O'REILLY®

Compliments of Pivotal

Thinking Architecturally

Lead Technical Change Within Your Engineering Team

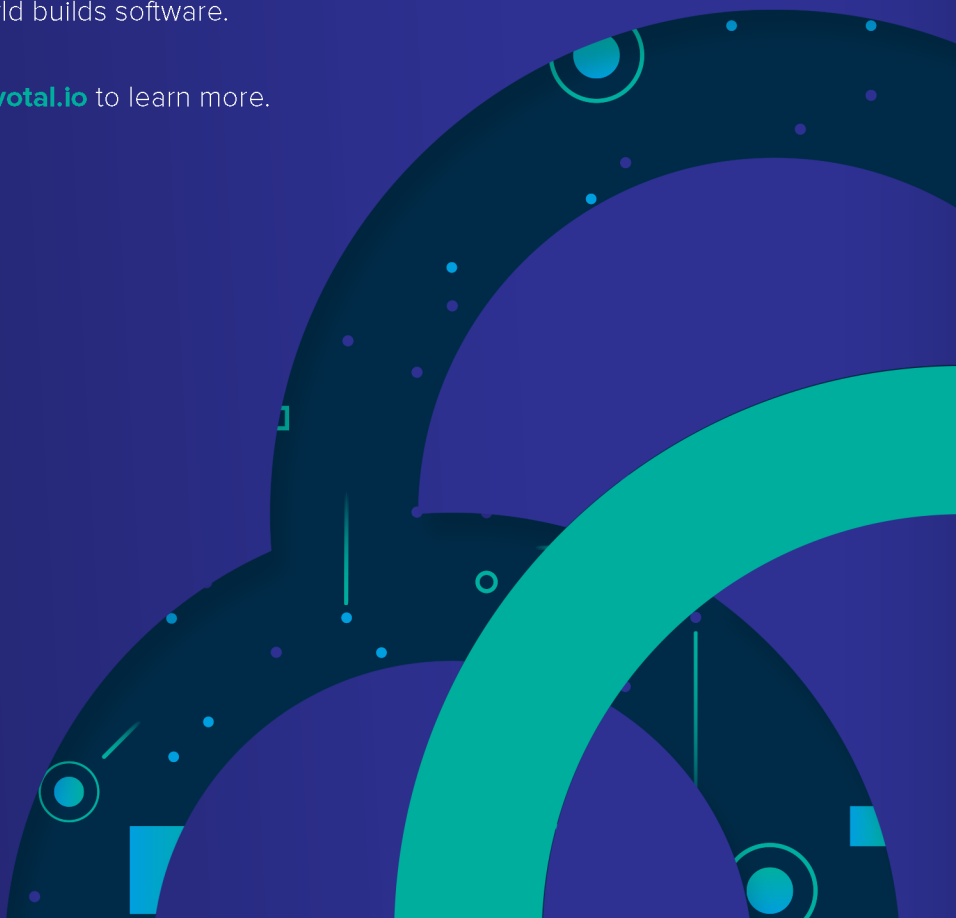
Nathaniel Schutta

Pivotal

Any App Every Cloud One Platform

Pivotal's Spring and Cloud Foundry, combined with our unique approach to agile development and continuous delivery, are transforming how the world builds software.

Visit **pivotal.io** to learn more.



Thinking Architecturally

Lead Technical Change Within Your Engineering Team

Nathaniel Schutta

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Thinking Architecturally
by Nathaniel Schutta

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Alicia Young
Acquisitions Editor: Brian Foster
Production Editor: Nicholas Adams
Copyeditor: Christina Edwards
Interior Designer: David Futato
Cover Designer: Karen Montgomery

June 2018: First Edition

Revision History for the First Edition
2018-05-14: First Release

This work is part of a collaboration between O'Reilly and Pivotal. See our *statement of editorial independence*.

978-1-492-04097-2

[LSI]

Table of Contents

Preface

Rich Hickey once said programmers know the benefits of everything and the tradeoffs of nothing…an approach that can lead a project down a path of frustrated developers and unhappy customers. But as an architect you must consider the tradeoffs of every new library, language, pattern, or approach and quickly make decisions often with incomplete information. How should you think about the inevitable technology choices you have to make on a project? How do you balance competing agendas? How do you keep your team happy and excited without chasing every new technology trend?

As an architect it is your responsibility to effectively guide your teams on their technology journey. In the following report, you will follow the arc from discovering a new technology through to maintaining technologies in your organization. You will learn the importance of tradeoffs, how you can analyze new technologies, and how you can effectively capture the inevitable architectural decisions you will make. You will also explore the value of fitness functions as a way to ensure the choices you make are actually reflected in the code base.

Before you read any further, I want to clarify a point about titles. In the software industry, titles aren't very well defined. Ask 20 developers to define architect and you will get *at least* 20 different answers. There are more than a few organizations that proudly say they don't have architects! Additionally, the title of "architect" can be closely guarded in some organizations.[1] As much as some people make very fine-grained distinctions, I hope people without architect in their title give this book a read and find something useful in it. As such, it is meant for tech leads, senior developers, junior developers, and of course, practicing architects, regardless of actual titles.

1 I was a "senior developer" doing architecture work for years before I finally was officially named an architect.

Whatever your title, you won't become an architect through words alone. You need an opportunity to practice the skills successful architects possess. That is why every chapter has a set of Katas, or exercises, to help you master the concepts. Katas originated in martial arts training, but the idea has spread beyond the dojo. As Fred Brooks famously said, "How do we get great designers? Great designers design, of course." This, of course, implies that the way to get great architects is to, well, architect a great system!

However, the average architect may only work on a small handful of applications. How are you supposed to become a master without the needed practice? Professional athletes spend far more time practicing their sport than they do playing in games that "count." Professional musicians and actors spend countless hours rehearsing before performing in front of a live audience. But beyond a perfunctory class, most architects learn on the job, meaning they make their mistakes on live projects. Is it any wonder so many software projects fail to deliver the expected value?

The concept of architectural katas arose from a series of workshops that Ted Neward, Neal Ford, Mark Richards, Matt Stine, Simon Brown, myself, and a few others have led over the last several years with the goal of giving participants an opportunity to work through a pseudo real-world problem and its architectural issues.

With that context, each chapter will present you with opportunities for effortful study. But you should also familiarize yourself with the Architectural Katas and what it takes to lead your own.

Acknowledgments

First and foremost, I want to thank my wife Christine for her constant support on my ever evolving journey. I could not do what I do without her in my corner. On more than one occasion she has shouldered more than her fair share so that I could present at some conference, teach a class, or write a book. To my son Everett for tolerating my sometimes erratic schedule and always welcoming me home with a giant smile and a hug. Even when I'm not home, I'm always thinking about you Ev! I lack the words to properly express my gratitude for my amazing family.

The team at O'Reilly Media is amazing and I appreciate their patience and guidance throughout the process of writing this report. Many thanks to Brian Foster for molding my thoughts into a more coherent narrative, to Alicia Young for turning my writing into something readable and to Nick Adams and Christina Edwards for shepherding me through the production process. If you'd have told me twenty years ago I would be part of the O'Reilly family I would never have believed it possible. Thank you to everyone that has welcomed me as a speaker, as a teacher, and as a writer.

I am where I am today thanks to an amazing group of people I am privileged to call my friends. To paraphrase Isaac Newtown, I have had the great fortune of standing on the shoulders of giants and I want to extend my heartfelt appreciation to those that have shaped my thinking on everything from software to fine dining to slinging slides. To Martin Fowler, Brian Goetz, Neal Ford, Matthew McCullough, Stuart Halloway, Justin Gehtland, Ken Sipe, Scott Davis, Mark Richards, Tim Berglund, Craig Walls, Venkat Subramaniam, Matt Stine, Glen Vanderburg, Dave Hussman, Ken Kousen, Mike Nygard, Kirk Knoernschild, Christopher Judd, Daniel Hinojosa, Raju Gandhi, Jeremy Deane, Peter Bell, Howard Lewis Ship, Pratik Patel, Brian Sletten, Michael Carducci, Danny Brian, Ted Neward, Joe Athman, and Aaron Bedra, I still don't know what I did to warrant such an amazing group of friends. The next round is on me! Assuming you read this. I also want to thank the many talented architects I've had the privilege to work with over the years. From Gary Gillard, my "first" architect on the first web project I was a part of to Jeff Escott for teaching me many of the subtler aspects of the job to my good friend Aasta Frascati-Robinson for being such an inspiration. Mark Stofferahn shaped my thinking on so many things from quality attributes to the cloud. Karim HadjSalem and Matt Gorman were quick to remind me sometimes the only option is to laugh it off. To Scott Leigner, Bob Baty, Cam Sonido, Steve Shaw, Rick Meemken, Dave Dahl, Mike Gitberg, Mike Samra, James Brown, Rob Harwood, Craig Gronlund, Hoa Ton-That, Pee T Lim, Ken Lamb, Don Smith, Eamonn Buss, Chad Hinkel, Chris Lyons, Raina Johnson, Joe Drechsel, Bobby Kaminsky, Ted Setterlund, Mark Schreifels, Bob Casola, Jim Payant, and Jill Spear, thank you all for your advice, your wisdom and your friendship. And to anyone I omitted from this list, my humblest apologies, know that it was a simple off by one error on my part and not a deliberate slight.

CHAPTER 1

Technology Changes

If you've been in the software world for any amount of time you know that technology changes. It is inevitable. Early in my career I became despondent because I thought I'd "missed the wave" of some technology or another. What I didn't appreciate at the time is that new technologies are a lot like buses—one comes every 15 minutes or so.

As architects, we have to remember that we cannot stop change, but we can manage it. In the rest of this chapter, we will cover the lessons the past can teach, a reminder that you can't predict what the future holds and why you shouldn't be so quick to jettison legacy skillsets.

Haven't We Seen This Before?

> Those who cannot remember the past are condemned to repeat it.
>
> —George Santayana

For most developers, time began with the first language they learned. That initial experience becomes our lodestar and we often compare new languages to that inaugural experience, which becomes our programming origin story. While our first language is indeed foundational, it can leave us short-sighted.

Many developers fail to appreciate that the "new" feature just added to their favorite language has, in fact, been part of programming for decades. When Java 8 added lambda expressions, some developers did not understand why they needed this "new fangled" feature. I'm sure more than a few Lisp programmers found that assertion amusing.

With experience, you will come to recognize that technologies often repeat themselves.[1] Reflect for a moment on distributed applications. Today it is fashionable to talk about microservices and serverless technologies. But step back for a minute and look at the evolution from bespoke servers to virtual machines to containers. If you squint a bit, you'll recognize the echo of service-oriented architectures. In fact, some people like to say microservices are just SOA done right. Remember all the blog posts, articles, books, and conference talks devoted to SOA?

But step back even further. Isn't SOA an awful lot like Enterprise Java Beans?[2] They were the "it technology" not that long ago, filling more than a few conference tracks. Even then though, our industry continued to build on the shoulders of what came before. Another step back and you might start to see the faint outlines of CORBA…

But that's not to say things haven't improved. Cloud computing removes many of the constraints that drove us toward shared infrastructure and all its consequences. Still, it is important to recognize how today's technology is informed by its past. The past shapes the future and with that historical grounding, you can make educated guesses about where a given technology is headed.

When you know where our industry has come from, you can start to recognize patterns. Noticing a recycled technology enables you to "skip" something you've seen fail before. You can also start to see why something that didn't work five years ago can work today.

But to see these patterns you have to study our past.

Learn From the Past

> Study the past, if you would divine the future.
>
> —Confucius

Unlike most other industries, ours doesn't do a great job of learning from its past. Every discipline studies its history except ours. A typical computer science curriculum might include a "history of programming languages" requirement but in general we give short shrift to the past. The focus is always the cutting-edge language, the latest framework, the new hotness. But without that exposure to the past, we can't appreciate when it is, indeed, repeating itself.

Software is an immature industry. Truth is, we don't like to talk about our mistakes. But we should. I will never forget August 1, 2007. That evening, the 35W

1 With the names changed to protect the guilty.

2 I apologize if that triggers unpleasant memories for any of you; I know I've done my best to move on from that chapter of my life.

bridge over the Mississippi river collapsed in rush hour traffic killing 13 and injuring another 145. I had been on that very same bridge earlier that week.

That fall, practicing engineers (as well as engineering students) around the world studied that tragedy. When the National Transportation Safety Board released their accident report I guarantee practicing engineers (as well as engineering students) around the world read it. And (I hope) the design flaw that ultimately led to that disaster will not be repeated in future bridge designs.

Mistakes Were Made…But Not By Me

Contrast that with software. When was the last time *you* were involved in a, shall we say, less than successful project? Did your company have a culture that allowed for an honest conversation, were you allowed to discuss the lessons learned from your experiences, or were you forced to say the project went swimmingly?

Years ago, I was on a project that, to say the least, did not meet all our expectations. After we wrapped up, my technical lead put together a postmortem presentation so that other teams could learn from our mistakes. He showed it to our manager and she…changed some things. Our manager shared the presentation with her boss who also made some modifications. By the time we actually presented our findings, the message was so watered down it basically professed the project was a rousing success!

I understand why we aren't keen on sharing our failures with the rest of the industry, but when we can't even have an internal discussion with our colleagues, we are more likely to make the same mistakes over and over again. We are also more apt to reinvent the wheel—unnecessarily.

The Technology Merry-Go-Round

Early in my career, I sat down with my then manager and asked her a very simple question: Why did you decide to become a manager? She said she got tired of constantly having to learn new things, that she was weary of the technology merry-go-round. I will admit, as a fresh out-of-school developer, I didn't understand her lament but having now lived through more front-end frameworks than I can count, I understand why she felt that way.

Technology *will* change and frankly that is what attracts many people to this industry. Learning new things keeps things fresh! But more than a few of us are guilty of practicing resume-driven design, of choosing a technology not for fitness to purpose, but to be able to add it to our CV.

The Futility of Predictions

As Yogi Berra once said, "It's tough to make predictions, especially about the future." This prescient quote should be brought to bear anytime someone on your team thinks they have *finally* found *the* answer in the technology space. To be honest, the number of "can't miss" technologies you've seen crater is proportional to the number of years you've been in the profession.

I distinctly remember attending a presentation where a prominent technology company introduced a fascinating new approach to front-end development. There was a palpable buzz in the jam-packed room. The introduction was a highlight of the conference and people started adopting it almost immediately. At the time, it seemed like *the* answer. A good friend of mine dove in head first in an effort to ride the wave. Conferences were littered with talks touting the new framework. Thousands of words were spilt in books, blogs, and articles.

A few years later, the script flipped. Teams were scrambling to remove a technology that had turned into an anchor on velocity. User interfaces had to be rewritten, kicking off another cycle of technology evaluation. Even the original company behind it quietly moved on and today you are highly unlikely to run into anyone developing net new code on it.

Based purely on developer excitement, that technology seemed like a can't miss proposition. But it did. At the same time it was announced, few would have believed we'd soon build applications with tens of thousands of lines of JavaScript. Or that Objective-C would soon be one of the most in demand languages. Or that Clojure would bring Lisp to the JVM.

We can't predict the future, but there is a good chance it will be different than today. Technologies rarely become extinct—see COBOL. But I can guarantee in five years we'll be all excited about something that hasn't even been invented yet.

While we have no idea what *specific* technology will dominate in the years to come, broad trends can guide your decision making. Consider the shift from handcrafted servers to virtual machines to cloud hosting. Based on cost structures alone, it is highly unlikely we will ever return to the era of treating servers like pets. And while we can't predict which technologies will dominate the industry it seems likely innovation will continue to cause disruption.

It is a fool's errand to make specific predictions in the technology space, but developers will no doubt continue to chase the latest hot technology.

The Value of Legacy Skillsets

Playing with the latest language or framework can be fun…but it is also frustrating. Anyone that has tried to install a 0.x framework is bound to have experi-

enced the pain of a missing dependency or some other small difference between the Hello World example and their laptop.

The software industry fetishizes the new and many developers fear a legacy skillset. No one wants to be the last of the X developers,[3] instead preferring to work with the freshest tech. But we need to appreciate the central tenet of "bleeding edge": the bleeding edge implies you will bleed. That might be perfectly fine when you are scratching your own itch at home, but you need to be more measured when suggesting technologies to your customers.

And just as it is far more exciting to announce the opening of a new bridge than it is to tout the maintenance of an existing one, many technologists prefer to be pioneers, preferring to be on the bleeding edge. While it can be more exciting to work on something new, it comes with its share of tradeoffs.

> "Our application has *four* UI technologies…"
>
> —Anonymous Technology Professional

I bet you can appreciate the preceding quote and I suspect more than a few of you can relate. How did my student find herself in this situation? Admittedly, no one felt empowered to say "no" and everyone wanted to enable innovation. While I certainly applaud teams willing to try new things, we cannot simply chase a fad; we have to understand what a new technique, technology, approach, framework, language brings to the unique situation we find ourselves in.

Dunning-Kruger Effect

The Dunning-Kruger effect is a well-known cognitive bias. In essence, it means that more competent practitioners underestimate their abilities while those with less expertise believe they are better than they really are.

Some of the inaccurate evaluation of skills relates to an ignorance of what competency actually looks like. As Donald Rumsfeld infamously said, "There are known knowns. These are things we know that we know. There are known unknowns. That is to say, there are things that we know we don't know. But there are also unknown unknowns. There are things we don't know we don't know." In other words, less competent individuals don't know what they don't know.

Keep the Dunning-Kruger effect in mind when someone on your team becomes enamored with yet another technology fad.

3 Though it can pay rather handsomely.

Legacy Skillsets Can be Valuable Too!

I understand why most developers prefer new technologies—employability. From the beginning of our career we are taught a healthy fear of a legacy skillset. We don't want to be the proverbial dinosaur of our organization.

While I generally advise people to stay current in our field, it is important to remember you *can* make a very good living working with "old," established technologies. Take COBOL for example. How many billions of lines of COBOL are there in the world? With fewer and fewer experienced COBOL experts in the workforce, the hourly rate of a COBOL programer is only going to go up.

Early on in my career I had a colleague who had a "retirement countdown screensaver." Shortly after the counter hit zero, we had a retirement party for him. Imagine my surprise when I ran into him a few months later. I asked him if he was visiting for lunch and he said "No, it's the darnedest thing, I'm working three days a week and they're paying me *more* than when I was a full-time guy!" I thought to myself…that's great, how do I get that gig?

My colleague had a legacy skillset—he knew COBOL, JCL, TELON, IMS, and a host of other mainframe technologies. He also knew the old systems intimately and he knew where all the skeletons were buried because he put most of them there. He was able to turn these "dead" skills into a solid payday, even after retirement.

But COBOL is just one example. Years ago, I was involved in a corporate merger and one of the systems we inherited was written in a technology we had zero experience with. We weren't alone. In fact, the install base was so small there were just a handful of contractors in the country. The time pressure of the merger didn't afford us the luxury of doing a rewrite immediately. We had to close the merger; only *then* we could address the technical debt. We had to make a tactical choice, leaving us with one option: pay inflated rates for a consultant that had the knowledge we lacked.

That said, once we completed the merger, one of our first priorities was removing the legacy dependency—and the very expensive contractor—from our portfolio. Again, it is in your best interest to have up-to-date skills. Sometimes we're too quick to remove an old technology from our resumes. Other times we stay in the past too long. The challenge is knowing when to jump from the old to the new.

Eventually that legacy language will disappear. If you are investing in an older technology, you need to be ready to move on. Keep a weather eye on the horizon—is it getting harder to find work with that technology? Check job boards and industry news sites to see where the trends are heading. Ping your network, are your peers noticing decreasing demand? It will likely happen a year or two earlier than you expect. Be prepared and continue to invest in your skillset.

If we are constantly experimenting with the new, we never develop any expertise in the now. Changing frameworks every few months might keep us at the forefront of buzzword bingo, but it is hard to be good at something you've just picked up. And as tempting as it is to just throw away the old and rebuild on the ashes of the past,[4] it is hard to deliver any business value if we are in a constant state of flux.

Over the course of the rest of this book, we'll discuss various techniques and approaches you can utilize to help you make the right decision for your projects.

The rest of this book follows the arc from discovering a new technology through to keeping things running smoothly. Chapter 2 will help you keep up with the software industry followed by how to pick technologies in Chapter 3. Then you will learn about evaluating technologies in Chapter 4, and Chapter 5 will help you introduce those new technologies into your company. Finally, Chapter 6 will discuss how to maintain those technologies in your organization.

Katas

As mentioned in the preface, each chapter will present opportunities for effortful study. Take some time to consider the current trends. What makes for a successful technology? What characteristics might indicate a technology won't work out in the end?

- Think back to a technology that, despite the fanfare, didn't take over the world. Why didn't it? What was lacking? Is there anything that could have changed the outcome?
- Consider a technology that *did* come to dominate. Why did it? What could have caused it to fail?
- Based on today's trends, what technologies do you think are worth investing in over the next year? Why?
- Based on today's trends, what technologies do you think are *not* worth investing in over the next year? Why not?

4 Sadly, "Let the past die. Kill it, if you have to." only works in the movies.

CHAPTER 2

Thinking Strategically

Guiding our projects through technology transformations is our challenge as architects. While it is tempting to flit from technology to technology, as architects we need to have a plan.

We need to think strategically about technological evolutions. In this chapter, you will learn how to keep up with your field, why you should build yourself a technology radar, and how to create a learning plan.

Keeping Up With New Technologies

It seems like every few days a new library, framework, or language is released. Every conference introduces dozens of new ideas. Companies are innovating introducing new products almost constantly. While you slept last night someone published a "new" methodology that solves all the problems with the one you use today.[1]

In a given week, you could go to ten meetups, watch days of video, and read thousands of posts. Your queue is fifteen deep with the latest must-read books and your phone has a months' worth of podcasts just waiting for your listening pleasure. But how do we cope with a rate of change that is, if anything, accelerating?

Hope is not a strategy.[2] As a senior technologist, you need to keep up with changes in your industry. Many occupations require continuing education to remain credentialed. Some occupations build professional development days into their schedule. But software doesn't have any kind of licensing nor do we have

1 With a money-back guarantee.

2 But it is what rebellions are built on.

any agreed-upon concept of ongoing learning. This might have something to do with the fact that many developers are self-taught.

There is no licensing body requiring software engineers or architects to stay current with our industry, but we certainly have a professional obligation to do so. But how? First, recognize that you *cannot* keep up with everything—there is simply too much change in our industry for one person to master all of it. *Sorry*.

To be honest, you will miss almost everything that gets published. If you read one book a week, that's only 52 a year. Over the course of an average working career that might net you around 2,600 books. Sounds like a lot, but it really isn't when you consider how many books have been published throughout human history. Even if you limit yourself to "current" books, heat death of the universe will happen before you finish reading all the things that were published in the last 12 months.

In order to deal with the tsunami of information, you need to focus. Pick one or two areas that you are really passionate about. I cannot stress enough—this is not a permanent decision. Near as I can tell there are only two decisions we can't reverse, if you decide in a week or a month or a year to move on to something else, great! Take an area or two that really catches your eye and go *deep*. Follow the thought leaders, read the foundational texts, watch the key presentations.

And skim the rest. While it is important to be generally *aware* of the broad strokes of our industry, you cannot be an expert in every single nook and cranny. While you should know there are several types of "NoSQL" databases, you don't have to be the go-to person in your organization on them (unless that is where your passion takes you). If front-end technologies excite you, awesome! Or maybe you prefer infrastructure automation, that's great! The point is, focus on whatever gets you excited.

Follow your instincts. If the topic doesn't speak to you, you will not make the time to study it. You will find almost anything else up to and including rearranging your sock drawer[3] to fill that time. Don't be lured in by an area that seems trendy—let your passion guide you.

The Myth of the Full-Stack Developer

If you've looked at a job posting lately, you've probably seen a reference to the "full-stack developer," someone who can seemingly do it all. The FSD knows five different front-end frameworks, Java, C#, Python, and spent ten years as a DBA with expertise in MySQL, Mongo, and Casandra. They also have seven years of

3 True story.

mobile experience on iOS *and* Android as well as certifications in several agile methodologies.

Keep in mind that most job listings are written by the human resources department and buzzwords make for helpful sieves to filter resumes. But let's be honest—there is no such thing as a FSD today, software is just too complex for that. There was a time when, to paraphrase Alan Kay, developers that were serious about their software built their own hardware. Developers also used to have to write their own operating systems and compilers.[4]

As industries mature, we inevitably specialize and software is no different. A modern application typically involves a JavaScript framework or two, some kind of middleware stack, and one or more database technology. It's unrealistic to think one person can master Angular or React in conjunction with half a dozen other technologies. And that's not even including our build technology, our deployment pipeline, and our infrastructure!

The solution today is to be T-shaped. You need a broad understanding of the technology landscape from front-end frameworks to cloud providers, but you also need depth, or expertise, in one or two areas. Specialization gives you the time necessary to become an expert on a given topic, while breadth gives you an appreciation for the limits of your knowledge. The combination is priceless.

Finding Focus

At this point you might be asking a critical question: How do I know where to invest my time? With an endless stream of new languages, libraries, and techniques it can be very difficult to separate the wheat from the chaff. We need a first-line filter. Years ago, Paul Graham wrote a piece called Java's Cover where he talked about the hacker's radar, stating: "Over time, hackers develop a nose for good (and bad) technology." In the post, Graham gives some useful criteria you can use to judge a something new:

- Is it overhyped? The best technology rarely needs extraneous promotion.
- Does it aim at the lowest common denominator?
- Are there any ulterior motives at play? More than a few technologies exist to fill a product hole.
- Do developers go out of their way to use it or are they required to by management? Passion counts for a lot in the success of a product.

4 We also had to walk uphill to the office both ways and our coffee came out of a percolator instead of a zen like pour over.

- Is there an abundance of accidental complexity? The best technologies allow you to focus on solving business problems, not the care and feeding of a language or framework.

Graham goes on to dismiss Java, saying: "I have a hunch that it won't be a very successful language." But he came to this conclusion having never written a line of Java. In fairness, later he concedes he could be wrong and that he is clearly judging Java solely by its cover. While clearly Graham was wrong about Java, he is absolutely right that we need some kind of sieve to help sort through the immense stack of things we could learn.

You can turn your attention to the community, to the buzz around a given technology, but that can be misleading. As Wayne Gretzky famously said, "I skate to where the puck is going to be, not where it has been." Think about it. Is the technology you're looking at where the puck was or where it will be? It is important to understand *why* people are excited about something. Is the technology an actual improvement or is it just the latest release from today's darling company? In many cases, the community is going to where the proverbial puck was, not where it is going.

You cannot possibly keep up with every twist and turn in the technology space. It is vital to cultivate sources you trust. Technology Radar from ThoughtWorks is a good source. Published semiannually, the Technology Radar filters real-world project experience through the expertise of Technology Advisory Board members and divides technologies across four quadrants: Techniques, Tools, Languages & Frameworks, and Platforms. Within each quadrant, technologies are placed into one of four rings:

Hold
: Something is very new and not ready for widespread adoption or something is at the end of its lifecycle.

Assess
: Worth working through a demo or two, examining if/where it might be suitable in your organization.

Trial
: This technology is ready for a full-on pilot project in your organization.

Adopt
: You should be using the technology today.[5]

5 Or, as one ThoughtWorker said to me, if you aren't using this technology I will make fun of you at the pub.

As a technology matures, it moves through the rings. Items that are new or have moved since the previous Radar are highlighted. Obviously the Radar cannot effectively show *every* technology, over time, items are archived.

Each volume highlights a set of themes that cut across the individual technologies listed. For example, recent themes include the normalization of cloud computing, the increased trust in the blockchain, and the use of open source in China. The themes alone paint a poignant picture of one view of the technology landscape. While it likely includes items that aren't relevant to your organization, every Radar has introduced me to two or three interesting technologies I wasn't following at the time.

Build Your Own Technology Radar

I *strongly* advise you to build your own Technology Radar.[6] Your organization probably has some existing documentation on the various technologies in your shop, probably buried in a spreadsheet or on a long-forgotten wiki. Take what you have, find a whiteboard, and bring along a pile of sticky notes.

Draw up the quadrants and the rings on the whiteboard. Don't be afraid to rename the rings or redefine a quadrant—you might not have enough Techniques to justify an entire section. Remove it or repurpose it for something that is more relevant to your company.

Give everyone a marker and a pile of sticky notes. Write down a technology and place it where you think it belongs on the radar. Do not limit yourself to tools currently in your portfolio, now is an excellent opportunity to spur discussion about that technology you've been researching. Once everyone is finished, you may need to consolidate where there is overlap. For every technology on the board, discuss each one and come to a group consensus about if it belongs and if so, where it should be placed. Rinse repeat.

When you are done, you can take a picture of each quadrant and send it to a wider audience (e.g., your architecture team or senior leadership) for feedback. Iterate as needed and then take the time to share your results with others. You may want to build a more formalized radar using a presentation tool or the ThoughtWorks visualizer. If you prefer, you can always make your own internal version by cloning or forking the visualization tool repository.

In addition to building a tech radar for your company, it is invaluable to build one for yourself. The process can help you formalize what you think is worth your time to explore. You should strive to balance the "cool" factor with employability when you are seeking to diversify your skillset.

6 For more advice on the mechanics, see Build Your Own Radar.

Commit to periodically reviewing both your corporate and personal radars. Once you establish the initial radar, it is far easier to repeat the exercise. Review sessions are a ready-made excuse to have a passionate conversation about new technology! Update, evolve, adapt.

The Technology Radar essentially acts like a canary in the technology coal mine informing readers about topics worth their time and attention. In the same vein, we can use community members as beacons too. Who do you admire in the software field? What are they working on these days? Crafting a presentation or writing a book are not small undertakings, if someone puts that much time and effort into the endeavor, it is for a good reason.

Litmus Tests

Regardless of whether you build a Technology Radar or not, spend some time thinking about what litmus tests you apply to a new technology. You often have to make snap judgments but applying consistent criteria can help you make better choices. Your experience will heavily inform *your* litmus tests but consider these:

- Is it straightforward to write automated tests against solutions based in this technology?
- Does the technology use open formats?
- Are these artifacts amenable to version control? Can I easily diff one version to another?
- Can I automate away repetitive tasks?
- Does this technology integrate into my continuous delivery pipeline?

Cut the Cruft

How many "streams" of information do you have coming into your life right now? How many people do you follow on Twitter, how many podcasts do you subscribe to, how many tabs are open in your browser right now? I used to subscribe to a bunch of magazines[7] but I've finally let almost all of them lapse—I wasn't making the time to read them. This fact became painfully obvious when I went through my magazine pile one weekend and realized I was months and even a year behind in some cases.

7 Yes, magazines still exist.

While that stack of paper may not have been hurting anyone[8] each one had a small "psychic weight." I knew they were there and often felt I was missing some really valuable nugget in those unread pages never mind the waste of paper. By culling the herd, I left myself with a far more manageable set of magazines I actively look forward to reading!

Maybe you aren't overwhelmed with dead trees, but I suspect you have a similar situation with podcasts, blogs, conference videos, and so on. Whatever your preferred streaming mechanism, spend some quality time curating it.

How do we end up with more information than we can digest? Part of it boils down to fear—we are afraid of missing something new/interesting/important.[9] But unless you are living in a sensory deprivation chamber, if something really big happens, you *will* hear about it. You might not be the first to know, but groundbreaking things have a tendency to invade your consciousness.

The A/B Stream

In addition to periodicals, I've also struggled with an overabundance of articles in my RSS reader. One day my feed reader pointed out that I had 15,539 unread items and suggested I "hit the panic button," marking them all read.

At that point I decided I needed to do something about this unmanageable torrent of bits. Now, I am sure that (nearly) every one of those posts had something interesting hidden within, some gem that would make me a better developer, husband, or father. I decided that day to create an A and a B folder chosen for no other reason than A sorts higher than B.

Within the A folder I assembled the roughly two dozen feeds I found myself returning to again and again. These were the authors that consistently wrote pieces I found worth my time. I quickly noticed a few of the people I followed were self-referential; that is, if one of them posted something noteworthy the other two or three would repost it. That allowed me to trim the list even further, allowing some of these people to do the work of editing or curating for me.

Everything else went in the B bucket. From time to time, I dip into the B bucket just to see if anything deserves to be promoted to the premier league (and occasionally I relegate something from the A folder). But I spend almost all my "feed time" in the A list. That isn't to say I read it all end to end every day, some of the things I follow post frequent "newsy" posts, and when I fall too far behind I no longer hesitate to mark all as read.

8 My wife would like to challenge this statement.

9 Call it technical FOMO if you will.

While you might find A/B too binary, having something in between, a purgatory of sorts, can help you decide if a feed deserves your time. Your attention is finite and is the most precious resource you have.

Regardless of the source of the information, make it earn its place in your world. If a podcast isn't interesting anymore, stop listening! Bored with a blog post? Move on to something else. Presenter not bringing it today? Respectfully move to a different session. Better to spend your time well than waste it.

Making a Plan

What do you want to learn? What will it take to accomplish this goal? On the surface, mastering a new framework or programming language can be overwhelming. And it doesn't happen overnight. We need to follow the same advice we give on an Agile project—break down the big things into smaller tasks. Like you've probably heard again and again from project managers, "Plan the work. Work the plan." Let's consider an example. Say you want to learn a new framework. Your plan might look something like this:

1. Set up your development environment (e.g., editor, compiler, other necessary tooling).
2. Create a "Hello World" project to make sure you have everything set up correctly.
3. Work through a more complex tutorial or two (or three) to explore more corners of the framework.
4. Build an application that scratches an itch you have on your project or at home.
5. Summarize what you learned along the way.

On that journey, you will inevitably run into issues, and you will gather links and other resources that helped you address them. Don't throw that work away, capture it! This doesn't have to be anything formal, notes in a text file will suffice. But take the time to write up your thoughts when they are fresh in your mind. They'll not only help you later, but could also help others in your circle. Don't be afraid to publish your notes as a blog, podcast, or screencast!

The start of the new year serves as a "temporal landmark" but it isn't your only opportunity to make a learning plan. If you've decided that you really want to do a deep dive on Clojure, there is no better day than today to begin. Start with a survey of the literature by researching what books are available. A quick search reveals there are several; looking at a few reviews should help narrow that down.

Now take a look at the community. Who works on Clojure? Rich Hickey is a good place to start. Walk his social graph and learn about who else to follow on

your favorite form of social media. Are there any local Clojure meetups? Add it to your schedule and make sure you do some networking at the next meeting. Subscribe to any relevant mailing lists or groups and follow the Clojure project on GitHub.

Don't forget about sources like YouTube and podcasts on the topic. Can you block off an hour a week to watch one or two? While you're at it, you might want to attend Clojure/conj. Depending on your organization's policies, it may take several months to secure funding to attend, so start early.

Last but certainly not least, see if anyone else in your office shares your passion for Clojure. While it can be rewarding to engage in some solo work, learning as a group is incredibly beneficial; a study group or a book club provides accountability. Just as you are less likely to skip a workout scheduled with a friend, if you know three of your colleagues read a chapter for tomorrow, you will be far more likely to do the reading yourself. Group learning gives you an invaluable opportunity to ask questions in a safe environment with people at the same stage of learning as you. Bouncing ideas off one another is a fantastic way to test and validate your understanding of a topic.

The Power of Book Clubs

Over my career, I've led or participated in several book clubs and I am convinced it is one of the most effective ways to learn something new. It isn't hard to get one started, all you need is a few people, a conference room, and a book on the topic you're interested in! Most companies will cover the cost of books and some will also spring for food, you just have to ask.

Groups provide motivation and accountability. When you are on your own, life has a tendency to interfere. A project deadline here, a happy hour there and next thing you know it has been a month since you've done anything on your learning plan. But when you have a group meeting Thursday morning to talk about a chapter, there is a pretty good chance you'll read the chapter as well.

When you start a book club, be prepared to act as the benevolent dictator. You might get lucky and rotate leadership from person to person but someone has to drive the discussion. Don't be afraid to set the schedule and to arrive at the meeting with some questions to spur conversation.

If reading isn't your thing, consider starting a video club instead. Between Safari, YouTube, and Vimeo, you are bound to find something of interest.[10]

10 Though you may need an assist from your security team to access them on your corporate network.

Schedule Individual Learning

Now that you have an area of focus plus a succinct inflow of information, you need to carve out the time to actually consume it. If you don't actively manage your exploration time, it won't happen. Setting up a development environment and walking through a tutorial can take the better part of a day.

You absolutely should schedule meetings on your calendar for learning. I have yet to meet an architect with hours and hours of free time and if you don't block out the time you probably won't get to it. Remember, time is an investment in not just your skillset but the likely future health of your project.

Create a routine and learning will become a habit. For example, you could block out lunch on Thursday for "tech talks" or take a quiet Friday afternoon for some hands-on time with that framework or library you've been dying to test drive.

As an architect, it can be nearly impossible to find contiguous blocks of time. I've long been a proponent of "coffee time." In the morning, with that first cup of coffee, spend 15 to 30 minutes sharpening the proverbial saw (e.g., go through your feeds, check twitter, or visit your favorite technical news sites) before the day gets away from you. Follow this approach day in and day out, and you will be amazed at what a difference it makes.

The Science of When

If I told you there was a virus running rampant in your local school district that substantially reduced tests scores of infected students, how would you react? If you have children, you'd probably want to get them vaccinated immediately and you might even consider removing them from their school. What if I told you it wasn't a virus, but a matter of timing?

In "When: The Scientific Secrets of Perfect Timing," Daniel Pink shows the importance of *when* decisions. For example, on annual standardized tests, Danish students with earlier testing times scored higher than those tested later in the day. We all experience predictable peaks, troughs, and rebounds in our day. Understanding your personal chronotype (lark, night owl, third bird) can substantially impact your success learning new things.

Learn as a Team

Learning doesn't have to happen in isolation, it can (and should) be a team event. Providing regular opportunities for people to explore is vital to the long-term health of your technology portfolio. Make it part of your culture. Google

famously practiced 20% time,[11] which incubated products like Gmail and AdSense. Some companies have Innovation Friday allowing their teams to contribute to open source, experiment, and learn new technologies. It can be a big ask in many organizations to block out an entire day but maybe you can carve out Friday afternoons or propose a regular "Tuesday Tech Talk." It doesn't have to be every week but it is important to establish a routine.

You might also consider regularly delivering architectural briefings. Despite the title, you do not have to be an architect to present! An architectural briefing is just a lightly formalized way of exploring something new and presenting the results to the team.

An architectural briefing should answer the following questions:

- Why should you use X?
- What do you need to know to answer the "why" question?
- What do you need to know to actually use X?

These should be short sessions, 45 to 60 minutes with time at the end for questions. While architectural briefings are not deep dives, they should go beyond the documentation. Everyone should participate, asking questions and bringing their own experiences to the table.

Everyone should be encouraged to present an architectural briefing. Passing the responsibility for exploration around the team encourages everyone to invest in learning. These are also fantastic opportunities for people to flex their presenting muscles. It also provides a relief valve for developers to chase a shiny new technology.

Take Advantage of Dead Spaces

From commutes to waiting for meetings to start, we have pockets of unused time throughout our day. Don't let that time go to waste, use it to help you further your goals. Early in my career, at the start of my day I would print off two or three articles. I would bring them with me from meeting to meeting and, since meetings almost never started on time, I'd use those random few minutes to work through my stack. By the end of the day, I'd generally have those articles read.

If you have a lengthy commute, take advantage of it. If you ride the bus, take the train or carpool, use that time to watch a video or read an article or two. Driving yourself to the office? Listen to a book while you shake your fist at traffic.

11 The practice encouraged every engineer to spend 20% of their time working on passion projects or other work that wasn't related to their daily responsibilities.

Going for a run at lunch?[12] Taking the dog for a walk when you get home from work? Use that time to listen to a podcast. Waiting for your child to finish up karate or soccer practice? Bring your tablet or your laptop and use that hour to explore something new.

I'm not suggesting you adopt a monastic approach to your daily life. Sometimes you just need to zone out for an hour. But small changes are the key to so many things in life. If you carve out an hour here, 20 minutes there, you will be amazed at what you can master.

Don't be afraid to invest in your own skillset. If you were a professional chef, you would bring *your* knives to work. While I believe companies benefit massively from their investments in professional development, sometimes the simplest path is to spend your own cash. Feel passionately about a conference but your company won't pay for it? Cover the costs yourself. Reach out to the organizer and ask about volunteering. Some offer volunteer opportunities that provide reduced cost (or free) passes for a few hours of your time. Use frequent flier miles to cover airfare and see if you can crash on someone's couch during the event. And while you're there, network with other attendees, perhaps now is the right time to switch companies.

We all learn in different ways. You may be a visual learner or you may learn best when you work hands on. Perhaps you immerse yourself in reading material or watch someone live code before you open up an editor. Maybe you prefer a classroom setting with a lively conversation.

Learn how *you* master new things and go that route. There are thousands of tweets, blogs, and articles a few clicks away. In most metropolitan areas, there are more meetups, user groups, and other gatherings than there are days in the month. Conferences, videos, and live online training offer thousands of hours of visual content for your viewing pleasure. Seemingly endless hours of podcasts await those that prefer to listen while they learn.

NOTE

Safari

I highly recommend Safari for its sheer variety of material and learning modalities. From books to videos to live online courses, it is a one-stop shop for technology education. Many companies offer subscriptions to their employees. If yours doesn't, ask them to! If they won't, consider it an investment in your career.

12 I do almost all of my podcast listening while I work out.

Leveraging Your Network

If you can't keep tabs on everything new in your industry, how do you make up for the inevitably massive gaps in your knowledge base? Leverage your professional network. Take advantage of the cumulative knowledge of your social graph. I guarantee your friends and colleagues know things you don't (and vice versa).

None of us can succeed on our own. Asking for help isn't a sign of weakness or incompetence, it is a key learning technique in a complex industry. Networking doesn't come naturally to all of us but the effort pays dividends.

If you aren't sure where to get started, go to a local meetup and make a new friend! Next time you go to a conference, eat lunch with someone you don't know and exchange email addresses. Look around your office—I bet there are some smart people, don't be afraid to invite someone out for a cup of coffee.

Staying Current

After spending however many words extolling the new, as architects it is vital we pay attention to currency. It doesn't get nearly as much attention as introducing an emerging technology, but it is vital you maintain the portfolio you have. Odds are even the least technical person in your company has heard of Heartbleed, Meltdown, or Spectre. And I guarantee your C-level executives are aware of the exploit at the bottom of the Equifax hack.

Step one is simple: have a currency standard. Many organizations have an N or N–1 stance, meaning that before a push to production, any dependencies must be within one release of the current major version. It can be challenging for projects to manage all of the various technologies that make up their application but publishing regular product roadmaps can help focus attention. You can also turn to your Technology Radar.

If you adopt a Laissez-faire attitude you will eventually find yourself with a mountain of technical debt. It is tempting to ignore currency, to pretend it is someone else's problem, but eventually it will be *your* problem. Just as it is far easier to pay off your credit card a little bit at a time, it is far simpler to maintain currency than to fall further and further behind.

Spend time thinking through how your organization will address questions like these:

- What technologies (or versions) do you need to move to Hold on our radar?
- Is there a new version of a core library on the horizon?
- When should you adopt the next release of your primary programing language?

- A vendor just announced end-of-life on a product you use. How will we migrate away from it?
- How much testing do you need to perform on a new version of a core library before you recommend it to your teams?
- What is your mitigation approach when a project falls behind on currency?

Katas

Now that you understand the challenges of staying current with the fast pace of the software industry, take some time to make a plan on how to keep up.

- Find an hour in your week for studying a topic that you are excited about. It can be Tuesday at lunch, Saturday morning, whatever works for you but put it on your calendar.
- Build a technology radar for your company.
- Build a personal technology radar.
- Schedule a book club to work through a new (or old!) technology book.
- Present an architectural briefing on a technology that interests you.
- Attend a local meetup related to a technology you are passionate about. Bonus points, volunteer to give a presentation.

CHAPTER 3

Evaluating Pros and Cons

Every technical decision involves weighing the pros and cons. In this chapter you will learn how to look beyond blind hype to the inevitable downsides of any technical choice. You should consider *all* the consequences of an option to make the best decision for your unique situation.

Evaluating Tradeoffs

> "Programmers know the benefits of everything and the tradeoffs of nothing."
>
> —Rich Hickey

Every choice you make involves tradeoffs. You cannot become so blinded by the benefits of a given technology that you fail to see its drawbacks. Successful architects are capable of seeing past the "everything is wonderful" sheen of a new exciting technology.

Saying No, Gently

Saying no is one of the hardest things you have to do as an architect. When a developer comes to you with an innovative solution it isn't fun to throw cold water on it. But sometimes that is *exactly* what you need to do. But it's important to know how to do it the right way.

A gruff "NO" is rarely the right approach and, in a perfect world, you make the developer think the no was their idea. Often it can be as simple as pointing out another option or making sure the team understands the full implications of their solution.

One time I had a developer put together a feature based on a number of libraries we had no corporate experience with. I applauded his effort and then casually mentioned the approval process we'd have to go through for all of the new libra-

ries he wanted to use. I was willing to put in the effort but reminded him it could take several weeks. Were there any solutions in house we could leverage? Sure enough there were.

Instead of spending cycles working through the approval process, I could focus on more pressing project concerns. We were also able to leverage in-house expertise on the existing libraries, shortening our learning curve. Instead of blazing a new trail through the forest, we followed a well-worn path giving the team more time to deliver functionality.

When evaluating a new technology, don't be afraid to lay out the pros and cons in a simple table, as illustrated in Table 3-1. It doesn't have to be an exhaustive list but taking the time to walk through them with your stakeholders can prevent a poor choice. Documenting the tradeoffs also communicates your rationale to your teams. If you fail to "show your work" people have a tendency to fill the vacuum with their own reasoning—and you may not like what they come up with.

Table 3-1. Pros and cons of a mythical library

Pros	Cons
Open source	No official support options
Reusable components	1.0 release months away
Growing community	Poor documentation
Familiar programming model	Limited test coverage

Every technology has some rough edges. Your job is not to varnish over the limitations of a given technology, but to acknowledge them. They may not be fatal to your use case, but you cannot be sure unless you put in the time to understand, and address, the advantages and disadvantages of a given approach.

You must shed light on the risks inherent in a new technology. By addressing them openly you can determine the best path to manage them.

Some Risks are More Equal Than Others

Not all risks are created equally, some really are more important than others. Give each risk two scores, one for likelihood and one to measure impact. Use a scale from 1–10, with 1 being little impact/unlikely and 10 being massive impact/guaranteed to happen. Multiply the two numbers together and you get a truer sense of the importance of a given risk.

For example, the impact of a sinkhole swallowing your favorite coffee shop is indeed a 10 but the likelihood is probably a 1, whereas the impact of someone being in your favorite easy chair is maybe a 7 while the likelihood could be a 6.

While the sinkhole is indeed an impactful risk, it probably won't happen and you should instead focus your attention on how to secure your preferred seat.

Risks cannot be eliminated completely, but they can be managed. Consider each risk and how you can mitigate it. Table 3-2 shows a simple version of a risk mitigation plan. A risk mitigation plan goes a long way toward quieting dissenting voices, but it can also help focus your thinking. And you might discover a risk that you aren't willing to embrace.

Table 3-2. Risk mitigation plan for X

Risk	Mitigation Plan
Our developers have limited experience with X	Invest in training and mentoring
Latest release of X just went GA	Perform additional regression testing
Concern that X will not work properly with legacy application	Perform proof of concept to validate use case

Ruler for a Day

When you first explore a technology, it is easy to see the pros. Walking through Hello World and a demo or two will give you a sense of how to work with the technology but often it takes more sustained exposure to see where the technology falls short.

The next time someone breathlessly extolls the benefits of their latest passion, a simple thought experiment will help you figure out how long they've been working with the technology. Posit that they are the all powerful ruler of their favored technology, and their word is law. They can finally add that one missing feature they desperately want, no questions asked. But before they can add the missing link, they have to remove something first.

Figuring out what to add is the easy part, but knowing what to remove requires experience. With most technologies, you have to dig deeper to find where the dragons live. This scenario works for everything from an IDE to a programming language to a JavaScript framework. When someone can give you well-reasoned answers to *both* questions, you know they've spent some quality time with the technology in question. If they struggle mightily with the second part of the experiment, politely ask them to dig a bit deeper.

This approach also makes for a fantastic interview question.[1] Their reasoning gives you tremendous insight into how they think about their favorite technologies.

1 Hat tip to Neal Ford for introducing me to this question.

The Technology Hype Cycle

Technology follows a rather predictable cycle. A team runs into a problem they can't solve with the current toolset. They create X, a novel solution that (unsurprisingly) fixes the issue they are trying to work around. A few tweets, a blog post later, maybe a screencast, and before you know it a whole host of developers are convinced that if they don't adopt this shiny new technology, they might as well just shut down their project. In the same way medical students often think they are experiencing the symptoms of the particular disease they are studying at the moment, we become convinced we must adopt whatever is trending.

The hype builds as more and more voices begin to reverberate around the echo chamber. Conference tracks begin to fill up with talks extolling the virtues of X, how it is clearly the answer to whatever ails your current project. Developers start clamoring to include the shiny new thing in their work. Some start to panic about their skillset—everyone is talking about X, I must learn it immediately!

But a funny thing starts to happen. As more and more projects adopt the latest fad, it is inevitably used in a context that it isn't suited for. When NoSQL databases first appeared, many projects blindly adopted them without considering whether they were well suited for their projects. Only after implementation did some projects "discover" their applications weren't great fits for eventual consistency, stale reads, or lost writes. A forced rewrite of your database layer tends to negatively impact project schedules.

Using the wrong tool in the wrong place can cause a few bumps and bruises to a complete project implosion. A project failure or three later, as predictably as spring follows winter, the "anti X" messages emerge. A few tweets, a blog post later, maybe a screencast, and before you know it a whole host of developers are talking about how X nearly destroyed their application. Before long a new technology appears that declares it solves all the problems of X.

Is X really a failure? Probably not. Developers simply misused it. Again, consider NoSQL databases. They were created to solve specific problems for specific applications. *If* your project actually has the characteristics that benefit from a NoSQL database, they work wonders! But when you attempt to use a NoSQL database in an instance where a relational database is a better choice, you will not have a great experience. That does not in anyway invalidate NoSQL.

When a civil engineer designs a bridge, they work with known materials with known properties. A given beam can support so much weight, a span of this length requires this type of foundation. We don't (yet) have those kinds of reusable blocks in software. Granted we've evolved beyond the point of writing our own operating systems and we continue to build useful abstractions from higher level languages that handle many low level programming issues to new server environments that free you from undifferentiated heavy lifting.

The software field finds the limits of a technology by (often unintentionally) pushing beyond them. Like ancient explorers, our maps are crude and filled with dragons. What happens when we sail to the edge of the map? There's only one way to find out. What happens when we try to build this type of app on that type of platform? Let's try it! Channel your inner scientist and perform an experiment:

- Form a hypothesis. Identify a key use case or gap in your knowledge about a given technology.
- Test your hypothesis. Spend a day or two implementing a solution using the technology.
- Learn from your experiment. Evaluate the outcome. Was your hypothesis correct? What should you test next?
- Rinse and repeat.

Proving Your Case

Sometimes the best way to understand a technology's limitations is to "build one to throw away," i.e., perform a proof of concept. Spending a few weeks working through a few representative problems can save you months later. It can also prevent a failed project.

Don't be afraid to use a proof of concept to show that a certain technology is absolutely the wrong choice. Many organizations will make a corporate-level decision and then declare that the new tool must be used for *everything*. Years ago, we spent two sprints proving that a certain tool was ill suited for building a web application. There was no doubt within the project team, but the decision makers wouldn't take our word for it. The shattered carcass of that proof made our point.

The best analogy I can draw for how we know where a technology breaks comes from a classic Calvin and Hobbes cartoon. Calvin's family drives across a bridge with a weight limit sign and Calvin asks his dad how they know how much weight the bridge can hold. His dad answers like a true computer scientist: they drive heavier and heavier vehicles across the bridge until it breaks, then they weigh the last truck and rebuild the bridge marked with the weight limit. Who knew Bill Watterson was a software pioneer!

At the end of the day, a given framework, language, or library is just a tool. The challenge for you is knowing when to pick up which tool.

Picking the Right Tool

If you own your own home, chances are you have a toolbox in the garage. You probably have a mishmash of tools collected over the years, a couple of screw-

drivers, a socket set, a set of pliers or two. But unless woodworking is your hobby, odds are you only have one hammer.

When you hire a carpenter to finish your basement, they show up with a truck full of tools. They have a veritable plethora of hammers. In fact, if they walked into your house professing to use their sledge hammer for everything, you would (rightly) ask them to leave!

But just as a master carpenter knows when to use their favorite framing hammer and when they need to reach for their finish hammer, we need to know when to use the right tool at the right time to solve our problems. We must understand the characteristics of a given problem and which technology is the right fit in that particular instance.

We all have preferences, just as a chef has a favorite knife, we too may reach for a familiar solution. But that doesn't mean we should try to carve a turkey with a paring knife. Guard against the natural tendency to frame every problem in a manner that leads to your preferred technology.

Choosing Wisely

You cannot afford to adopt a new technology simply because "everyone else is doing it." You have to analyze the unique needs of the project in front of you today and make the best decision you can with what you know now. There are several questions you should consider when making a technology choice:

- What does my current project *need*? Beware of gold plating and an excessive list of nice to haves.
- Is this a corporate decision or does it just impact a single application? Enterprise-wide decisions require you to balance a number of competing priorities and agendas.
- Does my organization already have a solution in the same space as this new technology? Having more than one answer to a question isn't a bad thing, but be prepared to provide advice on when to use one over the other.
- How does this technology fit in the overall corporate culture of my company? Never underestimate the impact politics will have on your technology choices.
- How will this technology evolve in the near term? We cannot predict the future but we can make some educated guesses.

Do your homework, get your hands dirty. Leverage your network and industry sources. Do not fall prey to analysis paralysis! Don't worry, whatever you choose

will be wrong over a long enough timeline.[2] At the end of the day, do what is best for your project and evolve over time.

&&!||

As much as we search for *the* answer, in reality the right approach almost always involves an "and" not an "or" especially for large companies. Should we use Java or C#? Which cloud provider should we deploy to? Which JavaScript framework is right for our company? Just as one size does not in fact fit all, it is highly unlikely a single answer will leave you satisfied.

Multiple answers does not imply you should encourage an individual project to adopt four different UI frameworks! Again, providing guidance is your job as an architect. If you have multiple tools that do similar things (and I know you do) make sure your teams understand when they should reach for one or the other. Sometimes the answer boils down to team preference.

Provide guardrails for your teams. Technology free-for-alls generally don't end well. Focus on paved roads. Here is a well-worn path, we know it works, we know how to support it. In this renewed era of polyglot technology stacks, consider a flowchart to help teams walk through a decision.

All of this won't stop teams from paving their own paths. Developers love to reinvent the wheel and skunkworks projects infest every organization. While most of the time these are fairly benign, they can be colossal wastes of time and money.[3] Depending on your corporate culture there isn't a lot you can do about people going rogue. But document the cost and the alternatives and protect your projects as best you can. If all else fails, fall back to a "you build it, you own it" approach. Let the team that created the mess pay for it.

Katas

Now that you understand the value of considering both the positives *and* negatives of a technology, spend some time working on your critical analysis skills.

- Build a flow chart detailing decision points a team should make regarding a set of similar tools.
- Take a technology that you think your company should be using. Create a pros/cons table. Is that technology a better fit than one currently in use in your organization?

2 Today's best practice is tomorrow's antipattern.

3 Case in point, building your own cloud platform. True story.

- Subject your favorite tool to the "ruler for a day test." What would you remove to make room for your killer feature? Why?

CHAPTER 4

Evaluating and Choosing Technologies

Without doubt, at some point you *will* have to evaluate and choose between technologies. You could just throw a dart at the wall or trust your gut but in the long (and short) term it is far better to follow a more objective approach. This chapter will explore what it takes to evaluate a new technology with a set of criteria you can use on your next assessment. It also includes a case study that uses this technique to compare Angular and React.

Evaluation Criteria

There are any number of things we can use as evaluation criteria. Depending on what kind of technology you are exploring, some will make more sense than others. Ultimately you will be guided by your own experience as well as the things that matter most to your organization. The following are a set of criteria I typically use when evaluating a new technology:

Documentation
: Is there any? The era of "the code is the documentation" is over. Is the documentation clear and concise? Is it up to date? Is there enough to get you started as well as enough to keep you going? What you need from the documentation will change dramatically from week two to year two.

Community
: What is the size of the community behind this technology? More importantly, is it thriving? A small but passionate community may be better than a large apathetic one. Does the community welcome new members or does it prefer to make them feel inferior for the first few years? What does the community value? Some languages encourage testing, others fetishize obfuscation.

Committer diversity
: Many open source projects have corporate backers. While that isn't a bad thing, it is important to ask what happens if that company decides this technology isn't core to their business anymore. What happens if the major corporate backer walks away? If the community is large enough, it probably won't have much of an impact. Development may slow somewhat but well-established projects will continue.

Codebase
: You won't get to read the code for most vendor projects but if you are evaluating an open source project, you should spend some time in the source files. Is it readable? Are there tests? Would your team be able to fix an issue if they had to? Could you fork the project if it became necessary?

Testability
: Does the technology in question encourage testing? Does it make it nearly impossible to write automated tests?

Update history
: When was the last commit? When was the last release? You should distinguish between dead projects and mature ones. Check the issue tracker. Are there heaps of open tickets? Are the forums filled with tumbleweeds? The project might be abandoned.

Maturity
: What is the version number? While some months-old projects boast of higher version numbers than decade-old established technologies, be cautious with anything with leading zeros. Newer tools are likely to change frequently and many of those changes will break your code. If there are major changes on the horizon, what is the upgrade path? How quickly are new releases coming out? Can your projects consume those changes at that rate?

Stability
: When a new version comes out, is it solid? Or does it require several point releases before it inspires confidence? In other words, how quickly would you put a new release into production?

Extensibility
: How hard is it to add something? Is there a plugin model? Or are you simply at the mercy of what the committers think matters most?

Support
: Can you purchase a support contract? I know many architects find them worth less than the effort to procure them, many organizations want someone they can call for help. Can you find help online? What do you find on Stack Overflow?

Training
: Can you get training for this technology? If there aren't commercial options available, can you build your own? Are there tutorials and examples you can follow?

Hiring
: What happens if you post a job looking for experience with this technology? Scan through the last set of resumes you received. Does this technology show up anywhere? What do *your* developers think about this technology? Do not underestimate their feelings; developers will vote with their feet. If your developers are avoiding this technology like the plague, you should reevaluate the process that led you to this point.

Corporate fit
: Does this technology align with your corporate culture? What does your company value in a technology? Some technologies will be a breaking change for your company. Will there be any political roadblocks? Does this technology compete with an existing solution in your organization? What guidance will you provide teams on using one over the other?

Security
: Does this technology have any known security issues? Are security vulnerabilities fixed in a timely fashion?

Licensing
: How is this product licensed? If it is open source, how will your legal department react to the specific license used? If it is a commercial product, perform due diligence on the vendor. Are they well funded? Do they have a strong track record? Do they have satisfied customers?

Usage
: Be aware of the lemming effect in software, that tendency to follow the crowd. However, if no one else is using a given technology put in the effort to figure out why. Software archeology of this sort is a mix of clever web searches, sifting through forums and consulting with your professional network. It may be a niche product or an undiscovered gem. Or everyone else might know something you don't yet. Be sure you know which bucket *this* technology falls into.

The Spreadsheet Approach

Spreadsheets make it easy to show how various options fulfill (or don't) a set of criteria. In general, put your options across the top and your criteria down the side. Some criteria may be more equal than others: don't be afraid to weight them. For example you might decide that hiring is a bigger deal than extensibility so you give hiring more emphasis. Often, this is a factor you multiply the score

by to give it a higher weighting in the overall score. I strongly advise the use of Harvey Balls in place of a numeric scale. Harvey Balls are just an ideogram ranging from an empty circle to a filled circle with one quarter increment in between (e.g., ◔ or ◑).

While we can argue over the meaning of "1" an empty circle is unequivocal. If you use 1–5, part of your audience will think a 1 is good even if you repeatedly say a 5 is the top score (or vice versa). If you must use a numeric scale, be consistent. Nothing is as confusing to your readers as a scale that randomly changes.

Tipping the Scale

Resist the temptation to put your thumb on the scale of an evaluation. You can always structure an "evaluation" to prove whatever you want. But in nearly every instance, your audience will notice you stacked the deck. If you have to rig the game, what is the point of an evaluation?

The criteria you choose and the weighting (if any) you apply is completely up to you. In some circumstances the criteria might be defined for you. But in general, you will have to determine the most important aspects to evaluate.

Proof of Concept

While you can always perform a paper-based evaluation, few things beat hands-on experience with a technology. I strongly advise you to acquire first-hand knowledge of anything new. Of course, you will *never* have enough time to work on a proof of concept, you will have to be efficient with your work, but you will have to focus on the aspects that will teach you the most about this technology.

Hello World won't teach you enough and you need to go beyond canned demos. Before starting a proof of concept, identify the areas you need to understand. What are the key use cases? What are your primary concerns? Tune the demo application to hit the major points that will give you confidence to make a decision one way or the other.[1]

A proof of concept should help you identify risks. As an architect, your job is to shed light on the risks, but others generally decide what to do about them. Keep track of the risks along with their impact and likelihood. You can document them in anyway you like: spreadsheet, wiki, whiteboard, etc. Every project does it their own way.

1 It should go without saying that the vendor should *not* perform the evaluation for you.

NOTE

Embracing Risk

Risks are *not* inherently bad. Every time you cross the street, drive a car, or go for a bike ride, you are taking some risk. We embrace risk in our daily lives[2] just by getting out of bed.

Risks should not paralyze you, but you have to manage them appropriately. You may decide you are unwilling to accept a risk or that you want to transfer that risk to a third party, but there is nothing you can do to remove all risk from a project. You can, and should, proactively identify them and work to mitigate them.

Politics

Many people get into software because it is an objective field. Your code compiles or it doesn't, your tests pass or they don't. But there is ample room for ambiguity in software.[3] While I wish it were not so, politics will play a role in almost every technology decision you are a part of. It doesn't matter how big or small your company is, you should prepare to deal with the nontechnical aspects of an evaluation.

Gauge the political winds at your organization. Talk with your peers, especially those with the longest tenure in your company. What technologies have had the roughest roads to adoption? Why? Consider the legacy technology you are trying to replace or augment. Who might see their role diminished in this new world order? How will they react to that? Can you identify who the most influential people are in your company? Aligning them to your goals can work miracles. Will the decision makers be receptive to the technology you want to introduce? If they aren't, you might be able to change their minds…but it will involve retail politics. Recruit allies. Can you tie this technology choice back to a pain point in the organization, even if it is a bit of a stretch? Do everything you can to align the technology to executive-level goals to ensure success. For example, years ago a new CIO made it clear that release weekends included far too much drama and he wanted to see improvement. A few of us used that as an opportunity to reintroduce a set of code quality initiatives we'd been advocating with limited success. By connecting that drive back to the CIO's goal of smoother releases, we made significant progress.

Every so often, the right answer is to just do it and let the chips fall where they may. Channel your inner Grace Hopper who rather famously opined: "It's easier to ask forgiveness than it is to get permission." Just be aware of your organization

2 Some more than others—I'm looking at you skydivers.

3 Just go ask one of your business analysts.

and your place within it. If you are already in a precarious situation, watch your step.

Evaluation in Name Only

At some point in your career you will fall into an "evaluation in name only," which has many of the same trappings of an actual evaluation. Sometimes the signs are hard to read but eventually they become unmistakeable. Negative results may be buried. A risk list might be moved to a less prominent location on the wiki. Outcomes from a proof of concept might be heavily couched.

As much as you may want to perform an unbiased appraisal, there may be vested interests in the outcome. You may or may not be aware of these motivations. In many of these instances, an "evaluation" is simply cover for a decision that was already made. Your work will be spun to prove whatever narrative has already been chosen.

Many organizations have very centralized (and hierarchical) decision-making structures. In most cases, you won't really know what went into the outcome of the decision.

If you are highly influential, you might be able to alter the outcome, but don't be surprised if you can't. At the end of the day there is only so much *you* can control. Don't overthink it, but try to understand the context of the decision and don't be shocked if you don't agree with it.

What you do next is a very personal decision. You may decide you can go along with the scripted result, in which case you should consider how you can insulate your projects from the technology. Perhaps another layer of indirection is in order. Add a proxy layer that hides the details of the technology from your application. If (or when) the time comes to replace the offensive library, you will only need to rework the proxy.

Some corporate cultures expect commitment to a decision but they may encourage a "dissent memo" where you can outline your disagreements. Simply having an outlet for your thoughts can be valuable.

Depending on how strongly you feel, it may make sense to explore other opportunities. Working for a company when you disagree with the corporate direction is challenging at best. If you aren't happy, it will show up in your work (and your personal life as well). Never forget that you are responsible for your career.

Case Study: Angular Versus React

How might this approach look applied to Angular and React? First things first, any comparison is highly subjective. There is no magic formula! Some people actually reject this comparison outright because "Angular is a framework and

React is a library." Pedantic yes, but don't be surprised when a colleague rejects your hard work based on a type coercion error.

Documentation is a plus for both tools. Both have good tutorials and extensive guides. The communities are largely comparable with Angular being slightly larger. Though both are backed by large companies, they have strong committer diversity. React's extensive contributing page is excellent.

React and Angular have clear codebases with a strong focus on testing and testability. Both are updated regularly although the Angular 2 conversion was disruptive.[4] Angular has a regular upgrade cadence with a deprecation period. A predictable schedule helps planning but can your projects consume changes at that rate?

Both are mature and stable. React often involves stitching together multiple libraries that in turn move at their own pace. In terms of extensibility, Angular allows you to swap out just about every component with something else if you wish and React is built around projects crafting their own lightsabers as it were.

As massive open source projects, each returns a plethora of search results. For better or worse, Stack Overflow has *far* more Angular questions. Is that good or bad? It may indicate a larger community, but it also might mean developers struggle to understand Angular.

How Many Questions Should We Find on Stack Overflow?

Stack Overflow is a fantastic resource for developers. Search on almost any topic and you will find good answers and strong opinions. When evaluating a new technology, spend some research time on Stack Overflow.

Finding no questions (or no answers to very few questions) should give you pause. A dearth of results indicates the technology is either very new or not popular enough to warrant a community yet. An endless list of questions about a library should also cause your spidey sense to tingle. It may indicate insufficient documentation or a framework that has far too many issues to be trusted in production.

Ideally, we want a "Goldilocks" set of questions. Enough to ensure us we can find help if we need it, but not so many as to give us acid reflux about the health of the technology.

Online and live training abound for both Angular and React. Hiring for either should not be difficult though you may find more Angular experience on resumes. Both are widely used with massive properties backing them.

4 Enough so that one friend said "Angular 2 broke me…HARD!" Your mileage may vary.

All in all, this is largely a toss up! While my subjective reasoning gives a slight edge to Angular, I *prefer* React. That said, it is hard to go wrong either way. Your project might have a very different set of needs and priorities, which is exactly why *you* need to perform your own evaluation before making a decision. While you should spend time exploring the choices others made, nothing replaces your own experience and intuition.

What Should You Choose?

There is no one answer to which technology you should choose.[5] Are you making a decision for *your* project or your entire company? The larger the impact of the choice the more consensus you will need to achieve. In these situations, spend the necessary time with the interested stakeholders to earn their buy-in. Overpowering them rarely works and usually just breeds resentment. Overcommunicate why you think your choice is the right one and be prepared to talk through it.

For project-level decisions, you still need to make sure your team understands why you went a certain direction. They may not agree with your decision, but you should be very open and transparent about how you arrived at it. Do your homework. Be strategic!

Katas

Now that you understand the importance of a thorough evaluation as well as the impact politics can have on a technology decision, put that knowledge into practice in *your* world.

- Compare and contrast two similar technologies.
  - What criteria did you choose? Why?
  - Which one came out on top? Why?
- Consider a technology you would like to introduce to your organization:
  - What political roadblocks exist?
  - How could you manage the politics of that technology?

5 Beyond "it depends" at least.

CHAPTER 5

Introducing Technologies

Once you've chosen a new technology, it is time to bring it into your organization. While you might have thought choosing a technology was the hard part, adopting it can be even more challenging.

Mitigating Change

Change is hard, and habits are hard to break. Have you ever laminated your New Year's Resolutions? It makes them far easier to reuse the following year. As challenging as it is to exercise more or eat better, it can be even more difficult to convince a whole organization to adopt any changes.

Change is also hard to maintain. We inevitably get worse before we get better and during what Seth Godin refers to as the dip it is very tempting to backslide. Consider a new exercise regime. After a lengthy layoff, that first trip to the gym is going to leave you feeling pretty sore. It is very tempting to just put your feet up on the couch and enjoy a recovery shake. But if you push through the discomfort and continue exercising, the soreness abates. Eventually your body hurts when you *don't* go for a bike ride or spend some quality time in the weight room.

The same phenomena exists in an organization. Introducing something new will inevitably lead to a period of discomfort. Some fear their job will be fundamentally changed or even eliminated. Others are worried their hardearned expertise will no longer be valued. Still others won't be happy with the process or the outcome and will reject the new thing on principle. Many people will eventually come around, but it won't happen overnight.

As a shepherd of the new, it is *your* job to work with people to help them understand why the change is needed and how they will benefit from it. Change can overwhelm people so be cautious with how many things you change at once. It is time for you to unleash your influence skills.

Crawl, Walk, Run

Change takes time. Be patient. Don't forget the journey you've taken to get to the point where you now embrace this shiny new technology. We all start at zero when it comes to learning a new language, framework, or tool. Never lose sight of the fact that your team isn't where you are today; they're where you were when you first saw this technology. They need to take their own path to get to you, but you can help them get there faster by avoiding the potholes and dead ends inherent in learning something new. For example, the documentation might be missing a key setup step or the process for integrating into your favorite IDE or text editor. You can save your peers hours of frustration by sharing your hard-earned experience.

In the same way that we aren't born running, we have to progress from crawling to walking to sprinting with a new technology. Consider any technology you love today. Hard as it may be to imagine a time without it, there was a moment where you didn't use it. And it is possible you may have actively disliked it!

I had this exact experience with Git. I use Git almost daily.[1] Today, I can't imagine using anything else for version control. But looking back, I didn't always feel this way. In fact, my first exposure to Git was a rather condescending tech talk that essentially said "if you're not using this set of shell scripts, you are an idiot." As a very satisfied Subversion user, I did not find this argument persuasive.

Time passed and a friend of mine created a conference talk on Git. As he is one of my favorite presenters, I sat in on a rendition. I was amazed at what Git could do. The power, the speed, the flexibility. I was starting to understand what the buzz was about. I began experimenting more and more with Git at home and even tried to figure out how to use it at work.[2] Over time I grew more and more convinced of the superiority of Git for version control. After learning about its ability to perform fine-grained commits, the incredibly powerful command-line interface, and the utility of GitHub, I am a convert.

Which takes us to today.[3] Frankly I can't imagine a project without Git. Using anything else would just feel wrong. I would constantly miss any number of things be it bisects, simple branching, or the ability to easily work offline.[4] But I remind myself that I didn't always feel this way especially when I'm introducing Git to someone. I remember my journey and try to keep it in mind when trying to influence others.

---

1 The very book you are reading lives in Git.

2 One of my friends even managed to inject it into his workflow thanks to the ability to sync between Git and SVN.

3 I reserve the right to change my mind at an undisclosed point in the future.

4 Like when I made edits to this book while flying over the Atlantic!

The Hammer Versus the Ninja

When it comes to trying to change someone's opinion, there are two primary options: you can use the hammer or the ninja. You can *order* someone to change or you can convince them the change was their idea. If you've ever been in a relationship with another human being (or frankly tried to train a pet) you know you can't really change another person. You can influence, you can nudge, but force rarely works.

The best way I can describe the difference between these approaches is by telling you a story from my son's early childhood. The daycare center we took our son to had a policy that your child was supposed to wash their hands before joining the proverbial general population. Kids are often walking petri dishes and good hygiene practices helps tamp down the inevitable spread of colds, flu, and other maladies.

As a dutiful parent, when we would arrive in the morning I would remind my son he needed to wash his hands. He would simply say "no," an experience I'm sure many parents can relate to. Needless to say, at that early hour, without the benefit of espresso, I did not appreciate his response. I'd put on my "dad hat" and remind him of my role in the relationship: "Everett, I'm your father, go wash your hands." Unsurprisingly, he would simply cross his arms and say "no, never."

Now I'm ready to deploy the foot (aka use the hammer) and escalate further. Luckily, my wife would intervene and prove she is definitely more adept at parenting. She would look at our son and say "Everett, I bet I can get *my* hands washed before you can." She would take one step toward the sink (like a ninja) and he would run over and start washing his hands gleefully saying, "I'm going to beat you Mommie!" Yes, yes you will.

If you want your colleagues to embrace your new technology, you are far better off taking the time to convince people it was their idea rather than trying to just force everyone to adopt it. To do so, you'll need to channel your inner ninja. Many years ago, I attempted to bring Groovy into a Java shop. I thought it would be a very natural extension to the toolset, but it was seen by many as a "weird workaround." Instead of pounding the table, I had one of our interns build a small program in Groovy.

In just a few days, he had a working prototype that far *exceeded* our original goals. I then had him reimplement one of our utilities in Groovy. In two days, he had duplicated what had taken another intern two weeks. And he built a helpful GUI on top of it. He then demoed his work to several key engineers who didn't know he had used Groovy. They were blown away and several started to leverage Groovy in their work with more than one taking credit for "introducing" it.

The Importance of Framing

Several years ago, a group of us attempted to bring GitHub Enterprise into our organization. We started from the premise that GitHub would ultimately be a better, more modern source code management tool. And while we had a thorough explanation as to why Git would be a superior alternative, we did not fully appreciate the gravity of the existing SCM tool. Our pitch was rejected.

An influential senior leader heard about our plight and decided to take a different tack. Instead of describing GitHub Enterprise as a *replacement* for the existing tool, he positioned it as a solution to a different set of issues. Like any organization we suffered from multiple solutions for a given problem largely because developers had no idea someone in another department had already solved it. He offered GitHub Enterprise as the solution, describing it as our code collaboration tool. Since it was solving a novel problem (and one our existing tool had no answer to), it was approved.

The moral of the story is how you *position* a technology matters a great deal, especially if you have an entrenched solution already in place.

Exerting Influence

To a certain extent, you are going to need to practice some retail politicking. Look around your organization—who are the most influential people? Recruit them to your cause! Take the time to reach out to them and discuss what you are doing and why you think it is the right choice. Keep them involved. It can be as simple as having a face-to-face conversation over a cup of coffee. Employ one (or several) of the following techniques:

- Make sure you have done your research and that you have a sound story. Use sources you know your audience trusts.
- Approach the conversation as equals relying on the strength of your reputation.[5]
- If you haven't been with the company long enough to have a reputation yet, bring along someone who does.
- Recruit credible allies to help bolster your position.
- Look for common ground and show how your technology can help the influencer solve a problem near and dear to them.
- Be prepared to help them further their goals. Reciprocity is a powerful tool!

5 Not sure what your reputation is? Ask. You need to know how you are perceived in your organization.

Odds are, you won't have direct "line of sight" to the ultimate decision makers, so you will have to practice influencing the influencers. You probably don't have regular meetings with the CIO, but if you can meet with two or three of her direct reports on a regular basis, you can accomplish plenty.

Developing a network within your company is vital. Who can speak for you when you aren't in the room? Don't be afraid to reach out to people within your company. Invite them out for coffee or lunch. Ask them if you can setup a regular 1-1 meeting. You'll be amazed at how people react when you ask them for their opinion!

Decision makers have trusted sources, and a little digging will show you who they are. Don't just follow the boxes and lines on the org chart either; you'd be surprised how much influence someone's racquetball partner can wield. Put in the time with the right people and you will have a far smoother road bringing that new technology to the forefront.

You also need to understand the nuance of your particular organization. What resonates? Some companies are driven by cost savings. Others care far more about developer productivity or speed to market. How can you shape your message to align with leadership's high-level goals? But remember, some technologies are essentially breaking changes to a company's culture. Is this one? If it is, be prepared for the organization to fight back. For example, bringing a heavy weight "enterprise" software solution into a startup likely won't end well.

Navigating Resistance

Some people will see your new approach as a threat to their reputation and they will throw fear, uncertainty, and doubt (FUD) at you and your technology. How will they attack it? How will you counter that attack? The following approaches have a track record of success:

- Outline the benefits of your technology and walk them through your decision making. Often people overreact because they think you made an arbitrary choice.
- Attempt to find common ground. Listen to their concerns—where is the reticence coming from? They will likely have some legitimate issues, don't dismiss them!
- Avoid aggression. If you push people their instinct is usually to push back harder.
- Check your communication style. Are you being respectful? Or are you being condescending? You may never have taken a college class on conversation, but as a senior technologist, your ability to communicate will dictate outcomes. Don't be afraid to take that internal workshop on presentations.

- Seek out feedback from trusted sources on your approach. What can you do better?

In some situations, you may have to embrace your inner Grace Hopper and just do it. People rarely argue with success! Understand the political realities of your situation but sometimes the only way to convince people is by proving it.

Marketing Your Ideas

You will have to market your new technology. While this may induce some panic, it doesn't have to be complicated. Try some (or all!) of the following:

Book clubs
: Book clubs are some of the best ways to introduce new ideas into an organization. You don't need much to get started: block out a conference room, pick a book, and invite some like-minded souls. Most companies will willingly furnish the books and many will even supply breakfast/lunch/dinner. Again, be prepared to drive the discussion but don't underestimate the power of a set of allies that can help you adopt a new technology.

Lunch and learn sessions
: Hosting regular "tech talks" can go a long way toward driving innovation in a company, and a recurring schedule helps cement the notion of continual learning. "Lunch and learns" do not have to be high ceremony events. A teleconference line, a screen-sharing application, and you're set. Consider starting with a smaller audience before you try to take things company wide and do not underestimate the effort of securing speakers! These are excellent opportunities for people to gain valuable experience presenting as well.

101/201/301 sessions
: These more advanced sessions are really a continuation of a regular Lunch and Learn but are generally targeted at a specific technology. As the naming implies, you want sessions that cover the basics through the advanced concepts. Be prepared to repeat 101 and 201 sessions regularly, especially in larger companies. Providing people ample opportunities to ask questions helps with the stock fear-driven response many initially exhibit.

Day in the life
: People often resist change out of fear. They look at the new technology and are concerned about its impact on their daily life. Take the time to meet with affected groups and talk to them about what they do now and how that may evolve in the future. In most cases, the day-in day-out won't change that drastically—it might just involve using a different tool. In many cases you will have materially improved their 9–5! Walking people through this eases the anxiety surrounding a new technology.

Meetups
: Have a new space that you are really proud of? Want to attract talent to your new initiative? Host a meetup. Some companies are very resistant to allowing anyone on campus, but it can be an outstanding recruiting technique. Your own employees are also more likely to attend an event down the hall! You can volunteer to give a presentation but even just the grassroots opportunities to discuss your technology evolution can change the culture dramatically.

Internal conferences
: For truly foundational changes to your technology ecosystem, you may want to undertake the effort to host your own internal conference dedicated to the topic. Do *not* underestimate the effort involved in even a small single track event. That caveat aside, an internal conference is a powerful tool of change. Give your team ample time to plan and prepare. Determine themes, consider having booths and demos, and don't forget to order cookies. Work with internal presenters as well as developer advocates from companies in the space.

Hackathons
: In nearly every industry, companies are embracing the hackathon. Running anywhere from 24 hours straight to a week, the idea is simple: give teams an opportunity to create, unconstrained by typical corporate bureaucracy. Block out a large enough space to house everyone, make sure there is ample power, WiFi, food, and coffee, and step back. I have participated in several hackathons and have always been blown away by the results. You will be amazed at what people can accomplish in such a short time.

Again, you will have to put in some time and effort marketing your new technology. It doesn't have to be flashy or formal, but it is a vital step.

Katas

Introducing new technologies isn't easy! Try one (or more) of the following exercises to help you advance your agenda:

- Volunteer to give a lunch and learn session on a promising new technology you have identified.
- Start a book club.
- Identify an influential person in your company. Offer to buy them a cup of coffee.
- Think about a technology you rely on everyday.
  - How would you introduce it to someone who was unfamiliar with it?

— What could you do to facilitate their learning journey?

- Spend an hour this week working on your professional network.
  — Reach out to someone you'd like to know better.
  — Grab lunch with a peer.
  — Go to that meetup you've been meaning to attend.

CHAPTER 6

Maintaining Technologies

Now that you've brought a technology into your organization, you have to maintain it. Using quality attributes, you can assess the architecturally significant requirements that will guide your decision-making process. But how do you maintain these quality attributes as your system inevitably evolves?

Quality Attributes

Quality attributes represent the architecturally significant requirements of a system. Some refer to them as the nonfunctional requirements, quality goals, constraints, cross-functional requirements, or quality-of-service goals. Colloquially they are referred to as the "ilities" since many share that suffix. However you refer to them, as architects, managing the quality attributes of a system is one of our most important jobs.

Your customers focus intently on the functionality of their system—which makes sense, it is the part of the application they see and interact with on a regular basis. Obviously you have to meet their requirements for the system, but as an architect you have to look beyond the immediate requirements of the system to understand the bigger picture. You need to consider the service-level objectives of the system.

There are dozens and dozens of quality attributes you could consider for a given project. Which ones matter most depends a great deal upon the type of software you are building. Some common quality attributes[1] include:

1 And no, they don't all end in ility.

- Maintainability
- Scalability
- Reliability
- Security
- Deployability
- Simplicity
- Usability
- Compatibility
- Fault tolerance
- Modularity

The list goes on and on! Your challenge is figuring out which quality attributes matter most on a given project and balancing them against one another.

As much as you might like to turn every knob to eleven, you can't. Some quality attributes have inverse relationships; if you maximize one, you by definition minimize another. Security and usability tend to collide for example. Ideally passwords would be 20-plus characters, include numbers, symbols, and a mix of upper- and lowercase letters, but the average human has no hope of memorizing a random string like that, which is why so many people have a cheatsheet taped somewhere near their computer.

You must balance the various forces at play in your applications. At the same time you have to educate your various stakeholders about the importance of the balance you've struck. Some quality attributes are obvious to your customers; if the system can't support their projected growth, that tends to be visible. If a competitor was recently hacked, you'll have an easier time discussing security and currency. The more "obvious" quality attributes tend to be easier conversations.

But not every ility is so easily understood by stakeholders. Take maintainability and simplicity. Unless you work on your own car, you probably don't care how easy (or hard) it is to change out the spark plugs. If we determine that these harder-to-see quality attributes are key to the success of our system, we have to practice influence.

Take the time to outline the benefits of your approach. Avoid techno jargon, shape the message to your customer's language. Look for common ground. Listen to their concerns and above all else avoid aggression because most people will just push back harder in response. We talked about influence in Chapter 5 and the same approach applies here as well. It can be hard to convince people but as a former manager used to tell me, it comes with the paycheck.

A Rose By Any Other Name…

For much of my career, I referred to the ilities as the nonfunctional requirements. And that term is still appropriate! However, a colleague convinced me there was a better phrase. We were walking through the architecturally significant requirements on an application when he made a great point. Most customers hear the "nonfunctional" part of the term and decide immediately that they only want functional requirements, stopping the conversation. But we have to convince them of the importance of something they can't readily see. Framing the quality goals as something nonfunctional implies they aren't required for the system to work.

If you refer to the same concept as a quality attribute, you change the tone of the discussion. Now you are framing the conversation around the *quality* of their functional needs. Yes, the system needs to do X, Y, and Z but with what quality of service? Your mileage may vary, but in my experience, quality attributes resonate with a wider variety of stakeholders.

Discovering Quality Attributes

How do you find quality attributes? To a certain extent, you can just work through the requirements looking for interesting bits. For example, take the following fictional bullet points about the iDon'tDrive system that supports self-driving cars:

- Car "phones home" on a regular basis.
- Sends a standard data payload including VIN.
- Demand is extremely high.
- Expect millions of cars on the road in the next 3 years.
- Marketing is full of great ideas.
- System must be available 24x7.
- Fleet generates a lot of information that can be mined.
- Data must be anonymized.
- Ensure only authorized users have access.
- Audit access to customer data.
- Push recall/update information to customers.

What words or phrases jump out at you? What makes your architect sense tingle? Perhaps the following:

- Demand is extremely high
- Millions of cars
- Available 24x7
- A lot of information
- Only authorized users
- Audit access

At first blush, it seems like auditability, availability, security, and usability will be key to this system. But you need to dig a bit deeper and explore the nuances of these (admittedly) vague requirements.[2]

Some of those terms are pretty self-explanatory; the system needs to be available 24 hours a day, 7 days a week, and millions of cars. But more than a few of these phrases will require you to investigate deeper. For example, what does extremely high demand mean? Thousands, hundreds of thousands, millions? Depending on who you ask, "a lot" can have drastically different definitions. Do you *currently* have millions of cars on the road or is that a stretch goal for year 5? What do you mean by authorized access? How much information do you need to audit?

It is important that you be realistic when you analyze the quality attributes of a system. Every business person is convinced that their idea will result in world domination. While you might love to have tens of millions of customers, you shouldn't overbuild capacity just to feed someone's ego. Perform your due diligence. What is the typical growth rate? What does the market expect? What happens if you aim too high or too low? How would you correct course?

Ranking Quality Attributes

Now that you have a list of quality attributes that matter for your current situation, you need to rank them. Before you fall victim to analysis paralysis, understand that any ranking is highly subjective. The point is not to produce the "right" answer on your first attempt but to have an artifact you can share with your peers and stakeholders. The discussion around your ranking is invaluable.

Once you have your initial thoughts worked out, schedule some time (an hour or two should suffice) to review with your peers. This could be a formal meeting or just a set of one-off conversations, whatever works best in your organization. Feedback will hone your thinking and most likely result in a some refactoring. That is a feature not a bug.

2 Let's be honest, many requirements are vague and unconvincing.

It is also important to realize the ranking may change based on the perspective of the system. For example, if you are a user of the self-driving car, you might end up with this ranking:

1. Usability
2. Availability
3. Reliability
4. Security

Again, this is a subjective list but it is a reasonable starting point. It is unlikely your users will read a manual and if your system isn't available they will find a different option. Reliability could arguably be pushed up the list but that is almost table stakes for this application. While security is important, it probably isn't a major factor for most users.

If you look at the system from the point of view of a service center, you might arrive at this list:

1. Security
2. Auditability
3. Efficiency
4. Usability

For employees, I've pushed usability down the list, not because it is unimportant but because I need to emphasize other factors. You could take service center employees off the floor for a few hours of training if you had to but you must prevent unauthorized access to the system. Notice the inclusion of efficiency. If you can eliminate a few clicks on every interaction, you can save the staff minutes, which add up quickly.

Consider adding a comment to each quality attribute describing why you ranked it the way you did. A sentence or two can help your stakeholders understand your reasoning. For example, see Table 6-1.

Table 6-1. Quality attributes for the iDon'tDrive system

Rank	Quality Attribute	Comments
1	Usability	Customers will not read a manual on how to use the iDon'tDrive system.
2	Availability	User must be able to summon a car 24x7.
3	Security	User must be confident that only they can access their account.
4	Reliability	System must work as expected when called upon to maintain confidence in the system.

It can also be beneficial to have a list of quality attributes that do *not* influence your decisions on the application. While we often focus on our goals, it can be equally as enlightening to list "not goals." Having a list of things you are not trying to accomplish can help during the inevitable "what if" discussions that accompany every project. Pointing to a list of things you aren't trying to solve can be a powerful reminder to everyone on the team.

Documenting Quality Attributes

Do not overcomplicate how you choose to document quality attributes. When in doubt, do the simple thing. Some architects use mind maps, which can help emphasize the connections between various competing priorities.

Before you use a piece of software to capture your thoughts, make sure you understand the implications. Can you share your map with colleagues that don't have the software package installed? Can you (reasonably) print out your map? Don't forget to factor in a learning curve with the tool. If a piece of software makes it harder than it has to be to capture your thoughts, move on to something simpler.

Don't shun a simple list. It may seem too basic but a table or bulleted list is often your best bet. Easy to build and easy to share, don't overlook an obvious approach.

Connecting Quality Attributes to Decisions

Once you have iterated on your ranked list of quality attributes, you should think about the implications on the system and consider the resulting architectural decisions. How will you support that particular quality attribute? What action or approach will you take to ensure the system will meet your needs?

When connecting quality attributes to decisions, we might end up with a table that looks something like Table 6-2.

Table 6-2. Architectural decisions for the iDon'tDrive system

Rank	Quality Attribute	Decision(s)
1	Usability	UX designers will be engaged and ensure the design requires no training to use.
2	Availability	System will be geographically dispersed across multiple data centers. Zero downtime deploys will be utilized.
3	Security	Standard three-zone security will be employed. System will encrypt personally identifiable information and follow all security standards.
4	Reliability	System will be geographically dispersed across multiple data centers.

Establishing Principles

As an architect, you know you cannot be in all places at all times. When I was a developer, I denounced "driveby architects" who rarely visited the project room, and when they did, lacked context. What I did not appreciate then was how thinly we've spread architects in most organizations; there just aren't enough to go around. Make no mistake: whether they have the title or not, people are making architecturally significant decisions daily on your projects.[3]

As an architect you can't be involved in every decision but you can establish principles, guideposts, guardrails, and north stars. You can create the environment within which your projects can thrive. But how do you know if projects are following your principles?

At some point in your education you were probably introduced to the second law of thermodynamics.[4] For those of you that offloaded that particular lesson to make room for other information, the second law of thermodynamics tells us the universe really wants to be disordered. And if you've been on a software project for any amount of time, you know our industry is not immune.

You've gone through the thoughtful effort to establish an architecture, but how do you maintain it? Architecture is often defined as the decisions that are hard to change or the decisions we wish we got right. But at the same time, we know change is inevitable and we cannot predict the future.[5]

Evolutionary Architecture

Maybe we should change our assumptions. Instead of agonizing over big upfront decisions, what if we designed our architectures expecting things to change? That approach is the central thesis of the book *Building Evolutionary Architectures* (O'Reilly).

The goal isn't to get everything right from the start. You must change the way you think about your systems and you must embrace change. Instead of a typical functional deployment, you might focus on deploying components and enabling features via toggles. You can now modify your application incrementally and perform hypothesis-driven development.

As the application is refactored in response to current needs, you need to assure yourself you haven't violated a central architectural tenet of the system. Knowing that you can't be involved with each and every line of code, how do you know if a solution violates part of the architecture? Enter the fitness function.

---

3 You'll know if they're making good choices in a year or two. Good luck.

4 Otherwise known as a teenager's bedroom.

5 Beyond the fact that it will be different than today.

Fitness Functions

The concept of fitness functions comes from evolutionary computing. When an algorithm mutates, you ask was this mutation a success? In other words, are you closer to or further from your goal? You can apply the same idea to software architecture by asking if you have violated a key quality attribute. Fitness functions capture and preserve the important architectural characteristics of a system.

First, you need to identify those key measures for project success. Take a look at your ranked list of quality attributes. How could you measure them for your application? Don't let the ability to measure something dictate too much. Just because you can measure it doesn't mean it matters![6] Once you've identified these metrics, you can set goals, or if you prefer, service-level objectives.

Now that you have a goal, you can create a fitness function. Basically, you are creating a set of tests you execute to validate your architecture. Does this particular design get you closer to your objectives? Did that refactoring result in a violation of one of your goals? For example, you could establish the following fitness functions:

- All service calls must respond within 100 ms.
- Cyclomatic complexity shall not exceed X.
- Hard failure of an application will spin up a new instance.

Fitness functions remind you what is important in your architecture. They inform your thinking about tradeoffs. Instead of baseless proclamations, you can have a discussion about the impact of a given change to the system. Does this particular mutation retain the balance you are trying to strike?

Some fitness functions are easy to create; others can be more challenging. For example, ensuring the cyclomatic complexity of your code doesn't rise above a small single digit number is a trivial thing to write and execute. Verifying that your application responds correctly to the loss of an availability zone can be trickier. The effort is worth it. In the same way a suite of unit tests frees your development team to refactor at will, knowing you haven't violated a key architectural principle is priceless as your application naturally evolves.

There are several things to consider when crafting fitness functions. Ideally you would automate all of your fitness functions, executing them as part of your deployment pipeline, but some will have to be manual. You need to consider how frequently your fitness functions run as well. Some fitness functions should exe-

6 Lines of code, I'm looking at you.

cute on every checkin; others will be triggered by various events such as passing a testing gate.

Certain characteristics of your architecture have to be tested in isolation while others require a more holistic approach. While you can test the individual response time for every service in your system separately, ensuring a given business transaction completes in a reasonable amount of time requires the execution of multiple services in concert.

For things that must be tested together you won't be able to test every possible combination. Be selective, let the value of the architectural characteristic lead you. Some fitness functions have a static result—they either pass or they fail—and others have a shifting definition of success.

Identify fitness functions as early as you can. The discussion of the tradeoffs around the fitness functions and quality attributes is invaluable to your understanding of the system. Doing so helps you prioritize features and may lead you to break the system up to isolate certain features. You can't know everything upfront, and fitness functions will emerge as the system changes. You should evolve your fitness functions along with your application.

It can be helpful to classify fitness functions as follows:

- Key—critical decisions.
- Relevant—considered but unlikely to influence the architecture.
- Not Relevant—won't impact our decisions.

Again, identifying things that are *not* relevant to this project can be very powerful.

Keep your fitness functions visible. The entire team should be familiar with them. You should also regularly review your fitness functions aiming for at least an annual check. Are they still relevant? Are there new dimensions you need to track? Are there better ways of measuring or testing your current fitness functions? Iterate as necessary.

Katas

Introducing a new technology is only part of the process. You have to be sure you are able to maintain your solutions during the lifecycle of your application. Take some time now to try one of the following katas:

- Identify the quality attributes for your current system. Rank them.
- Discuss your ranking with a colleague.

- Identify as many fitness functions as you can that test the most important quality attributes on your current project.
- Add at least one fitness function to your pipeline.

CHAPTER 7
Conclusion

You've covered a lot of ground over the pages of this book! Whether you are an architect, a tech lead, a senior developer, or a junior developer, I hope you understand the odyssey a technology takes from discovery through analysis onward to integration with your projects. I hope you are now better equipped to guide your teams on their technology journey.

I challenge you to apply what you have learned here to your work. You now have a variety of techniques you can apply in your ongoing efforts to remain current in the software industry. Take an hour this week to listen to a podcast or start a technical book you've always wanted to read. Cull your information feeds, eliminating the chaff. Start a book club. Present at a meetup.

Adapt the evaluation criteria presented in this book to *your* unique situation. Build a technology radar. Compare and contrast two similar technologies. List the tradeoffs for that new JavaScript library everyone is raving about. Create a plan to adopt a new technology in your organization. Bring that new language or framework into *your* company. Grab lunch with a peer. Create some fitness functions for your current application. Last but not least, pay it forward: pass on what you've learned here to a colleague.

The software industry moves fast and you cannot let it pass you by. Be strategic in how you apply new technologies to your projects. Think architecturally.

About the Author

Nathaniel T. Schutta is a software architect focused on cloud computing and building usable applications. A proponent of polyglot programming, Nate has written multiple books and appeared in various videos. Nate is a seasoned speaker regularly presenting at conferences worldwide, No Fluff Just Stuff symposia, meetups, universities, and user groups. In addition to his day job, Nate is an adjunct professor at the University of Minnesota where he teaches students to embrace dynamic languages. Driven to rid the world of bad presentations, Nate coauthored the book *Presentation Patterns* (Addison-Wesley) with Neal Ford and Matthew McCullough. You can follow him on Twitter at @ntschutta or on his website.