.professorbeekums.com.

Programming | Web Development | Coding | Software Development | Software Engineering

Follow

Written by Beekey Cheung

284 Followers · Writer for Better Programming

Building a way to overcome information overload at https://www.dynomantle.com/

More from Beekey Cheung and Better Programming

Beekey Cheung

Certificates Don’t Prove Competency

I’ve written before about how I think college GPAs are a useless metric for hiring managers.

3 min read · Jan 20, 2022

13 1

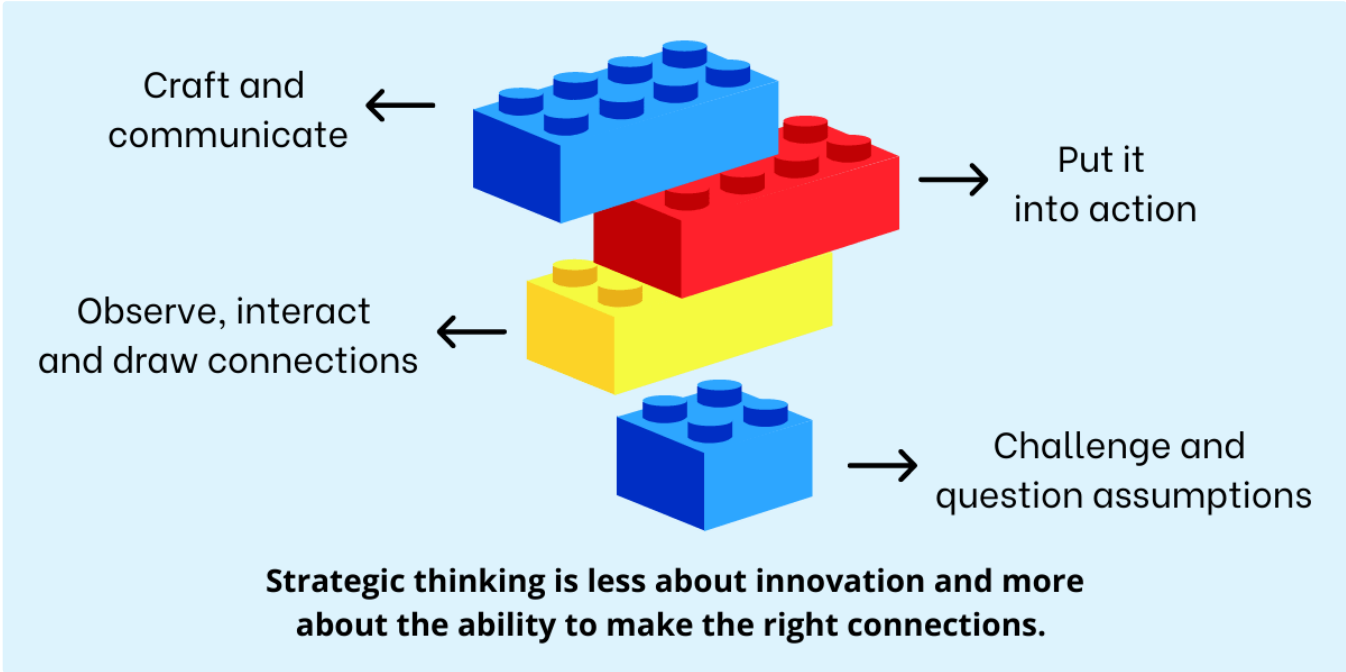
Allen Helton in Better Programming

ChatGPT Changed How I Write Software

Generative AI has completely changed the way I approach software design. I don't think I could ever go back.

· 8 min read · Jun 1

1.7K 20



Vinita in Better Programming

How To Develop Your Strategic Thinking Skills and Stay Ahead

Cross the boundary of your comfort zone to think about an idea to its extreme without mental guardrails to put it down.

✦ · 10 min read · May 31

1.2K 9

Beekey Cheung

Side Projects Make You Better At Your Full Time Job

In 2014, I had been hearing a lot of talk about Go. It sounded like a really exciting language that I wanted to try.

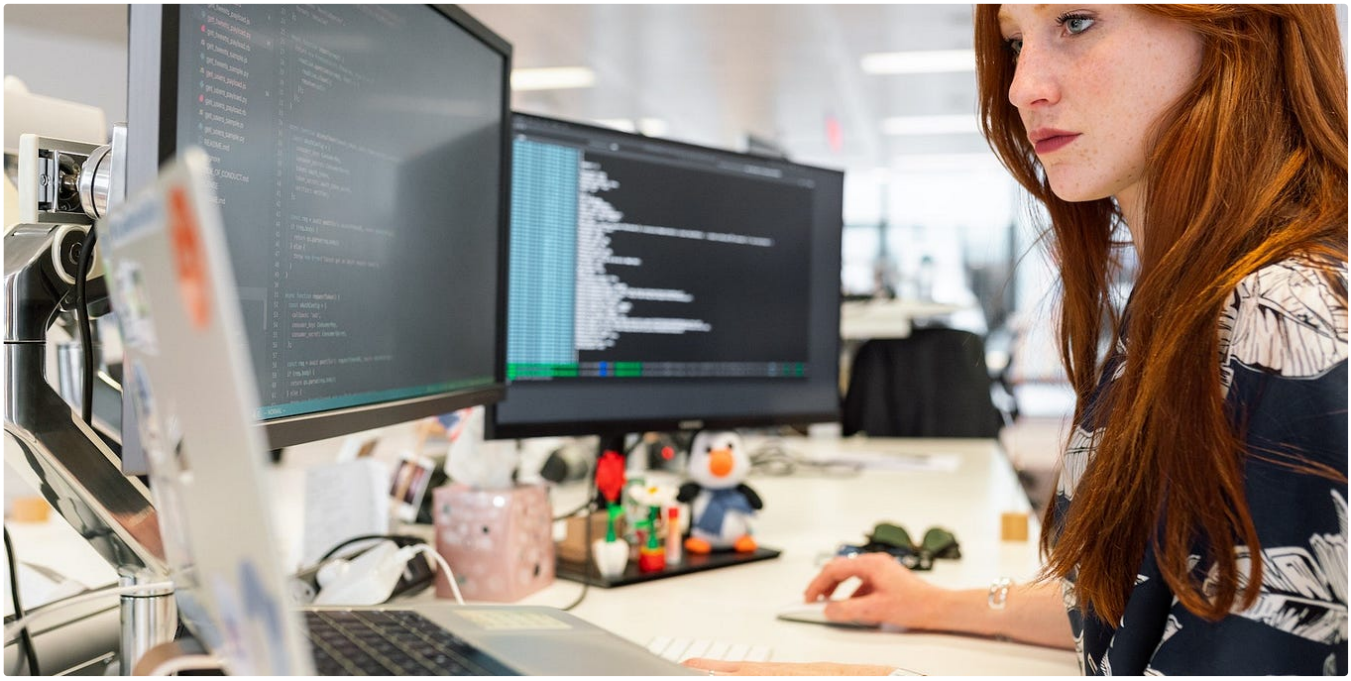
5 min read · Mar 21, 2022

212 2

See all from Beekey Cheung

See all from Better Programming

Recommended from Medium



The Coding Diaries in The Coding Diaries

Why Experienced Programmers Fail Coding Interviews

A friend of mine recently joined a FAANG company as an engineering manager, and found themselves in the position of recruiting for...

· 5 min read · Nov 2, 2022

4.5K 99

Arslan Ahmad in Level Up Coding

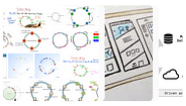
Don’t Just LeetCode; Follow the Coding Patterns Instead

What if you don’t like to practice 100s of coding questions before the interview?

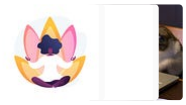
· 5 min read · Sep 16, 2022

2.8K 10

Lists



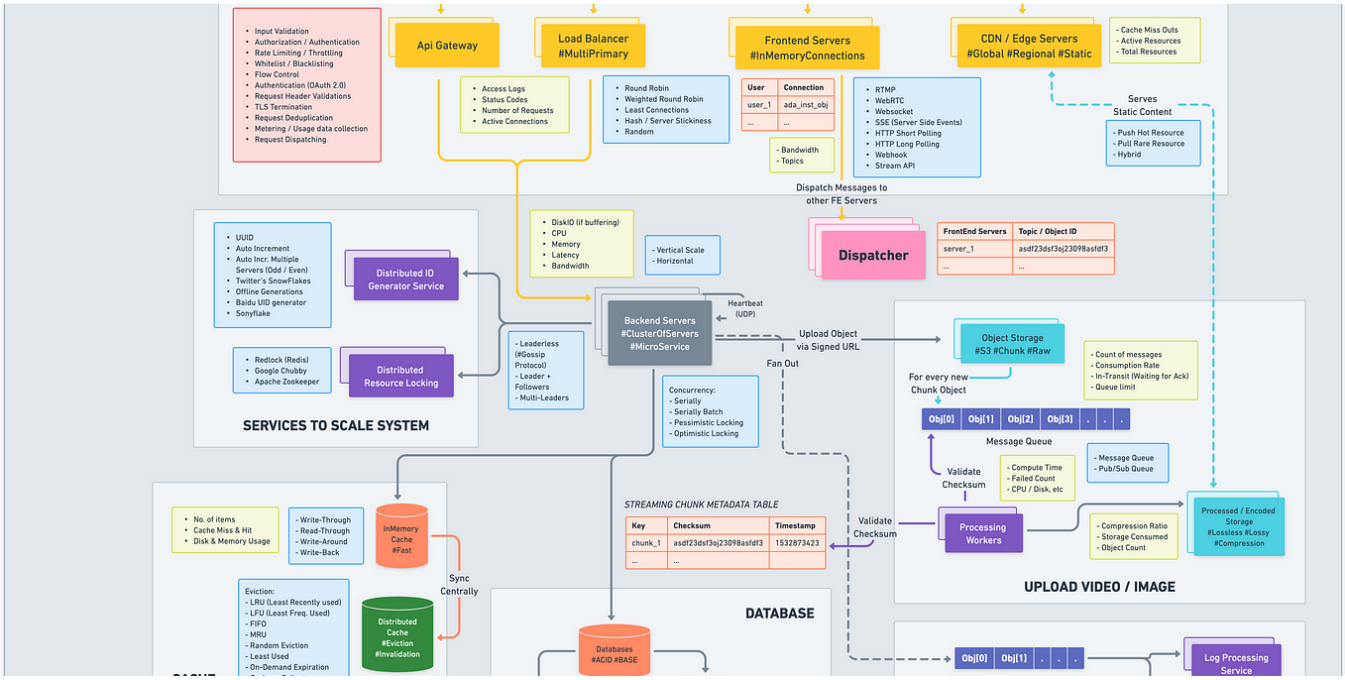
General Coding Knowledge
20 stories · 34 saves



Stories to Help You Grow as a Software Developer
19 stories · 152 saves

It's never too late or early to start something
10 stories · 10 saves

Coding & Development
11 stories · 21 saves



Love Sharma in Dev Genius

System Design Blueprint: The Ultimate Guide

Developing a robust, scalable, and efficient system can be daunting. However, understanding the key concepts and components can make the...

✦ · 9 min read · Apr 20

4.7K 32

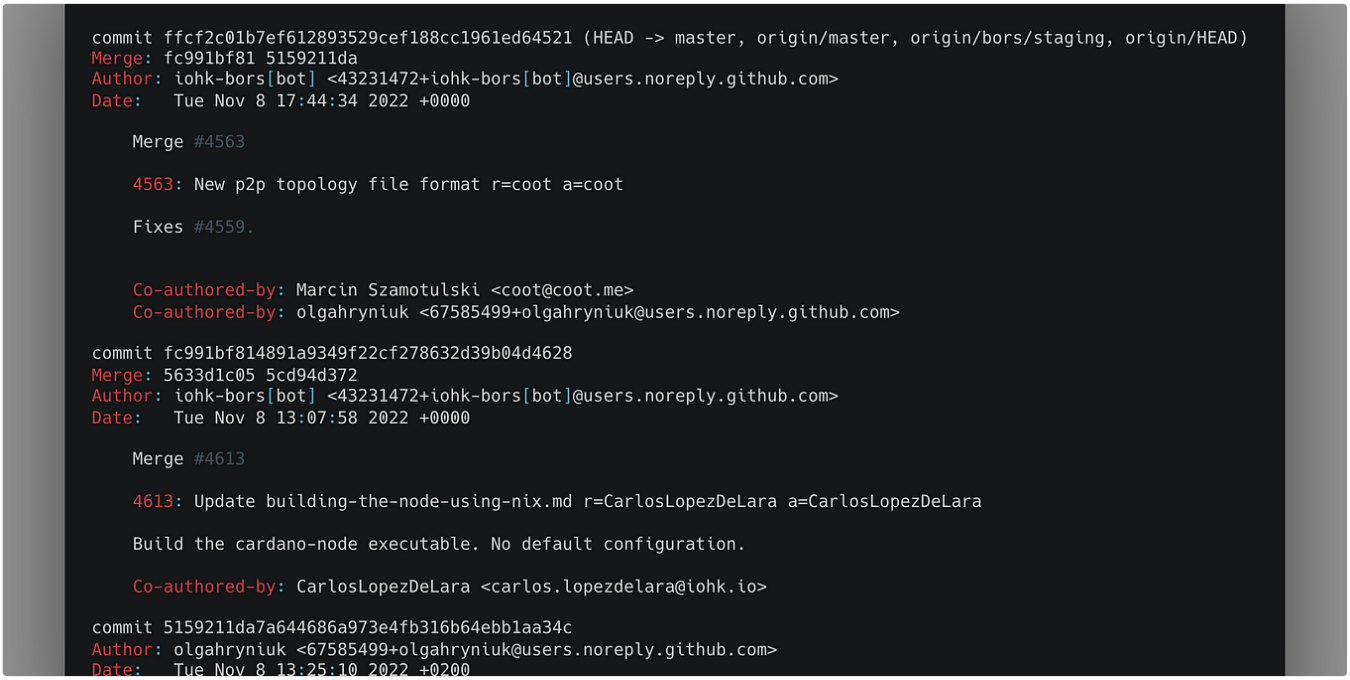
Anthony D. Mays

How to Practice LeetCode Problems (The Right Way)

tl;dr: You're doing it wrong. Use "The Six Steps" any time you practice LeetCode questions, preferably with another person. Keep an...

✦ · 13 min read · May 10, 2022

1K 9



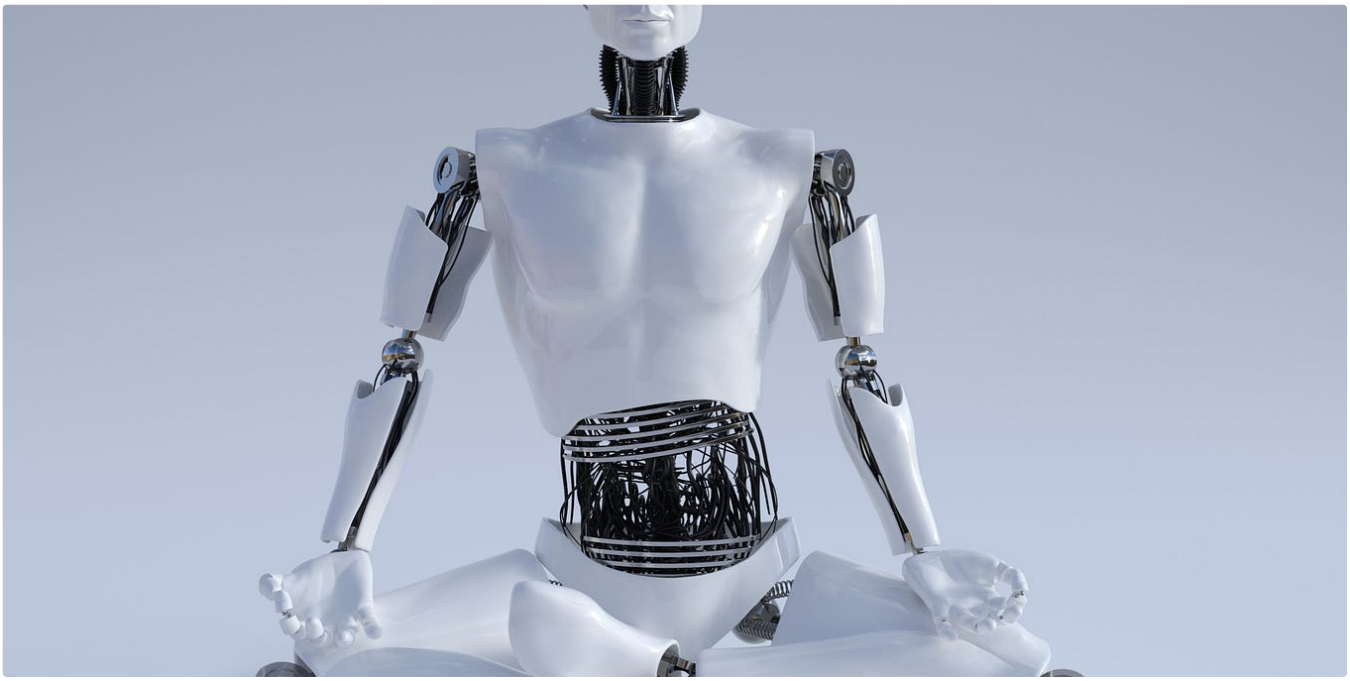
Jacob Bennett in Level Up Coding

Use Git like a senior engineer

Git is a powerful tool that feels great to use when you know how to use it.

✦ · 4 min read · Nov 15, 2022

6.8K 67



The PyCoach in Artificial Corner

You're Using ChatGPT Wrong! Here's How to Be Ahead of 99% of ChatGPT Users

Master ChatGPT by learning prompt engineering.

· 7 min read · Mar 17

26K 455

See more recommendations