How to tell what level of developer you are, junior to senior

The developer you are today is a cleverer, more capable version of the developer you were yesterday. The scope of task you can handle today is ever so slightly bigger and more challenging than your peak yesterday. It's one of the most exciting aspects of becoming a software engineer. The career progression opportunities are massive. Likewise, the satisfaction you can get from solving very hard problems goes right up to the edge of as-yet unsolved problems by any human. For example, human eyes had never actually *seen* a black hole until Dr. Bouman plied her craft to create an algorithm to do it. Not a lot of industries have this quality. It's quite special.

A related aspect of working in tech is that your professional level is independent of many factors that would decide your career growth in other industries. For instance, you don't have to be 10-years-senior in your career to be a 'senior' programmer. Likewise, being a developer for 10 years doesn't make you senior by default. You don't have to have diplomas and accreditations to do the work. You just have to have the skills and be able to demonstrate them.

Honestly, it's perhaps the thing about the technology industry that I love the most. Nepotism might make you the foreman on a construction site or the manager of a restaurant, but it won't help you understand how to scale a poorly architected codebase. Tech is *far* from unbiased, but it's significantly meritocratic compared to many other fields.

The fact that great people can progress very quickly can also make it hard for people to know where they are in their journey. Not every team knows how to nurture its rising stars. Not every manager knows how to foster someone who is stuck between two levels. What this post aims to do is give some clarity about the different software engineering levels in terms of skills and performance. About team dynamics and independence. If you're lost in a world of "We're seeking developers with 5+ years of experience," it might offer you a counterpoint that focuses on what you're actually capable of rather than some arbitrary tenure.

***A brief aside about terminology**: In this post and in general, I'm going to use the terms "developer", "programmer", "coder", and "engineer" interchangeably. In all cases, I'm talking about someone who knows how to turn ideas into code that solves real problems. Some people make very fine grained distinctions between these titles. The people I find investing significant energy in these kinds distinctions always happen, by random chance I'm sure, to be on the high side of the chasm they're insisting exists.*

Intro made — let's walk through the broad strokes of the technical career progression. Each of these descriptions is meant to be an entry point into the conversation about the level. You can follow the links in each section below to a more detailed description of the levels.

Junior Developers

The junior developer level is all about learning things in order to immediately apply them to real work. It’s a very exciting level because every day contains both mysteries and answers to those mysteries. It’s all about overcoming the relentless confusion of what these tools are and how they work. What this “internet” thing is. How this server works. How that CI pipeline works. The junior developer is constantly discovering connections between things that blows her mind.

As a junior developer, she’s living in a world where some people around her definitely know the answers to her questions all of the time (or at least they ought to…). The junior developer is often making a tradeoff between how much confusion she’s feeling and how much she’s costing the more senior people around her by bothering them for explanations. As you might guess, this means that her experience of being a junior is in many ways dictated by how supportive her team is toward her. Good, nurturing tech leads and patient mid-levels can make the junior’s experience really exciting and challenging. The junior engineer is likely looking to these folks daily or roughly every other day for help.

Exiting the junior developer level means finding some foundation within the chaos. It means learning enough of the practical art of software engineering so that, mostly, she knows what to do when she sees a problem to solve. When the junior coder starts to know instinctively how to solve most of her day-to-day work without diving headfirst into documentation and Stack Overflow, she’s probably trending toward mid-level. By contrast, if she’s still mucking about in MDN, W3Schools, and copy-pasting stuff to see what works, she’s certainly still a junior developer.

For a much deeper dive about the junior developer level here, check out this expansion on the topic.

Mid-Level Developers

Where the junior developer level is all about learning, the mid-level is all about learning how to harness all those new skills efficiently. The junior dev learnt to write good code fast. The mid-level dev now has to learn *what* code to write. *What* problems to solve. The mid-level developer is a bit like a kid in a candy store. Suddenly, she has all these new problem solving abilities. Suddenly that random error isn’t so random — it’s something that can be debugged. That mysterious issue with Git that required her to dump all her changes and reclone the repository isn’t so mysterious now. She can fix her repo and get back to work. Her productivity is way up compared to her junior-level self. Most of the time she’s just motoring through work, occasionally hitting some place where she’s not sure what to do. She might need support from a senior engineer every few days, maybe once per week.

The mid-level engineer is still in search of answers, much like the junior. However, the mid-level dev is looking for “why” answers where the junior is still working on “how” questions. The mid-level wonders why the backend uses a particular object schema within the REST API. She wonders why the code for this key piece of infrastructure is so old and broken down, yet no one has fixed it. She wonders how she could refactor this UI library to be more elegant, since the current implementation is hacky and brittle.

It's exactly this newfound ability that makes the mid-level challenging. The mid-level dev can now handle abstractions and complexities that were inaccessible before. Now she has the higher-order problem of needing to decide *what* work to do. At the mid-level, it's not uncommon at all to see engineers diving deeply into some interesting problem without thorough consideration about whether or not that work is actually worth doing. The discernment between "Ooh, this is fascinating!" and "Ooh, this is fascinating and let's never do it," is a key example of the difference between mid-level and senior. Bridging the gap between mid-level and senior is largely about sharpening that judgment. It's also about learning how to find answers to key questions. Where the mid-level engineer often comes with tough questions that demand answering, the senior engineer will often already know the answers. And when she doesn't, she rolls up her sleeves and finds them.

Update: It was pointed out to me by a very successful engineering director that my writeup carries forward a certain bias of tech companies. That is, we push people hard to attain technical seniority and then spring on them the challenge that they'll also need "people skills" (whatever those wacky things are!) to become senior. I think this criticism is absolutely fair and the bias is absolutely present. As a mid-level developer, you may be under-developing your tech lead skills. You might find that your lack of such skills are blocking you from proceeding to seniority, even if no one is telling you so.

For more on what it means to be a mid-level developer, check out this expanded discussion.

Senior Developers

Once an engineer has acquired significant technical abilities, mastery of her tools, and enough people skills to facilitate a small team of engineers, she's likely crossed into the domain of seniority. Seniority as an engineer is a very mixed landscape. What's "senior" at one company is not "senior" at another company. If you talk with an advertising agency about what their senior coders are capable of, you'll find they have a different set of expectations than, say, Google. What I'm describing in this post is *technical* seniority. Closer to what FAANG-esque companies would think of as "senior" or "tech lead". My definition of senior would be insufficient in, say, a hybrid project management/software consultancy where client management and presentation skills are prized. However, this definition of senior does cover the most critical quality: *To be senior, you need very good technical intuition and even better judgment.*

It may surprise some readers that many senior engineers are no more skilled in terms of raw coding ability than their mid-level counterparts. It's true though. Once she's made it to the upper regions of the mid-level (on a decent team, with code review), she's probably about as good at writing code as she'll ever be. As a senior engineer, she can continue honing her coding skills. Especially with respect to managing higher complexity code and higher complexity codebases. There are many way to deepen her skills in terms of raw coding ability. It boils down to what sort of senior engineer she is. In the expansion post, we'll talk about different types of senior engineer and what makes them who they are.

The senior engineer learns a new dimension of skills. She's faced with hard constraints, like timeframes for delivering work that don't allow everything to happen. She has to murder some darlings. She's faced with a panoply of things she and her team *could* do. She has to decide what work actually *needs* to get done. She satisfices rather than indulging. She chooses pragmatic solutions rather than elegant ones. Most of these skills are learned the hard way. She learns mentorship skills because now she has to mentor. She learns behavioural interviewing skills because now she has to hire.

A mid-level engineer is likely to build elaborate castles in the sand — nifty and whizbang abstractions to solve small, often irrelevant, problems. She obsesses about particular implementation details trying to optimise them away or achieve the unachievable. After a year or two of this, the mid-level engineer wearies of her toils. She looks back at the hundreds of commits. At the brilliant microframework she created which no one is using. At the hours she spent trying to find an elegant solution to a completely intractable problem. "No," she says, "No more."

When an engineer turns toward senior, she grows a specific kind of incisive judgment. She's well past being technically competent enough to understand the topics being discussed. Instead, she's listening for time being wasted. For holes in the collective understanding of a problem. For senior engineers, it's no longer about the naive joy of doing technical things. She has her sights set on delivering real work. She's the pillar of her team not because of the work she does but because of how thoughtfully she spends her energy.

If you'd like to read more about the senior engineering levels, check out this detailed expansion.

Conclusion

I hope this is a helpful discussion of the different engineering levels. I haven't found many other write ups online that discuss these points meaningfully. As an "All 'Rounder" senior engineer turned manager, it means a lot to me to be able to help other people grow in their career. This writeup is an attempt to take away some of the mystery about what separates the juniors from the mid-levels from the senior folks.

If you enjoyed this writeup, I'd love to hear from you. If you hated it, I'm all ears, too. If you have questions or you want more details, please just ask. I'm happy to expand on any topic here.

What is fast?

"Fast" here doesn't mean at 100WPM. It means efficiently. Some of that is wall-time. For instance, the first time you ever have to write code to interact with a binary tree of data, you're super slow at it. You're having to really use your brain to think about the data structure. About how to manipulate it. You're slowed down by the cognitive overhead of learning that data structure in the first place. Once you're familiar with it, you're able to move way faster. You've already done that hard brain work and now you can focus your brain time on other things.

The second way you're slow when you start out is that you don't know the APIs. How do you look at the top entry in a stack? Oh, right, there's a "peek" method. That's different from an Array. The first time you come across something like Promises in JavaScript, it's a huge amount of work to make sense of what all the `then` and `finally` stuff is all about. The lack of familiarity makes you move slowly.

As you progress towards senior, you're not slowed down by this. At some point, your speed of delivering code is actually wall-time bounded by how fast you can type the code out. When you get to that place, you're probably a "fast" coder. I can't think of a single senior engineer who considers their typing speed to be the limiting factor on their output. Instead, they tend to be slowed down by ambiguities, unexpected outcomes that emerge from complex systems, etc.. Those are more or less fixed costs, no matter how senior you are. However, the senior engineer doesn't lose any time to trying to figure out why their for-loop isn't working, only to find out they had their `>=` flipped the wrong way. Junior engineers lose time to these sorts of trivial bugs all the time.

Likewise, senior engineer code tends to go through code review very quickly. Not because they're blessed and privileged and therefore get to drive in the fast lane. Instead, they produce code with fewer defects and anti-patterns. There's less to criticise in their code, which means the conversation code review can be focused mostly on the content of changes, not the trivial details like "You don't need this `if` statement" or "There's an uncaught exception here when the input is null." This comes down to a base line of professional coding ability that every senior engineer will have. Whole books are written about it. Things like Code Complete are fantastic resources for being able to write your code in a clean, easy-to-maintain way that makes it easier to review.

So, in summary, “writing good code fast” means having most of the problem already in your head when you start to type. It means having enough familiarity with your language of choice and with professional coding standards to produce code that works, is clear, and easy to reason about. When you get to this point, the time you spend coding is mostly figuring out what work to do and how to approach it the right way. That work is the same whether you’re junior or super senior.

Are you a junior developer?

The key hallmark of juniority is the quest for clarity. To use the language of my homeland, junior developers don’t know their asses from a hole in the ground. Everything is a mystery. Why your Git repository has been decapitated is a mystery. Why your code, which you carefully copied and pasted from several highly voted Stack Overflow answers, still isn’t working is a mystery. Why your code, after random permutations, has begun working — just as mysterious. The junior developer arrives starry-eyed in a world of chaos. What she lacks most of all, usually, is a sense of her place in it all. What does “good” look like? Who should be a role model? Is my workload appropriate for my abilities? Is my tech lead giving me good code review and useful feedback?

This level is fraught with so many pitfalls, it’s honestly shocking we don’t churn more coders out of the industry entirely. (Maybe we do and we just don’t know it? Hard to measure the absence of something… .) Anyway, it’s very easy to learn the wrong things. Or follow the wrong role models. For example, I was recently in a conversation with a junior coder whose tech lead was encouraging her to become a scrum master because it would give her opportunities for “leadership.” This junior developer is still learning the basics of programming itself. How could learning “leadership” be the most important thing?

On top of that, thinking of the scrum master role as “leadership” is laughable. In my career, I’ve interviewed at least a hundred people with significant “scrum master” experience. What they all had in common is that they sucked at actually writing code. In the case of this junior developer, it showed me that her tech lead doesn't actually know what it takes to be technically senior, not really. But how could this junior developer know that she was being misled? Super dicey situation.

The thing junior developers *actually* need most is lots of coding tasks — each sized just a smidgeon above their ability level — and high quality code review. **If you want to transition out of being a junior developer quickly, the One True Way™ is learning to write good code and to write it fast.** People management is not The One True Way™. Project management is damn sure not The One True Way™. If you're early in your career, your path upward is purely about hard skills.

The reason for this is that as you progress up the ladder of engineering, you need real problem solving skills. The problems only get harder, more ambiguous, and more complex. If you derail into things like *process* before you acquire significant skill at the core craft of programming, you're putting a glass ceiling on your own career trajectory. How can you become a good tech lead and mentor if you haven't learned your own craft deeply? How can you evaluate candidates you need to hire into *your* team if you have only a shaky understanding of the fundamentals? How you can break down big, complex, company-level problems into coherent parts if you don't really understand your core tech stack and tools? The short answer is: you can't.

Oh, you can cheat. You can learn… other… skills. You can learn to be a scrum master. There's loads of jobs out there for "senior" people with erstwhile coding skills who spend their lives chasing the actual developers about Jira ticket "t-shirt sizes" and sending invites to retrospectives. Hey, if that's your jam, by all means, be my guest. But the hard truth is that every minute you spend mastering the *process* of software development before you've achieved significant skill in the *craft* of software development is a minute you've lost. Just be aware that your technical problem solving abilities stopped growing as soon as you put on your ceremonial scrum master fez and stoll (or whatever they wear).

Another common pitfall of the junior developer is over reliance on tools. I often find when I interact with junior folks that they're using VS Code with AI-assisted code completion turned on. That they're using Git UI tools like GitKraken or similar. On the surface, these tools seem to help out the junior developer by giving her something to start with. She begins to have little internal conversations like, "Hm… what were the arguments of `Array.slice` again? Oh wait… was it actually `Array.splice`? Ah, well, VS Code will tell me." Or maybe something like, "Did I have a branch for this work already created? Am I out of sync with `origin/master` ? Ah, my git tool will tell me."

The problem is that the developer begins to rely on the tools to be able to produce any work at all. Take away the tool and suddenly the developer is lost. She doesn't remember how to do a `for-loop` from `i` back down to `0`. She just hits `Tab` when VS Code suggests the right one. What this means practically is that when the junior developer has to dive into a new codebase, she doesn't actually know how to read the code thoroughly. Or when she's being tasked with code review from someone doing something a bit tricky, she can't really make sense of the code. Because she wasn't practicing the craft of software engineering all along. Instead, she was practicing the craft of picking the right answer from a list of suggestions.

Many junior engineers struggle in interviews and have much more limited impact on their teams because they're only able to produce code when their tool nudges them in the right direction. Moreover, sometimes that nudge is actually wrong, but the developer doesn't catch it because she's not reading the code carefully. She's hitting `Tab` when it feels like the right code. **Strong advice here: If you want to level up out of juniority, do not use auto-complete IDE integrations.**

The junior developer starts to come out of the wilderness of juniority when she starts to have a clear picture in her mind about what code to write as soon as she hears a task description. When she begins catching herself making the same old mistakes in her code *before* sending it out for code review. When her questions transition from "what is this mysterious `async` keyword?" to "I wonder if there's a way to parallelise all these calls in a single `await` statement…?" When she starts to understand what task she'll take on next based on the task she just finished.

In other words, the junior developer's life is marked by the constant state of taking in new information. Hard facts. Pointers about code structure. Little gems of wisdom and command line wizardry. A junior developer's life is a daily influx of new knowledge balanced with a daily outflow of new-to-her code that solves very specific problems.

The sort of tasks I'd expect a junior to be able to handle

- Fix a data-layer bug causing an occasional error on this screen of the app

- Extend an existing feature by adding a new piece of UI and querying the database to populate it
- Refactor these files of older-style syntax into our new, modern coding style

The junior's tasks are atomic (write some specific code to solve some specific problem). They're clear and well-scoped. They're focused on specific outcomes that are already known to someone more senior than them. This is important, because junior folks get lost. They get confused. They make messes of things. Because of this, they need tasks that already have clear paths to success. Their day-to-day conversations with more senior team members ought to be things like, "I'm trying to finish <some specific task> but I'm getting a weird error that I don't know what to do with." Which is typically followed by a senior team member looking at the error, thinking of how it must have come to exist, and suggesting remediation steps. In the best case, offering the junior a mini lesson in whatever just went wrong.

The sort of tasks I'd expect a junior *not* to be able to handle

- Figure out if a new feature is possible by looking at the data schema and figuring out query performance constraints
- Work with a product manager and designer to come up with a new application feature
- Select and adopt a new abstraction to replace an old one that is causing the team headaches

The key distinction between tasks that are good for junior developers and tasks that aren't is the scope of ambiguity. Junior folks are in deep, deep ambiguity just learning their craft. They're working hard everyday to learn the nuances of the programming languages they're working in. Git is an abyss. Command line tools are a quagmire. Code review is a constant game of whack-a-mole where the junior coder's code is the mole and senior developers' hard-won coding experience is the mallet.

In other words, a task that is a stretch to a junior developer is one in which they need to learn something new just to complete it. For a mid-level or senior developer that same task would just be a matter of time, not learning, because they've already hit those milestones.

On the Cusp of Mid-Level

As junior developers transition toward mid-level, several things change. First and foremost, their pace of work goes up. Way up in some cases. Where true junior devs spend most of their brain time just figuring out how to use their tools to solve specific problems, mid-level engineers have some real skill with those same tools.

One key consequence is that they get a bunch of brainpower back. They can now spend more of their mental energy thinking about how to solve higher-order problems. They're encountering completely new things less frequently. They have learned lots and lots of coding patterns and key lessons. It's a bit like gaining "muscle memory", but for your brain. This translates directly into speed. Mid-level engineers can typically take any junior developer’s task and rip through it in a tiny fraction of the time that it would take a true junior developer. When you start knowing everything required to complete a task, your speed of work becomes bounded only by the time it takes you to understand the task, type the code, test that it works, and get it committed. You’re not mired in seemingly endless documentation or asking for help from senior people as often.

Importantly, junior developers trending toward mid-level will start to understand not just the task they’re working on, but the shape of the whole project they’re working on. The level of ambiguity they’re taking on shifts to bigger, more important things like how to solve more of the problem with less work.

Are you a mid-level developer?

Graduating from juniority into… uh… mid-level-ity? means reaching some key mile markers. Mid-levels have solid core programming knowledge. Their overall problem solving ability has progressed a lot. They generally have a sense of how to solve a problem when they hear it described. Mid-level folks also have a fairly solid knowledge of the tools of the craft. You can distinguish the junior from the mid-level largely by this amount of confidence she has in her abilities. The mid-level dev knows she can produce good code fast. She's not stumbling over Git rebase errors for hours. She's not confused about the difference between Array.map and Array.reduce any more. She mostly just cracks on with the work. Consequently, the mid-level developer is beginning to feel hunger for more interesting challenges.

Where the junior developer's pitfalls are largely problems of ignorance, the mid-level engineer's pitfalls are largely ones of knowledge. A junior developer might lose a week of her time because she read a Stack Overflow answer that said some framework will solve what she thought her problem was. Days later, she finally realises that her initial premise about the problem was wrong and that the framework doesn't actually do what she wanted it to do at all. A mid-level developer is much more likely to understand the immediate problem correctly and then try to solve the *class* of problem with a clever abstraction. All that hard-earned knowledge and ability comes in handy as the mid-level developer starts exploring just how fancy of a solution she can come up with for the problem she's trying to solve.

The pitfall here is priority. Yes, the mid-level developer often has the curiosity and ability to spend hours and hours solving some problem to a very high degree of technical correctness, extensibility, and general panache. However, the senior person usually has the visibility into the overall problem space to decide not to solve this problem at all. Where the mid-level person spends days crafting a clever solution, often a senior person will come along and realise that they can avoid doing any of that work with a small hack or workaround. The senior person isn't trapped by their curiosity the way the mid-level person is.

Overall, in any technical team, the majority of the code in the codebase will tend to be written by mid-level people. The mid-level engineering level is special because it features a combination of high coding ability and high enthusiasm for new problems. The relationship between senior developers and mid-level developers tends to be that the senior person lines up some work for the mid-level person to tackle. Their conversations tend to be about how the big pieces fit together. About the "why" of the work more than the "how" of the work.

There's something special about mid-level engineers. That is — many engineers never make it beyond mid-level. At mid-level, you start to see human limitations coming into play a bit. The problems mid level engineers face are sufficiently harder than what juniors face such that some mid-level engineers will be at their peak. The level of complexity goes up significantly when you transition from "how do I do this work?" to "how does this work fit into the bigger picture?" If you didn't learn your tech well enough (in the junior sense described previously), you’ll struggle to progress beyond mid-level.

At mid-level, many engineers divert into alternative paths. Some drift toward people management, if they’re at a company that permits managing people before becoming technically senior. Some move into project management. Some move into product management. Some drift toward design-forward frontend roles that don’t require greater technical depth.

For the developers who stay on the One True Way™, the duration of their time at mid-level will be spent building things. Mid-level developers have interesting workloads in that many of their tasks will be identical to the sorts of tasks that junior folks would have to take one. The difference is that the mid-level coder handles such tasks without needing any support. Mid-level developers work also shares a lot of common ground with senior team members, except that the senior engineer knows the plan and is driving the work in the right direction, where the mid-level developer is following that lead and trying to make sense of how the work all fits together.

Mid-level engineers usually receive support from senior folks that looks something like: “I have a question about the API we need to offer for our data ingestion endpoint. Why are we asking for a JSON object instead of just accepting the four arguments we know we need?” To which a senior engineer might have a response like, “That JSON object exactly matches the output of another piece of the system. If we accept it as-is, we don’t have unpack and repack it along the way. Also, if that object’s schema changes, we don’t have to maintain this code because they’ll be using a shared definition.”

The key point in this hypothetical design discussion is that the mid-level knows *what* code to write. At this point in her career, she's wondering about *why* to write the code. She's thinking ahead trying to understand how the pieces fit together and how to build the whole system in a good way. Likewise, she doesn't yet have these answers. She's working with a senior engineer who is already thinking through those complexities. In the ideal case, the senior engineer has the time and interest to explain things and share context until the mid-level developer is operating with as much clarity as possible. Good senior engineers help smart mid-level folks move forward very quickly.

The sort of tasks I'd expect a mid-level coder to be able to handle

- Create an adapter layer that helps this piece of code communicate with that API
- Work with the designer to implement this complex UI feature, keeping an eye out for design choices that will be hard to implement
- Walk through the whole codebase adding request logging to every endpoint and API

These mid-level tasks have significant ambiguity but they're also scoped within a project. They're not solving key problems themselves, instead they're a piece of a larger thrust of work. This is a key difference between mid-level and senior engineer tasks. Senior folks will be accountable for outcomes, where mid-level and junior folks will be accountable for subparts of those outcomes. As the mid-level dev progresses, she will move closer and closer to delivering work that solves a whole problem.

The sort of tasks I'd expect a mid-level dev *not* to be able to handle

- Choose a technology and design an API with it that satisfies a number of business constraints; implement that API efficiently, migrating existing data as necessary to achieve a clean implementation
- Debug and fix a data inconsistency issue that seems to span several systems, each maintained by different teams
- Lead the development of mobile apps based on existing desktop interfaces, choosing a tech stack that maximises code re-use

Where the mid-level developer falls down is where the ambiguity is too great for them to manage it. This might mean organisational or business ambiguity. Or it might mean technical ambiguity of a sort that exceeds what that mid-level engineer is ready to handle. The difference between a mid-level and a senior developer is entirely ambiguity management skills. It's a difference in terms of practical ability In other words, some technical details that are ambiguous to a mid-level would be obvious to a senior developer. This difference also shows in ambiguity management. That is to say where the mid-level engineer gets overwhelmed by the ambiguity and can't make progress, the senior developer will stay cool and find their way through.

On the Cusp of Senior

As a mid-level developer is heading for seniority, some key behaviours start to change. The most noticeable one will be the type of questions they ask.

Junior engineers will mostly ask “how” questions. “How do I use this library?” “How do I debug the error message?”

Mid-level engineers will tend to ask “why” questions. “Why should I implement this code this way?” “Why do these parts of the system fit together this way?” “Why aren’t we doing that other work before this work?” Mid-level engineers grow by tuning their radar for how decisions are made. For priorities. For tradeoffs.

Tradeoffs and priorities are the senior engineer’s life. Their questions tend to be “What happens if…” and “How could I know if…” questions. “What happens if we spend a week solving that caching problem? Will we be able to deliver the whole solution on time?” “How can I know if this data consistency problem is actually upstream of the system I’m currently looking at?” Where mid-level engineers are often asking for answers, senior engineers are usually looking for means to answers. They’re trying to work ahead of the current situation, figuring out not just what comes next *but at what cost* relative to everything else that could come next.

Mid-level engineers have some hard-won technical abilities. They earned them along the way out of juniority. They’re raring to dive into complex problems and build elegant solutions. One of the milestones that mid-level engineers have to reach before they can turn senior is seeing exciting technical opportunities and carefully deciding what work *not* to do. Where the mid-level engineer might fascinate for hours or days (or, in rare cases, months) on a difficult problem, a senior developer will pause to consider the costs to the team, the project, the codebase. Mid-level engineers usually don’t have enough context to make these kinds of tradeoffs. As they begin to sharpen their senses for these kinds of things, they’ll start to notice other engineers spending time on things they shouldn’t be. They’ll start to see places in their own work where they wasted time solving the wrong problems.

Update: A super common mid-level engineer trap is lack of people skills. Many mid-levels will struggle to transition smoothly into senior status because of underdeveloped interpersonal skills. Maybe she gives super harsh feedback that is technically correct, just soul crushingly blunt. Maybe she avoids key meetings because she doesn’t like to waste time talking and prefers only to write code and solve problems. Maybe some aspect of her personality is holding her back (e.g. excessive negativity, lack of time management, excessive egocentrism). I’ve seen mid-levels struggle with each of these.

It’s extremely common for companies to fail at mentoring these people. It was pointed out to me by a very successful engineering director that my posts here carry the same bias the industry has — we spring the expectation of people skills onto engineers without ever calling them out explicitly. We just assume that once you’re super technical, you’ll sprout wings and learn to fly, too. It’s not fair, but it’s super common. The mid-level engineer trending toward senior needs these skills. She needs to be able to *give critical feedback with compassion. She needs to be able to effectively run meetings and execute planning. She needs to be able to coordinate people. Technical skills are the key skills of juniority. Across her time at mid-level, the developer has to continue to level up her technical skills while also starting to acquire the leadership skills she'll need to succeed as the head of a team.*

At some point, she'll have all the technical chops she needs to solve most problems she comes across. She'll have the ability to discover answers where she doesn't have them yet. She'll be able to think critically about the tradeoffs between different approaches to solving a given problem. She'll be able to think about how to support junior and other mid-level engineers as they progress through the subparts of the work. She'll be able to deliver technical solutions leveraging all of the above. With a couple successes at doing this, she'll have proven her track record enough to say it wasn't just luck. At that point, the engineer will have attained the rarified air of seniority.

Are you a senior developer?

When an engineer transitions from mid-level into senior, their job tends to change. Mid-level engineers can get away with the occasional dalliance into some pet problem. Or some period of low productivity because they're lost and confused and need some guidance. With seniority comes higher expectations. The lowest expectations of senior developers are that they can run a project and lead a small team to bring the work to life. They're stable and consistent. They're rarely stumped by technical problems and, even when they are stumped, they have the tools to get themselves unstumped. They don't lose their composure or resolve just because they don't know the answer and it's not revealing itself without a struggle.

Human limitations play a significant part in who becomes a senior developer. Even more so in deciding who becomes very senior. A hard truth is that you have to be pretty clever to make the cut as a senior developer. You have to be able to handle high amounts of complexity and ambiguity. These take the form of technical implementation details, team coordination, and tradeoffs of all sorts. Many people don't get excited by that kind of work and even some who are excited by it simply can't do it. Looking up from the bottom of the senior rungs of engineering ladder, the number of people in such roles gets smaller and smaller with every step. Maybe 1/3 of everyone who ever starts out as a junior will ever make it to senior. The falloff as you look towards “staff engineer” or “principal engineer” is much, much greater. At a company like Facebook which has thousands and thousands of engineers, fewer than 10% make up the top brackets of their engineer ladder.

That said, the day-to-day reality of being a senior developer is mostly about good judgment. In fact, good judgment is at least as important as technical abilities in the senior level. You often find senior engineers who are no more technically skilled than strong mid-level engineers. The difference between them will be that the senior engineer uses good judgment in choosing exactly the right problems to solve. She keeps the team motivated and focused. She delivers. Her mid-level counterpart might lack literally every one of those qualities, despite being just as good as a coder.

Every senior engineer learns the skill of giving the people around her some room to struggle, but always with guard rails. She learns the art of deciding when to intervene and when to let people toil.

When her mid-level teammate says, “I’m going to dive into solving <this interesting problem that I’m excited about>!” Her brain does the computation: “When we spend a month on that problem, we still won’t have solved this other blocking problem… but the abstraction you’re proposing could really pay down some technical debt… Hm… Okay. Worth it.” She probably does and says nothing. Instead, she watches the magic unfold.

When she hears, “Ugh… I hate this stupid application. It makes no sense,” her mind automatically translates it as, “I’m a junior engineer and I’m getting frustrated because this is really hard. I need an encouraging nudge in the right direction.”

When she hears one of the mid-level engineers on her team exclaim, “If we switched to using <some hot new library>, our code would be so much cleaner,” she might feel an ache in her bones that says, “Even if that new library was made of solid gold, it wouldn’t be valuable enough to warrant the weeks of refactoring. And, should you want to know the *real* blockers in our codebase, here’s a prioritised list that’s been keeping me up at night: …”. Here, maybe she nudges that engineer in the direction of something that would really help the team without costing weeks of refactoring.

In each of these scenarios, the skills the senior engineer is applying are: perceptivity, discernment, and leadership. Note that all of them are technical, even the emotional support to the junior. Because the senior engineer is very technical, she can tell the difference between frustrated complaint and legitimate technical grievance. When someone gets a senior title, but not senior technical abilities, they lack this perspective. In those teams, often technical decisions are made that result in lateral movements or even worsening of the existing tech. How do you know if someone’s bold new initiative is worth it or not? Good senior engineers do the brain work to figure it out.

The last point I’ll make about senior engineers is that they don’t make messes. This can be technical messes or interpersonal messes. This is sometimes described as “the first rule is that you figure out the rules.” Some folks will chafe at this. It might seem unfair. It might seem like senior engineers are meant to be psychic. How are you meant to know what is expected of you if no one takes the time to explain it? The counterpoint to this is that there are usually countless ways to deduce the correct expectations. Doing so requires the senior engineer to observe much, much more than she engages. She has to be mentally and emotionally poised enough to think several steps ahead of her actions, anticipating their outcomes. She might feel just as outraged by a management decision as her junior engineer peers. Yet she'll mostly be silent about it, where the junior and mid-level engineers might rage and cause interpersonal chaos.

Her less-senior counterparts are not wrong, they've just misunderstood how real influence is asserted. The senior engineer thinks about how to create the outcome she wants without causing undue drama along the way. This often means she bites her tongue when she'd love to lash out. Among very senior engineers you'll often find extremely strongly held opinions. What you also find is that these engineers think very, very carefully about how to bring those opinions to bear within a team. If the senior engineer causes offence and backlash, she's failed at being effective in her role. She's likely to be just as driving and revolutionary as people who are a hundred times more visible. The levers she chooses to pull are the ones that lead to the necessary changes, often without anyone ever needing to breathe a word about it. Most mid-level and even entry-level senior engineers miss this.

Senior Coder Caricaturisation

A fun detail about senior engineers is that they exhibit a huge diversity in skills. Most junior and mid-level engineers go through the same struggles and very similar growth trajectories. By the time those engineers become senior, they've found their passions. They've also learned what working styles they connect most with. Because of this, there’s a really neat diversification that happens at the senior level.

One of the most important distinctions is that some senior engineers drift toward project leadership and others drift towards deep technical mastery. The sorts of engineers excel at project leadership tend to build stronger interpersonal skills. They tend to acquire more mastery of navigating the org chart of their company. At building consensus. At communicating and promoting their work. Most technology project don’t require deep tech. Most projects just require a vision for the outcome and steady progress toward it. Especially in larger organizations, the obstacles are interpersonal or political. Many senior engineers specialise, really, in navigating these treacherous waters, delivering projects consistently. Sometimes they’re successful *despite* their company.

The other sort of senior engineer tends to drift towards elite levels of pure technical ability. They’re the people who know how their code works all the way down to the machine instructions. They’re the ones giving incisive code review that makes other senior engineers pause to say, “Whoa.” These sorts of senior engineers inspire the people around them, but not because of their powerful presence in meetings or their rallying presentations. Instead, they inspire people because they achieve technical outcomes that shock and delight. These are the sorts of people who led the way on Machine Learning as a discipline. The sort of people who create world-changing libraries like React. Or for that matter who invent programming languages like Rust. Often these engineers operate very near to academia, taking theoretical tech and turning it into applied tech. To this ilk of programmer, there is no such thing as topping out in terms of coding ability or technical mastery.

There exists a tiny subset of senior engineers who do both of these things. They tend to be the sort of people you see on stage at major tech events. They're the leaders of influential open source projects. People like Andrei Alexandrescu and Rob Pike come to mind for this rare intersection.

The distinctions run deeper than these two broad buckets. Listed below are some coarse categories of engineer that show up in the senior echelons. These categories have hazy boundaries. Think of it as a very crude K-modes across engineering personalities in order to surface the clearest subgroups.

The All 'Rounder

The most common form of senior engineer is The All 'Rounder, sometimes called 'full stack generalist'. She's the sort of engineer that can make good progress on building new systems or maintaining old systems alike. She can solve hard bugs across the stack. She can give solid code review feedback. She can break down tasks and delegate to folks junior to her. She's the kind of engineer you'll happily hire as many of as you can because they show up and make good things happen consistently.

A subset of these folks will drift into people management. Especially at startups, it's not uncommon to see a tech lead serving in a managerial role. Likewise, a subset of these engineers will drift towards product and/or project management. They'll have the perspective and people skills to effectively lead a team in delivering a technical outcome like a new product or a new piece of infrastructure.

It’s not uncommon to see All ‘Rounder engineers bouncing between people management and tech lead roles. Between TPM roles and senior engineer roles. Their core skillset includes deep technical knowledge, solid interpersonal/communication skills, and deep pragmatism with respect to getting things done. Because of this, they have a lot of ways of offering value within their company.

The Code Whisperer

This brand of senior developer focuses most or all of her energy on navigating an existing problem space, making things better. She finds bugs and jagged edges that need honing deep within the machine. Her work is almost exclusively technical and largely individual. This type of engineer likes diving deep into the technical weeds, discovering the brilliance and sins of engineers past, like a code archaeologist. When there’s a problem no one else can explain, this person is usually the one to call. When left to their own devices, they discover fixes and optimisations that blow away other engineers. They might produce a quarter of the work of their brethren, but that work’s impact tends to be many, many times more valuable.

The Earth Mover

Some senior engineers love moving heaven and earth. They like taking old broken down systems and turning them into new ones — the hard way.

The love taking a vague problem area and writing all the code to solve it. They want hard problems to solve. They want technical complexity. But most of all, they want to deliver output. They may or may not be good at rallying teams, but they certainly inspire their team by relentlessly plowing through hard tasks. They're good at managing big, complex, multistep refactors. They're probably less technical than "The Code Whisperer" but they're also much, much more productive.

The Domain Expert

Some senior engineers will find their calling by losing themselves in some technology. Maybe it's databases. Maybe it's machine learning. Maybe it's operations. The Domain Expert tends to be really, really good in a narrow area. They often play critical roles on their teams because their knowledge is desperately needed for the success of the whole. Incidentally, hiring a Domain Expert is tricky unless you already have one in the same domain. Successful Domain Experts within a team tend to offer their knowledge constantly to others. Code reviews. Tech talks. They tend to raise the bar around the team within their area of specialty.

What comes after "senior" ?

Levelling up within the "senior" bracket of engineer is largely a function of what type of engineer you are. Each of the coarse categories described above has its own career progression. However, as you go from senior to even more elite statuses like "staff" engineer or "principal" engineer the common factor is that you're consistently offering greater and greater value to the company. Solving more important problems. Enabling other developers to work smarter or faster. Saving the company from big problems when few others (or in some cases when no others) could help.

First degree senior developers tend to lead a small teams of coders. More senior engineers can offer value by tech leading larger projects, rallying larger teams of engineers behind them. Or by gaining deeper technical knowledge allowing them to solve problems other engineers can’t. Or by playing a key role in delivering technical outcomes over and over again. Exploring how to evaluate very senior engineers is a pretty interesting topic that warrants its own detailed write up, so I’ll leave off the topic here.