Coaching through coding

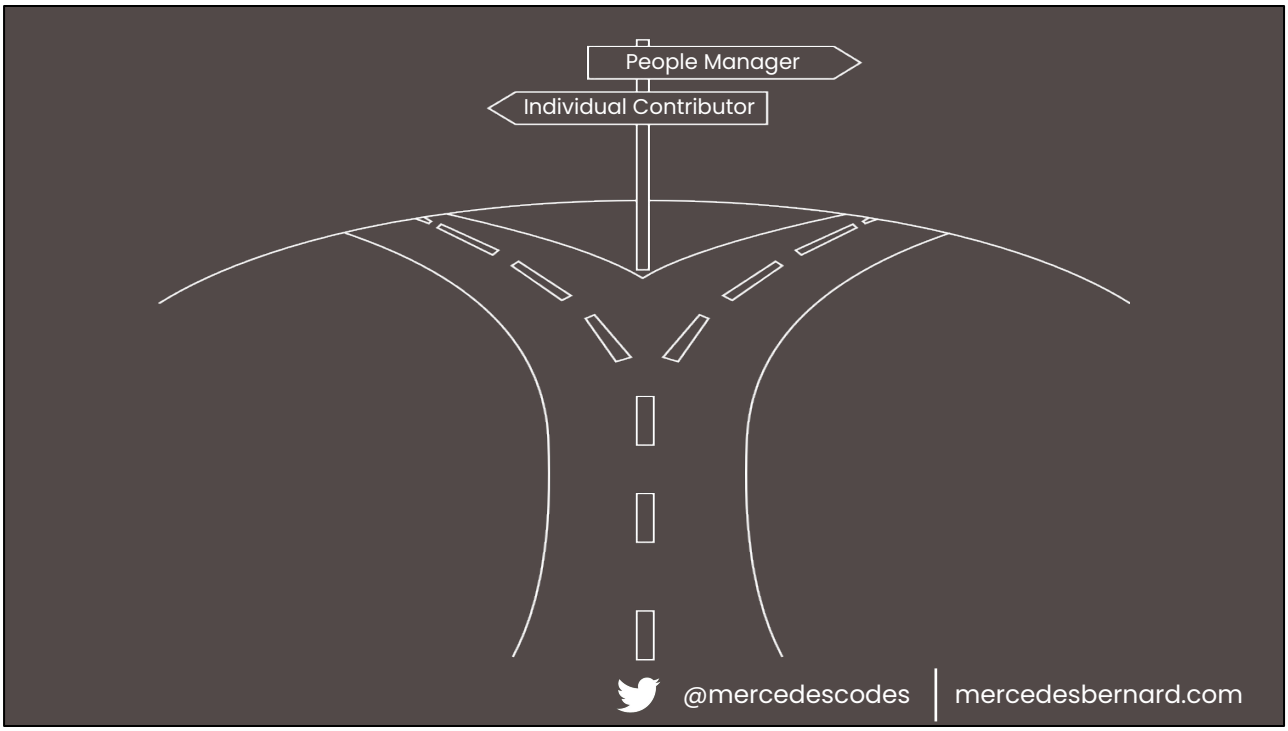
 @mercedescodes | mercedesbernard.com

Hello! Today we're going to talk about finding opportunities for coaching and mentorship in the daily technical activities of our work as individual contributors. This talk is written for senior+ engineers, senior, principal, staff engineers, etc. But I hope everybody here today will find at least one new strategy to bring to their team to create a more welcome learning environment for everyone regardless of level.

My name is Mercedes Bernard. My pronouns are she/her. And I'm a Principal software engineer/engineering manager with a digital consultancy in Chicago called Tandem.

If you would like to follow along, you can find my slides on my website and I also tweeted out a link right before this talk if that's easier.

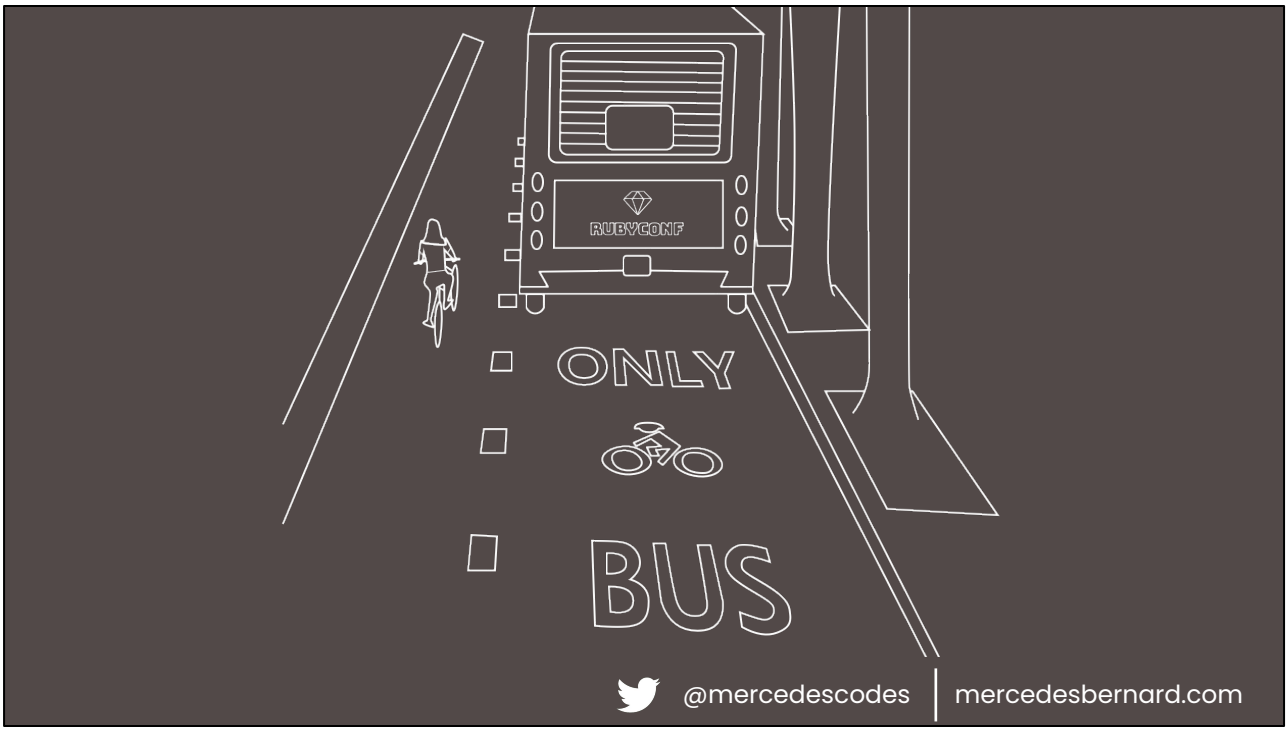
Feel free to Tweet at me, live-tweet the talk, or shoot me messages in Slack. I'll be doing a virtual Q&A right after this talk and will catch up with all of you then.



As we progress in our careers, we're often met with a fork in our career path. Do we want to progress into people management or do we want to become a technical expert and stay on the individual contributor (IC) track?

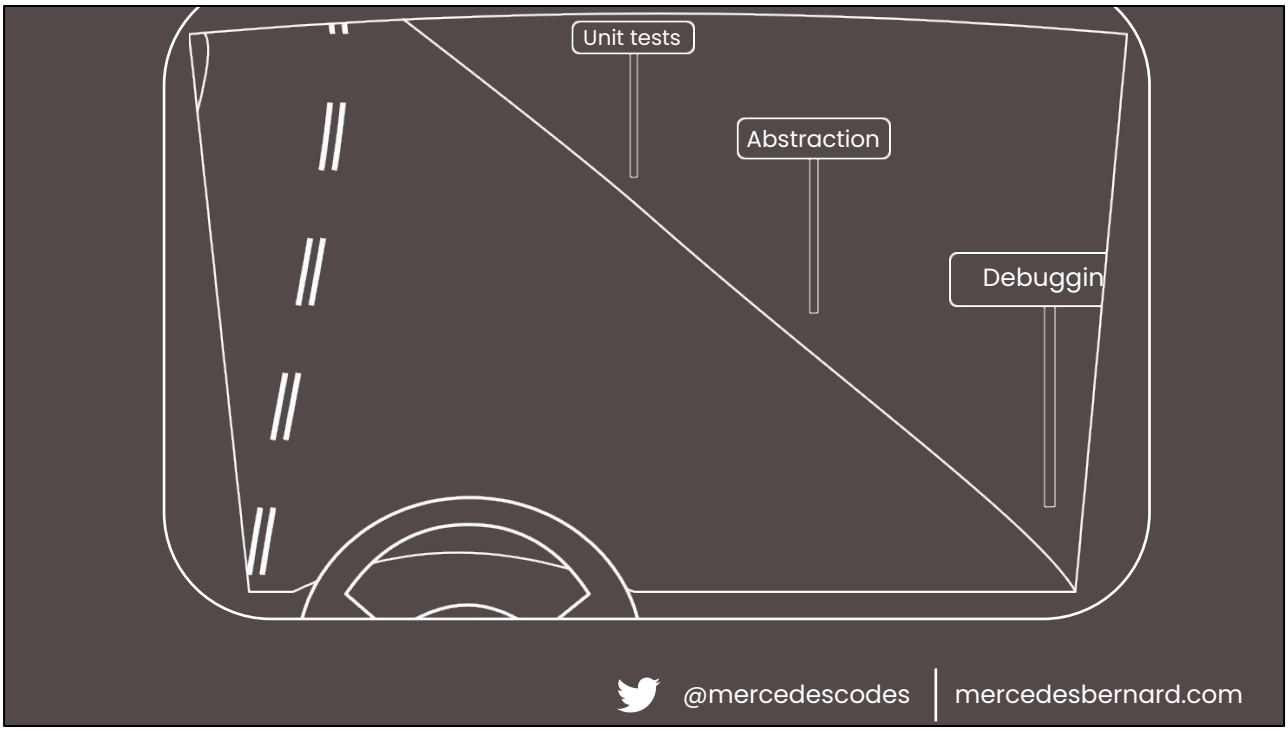
For anyone who isn't familiar, "Individual contributor" is the term used to describe the role of a software engineer who contributes their deep technical skills and ability to a team but doesn't have people management responsibilities.

Personally, I find this distinction between people management and individual contributor somewhat limiting. I think folks who manage people can still be technical. And I think people who strive for deep technical knowledge still have a responsibility to share that knowledge with others.



What if I told you that instead of a fork in the road, the choice to pursue a management or IC role is really like deciding whether you want to change into the bus lane or stay in the bike lane?

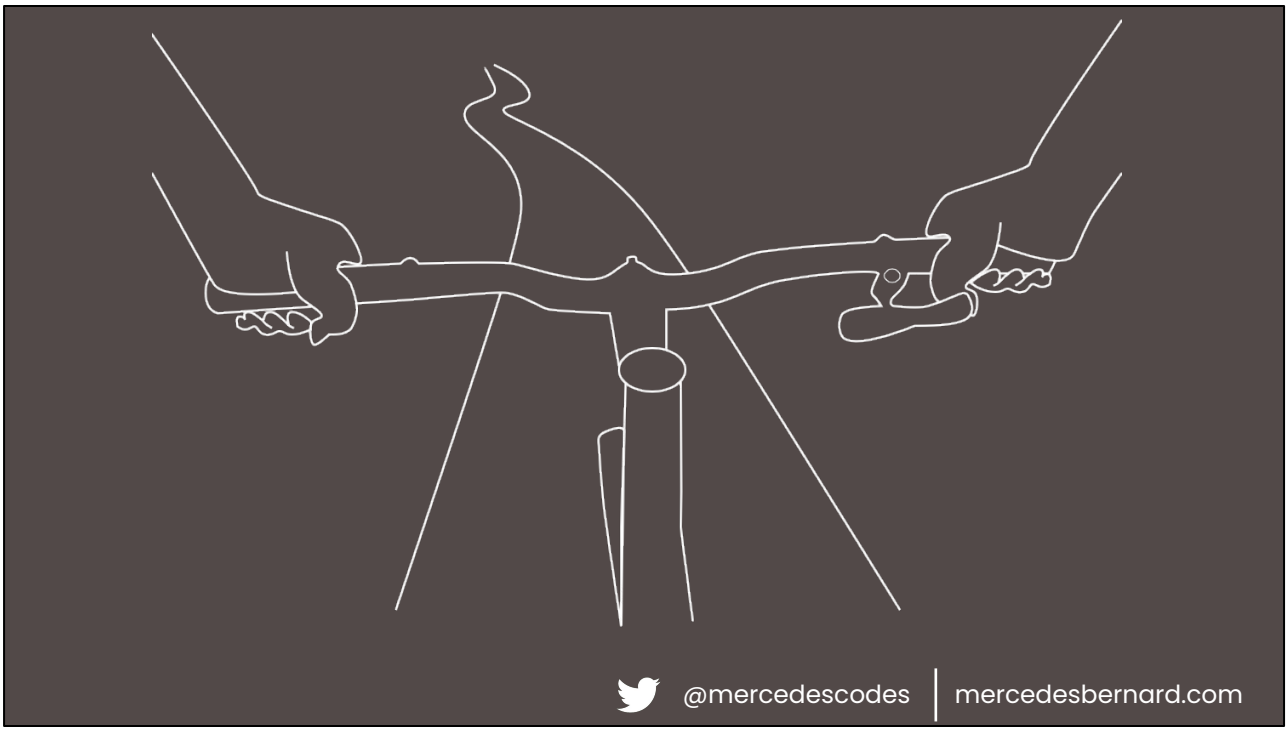
Hear me out.



When we go into people management, we're responsible for supporting the career growth of many different people at different stages in their careers. It's kind of like you're driving a bus and helping your team get to where they're going.

When they need to work on a skill, they can get off at that stop and invest time in honing that skill. Then when they're ready, you pick them up on your next route through and mentor them on possible next steps they could take. And they'll get off at another stop when they find another skill they want to grow and invest in.

A good manager shows you where you can go and helps you get there. They find opportunities for you and offer directions if you need them.



Being an IC is more like commuting on your bike. On your bike commute, you can take the quickest route to get to your personal, desired destination. You can find all of the shortcuts and all of the convenient detours to avoid tricky intersections.

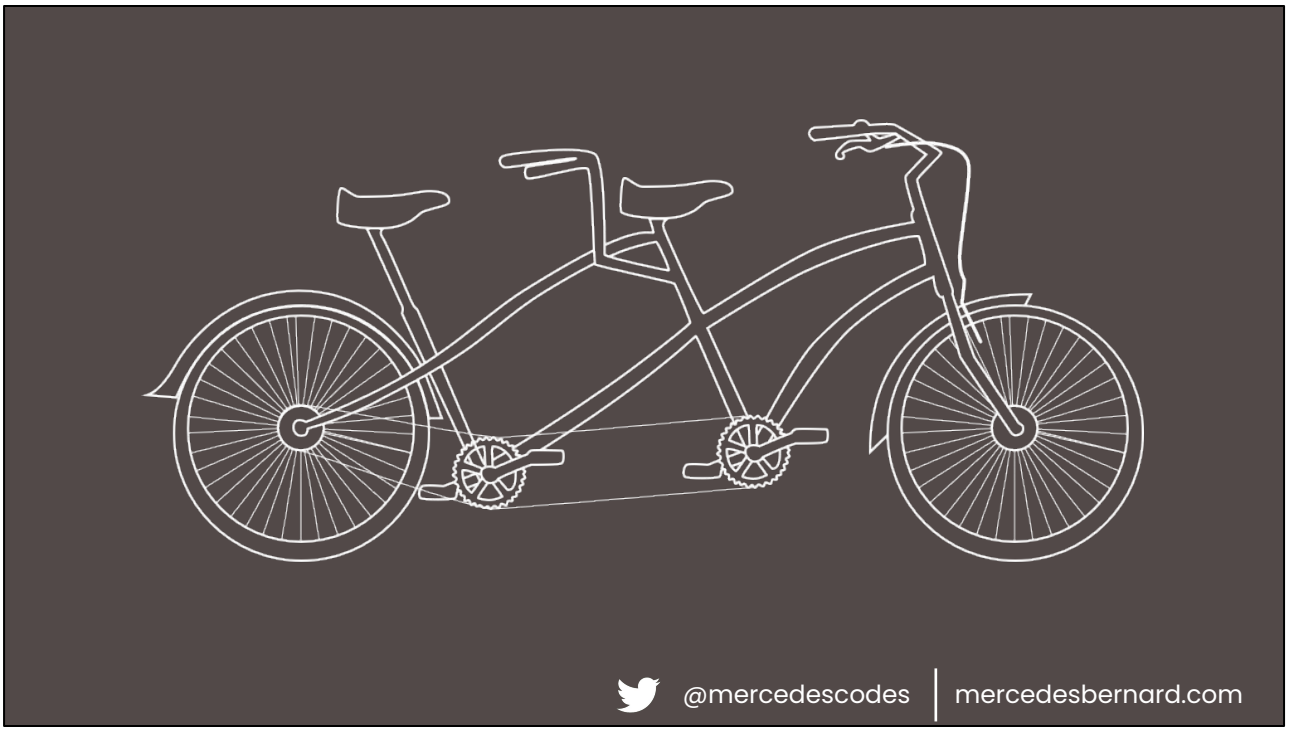
It might be more effort for you to bike to work than take the bus but you can take a fast, straightforward route that works for you.

Often, it's not in the team's best interest for one person to get from point A to point B as quickly as possible. As ICs, our success is often measured by the technical problems we solve.

I think our industry has taken the "individual" part of individual contributor too far, assuming that a top performer is someone who can do the work of 10 others rather than someone who can enable and empower the work of 10 others successfully.

I take heart in the fact that the 10x developer has become a meme at this point. But there are still too many people responsible for building teams that are looking for their unicorn.

Team success is marked by continuous learning and growth. And our senior ICs should support the team success by finding ways to contribute to everyone's professional development and learning.



Sometimes the most enjoyable commute is one you take with someone. I'm proposing that we start taking more tandem bike rides together, slowing down just a bit to ensure that the whole team succeeds.

Mentorship vs. Coaching

@mercedescodes | mercedesbernard.com

Throughout this talk, I'll be using the terms mentorship and coaching. "Mentorship" is a fairly common concept that's tossed around a lot in our industry and we're starting to hear about "coaching" more and more. They both fall under the mentorship umbrella, but there is a little nuance that I think is important to call out.

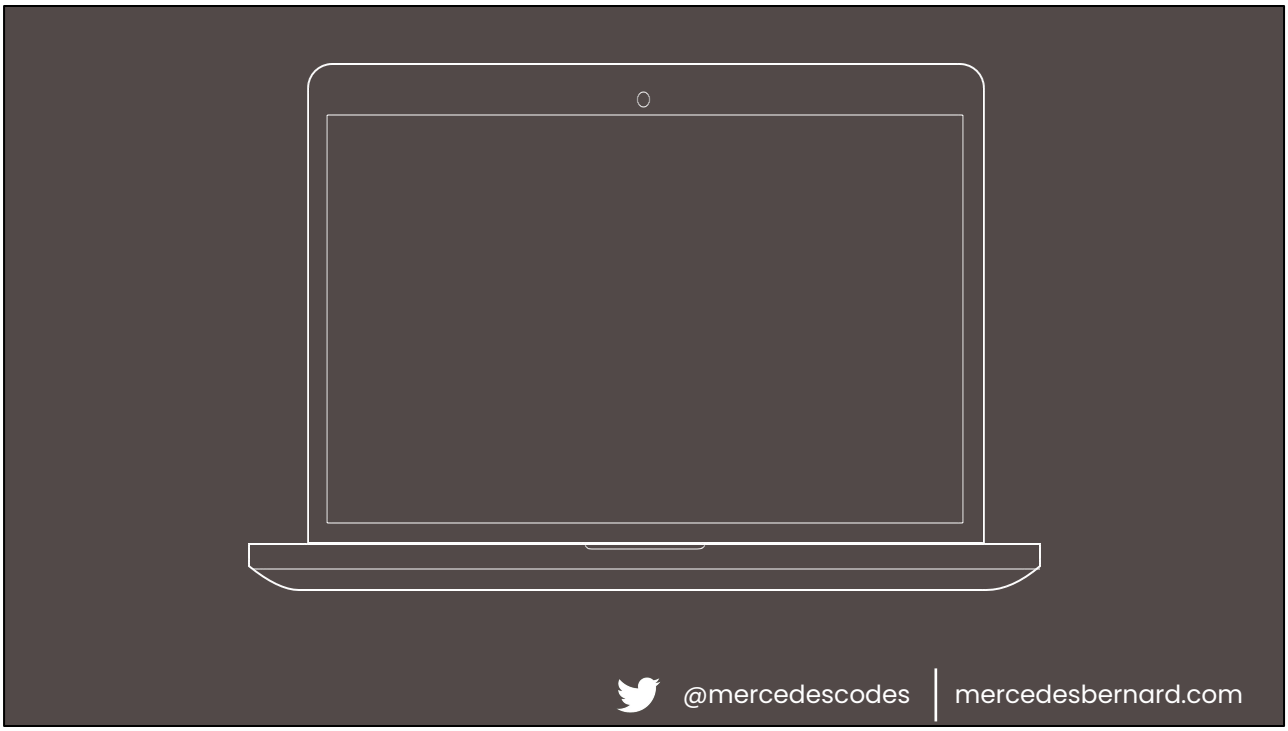
When we hear the term “mentorship,” a lot of us tend to think about 1:1 conversations over a cup of coffee or tea where the mentor tends to do most of the talking.

This is because mentorship is very relationship-oriented. It is reliant on getting to know one another well in order to share goals and offer advice. Conversations like these require a lot of trust and vulnerability, so mentorship relationships tend to be more long-term.

Coaching, on the other hand, is less about the future and more grounded in the present. Instead of offering advice for potential goals and career growth, coaching fosters the growth of current skills by providing feedback and guidance for the task at hand.

Because coaching is more skills-focused, it doesn't require as much of the long-term commitment for relationship building. This makes coaching an incredibly powerful tool that can be used in many different interactions.

While managers support the career growth of their team through mentorship, everyone has the opportunity to support their team's technical skill growth with coaching.

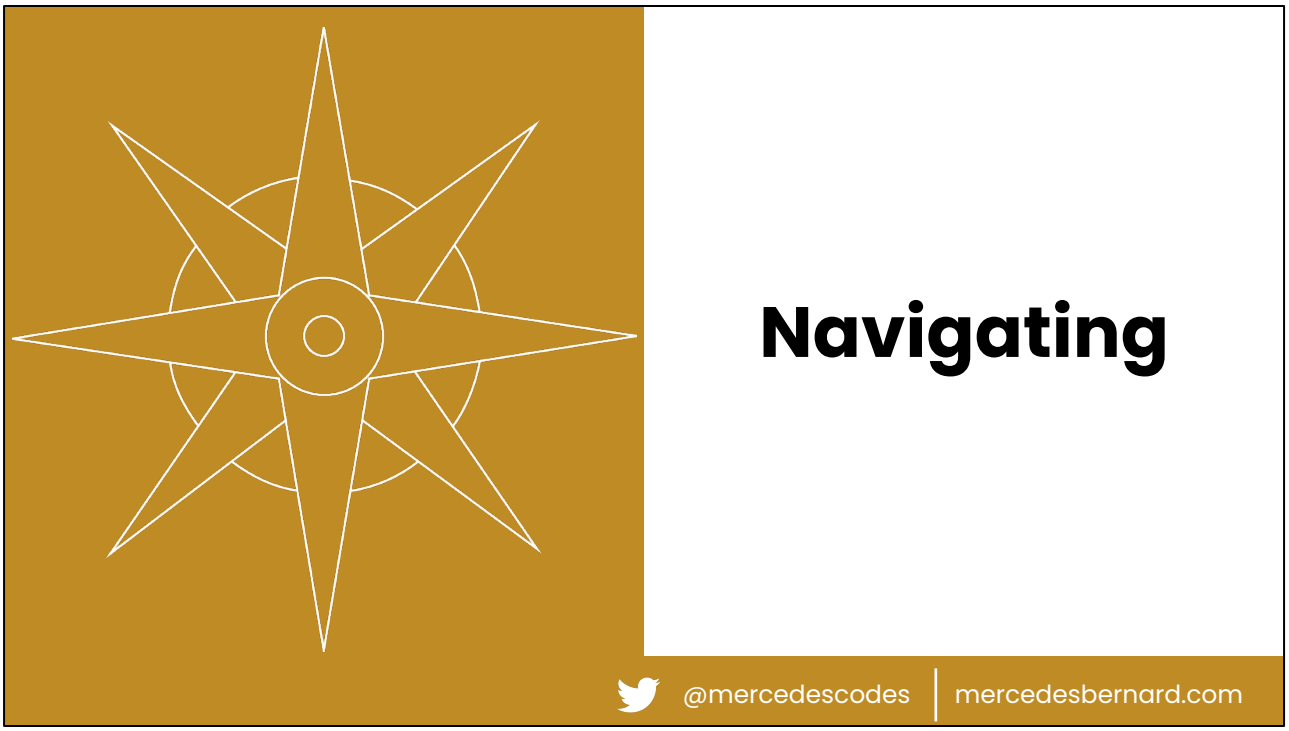


As a senior+ developer, we write a lot of code but we also have a whole variety of other technical responsibilities that we participate in. And every single one of those is an opportunity to coach someone and level-up their skills. I'm going to walk through examples starting with very code-heavy activities and moving to less code-heavy activities.

Pair programming

@mercedescodes | mercedesbernard.com

Pair programming is a wonderful coaching tool when approached correctly. Collaborating on a piece of code with a less experienced team member provides a safe learning environment. You can provide suggestions and feedback without going into full Teacher mode which keeps the session comfortable and incredibly valuable.



As a senior+ engineer, you should spend a lot of your pairing time navigating.

How do you decide if you should be navigating on a specific task? Tasks that may be too challenging for your pair to implement solo or that may solicit a lot of code review feedback are good candidates for navigating to help them to level up their ability to work on complex tickets. I tend to think about complex stories as those that touch many layers of your system but it may also be ones that require a lot of domain context or are using a tricky design pattern or some other scenario specific to your project.

In the navigator role, you have the opportunity to coach about code architecture and organization. And you also leave your pair in control of the session as the driver. Power dynamics during pairing are real, whether they're based in gender, race, organization hierarchy, personality differences, or other factors. Given the reality of our industry, the power differential usually skews to privilege the more senior engineer. By having your pair drive, you can create a safe pairing space by setting expectations that they control the pace and they can stop the session to ask questions or get more clarity any time they want it.

While you're navigating, stay focused on the big picture thinking. Let the driver decide on naming and syntax. Be ready to help where needed but let them focus on the details. This can be especially helpful for someone early in their career to get comfortable making decisions between 2 similar implementations (should I use a `.map` or an `each`) or to get the muscle memory for certain things we take for

granted like creating Rails migrations or spec files.

And as they are making decisions, you can provide feedback and context to model thinking through the tradeoffs between choices.

Where you take a more active role in the pair session is deciding how the individual pieces of code the driver is writing will fit together. You can coach them on deciding when to create an abstraction or when to make a service object for easier readability or maintenance. Hearing how you make your architecture decisions and how you put the pieces together will strengthen their architecture skills.

This might be counterintuitive but as a senior+ engineer, I think you should try to spend 50% of your time driving during pair sessions.

When should you be driving? Tasks that would be a stretch for your pair to implement solo are good candidates. And when I say stretch, I mean within their scope of skills but something they would be challenged by: a new pattern they haven't used before but exists within the codebase or a problem they haven't solved before but is similar to something else they've done. Driving on these types of tasks is a good way to foster your pair's technical thinking and communication by providing support in that stretch space.

Other good candidates for driving are areas of the code that your pair is more familiar with than you are. As senior+ engineers, we need to be comfortable recognizing the expertise of our team members and celebrating it. This does wonders for combatting imposter syndrome and leveling the playing field so that everyone on the team feels safe and comfortable sharing opinions and lending expertise in conversations.

However! When I say drive, I mean the very traditional pair programming definition of driving where you are concerned with the smaller issues at hand: syntax and naming. This makes space for your pair to practice their technical communication skills. Vocabulary is really hard in software engineering and we all need to practice explaining our technical ideas to become skilled at this type of communication. We also need the practice to be able to identify what we don't know and how to ask questions to help us find solutions.

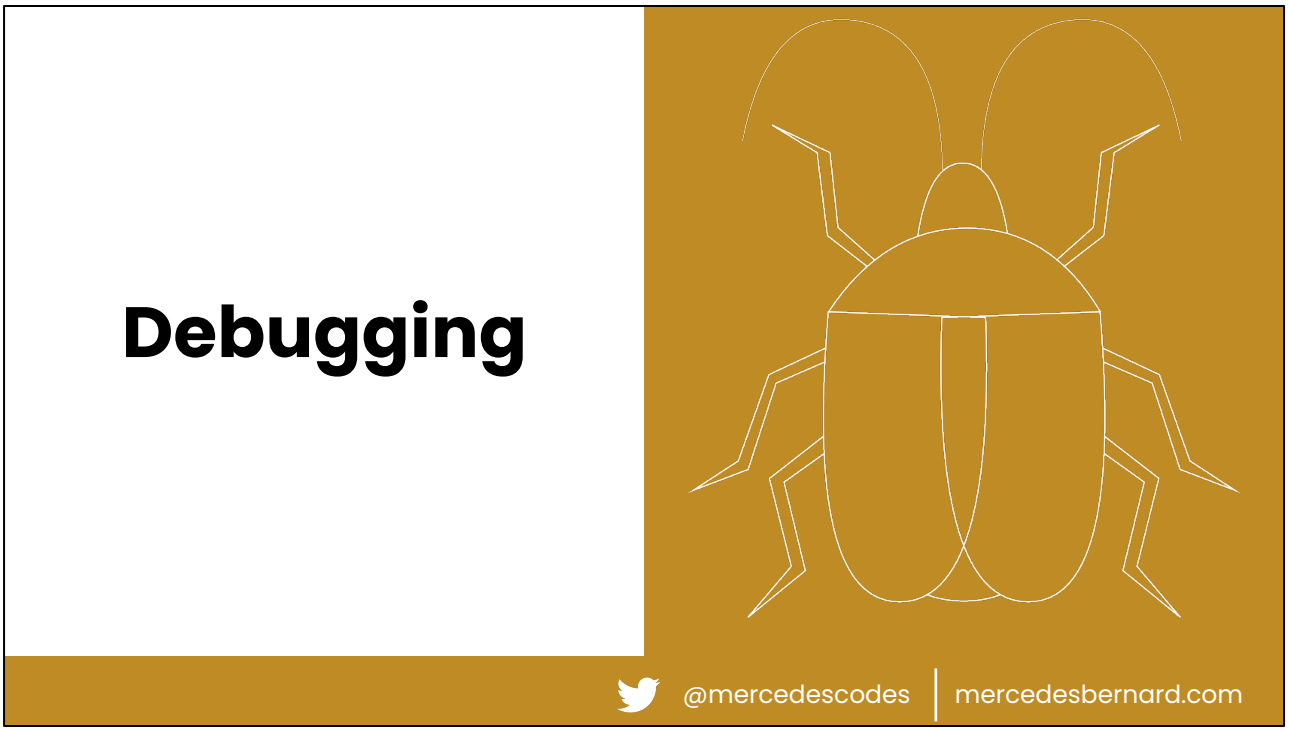
When you are driving, resist the urge to fill silences or “give them the answer”. Use questions to understand what the navigator wants to do next and mirror their statements to check for your understanding. Provide them with the vocabulary if they get stuck and use pseudocode liberally to capture the navigator’s ideas. Sometimes seeing code can help people better articulate what they’re thinking.

As you’re driving, explain why you are naming things the way you are or how you’re making decisions about which lines of code you’re writing or why you’re passing the parameters you are. This is an opportunity for you to coach on readability and simplicity to help the next developer understand your code.

If you find yourself taking too much control of the pairing session and your pair just watching you code, you should pause.

Are you navigating so much because this task is too complex and not a good stretch opportunity? If so, talk to your pair and ask to switch to navigating so that you’re both engaged in the session.

Are you navigating because it’s hard to let go of control? Take a break, grab a drink of water, and when you start pairing again, acknowledge that you were navigating too much, apologize and be more aware as you continue. It takes practice and no one’s perfect. Recognizing when you’re falling into this pattern will help you break the habit.



Debugging sessions tend to blur the line between driver and navigator a bit more than stories since neither of you may know what's happening or what's causing the issue.

If you are in the navigator role during a debugging session, you can explain how you're interpreting the symptoms that you're seeing: error messages, odd behavior, steps to reproduce, data discrepancies, etc. You can lend your expertise by pointing the driver where to look and then describing why you think something is or isn't important to continue investigating.

When you're driving, your pair can practice critical thinking and problem-solving skills since they'll be reading error messages, interpreting their meaning, and directing you where to look for the cause of the issue. This is incredibly valuable because these are the kinds of skills that aren't taught in most education programs. Having your pair make decisions about what to investigate next makes it easier for them to remember next time that it might be a data issue or to isolate whether the bug is happening on the client or server by checking the network tab.

My personal preference during a debugging session is to have my pair navigate.

I was recently pairing with an apprentice and they were driving while we were debugging an interesting React bug. They were so interested in "fixing the problem" that they were making changes faster than the hot-reloading could catch up and within a couple minutes, I had no idea which change we were testing in the browser because Webpack was still rebuilding… something… In that moment, I asked them to

pause and explain why they made the last change did. We laughed because they were kind of doing a “throw it at the wall and see what sticks” approach but it wasn’t super helpful because we couldn’t keep track of what they had tried or why they were trying it. We swapped roles in the pairing session and I was able to have them navigate and explain what changes they wanted to make and we were able to be more systematic and isolate the bug.

In my experience, early career devs try to move through debugging steps really quickly because they feel pressure to “find the answer.” As the driver, you can slow them down a bit and make sure that you are methodical as you are checking your assumptions and trying to reproduce the issue.

Code reviews

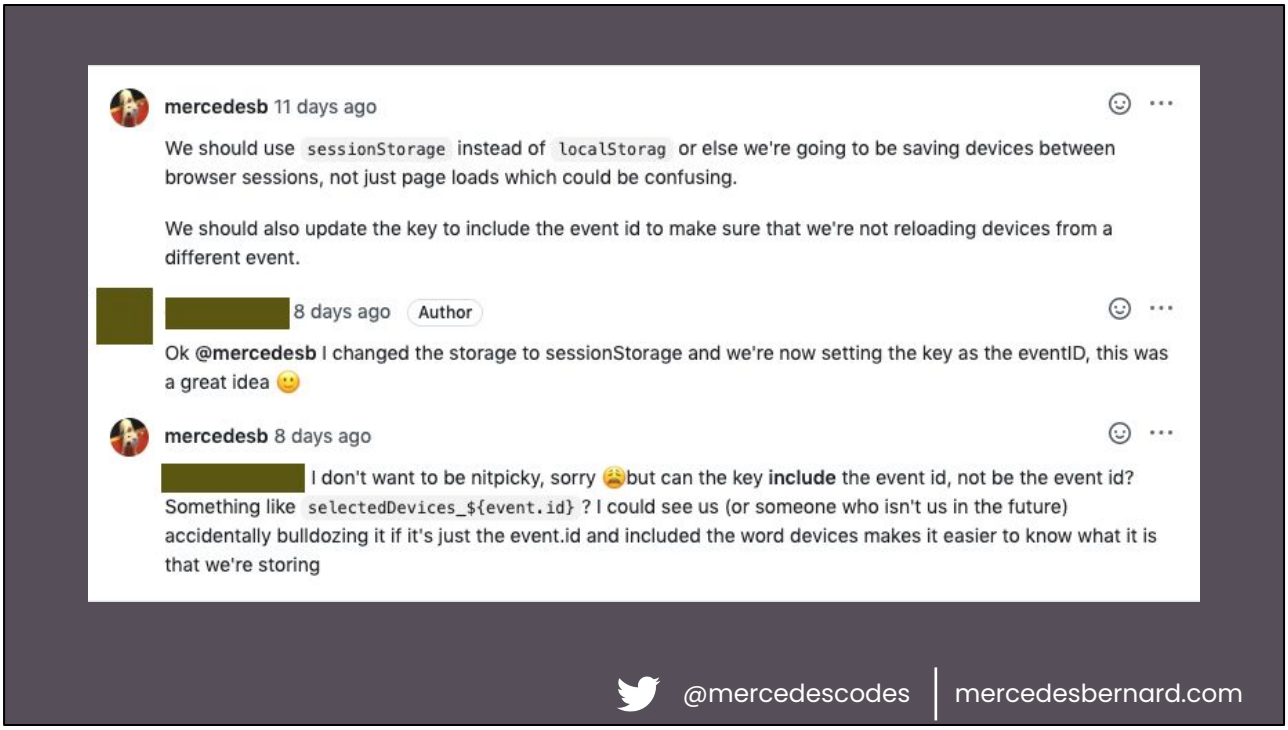

@mercedescodes | mercedesbernard.com

Code reviews can be intimidating. Going into Github (or your source control tool of choice) and seeing that big red “Changes requested” can be discouraging if not treated with care. Without proper consideration, the feedback given lacks context and can be less than helpful. But when prioritized as a coaching tool, code reviews provide some unique opportunities for sharing knowledge and developing technical skills.

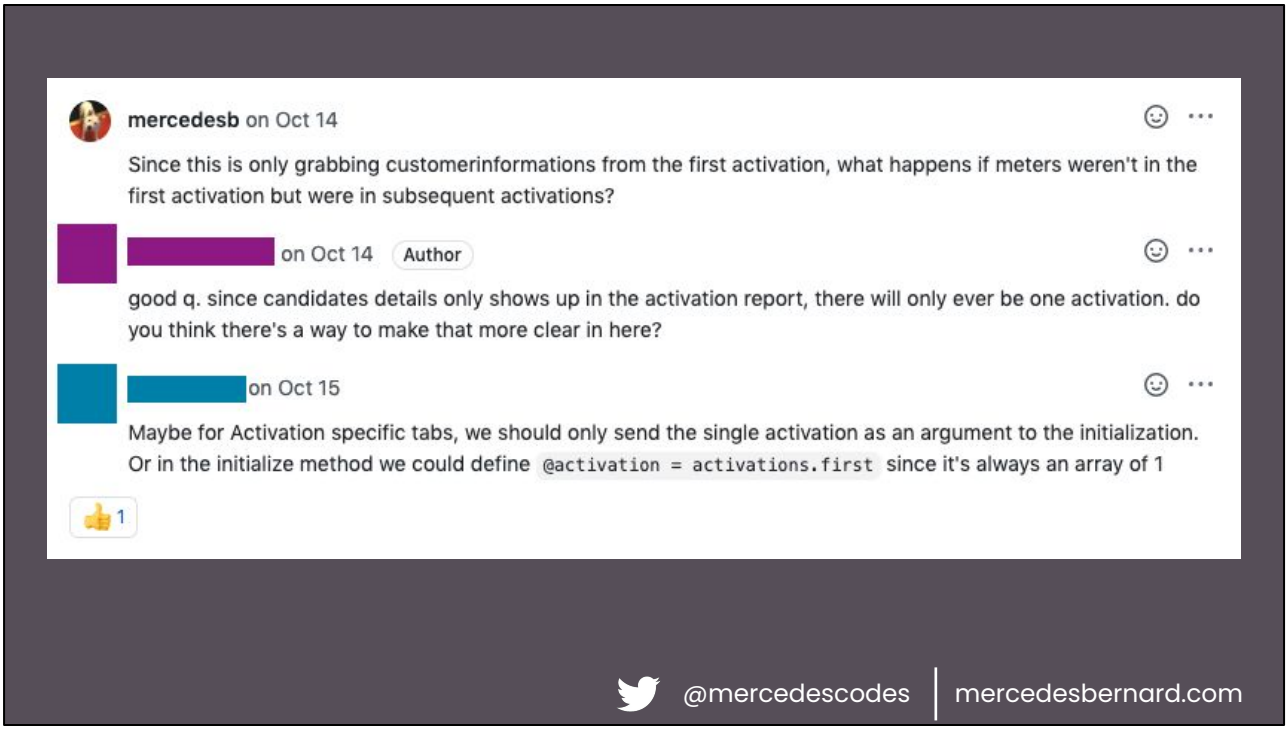
Some of the benefits of code reviews compared to pairing are that they can be asynchronous so you don’t have to align your calendars which can be especially challenging now that most of us are remote and many teams are juggling timezones.

And if you are using a tool such as Github’s pull requests, code reviews also serve as long-lived written documentation and feedback that the reviewee and others on the team can refer back to.

When providing feedback about something that you think should be changed, make sure that you are specific about the reasons why. Instead of using vague terms or talking about “better,” explain what the benefit your proposed change will have. Explaining the benefits and potential tradeoffs and why you prefer the proposed solution models technical decision-making skills for the reviewee.



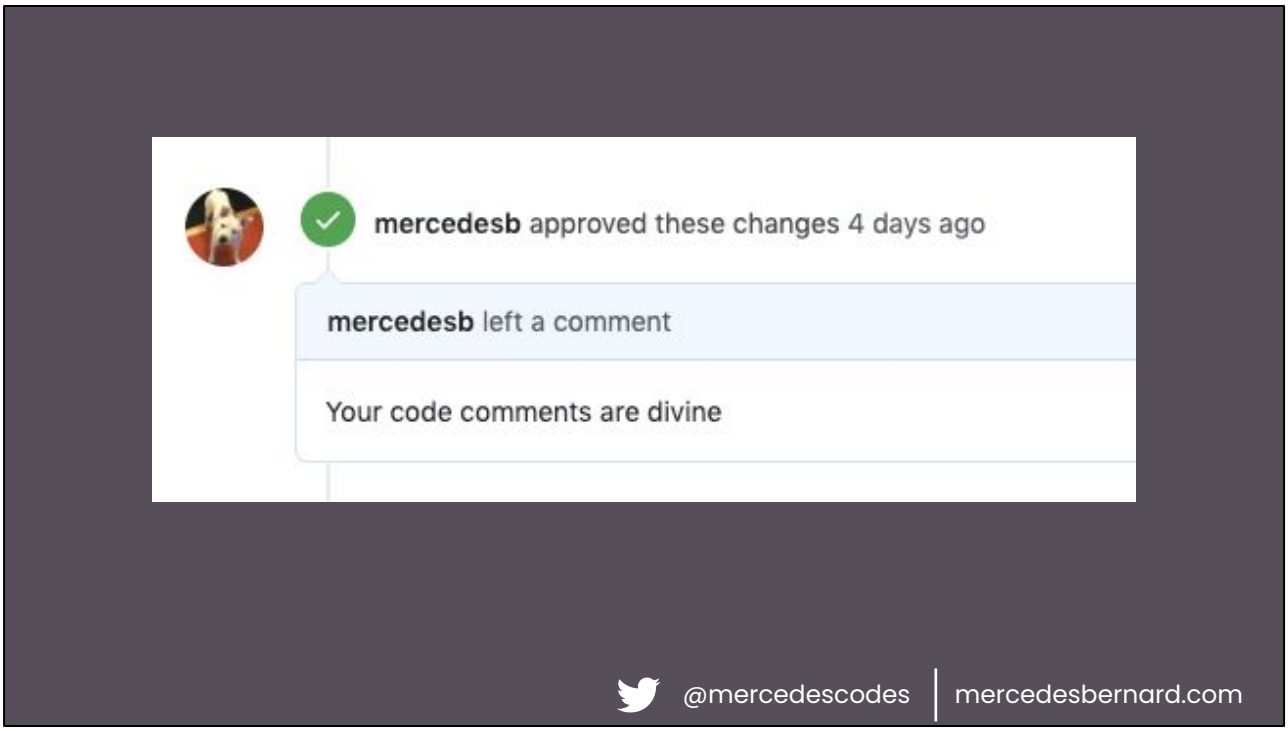
Without ever sharing the reasoning that goes into a decision, it's hard for someone to learn the various considerations to think about when they are coding. Taking this little extra time to model this behavior helps them develop these skills when they are making their own implementation decisions next time.



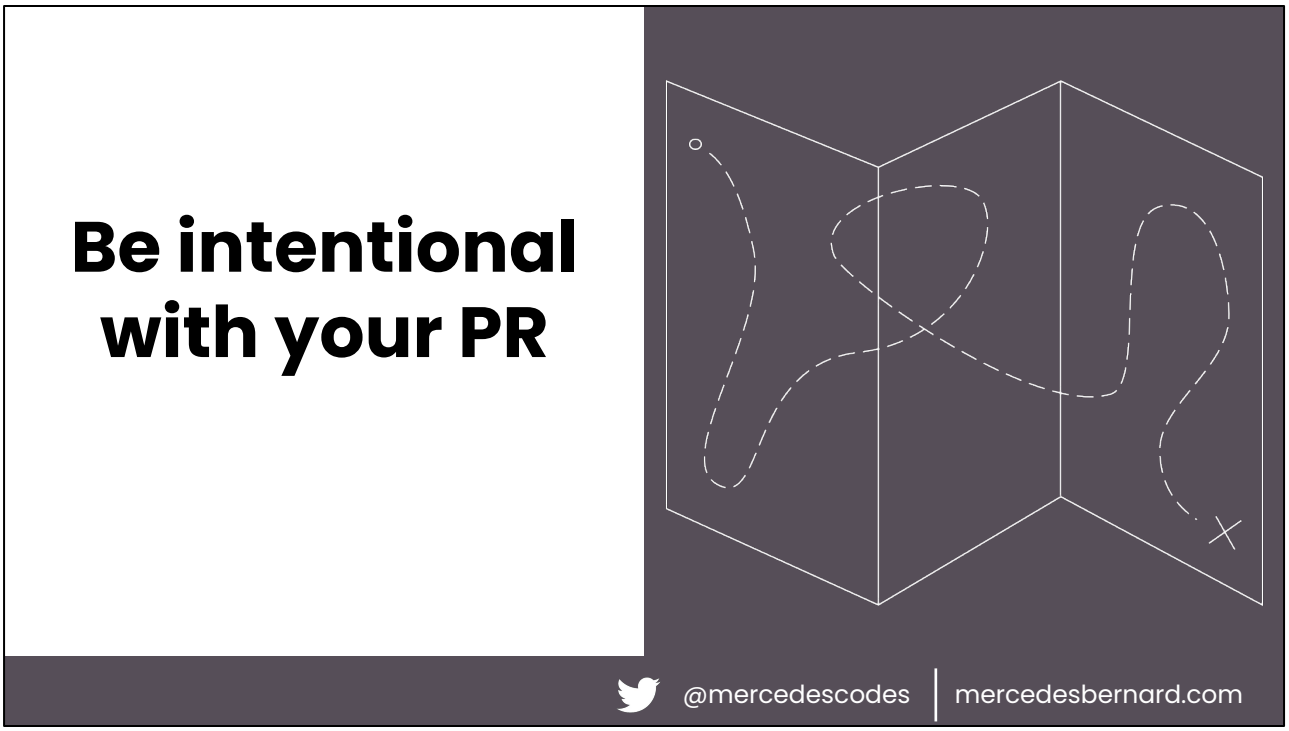
It's also important to balance prescriptive requested changes with open-ended questions from both a kindness and a coaching perspective. Code is subjective and there is no "right" way to do it, open-ended questions allow you to make suggestions without forcing someone into your choices. They are also a useful coaching tool to stretch someone's skills as they progress in their career. Rather than providing an answer for them, you can prod them to think about something they might not have considered, such as performance. Pointing out nested iterations and asking, "How might we reduce the nesting levels here?" provides them some guideposts to think through alternative implementations.

Open-ended questions are more process-oriented than results-oriented. They facilitate a dialog about the code and make space for a larger conversation which paves the way for more coaching opportunities through pairing or implementation planning.

Code reviews aren’t only for requesting changes and picking apart code. There is always something positive to find while doing a code review. It can be a well-written test, a great variable name, or just the evidence of progress that someone’s coding skills are leveling up, such as their methods are getting clearer and easier to understand.

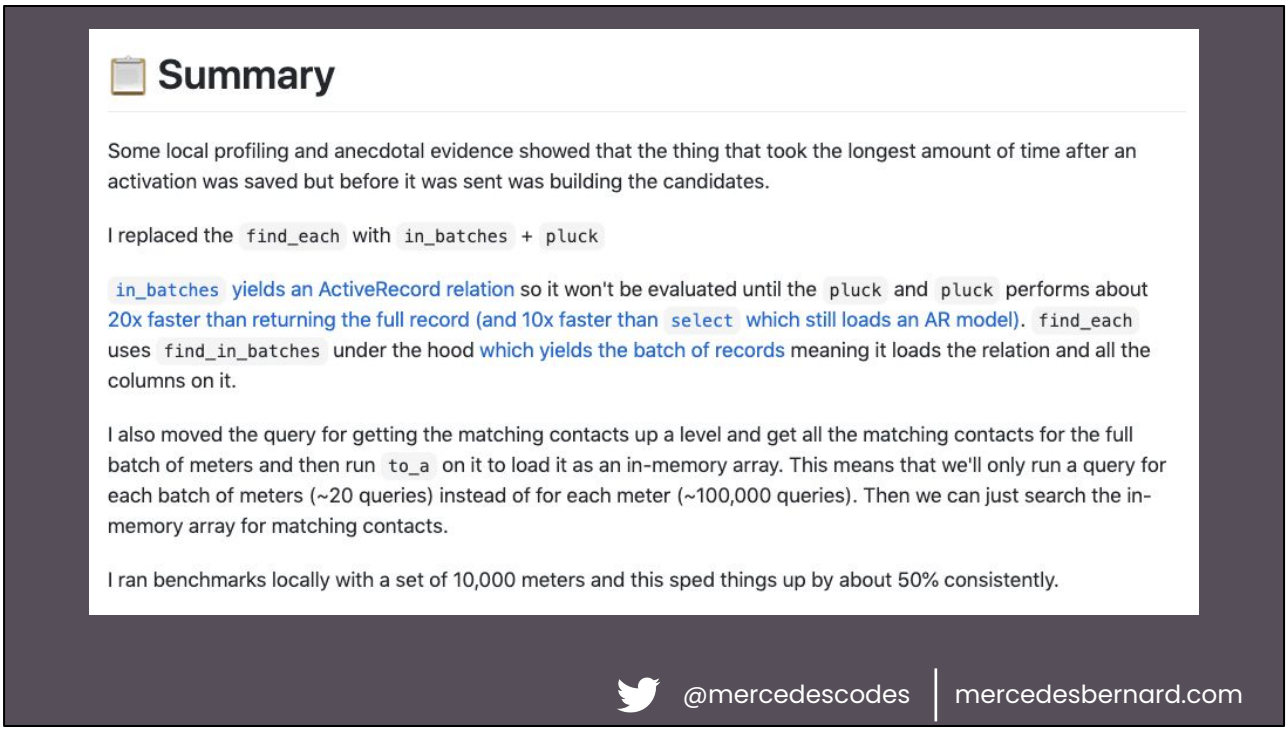


It's important to share positive feedback so you reinforce the learning and growth that's occurred. A small comment about how great this variable is because I could tell exactly what it was from the name or about how helpful the code comments were are great reminders for the reviewee that other people will read and use this code and that naming and documentation are important skills.



Finally, giving feedback in a PR isn't the only way to use code review tools. You can also use them proactively to add comments to code that you've written to guide and coach your team.

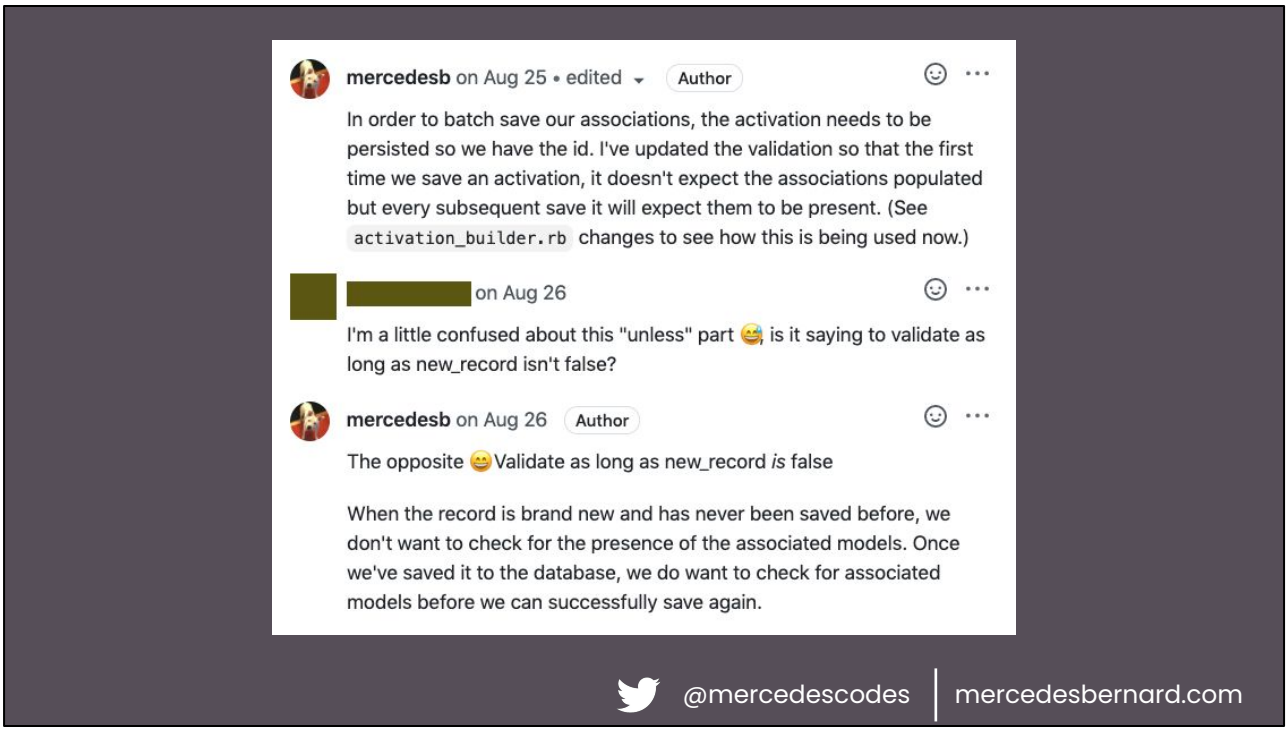
First, make sure your PR description is helpful. Make sure it not only explains the high-level change you made but **why** you made the change. What is the use case or the added benefit that this change will give the user? What bug behavior does this resolve? Provide context so your reviewers can review more than just the lines of code present, but could also think about possible enhancements.



And provide links to helpful documentation. If you used Stack Overflow answers to help you find a solution, link to them. If you found yourself digging deep into documentation to find an answer, link to it. If you know you're using a specific design pattern, explain it and why you chose it, and maybe link to a helpful blog post in case someone wants to learn more about it.

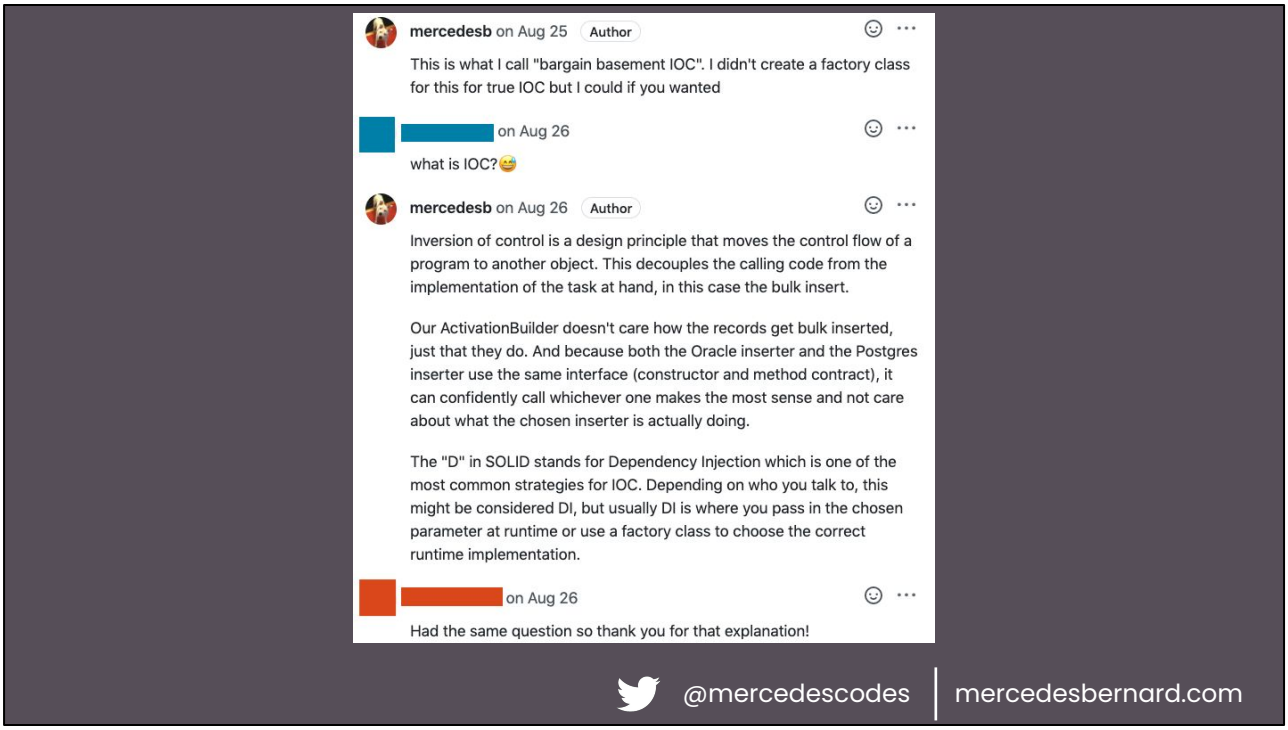
As you do this more and more, closed PRs become treasure troves of knowledge that can be used by the present team to expand their skills or by future team members to gain an understanding of legacy code changes that might not be intuitive upon first glance.

By proactively sharing documentation, you're also showing that everyone has to look things up and it's ok not to have an answer. Success is measured by **finding** answers, not by **having** them. And linking to relevant documentation can help your team become more comfortable reading docs and more skilled at determining what is and isn't helpful documentation which will increase their efficiency.



If your PR is still especially meaty, don't be afraid to go through a code review for yourself first and leave comments on tricky bits of code or configuration explaining what they do.

I recently had a PR that was focused on improving performance for one of our key use cases and it was especially complex because our client's on-prem database is Oracle which is a non-standard choice for Rails. Some of our go-to Active Record strategies weren't available in the Oracle adapter and we couldn't add extra indexes to their database so I had to get creative. I went through and explained my choices in the comments and what each part was doing so that it would be easier for reviewers to follow the diff and understand what they were looking at.



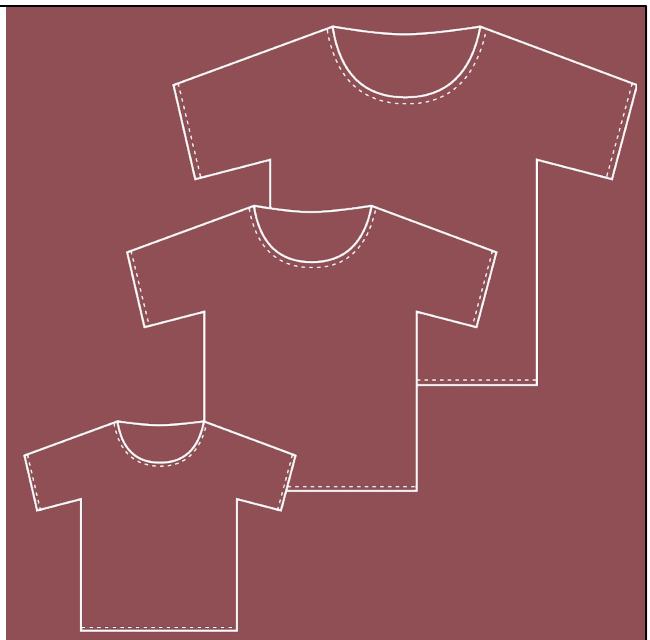
When someone just skims your code and does a “LGTM” (looks good to me) approval, no one benefits. And the more questions you can prompt on your code, the more opportunities you have to make sure that everyone on the team is on board with the implementation and understands the choices made. Everyone wins with more questions so find ways to encourage your team to dig into the code you write so you can share insights with them.

No-code technical tasks

@mercedescodes | mercedesbernard.com

When we aren't writing code, that doesn't mean we're not still engaging in technical work. There is a lot to learn and share from the code-adjacent tasks that we do every day.

Estimating



@mercedescodes | mercedesbernard.com

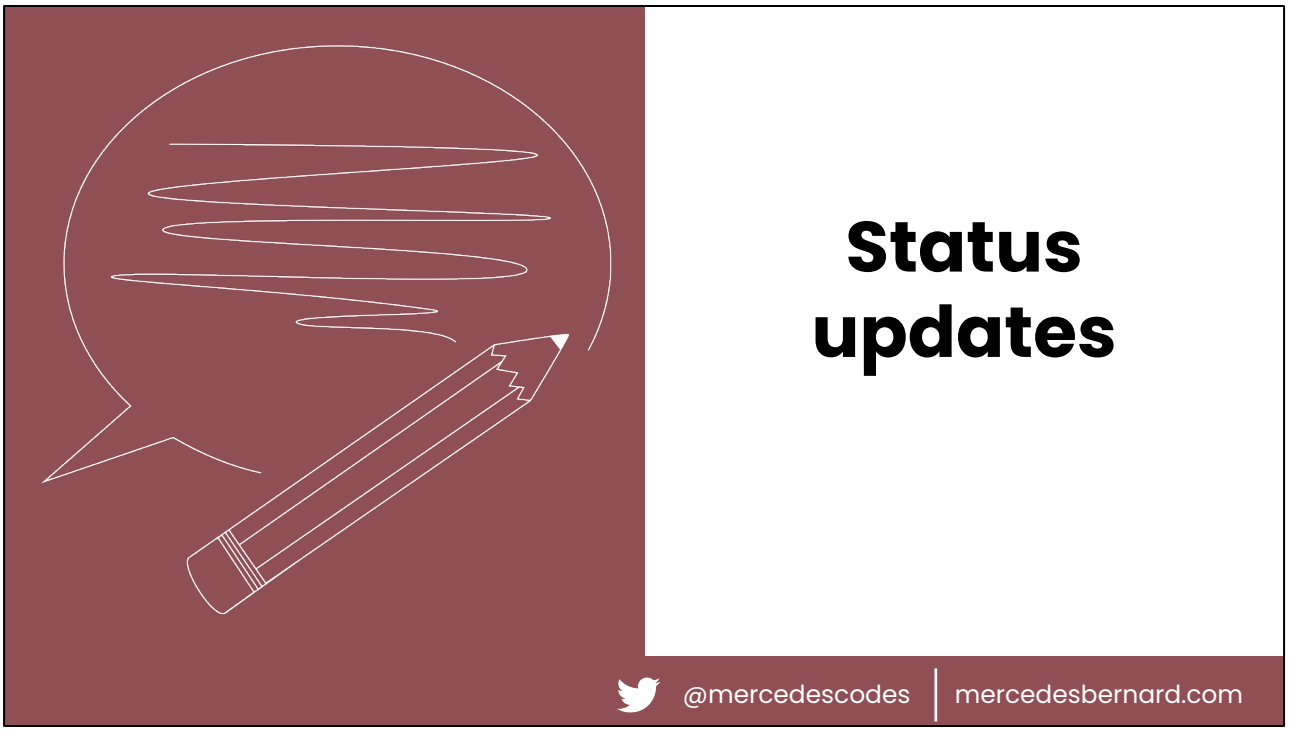
Estimating is a really hard skill to learn since estimates are very subjective and everyone approaches them differently.

During an estimation meeting when someone estimates low and you estimate high, have them explain why they think the task is low effort or complexity. Understand and acknowledge where they're coming from. And then explain why you estimated high. This is a good opportunity to explain non-obvious considerations or remind the team about all the different layers of the application this functionality will touch.

Sometimes when we're early in our career, we tend to oversimplify tasks in our head, forgetting that we need to migrate data or that there is no API for this UI yet or that the stakeholder always waits until the last minute to change their mind so we should start to anticipate that pattern.

Sharing all of your estimation considerations can help your team start to broaden their thinking when they're estimating and consider things that they haven't before.

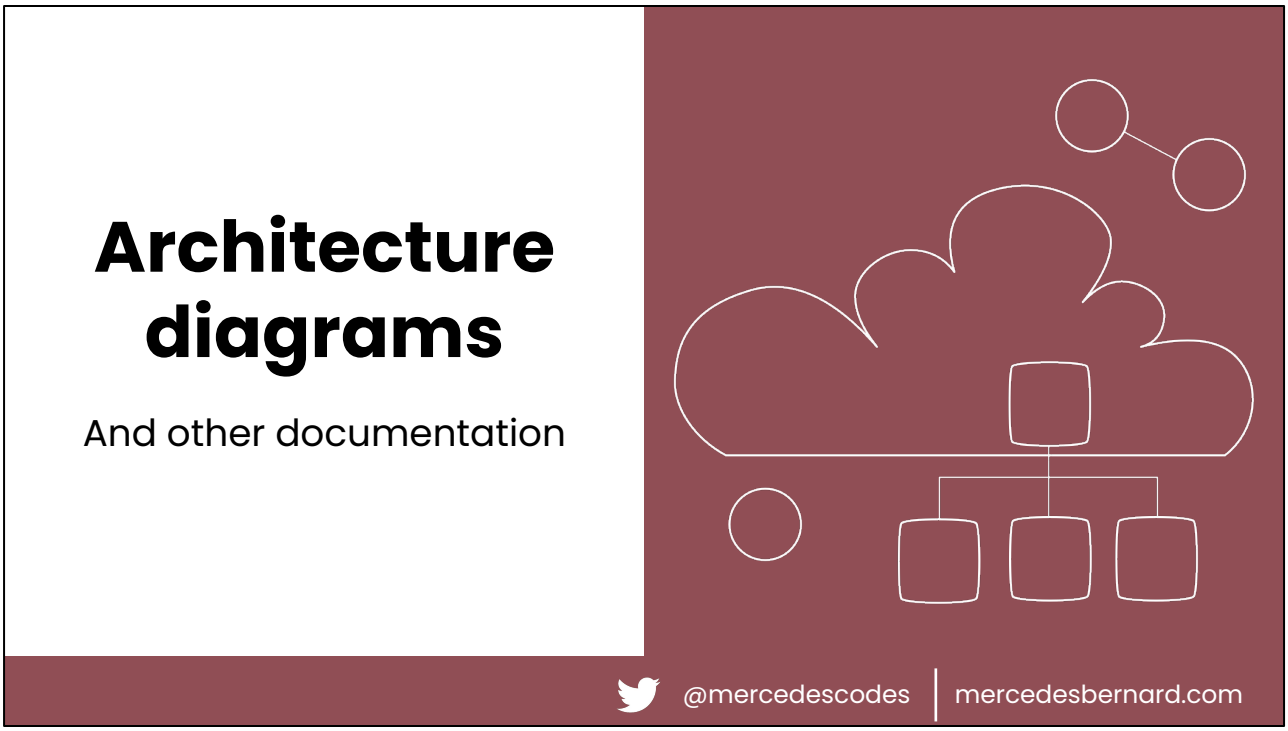
And opening up the estimating process for conversation also makes space for your team to share their expertise with you. Sometimes you'll learn about a part of the code base that they are more familiar with. When they practice pushing back on your assumptions by explaining why your estimate is too high because you're not considering code reuse or other similar patterns already in the code base, they are flexing those muscles in a safe space and growing for when they might need to use them with stakeholders later.



Status updates (in stand up or other meetings) are a great place to share with the team what you've tried and what did or didn't work. Sharing things that didn't work is just as important as sharing what did because it shows how you approach a problem. Learning how to problem solve and learning new problem-solving strategies is key to success in our work.

You can also coach your team on timeboxing by explaining when you knew it was time to change tactics and try something different which will help them learn to manage their own time and lead to a more productive team.

Sharing your status updates creates space for people to express interest in work that they aren't involved in and gives you the opportunity to include them in it. This helps make sure that everyone on the team is getting the chance to develop a variety of skills and also helps build your awareness of each team member's specific interests so you can sponsor them for opportunities that align.



As developers, we hate documentation. I don't know very many people who enjoy taking the time to write it well and it often gets deprioritized in favor of more code. But creating architecture diagrams and functional documentation is a valuable opportunity for pairing. It can help your team see the big picture of the entire project and encourage everybody to remember your users which increases our empathy when we're building it.

Creating documentation helps solidify what they know about the system and creates opportunities to ask questions about the things they don't know. Increased confidence in their knowledge of the system they work in every day increases comfort and safety for them to ask more open-ended questions for why parts of the system were built that way.

And documentation doesn't just come at the end of a project or feature :) If you are creating diagrams at the start of the project to think through system configurations, you have the opportunity to discuss the pros and cons of each approach and help your team understand why we might choose one approach over another.

On my team, we've been talking a lot about the constraints of our current project and when those constraints make it appropriate for us to use a microservice, and when a microservice may increase our maintenance effort without any added benefit.

For example, our current client has a large legacy system for managing their data made of up many many different databases with multiple different database

management systems underneath. And they've been known in the past to change which database they'd like us to query for critical data with little warning. So rather than spend time configuring Rails to handle multiple databases and try to figure out how to manage multiple ActiveRecord adapters, we've chosen to use some selective microservices to manage database connections. Being able to talk through these constraints and how they lead us to the microservice decision has been really productive and led to many additional architecture talks that will help on future projects.

Engaging in conversations about architecture and infrastructure considerations will deepen your team's understanding of those concepts that they will be able to bring with them to their next project.

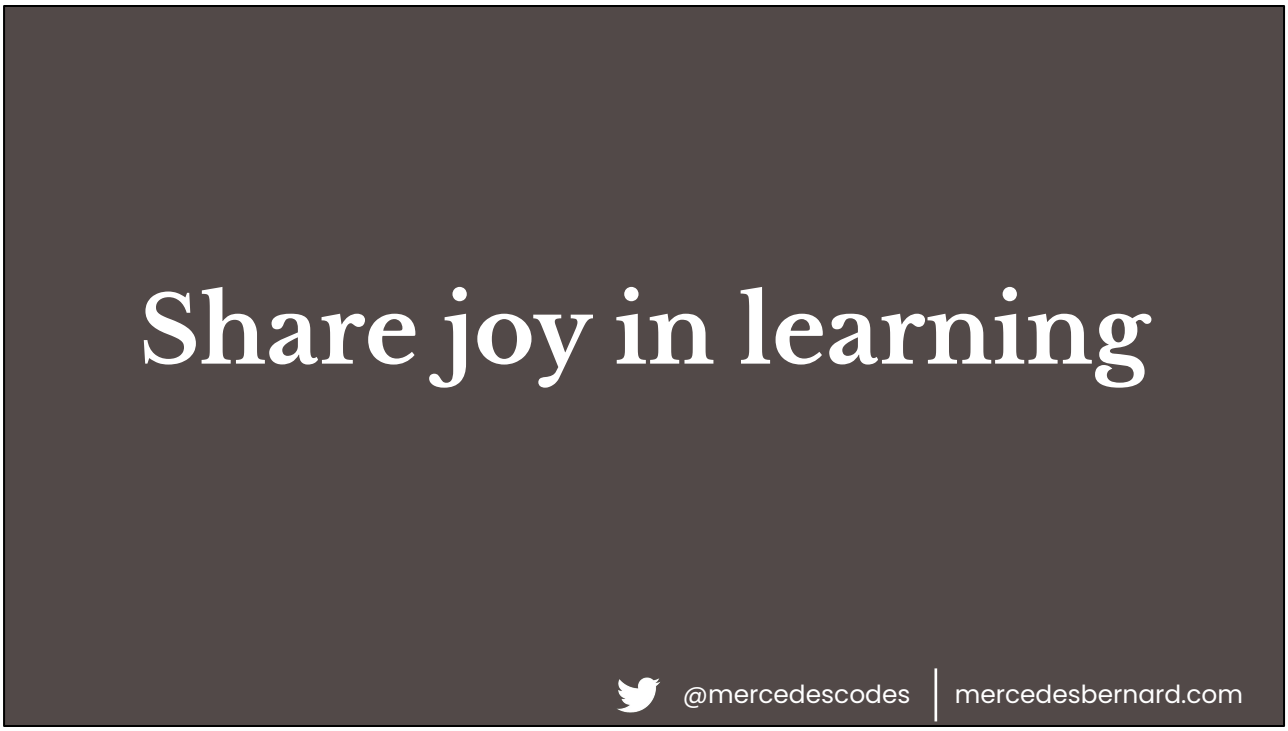
Finally, pair with someone on the team when you're doing research or investigating a spike. I've said it before but it's worth emphasizing: being able to share how you research and make decisions is really beneficial to your team.

We rarely learn these skills in any code school or college program and have to learn by doing. Sharing your tips and what you've learned over the years by researching with them will help them develop these skills faster.

Think about the last time you went Googling for an answer to a tricky technical question. How were you able to glance at the first 3 Stack Overflow questions and know that they weren't helpful to your specific use case? When you look at documentation, how long do you spend digging before you realize that the feature you are looking for either doesn't exist or is undocumented? How do you determine when it's worth cracking open the source code of an open-source library to find the method parameter you're looking for because it was alluded to in their docs but never explained sufficiently? These are all things that come with experience and we can save our team time and provide helpful guidance for them to learn this kind of intuition.

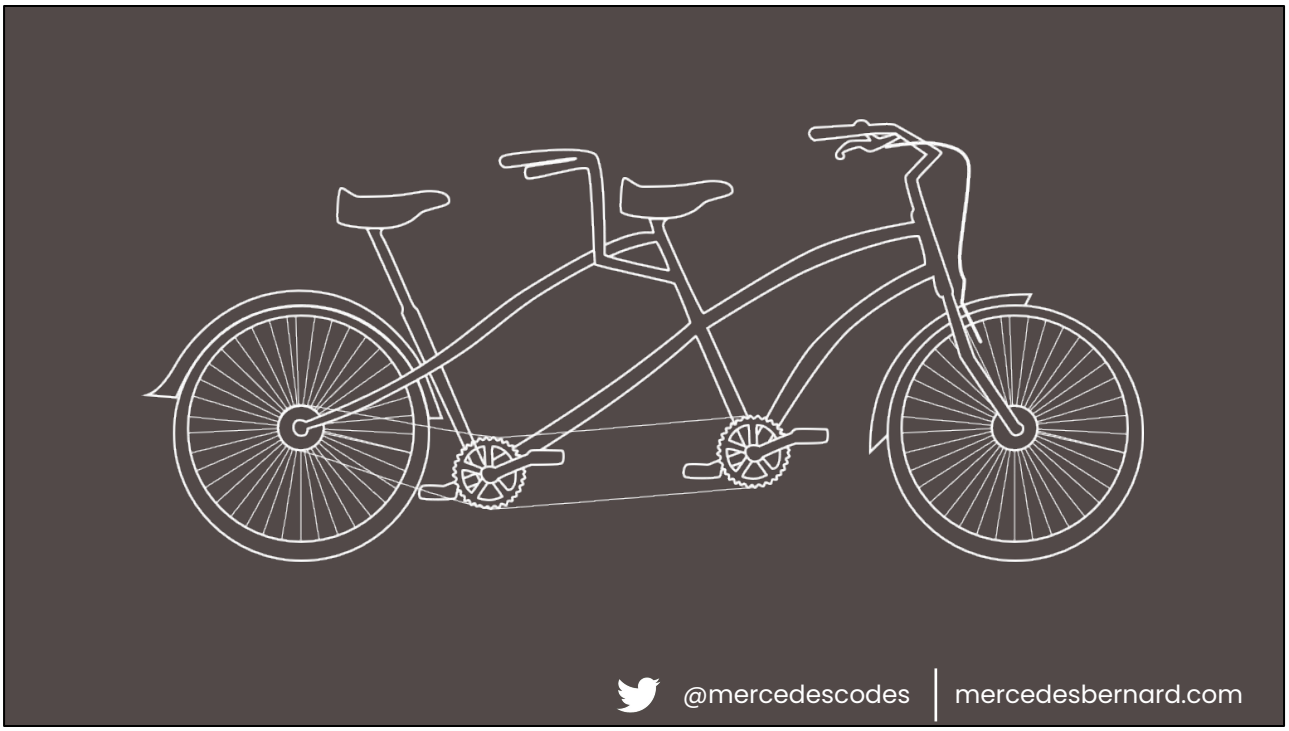
One of my favorite tricks, when I'm stuck trying to figure out how to use a library, is to go to their source code and check out their test suite (hopefully they have one!). Tests can be a quick and easy source of documentation to see a variety of ways to invoke the library and can help you find your use case faster and with less trial and error.

Pairing while researching and looking for questions also helps create a more level environment where you and your pair can be confused and learn together. This is great for trust and relationship building and also boosting resilience so that imposter syndrome doesn't kick in the next time your team member doesn't have an answer. Create a welcome space for people to say "I don't know, let me go find out."



At the end of the day, I think the most important coaching tool is to just share joy in learning with the rest of your team. To be successful in tech, we need to be continuously learning.

No matter how small something you learned is, tell your team about it. Get excited that you hooked up a radio button in a framework you've never used before. Share that feeling of accomplishment when you finally identified that ridiculous edge case in the legacy system you're integrating with. Show off a silly side project you're making. Wow your team with a new hobby you picked up. We don't have to be experts all the time. Our jobs are fun because we get to learn and solve problems all day long, share that with your teams.



Your commute becomes more enjoyable when you share it with someone, so does learning and growing.

We don’t all want to be bus drivers with the responsibility of helping to get our team members to the next stop on their career, but that doesn’t mean we can’t travel with them for a little while and still find opportunities to support their career growth from our own bike seats.

We should think of our technical work like riding a tandem bike and look for people to share a ride with for a little while.


Thank you
#_rc_coaching-through-coding
@mercedescodes
mercedesbernard.com