Chapter 12. Learning to Learn

> Live as if you were to die tomorrow. Learn as if you were to live forever.
>
> Mahatma Gandhi

If you’ve been in the technology space for any length of time, you’re familiar with the reality that technology changes. Constantly. Which means you will be challenged to learn new things regularly. Many engineers believe they are paid to write code, which is true. However, you are also paid to *think*. You earn a living by using your brain, so you have to feed your brains with knowledge, which means learning.

Arguably, the ability to learn new things quickly is the most important skill of a successful engineer. Additionally, you need to not only understand new things but also apply what you’ve learned to real-world scenarios. Many people benefit from the “see one, do one, teach one” approach.

The Shiny New Thing Paradox

Software engineers are attracted to “new” like moths to a flame, and when something first appears, it is often accompanied by a flurry of activity that makes it seem like you’ll be left behind if you don’t immediately embrace it. In the early days though, technologies might not be ready for prime time; the rough edges haven’t been sanded down yet. Over time, the hype fades. But that is often when the new thing is most ready for adoption. Never forget, new technologies are like buses: another one will come along in 15 minutes.

Cramming Doesn’t Work

Odds are at some point in your educational career you crammed for an important test. And it might have succeeded—at least in the short term. But did you retain the information? To truly learn something, you must encode the information: it must be elaborate, meaningful, and have context. Stories are incredibly effective and have been used for millennia to teach and share. Learning often involves repetition and, when spaced out appropriately, increases knowledge and retention.

How do stories apply to you as an engineer? It’s one thing to have someone tell you to be very careful when using commands like `rm`, especially with the `-f` flag. It is something altogether different for someone to share the story about how they inadvertently wiped their laptop hard drive while working on a script that included `rm -f`.[1]

Despite your best efforts, you will forget things, which is actually a feature, not a bug! Your brain is incredibly good at purging information you no longer need. While some bits may never leave,[2] and others vanish far too quickly,[3] your brain is actively “taking out the trash,” as it were. Information decay is predictable, but it isn’t the same for everyone or every fact. Computers can help by repeating questions at proven intervals (which, along with some pretty slick gamification, is how some people succeed at learning new languages). Some of you have likely availed yourself of the low-tech but very effective technique of leveraging flash cards!

If you find yourself struggling with something, it is often better to put it aside for a bit rather than continuing to beat your head against the wall. The simple act of taking a shower, going for a walk, or making dinner can distract you enough to unleash the answer. Your brain has two distinct processing modes: R-mode and L-mode. *R-mode* is the nonverbal, search-and-retrieve aspect of your brain. It runs in the background and isn’t directly controllable, which leads to some unpredictable results. When you’re watching a movie and you can’t quite remember that actor’s name—and then five minutes later it just pops into your head—that’s the R-mode at work.

The *L-mode*, by contrast, is verbal, analytical, and linear and is focused on logic and computation. By distracting the L-mode, you can free up the R-mode to do what it does best. Think about the problem and then do something routine like taking a shower or cleaning up the house.

As an engineer, you can use these two processing modes to your advantage. If you find yourself stuck on an issue, pushing harder might not be the most effective solution. Don’t be afraid to put the issue aside for a bit, distract the L-mode by doing a more mundane task (or take a walk), and see what happens. More often than not, a solution will spring to mind later.

Skills Acquisition

> The best investment you can make is in yourself. The more you learn, the more you earn.
>
> Warren Buffett, American investor and philanthropist

There are several models of skills acquisition that all have similar concepts. One example that comes out of the martial arts is *Shu Ha Ri*, or as Bruce Lee once said, “Learn the principle, abide by the principle, and dissolve the principle.” Another way of thinking about this comes from Clark Terry: Imitate, Assimilate, Innovate. In the Shu or Imitate phase, students follow the instructor exactly. The focus is on duplicating the teacher without much concern for the why or the underlying theory.

In the Ha stage, you start to branch out. You understand the basics and have enough muscle memory to repeat the core movements. Now is when you begin to fill in the theory and start to seek out other teachers or sources of information, which you can then assimilate into your understanding of the topic. At the Ri stage, you are learning from yourself, essentially. Your practice drives the learning instead of following the path laid out by others.

The *Dreyfus model* expands on this concept by dividing learning into five distinct stages (outlined in Figure 12-1) that work in a progression. In the Novice stage, learners require recipes: do this *exact* thing. If you’re learning a new sport, this is the stage when your coach might move your foot a few degrees or have you repeat a move seemingly endlessly. At the Advanced Beginner stage, you start to move beyond the rigid rules; you start experimenting to see what happens when you change your grip or modify a stance.

At the Competent stage, you can start to troubleshoot. You may recognize what your mistake was by watching the ball flight. If you make it to the Proficient stage, you can self-correct. You can see or feel the mistake and fix it without needing a coach to point it out. If you work at something long enough, you'll enter the Expert stage, where it all happens by intuition. As a golfer, you just think, "I need this ball to cut" and…it cuts! In fact, at this stage, you'd likely struggle to explain what or how you accomplished something; it just seemed right.

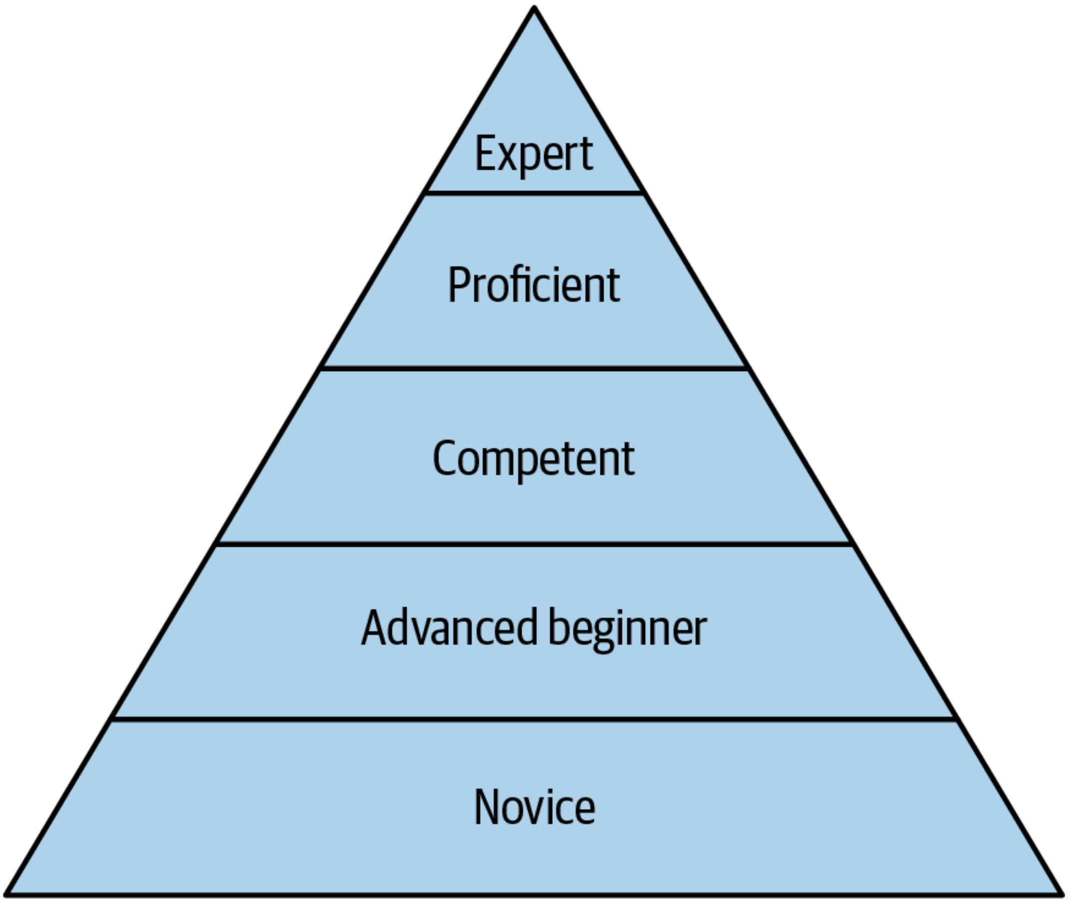


Figure 12-1. The Dreyfus model captures the journey of acquiring new skills

It should be clear that telling someone who just picked up a golf club to "hit a cut by thinking about hitting a cut" won't work very well. When you are just starting out, you need step-by-step instructions, which is why so many languages and frameworks have Getting Started tutorials.[4] Rules and recipes are vital to beginners, but they can absolutely stifle experts because they inhibit their hard-earned intuition. And just as with improving your short game, simply watching a video or reading tip will take you only so far: you have to practice!

Think about how you learn a new framework. Do you immediately jump into building a complex application? Probably not. You start with the proverbial "Hello World!" example.[5] At first, you're copying code directly out of a tutorial, relying on search or a chatbot to troubleshoot errors. Over time though, your intuition develops, and you start to recognize patterns; you'll know where the problem is as soon as you see the error message. After even more time, you'll discover where the edges of the map are, the parts of the framework that you'd change if you could. As you approach mastery, you'll no longer be translating what you want; you'll simply be expressing your intent through the framework. When given a task, you'll "see" how to do it.

The Learning Habit

> The most important skill to have as a programmer is the ability to teach yourself new things effectively and efficiently. You're going to be constantly growing and picking up new technologies. The ability to do that is more important than any individual tool or technology.
>
> Ali Spittel, software engineer

While there are few hard-and-fast rules, becoming an expert takes a significant amount of time, which is why most people don't go much beyond advanced beginner. Given the pace technology moves, that has startling ramifications. Technology moves *fast*; how much time do you get with something before you move on to the next big thing? That isn't to say a successful career requires you to become an expert on every aspect of software. Over your career, your expertise will grow in some areas while other skills remain in the advanced beginner phase, and there's nothing wrong with that!

The challenge is getting the most out of your limited learning time. Make a list of what you want to explore, prioritizing what you want to learn over the next few months. Your list and priorities will be fluid; as your interests change, so too will your learning goals.

Not all learning requires the same level of focus or attention: the depth you choose to pursue changes based on the topic. Christopher Judd refers to this approach as a Learning Depth Strategy:

Survey
: Listen to podcasts, attend user group or conference sessions, watch technology conference keynotes, peruse the release notes.

Shallow dive
: Watch short videos, read blog posts and articles, work through tutorials.

Deep dive
: Attend a live, in-person or online workshop or training course, read books about the topic.

Ultra deep dive
: Deliver a presentation on the topic or lead a workshop or training class.

Learning has to be a habit as humans forget things. In the 1880s, Hermann Ebbinghaus studied memory, discovering people forgot half of what they learned a mere 30 minutes after learning it! However, when you are repeatedly exposed to the material, the decay slows considerably. He proposed the idea of spaced repetition: essentially, reviewing the material at gradually increasing intervals after you learn it reinforces the concepts before they are completely lost. Consider how you could schedule this approach throughout your year:

Weekly
: Listen to podcasts, watch a video or two, read an article or two.

Monthly
: Read a technical or business book, attend a user group or meetup, take an online course.

Yearly
: Attend a technical conference, learn a new programming language, invest in a side project, get a certification, teach or deliver a technical presentation.

It can also be helpful to "learn out loud." Consider posting your progress on social media; you will often discover a community of like-minded people offering support and encouragement. As you uncover new things, think about creating a video or write-up of what you've learned; don't be surprised when a future search on a question returns something you've created![6] You'll be thankful you took the time to document what you did.

The Technology Merry-Go-Round

Nate here. Early in my career, I was talking to my then manager and asked why they'd moved out of tech into management. They told me they were tired of having to constantly learn new things, of always being on the technology merry-go-round; they'd seen enough new things! While I didn't fully appreciate that insight newly out of university, I completely understand the sentiment today. How many languages or frameworks or frontend libraries or datastores have you learned in your career?

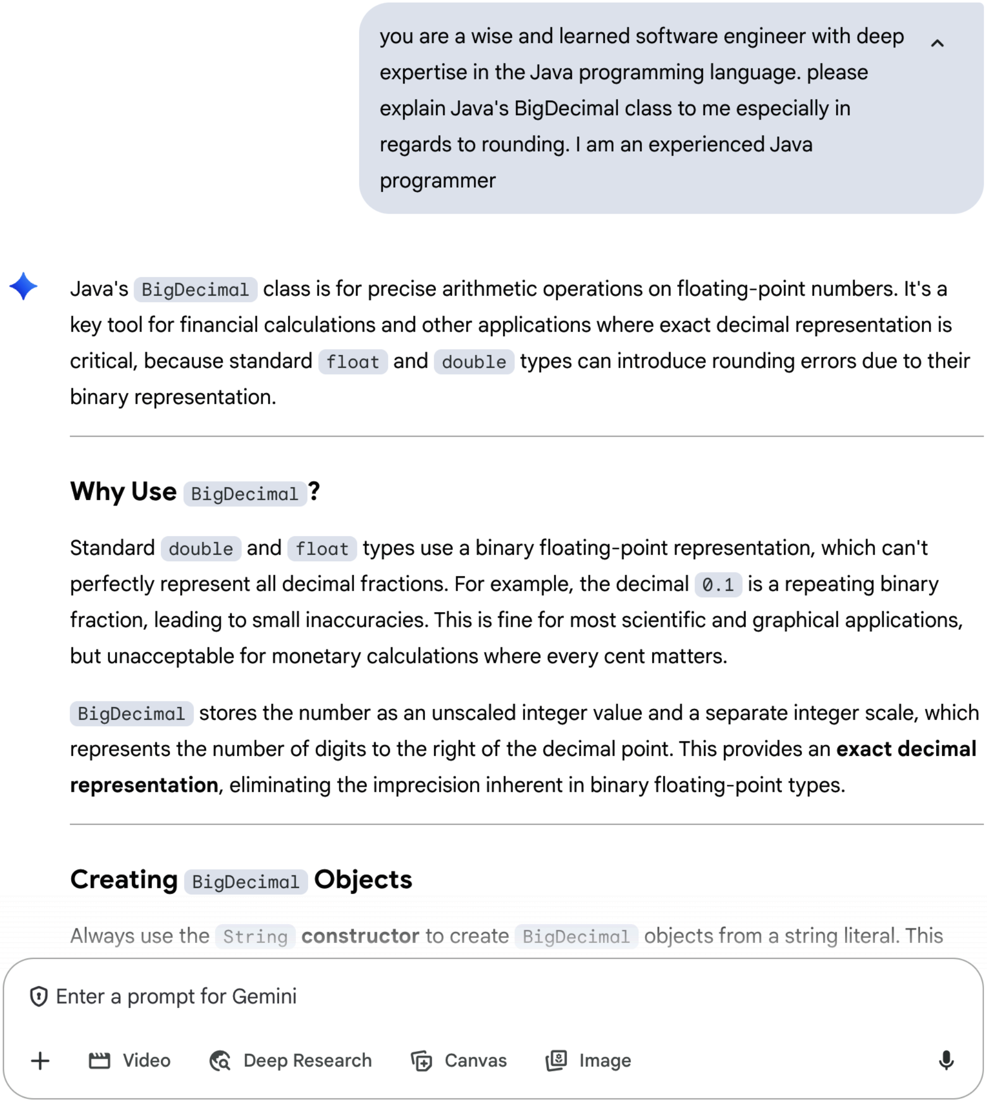
That isn't to say management is the path of least resistance: there are plenty of unique challenges guiding people. But being a software engineer means a near constant stream of adding to your knowledge portfolio. It can be very invigorating, but don't underestimate how hard it can be either.

Knowing there will always be more to learn, how do you keep up? You need to make it a habit. Develop a routine that works for you. Block out time on your calendar to read and explore; you may have to schedule over it from time to time. Friday afternoons are generally quieter,[7] but maybe for you it is Tuesday morning or Thursday over lunch. Ultimately, you must be the guardian of your own time. Some people plot out a year's worth of learning on a Trello board or set reminders throughout the calendar. Experiment with what works best for you, but whatever you do, set aside specific blocks of time to learn.

You should strongly consider a practice we refer to as *morning coffee*. Take the first 15–30 minutes of your day, when you're sitting down with your preferred beverage (whether it's a shot of espresso, tea, an energy drink, or something else) and peruse the news, whatever that means for you. It could be spending some time on social media, listening to a podcast, walking through a tutorial, or reading an early edition of a new tech book. But make it a habit and do it before your day gets away from you. You've had those days that just seem to disappear down a rabbit hole of fire drills! Prioritizing that time will also help on those days where you were so busy you felt like you didn't get anything done; at least you've learned something new!

Learning Through AI

AI can be a learning accelerator. While it isn't a substitute for hands-on experience, using a chatbot can help you research topics. Chatbots can be great resources for tailored learning paths and deep-diving technology in a conversational, digestible manner. For example, let's say you need to use a Java class you aren't familiar with or want more info on a reference model. You certainly can read the docs, but you can also have AI do it for you, as seen in Figures 12-2 and 12-3.



you are a wise and learned software engineer with deep expertise in the Java programming language. please explain Java's BigDecimal class to me especially in regards to rounding. I am an experienced Java programmer

Java's `BigDecimal` class is for precise arithmetic operations on floating-point numbers. It's a key tool for financial calculations and other applications where exact decimal representation is critical, because standard `float` and `double` types can introduce rounding errors due to their binary representation.

Why Use `BigDecimal`?

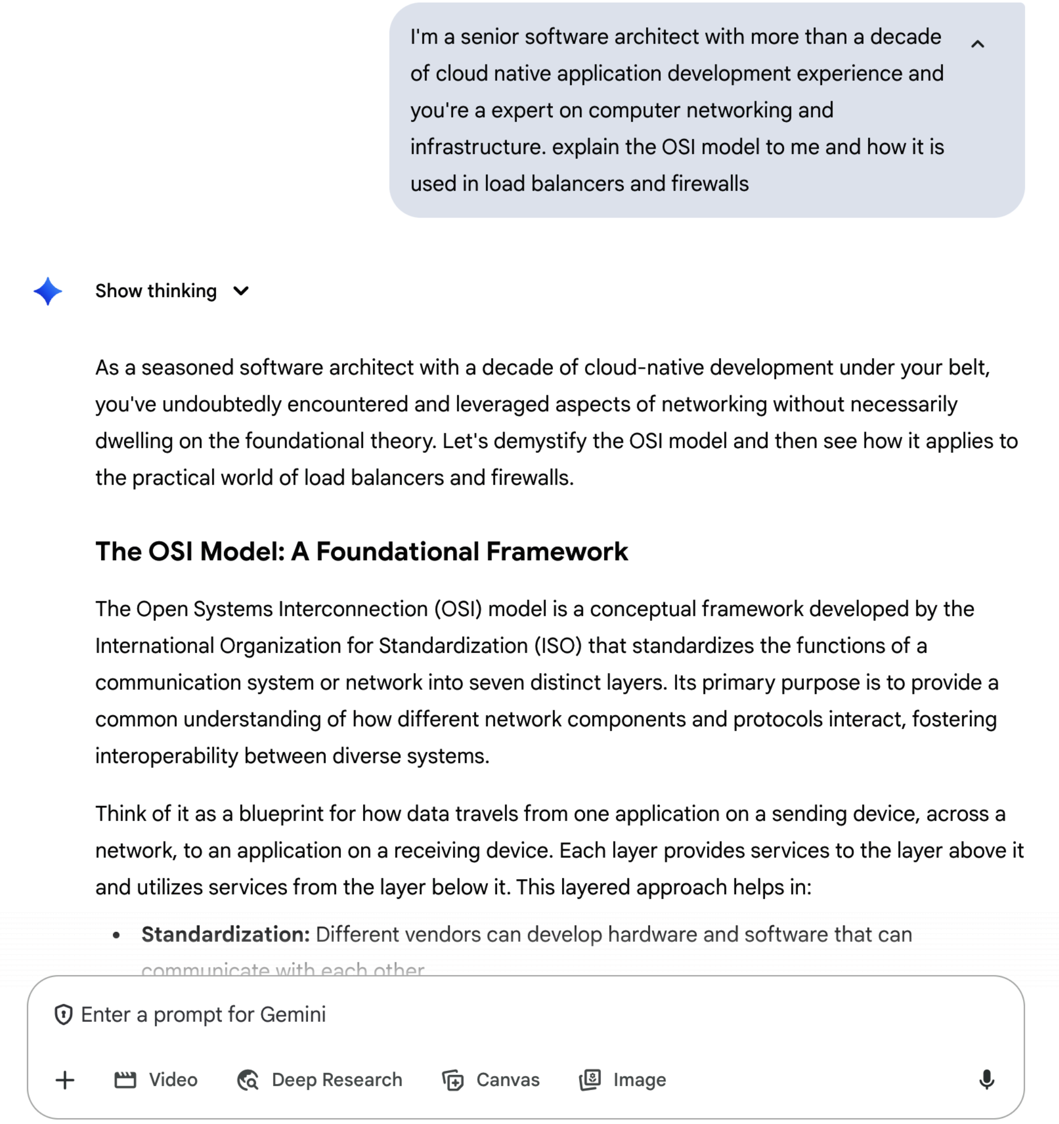
Standard `double` and `float` types use a binary floating-point representation, which can't perfectly represent all decimal fractions. For example, the decimal `0.1` is a repeating binary fraction, leading to small inaccuracies. This is fine for most scientific and graphical applications, but unacceptable for monetary calculations where every cent matters.

`BigDecimal` stores the number as an unscaled integer value and a separate integer scale, which represents the number of digits to the right of the decimal point. This provides an **exact decimal representation**, eliminating the imprecision inherent in binary floating-point types.

Creating `BigDecimal` Objects

Always use the `String` **constructor** to create `BigDecimal` objects from a string literal. This

Figure 12-2. An example of a chatbot explaining a class



I'm a senior software architect with more than a decade of cloud native application development experience and you're a expert on computer networking and infrastructure. explain the OSI model to me and how it is used in load balancers and firewalls

As a seasoned software architect with a decade of cloud-native development under your belt, you've undoubtedly encountered and leveraged aspects of networking without necessarily dwelling on the foundational theory. Let's demystify the OSI model and then see how it applies to the practical world of load balancers and firewalls.

The OSI Model: A Foundational Framework

The Open Systems Interconnection (OSI) model is a conceptual framework developed by the International Organization for Standardization (ISO) that standardizes the functions of a communication system or network into seven distinct layers. Its primary purpose is to provide a common understanding of how different network components and protocols interact, fostering interoperability between diverse systems.

Think of it as a blueprint for how data travels from one application on a sending device, across a network, to an application on a receiving device. Each layer provides services to the layer above it and utilizes services from the layer below it. This layered approach helps in:

- **Standardization:** Different vendors can develop hardware and software that can communicate with each other

Figure 12-3. An example of a chatbot explaining the OSI model.

You do need to be aware of hallucinations, and you should double-check with the source material, but there's no doubt AI can help you learn new information and brush up on concepts from your past.

Fear of Missing Out

There are a lot of bits out there: new languages, new techniques, new approaches. How do you keep up? Your preferred learning style has a lot to say about your approach. Some people prefer to read articles and books, some prefer to listen or watch, while others learn best by doing. There is no shortage of learning sources at your disposal, from searching the web, following a reading list, listening to podcasts, watching videos, attending user groups and conferences, and leveraging industry resources like the Technology Radar.

Attention is a precious resource, and you can't just scale it up. Don't waste it. You cannot read or watch or listen to everything; you are going to have to be selective. Many developers have a legitimate fear of missing out (FOMO), where if they aren't actively consuming "everything," something vital will slip by.

Sorry to burst your bubble, but you literally cannot read or watch or listen to everything. In fact, in the time it takes you to read this paragraph, roughly another week's worth of video will have been uploaded to YouTube! And that doesn't take into account podcasts or Getting Started guides or social media or any other content you might want to consume.

Technology FOMO is real, but take solace in the reality that if something *really* big happens in technology, you *will* hear about it. You might miss the initial announcement, but that's OK. Heck, there are areas you aren't overly interested in but you still heard about! Don't be afraid to leverage your network too. Odds are your friends and colleagues all have different interests than you. Take advantage of that. In fact, you could share interesting information you find in your specialization areas with them, and vice versa. There's nothing wrong with sharding your learning across multiple people!

> Focus means you won't get to surf every wave in the ocean.
>
> Kelsey Hightower, American software engineer, developer advocate

Your time and attention are incredibly valuable resources.[8] And they are resources: you can't just scale them up. You cannot afford to waste your attention; you must be selective with how you spend it. Don't practice resume-driven development; you need a *why* to learn something new. There will always be another hot new thing on social media, and it's OK to wait to learn something until you need it. Evaluating a new technology is tricky, but with experience you'll start to spot the patterns, and you'll see how this "new" thing is actually very similar to something from a few years back. If your instincts tell you something is overhyped, there's nothing wrong with taking a beat before investing time in it.

Your professional network can be very useful here as well, as sometimes you can outsource the learning to someone else. If you know a friend is exploring a space you're not sure of just yet, touch base with them periodically to see how it's going. Their experience can guide you, and in some cases they'll work through problems, allowing you to skip over some rough patches.

> Attention is a bit like real estate, in that they're not making any more of it. Unlike real estate, though, it keeps going up in value.
>
> Seth Godin, American author, marketing expert, and entrepreneur

Let Your Passion Guide You

Nate here. I once tried to learn more about databases so I sat down on a random day and started going through a book on the topic. After about 20 minutes, I put the book down and literally went upstairs and organized my sock drawer. Not a euphemism: I decided my sock drawer was a higher priority. That was an unambiguous signal I did not, in fact, have a passion for the particular topic of databases. While you can schedule some learning time pretty easily, that still leaves a rather important question: what should you be studying? Ultimately, this is up to you, but you should let your passion guide you. If you aren't excited about a topic, you'll find it difficult to make time in your day to practice.

Pick something that energizes you and go deep on it. Follow the experts, listen to the podcasts, watch the videos, and read the documentation. Read "Find What You're Passionate About". Keep in mind you can switch to something different at any time. There's nothing wrong with putting a book down or closing a tab or leaving a video half watched; there aren't extra points for finishing. If you're still excited about a given topic, great, keep going! If it no longer sparks joy, fantastic, pick something new. Rinse and repeat.

Just as you cannot keep up with every new thing in software, you also can't adopt every new thing either. Your passion about a given tool or technology doesn't automatically mean it is a good fit for your project or organization. Many developers have, at one time or another, practiced resume-driven development. In other words, they adopted a new technology because they wanted to add it to their resume, not because it would meaningfully benefit the project.

You must be a critical consumer of new things. Software is a constant set of trade-offs, and despite the way it is often taught, there is rarely a "best" way.[9] Don't focus on just the positives of a technology, but consider all of the consequences. Be an informed consumer and remember Rich Hickey's wise words: "Programmers know the benefits of everything and the trade-offs of nothing."

Where Should You Invest Your Time?

Intuition grows with experience, and over the years you will start to have a sense of what things have staying power and what is probably just a flash in the pan. Ultimately, you'll see where the trends are taking the industry. You should temper your hunches a bit, though. Paul Graham once wrote of the hacker's radar, describing some of the criteria he considers when looking at a new technology.

After applying that hacker's radar to Java, he concluded that "I have a hunch that [Java] won't be a very successful language." Now, he made this statement in 2001 when Java was fairly new. More than 20 years later, with the full value of hindsight, Java is obviously one of the most successful languages of all time. If someone as experienced and successful as Graham can be that wrong, you may want to temper your own intuition a bit.

That said, judging covers can be useful. In software, patterns often repeat themselves, and things tend to swing like a pendulum between extremes. It is common for old things to be dusted off, refined, renamed, and introduced again as if no one has ever done that before. Consider this description:

- "An approach that allows software components written in different programming languages and running on various computers to communicate and work together."

What technology do you think that definition describes? Are you thinking about microservices? It fits! However, that's a definition of Common Object Request Broker Architecture (COBRA), which was popular in the mid to late 1990s, many years before anyone had even contemplated a microservice architecture.

With experience, you will start to see how a new technology is just like one you used a few years back.[10] Maybe problems with that technology have been resolved, or maybe not. You should ask probing questions. Just because something is new doesn't mean it will benefit your projects.

The wisdom of the crowd *can* be a helpful guide. What is capturing the attention of the community? That can lead you to interesting things. But sometimes the masses get caught up in the hype. Again, apply your own experiences as well.

You can avail yourself of industry resources such as the Technology Radar from Thoughtworks. Updated twice a year, the Technology Radar is a snapshot of tools, techniques, platforms, languages, and frameworks. The radar assimilates experiences across multiple projects to generate as objective an analysis of technology as possible. Tracking the Radar can introduce you to new things as well as providing a sense of where those technologies are in the adoption cycle.

Build Your Own Technology Radar

Hopefully, your company offers a rich set of learning opportunities that you can take advantage of. However, as a software engineer, you are ultimately responsible for your career journey. With a never-ending river of new languages and technologies, it can be incredibly challenging to decide how to allocate your precious attention. It is very easy to get stuck in analysis paralysis, something you can short-circuit by building your own technology radar.

It doesn't have to be overly complex nor does it need to be as graphically impressive as the Thoughtworks product. But establishing the habit of periodically checking in with what *you* find interesting and exciting can make all the difference in your learning. You can read more about how to do so in Chapter 14.

Practice Innovation

As you learn and explore, you will find something that absolutely will benefit your project, your team, or your organization. Bringing those ideas into your organization can give you increased visibility in your company, helping you earn raises and promotions. But how do you go about introducing new things?

A book club can be one of the simplest and most effective ways to bring in new technologies. Find a group of like-minded engineers, pick a technical book, and have at it. Try to meet weekly, perhaps first thing in the morning or over lunch. Nearly every organization will cover the cost of books, and many are even willing to pick up catering for lunch or breakfast. Book clubs also serve as motivators: it's a lot easier to read a chapter when you know you're talking about it with your team on Tuesday.

Many organizations practice some variant of *innovation day*, often called *hacker days* or *hackathons*. No matter the term, the concept is the same: provide people a safe, hands-on space to explore new technologies. Hacker days act like a relief valve for the inevitable desire to play with new technology. Without it, many developers will try to just sneak that new library or language into a project, betting on the utility of asking for forgiveness later rather than asking for permission.

If your company doesn't have a recurring hacker day concept, introduce it! Pitch it as an experiment and start small—maybe even just a Friday afternoon. Involve as many parts of your organization as you can; you'll be surprised by just where some game-changing ideas lay dormant. While the resulting code may never see production, you will learn something. Maybe a new framework seemed really promising, but when it made contact with your particular context, you quickly realized it would require a near ground-up rewrite. Perhaps that new language elegantly solves a tricky problem you've been struggling with for weeks. You won't know until you try it.

Periodic project showcases, whether department-wide or companywide, are an excellent way to learn what technologies are being used in your organization. The mechanics range from short, lightning-talk-length demos all the way to a set of booths, where different groups present their work. Demo days are not only a great way to learn about the technologies being used in the company, but also an opportunity to identify the people you can go talk to when a need arises on your project.

Hackathons aren't the only innovation option. Regular tech talk series are an invaluable way to introduce new concepts into a company. Whether the speakers are sourced from your colleagues or industry experts, providing a steady diet of lunch-and-learn sessions is one of the simplest ways to keep up with the changing technology landscape.[11] Again, if your company doesn't have an existing talk series, create it! Volunteering to deliver a talk on a technology you're trying to learn is a powerful motivator.

Lunch-and-learn sessions don't need to be companywide to be effective. More localized approaches are great too. Have you read an interesting article or watched an insightful video? Share it with your team. Looking for some extra motivation to read a new book? Start a book club with a few like-minded souls, trading off who is responsible for leading the discussion. If you're really excited about new technologies, start a biweekly tech discussion meeting!

Most decent-sized metropolitan areas have a regular schedule of local meetups and user group meetings. Odds are, you can find an interesting tech talk most evenings (and you're likely to get dinner or at least some refreshments out of the deal).[12] While the large industry events in Las Vegas or San Francisco may get more attention from the tech press, there are many excellent regional events that won't require you to spend thousands of dollars on travel.

If you've already got a well-attended lunch-and-learn series, and you've attended a regional event, you could take things up a notch and organize an internal conference. These can range from informal half-day events with a handful of internal speakers to full-on multiday, multisite affairs including industry experts from outside your organization.[13] Do not underestimate the effort it takes to organize and deliver on even small internal events. But the payoff can be huge.

Internal events bring people together and help new ideas spread. They offer a chance for people to interact with those they don't normally encounter day to day. They are also an excellent chance for you to put your presentation skills to the test. While presenting provokes primal fear in many, it is an excellent motivator if you're trying to learn something new. Not to mention the visibility can accelerate your career growth, setting you up for a promotion or increased project responsibility.

Architectural Briefings

Some organizations have a formalized approach using architectural briefings. Essentially one person does some research and presents the results back to the team. And no, you don't have to be an architect to deliver an architectural briefing. While there are any number of questions you can explore, you should be sure to hit on these:

- Why should you use this technology? Why should you *not* use this technology?
- What do you need to know in order to answer the previous question?
- What do you need to know in order to use this technology?

The resulting presentation doesn't need to be a multiple-hour affair; 45–60 minutes is more than sufficient. It isn't a deep dive how-to but should get beyond the basic Getting Started documentation. These sessions should be interactive and encourage two-way participation. Attendees should be taking notes and asking good questions, bringing their own experiences to bear on the topic.

If you get past the briefing phase, it’s time to roll up your sleeves and get your hands dirty. It’s time for a workshop. Unfortunately, you will never get enough time to perform a workshop. Trying to decide between two competing frontend technologies? There’s a surefire way to make that choice: build the app twice, once with each, and throw away the one you don’t like. Good luck getting that approved.[14] You will have to determine the most important things to prove with your limited time.

Focus on exploring the key features and have a simple setup for participants. Clearly state your objectives and offer follow-up material for those who wish to dig deeper after the session. Don’t be afraid to try things out and poke around the nooks and crannies; you may be pleasantly (or unpleasantly) surprised by what you discover. What works? What doesn’t? What happens when you venture off the happy path?

Ensure that your workshop is subject to any constraints the technology will encounter on the target project. If you will be adding this to an existing legacy project, test the technology with those legacy limitations. Getting something to work in a greenfield environment can be helpful but may leave you with an incomplete picture.

Legacy Project Constraints

Nate here. Years ago a participant in one of my workshops shared their experience trying to use a new user interface technology on their project. They had hit a wall with the library they had chosen years before and needed to make a change. The team surveyed the field, narrowing the options to three. They divided up the libraries and performed some greenfield workshops. Eventually they settled on one option that they planned to sprinkle into their application.

Six months later they now fully realized this library couldn’t be added in here and there; to truly use it, they would need to completely refactor their application. They admitted they’d been caught up in the hype around the library they chose, but it’s also clear they should have performed their exploitation using a fork of their existing project. They likely would have discovered the need for a greater refactoring much earlier. Now, it is possible they would still have chosen that library! But at least they would have gone into it eyes wide open.

The moral of the story: just because you can get the to-do list to work greenfield doesn’t mean a given technology will survive contact with your environment. You play how you practice; don’t try new things in a vacuum.

Once you’ve had some hands-on time, you can make informed decisions about a technology. If everything checks out, it’s time to find a trial project in your organization. Make sure the trial project is a good fit and ensure that a failure won’t doom a critical strategic project.

Practice Grace

Keeping up with the rapidly changing technology landscape is important and, frankly, a vital part of your software engineering skill set. However, it can be very overwhelming. Be kind to yourself; practice some grace. It can be incredibly easy to burn out by trying to stay on top of an industry as diverse as software. Learning should be fun, not a point of stress in your life. Don’t let it come at the expense of a good night’s sleep or your mental (or physical) health.

There's nothing wrong with waiting until the last responsible moment to learn something. Technology is full of hype cycles; don't be afraid to let something play out before you commit your time to it.[15] Patience is a virtue; the software space is littered with the abandoned remains of last month's new hotness. There's no statute of limitations on when you can learn something: if something proves it has staying power, you can always pick something up next week or next month or next year.

Wrapping Up

Technology changes. Fast. Over the course of your career, you will work with a wide variety of tools, languages, and technologies; your ability to learn, adapt, and adjust is vital to your success. Keeping your skills up-to-date must be a habit, so block out time in your schedule to play with new things. A regular cadence is more sustainable than cramming a year's worth of learning into a long weekend. Enjoy the process!

Putting It into Practice

Ultimately, to learn a new technology, you have to literally put it into practice. Consider building a personal app to scratch an itch; maybe you want to manage your burgeoning wine collection or you want to build an application to keep track of your child's sports statistics. If you don't have any burning needs, reach out to your local school or nonprofits, as they're often looking for technology help and will usually give you wide latitude to try things out.

Take a moment to survey the technology field; what is generating the most interest today? Does it speak to you? If not, pick something that does! Go deep on the thing that excites you, follow the thought leaders on social media, and subscribe to podcasts, streams, newsletters, and videos. Is there a local user group you can attend? Are there any conferences focusing on it? What about virtual events? Immerse yourself in that space. Try building a personal project using it. Give a presentation on it to your colleagues, at your local user group, or at a conference.

Every month or two, reassess. Are you still passionate about that technology? If not, pick something new! Rinse and repeat. Don't be afraid to return to something you explored earlier; you won't have to relearn everything, and you'll likely expand on your knowledge.

At the end of the day, nothing is quite as effective as spending a few days or weeks producing working code. It doesn't have to be elegant, doesn't have to cover every possible edge case, and can be just enough to prove the technology out. There are no shortcuts to learning, no way to just "load the information into your head."

Learning out loud can also be an effective teacher. Don't be afraid to join forces with a couple of friends or colleagues working through a small project together. Even just one additional set of eyes can help, so pairing is an effective approach. Consider open sourcing your project or at least hosting it on a public repository so others can comment and contribute. You could also consider live streaming your learning; while it can be intimidating, the communal effect is powerful. Others have likely encountered similar problems and will typically offer their advice. Just the simple realization that you aren't alone on your learning journey can be incredibly reassuring.

Additional Resources

- “Thinking Architecturally” (report) by Nathaniel Schutta (O’Reilly, 2018)
- *The Pragmatic Programmer* by Andrew Hunt and David Thomas (Addison-Wesley Professional, 1999)
- *The Passionate Programmer* by Chad Fowler (Pragmatic Bookshelf, 2009)
- Thoughtworks Technology Radar
- *The First 20 Hours: How to Learn Anything…Fast!* by Josh Kaufman (Portfolio, 2014)
- *Pragmatic Thinking and Learning* by Andy Hunt (Pragmatic Bookshelf, 2008)

[1] Someone your authors may or may not have direct experience with themselves.

[2] Say, a phone number from your youth or the name of your favorite elementary school teacher.

[3] What did you say your name was again?

[4] Plus, who doesn’t need another to-do list, amirite?

[5] How many to-do list applications have you built?

[6] Both of your authors can attest to finding some of their own material they had completely forgotten they’d created.

[7] Anyone who schedules a meeting on a Friday afternoon has committed a hostile act.

[8] Time is a nonrenewable resource. Use it wisely.

[9] More often than not, you are choosing the least worst approach.

[10] Or a technology initially created for one purpose may later re-emerge to solve a different problem. For example, the internet was repurposed for civilian use after being initially developed by the military.

[11] Or as one organization termed it, “chew and spews.” Not our preferred moniker, mind you.

[12] Volunteering to give a talk at a local event is an excellent on-ramp to conference speaking if that path is of interest to you.

[13] Your authors have participated in *many* of these events for our clients; reach out if we can help.

[14] If you ever succeed, please drop your authors a note; we’d love to hear more.

[15] Time is a nonrenewable resource. Spend it wisely.