Chapter 9. Software Architecture

> Architecture is about the important stuff…whatever that is.
>
> Ralph Johnson, computer scientist and co-author of *Design Patterns: Elements of Reusable Object-Oriented Software*

Software architecture is a massive topic. One of your authors teaches a class on the topic at the University of Minnesota and can only scratch the surface in a semester of graduate school. Seemingly everyone has their own definition of the topic, but they all agree it is important to a project's long-term success. This chapter won't make you a software architect, but it will ensure you'll understand the importance of trade-offs, why the answer is almost always "it depends," and why quality attributes are key to building applications that can evolve.

What Is Architecture?

> Have you ever put 20 Agile engineers in a room and asked them each to write down their definition of architecture? I did once. I got 20 different answers.
>
> Matt Parker, author and engineering leader

As discussed in Chapter 4, the software industry is very young and, as such, terminology is often borrowed from more mature disciplines. *Architecture* is a perfect example: the term comes from the building industry. Before digging the foundation of a new home, an architect designs the structure, making sure the resulting house conforms to the local building codes while also meeting the needs of the future occupants. The architect is responsible for the big picture, the structure, the vision of the project—in other words, the things that are hard to change later, such as electrical, plumbing, and ventilation. Once the concrete is poured, it's really hard to refactor!

In software, architecture is much the same. On a software project, the architect is responsible for the overall design of the system, taking into consideration all of the often unstated requirements related to scale, performance, security, and the other quality attributes often referred to as the "illities." While past experience will help you navigate those waters, every application is different. When you encounter an error message from your datastore, an internet search will likely find an answer.[1] However, no amount of querying will ever give you a satisfying answer to "Which datastore should I use for the Foo App?"

> Architecture is the stuff you can't Google.
>
> Neal Ford and Mark Richards, authors of *Fundamentals of Software Architecture, 2nd Edition*

Architecture is sometimes defined as the decisions that are hard to change later. Much as a single-family detached home has different requirements than an apartment building, a software architect must understand the overall landscape of

the application. While software can be refactored more easily than a house, some architectural decisions can be devilishly hard to rework later.[2] For example, imagine you decide to build your application by breaking it into many small, independent pieces (called microservices) written in Java on top of Spring, only to change your mind six months into the project; you have a challenge ahead of you. Architecture requires critical thinking! Architects identify the key quality attributes for a given project and ensure that the design meets those requirements.

Ultimately, architecture is about making decisions about things that are difficult to change later based on trade-off analysis. As such, architects will take time to explore the problem space in an effort to ensure that the application can meet more than just its functional requirements. An architect will often ask questions like these:

- How many users does this application have to support? How many of those are concurrent?
- Where are those users located? Are they mostly in the same area, or are they spread out around the world?
- What is the availability target for this application?
- What can cause a spike in demand or usage of the application?
- When will this application typically be used? Business hours? Any time of day?
- What constraints exist? What does this application integrate with?
- Which cloud provider(s) does the organization contract with?
- What is the competitive landscape? What does your competition require you to match or exceed?
- What is the ideal deployment environment?
- What is the current deployment environment?
- What data should the application store? How long should that data be retained?
- What is the carbon footprint of the application, and how can it be optimized?
- What is the cost of this application, and how can it be optimized?
- What laws and regulations must be adhered to?
- What are acceptable response times for various aspects of the systems?
- What security policies must this application meet or exceed?
- What privacy concerns does this application have?

These questions have no generic answers, as your applications are unique! That is why there is no “right answer” when it comes to choosing an architecture. You must weigh the pros and cons of every decision. For example, if you’re building a document management system for 300 people in Brazil, you’d locate your application in data centers in or near Brazil, you’d want to consult your legal department to understand any specific laws or regulations in the Brazilian market,

and you'd want to know how many documents the application was expected to process a month as well as how large those documents typically are. It'd also be wise to ask if there were any plans to expand beyond Brazil, if there would be a need for any document translation, and what type of document analysis would need to be supported.

Trade-Offs

> Programmers know the benefits of everything and the trade-offs of nothing. Architects need to understand both.
>
> Rich Hickey, creator of the Clojure programming language

Architecture concerns itself with the questions that don't have a simple answer, which is why architects answer nearly every question with, "It depends."[3] In school you learned there was an answer that got you an A, an answer that got you a B, and an answer that got you a C. However, in software, to quote Neal Ford, "There are no right or wrong answers in architecture—only trade-offs." And to further complicate matters, in software, there isn't a "best" answer; in most cases you will have to accept the least worst answer. Every decision you make on a software project involves a trade-off.[4]

An architect's job is to perform the trade-off analysis on the (many) options you will consider for a given project. Every architectural style can be said to have various "star ratings" across various architectural characteristics, but the analysis isn't just as simple as picking the one with the most stars. Certain characteristics might matter *more* for your application than others, thus pushing you toward a style that may appear less than ideal at first glance. Many projects fail in their attempt to create an academically perfect architecture, something that isn't attainable. Good architects make the best decisions they can while still meeting project deadlines.

The Accidental Architect

Many organizations can be a bit…territorial…with titles, and it is very common for engineers to perform architectural tasks even without holding the title of architect, something your authors can personally attest to. We've also never seen an organization that had more architects than they knew what to do with, and most architects are spread thin across many projects. You may find yourself with limited (or no) access to "an architect," making you the accidental architect. If you find yourself playing the role of architect, make sure you're taking the time to see the bigger picture and don't be afraid to get feedback from your team and others in your organization.

You must perform due diligence to weigh the trade-offs. For example, engineers will often pull out common functionality used across modules into a reusable library to facilitate reuse and simplify maintenance. However, should that library be a service that is called at runtime or a library that is built with the caller? In other words, should it be a runtime dependency or a build-time dependency? It depends!

But what does it depend on? You can assess a few factors in this situation. You might first ask how *volatile* the library is, how frequently does it change? Relatedly, how critical is it for callers to have the latest version? If the library is in near constant flux, a runtime dependency would make sense; you can deploy new

versions of the library as needed, and you can rest assured every caller will have the latest version at the same time.

However, you've introduced an out-of-process call into your application, which provides surface area for a new set of errors. What does your application do if it cannot reach the external library service? Does that library need to be geographically distributed? Depending on what the library did, you might have to include an alternative, a cached answer, or a default result. If you choose to make the library a build-time dependency, you remove any network hops, which removes a set of errors and is simpler to test. Upgrades to that library are now more difficult to roll out to an organization, though, and there's a strong likelihood different users of that library will deploy with different versions.

> Software architecture hot tips:
>
> - Good things are better than bad things, except when they're not.
> - Also, nothing is good or bad.
> - It depends.
> - The answer to every question is "it depends," except for when it doesn't. It depends.
> - Name three things you like. You can't have them at the same time.
> - No.
> - There are many definitions of software architecture, but none of them are correct.
> - There's no such thing as software architecture.
>
> Ken Scrambler

Which approach is "correct"? It depends! For example, consider a utility that calculates sales tax based on the shipping destination. Clearly that can change on a regular basis, and it's important for the system to use the most up-to-date rules, but if a network issue prevents your application from calculating tax, there better be a backup in place, or you should expect an urgent message from people high up in the organizational chart. What would you recommend in this situation?

Architecture Versus Design

The boundaries between architecture and design are fuzzy. On any given project, design decisions will affect the architecture, and vice versa. This is sometimes referred to as the Twin Peaks Model, describing the many chicken-and-egg situations often found on a software project. In this case, the current design may limit your architectural maneuverability, and the current architecture will impact your design space! Some decisions will neatly fit into either the architecture or design bucket, but the vast majority live along a continuum between the two. For example, choosing which architectural style is best for your application is clearly an architectural decision, while choosing a data type in a class is a design decision.

If the distinction between architecture and design is often a matter of degrees, does the distinction matter? It depends! In many cases, it comes down to who is best positioned to make the ultimate decision. Architects and developers should be prepared (and willing) to work together on those decisions that fall between the two ends of the continuum.

Another lens you can apply is to ask yourself: is this decision strategic or tactical? *Strategic* thinking requires you to think ahead, focusing on long-term goals and the overall direction of the project. In many cases, the strategic thing to do might take a bit longer or even require you to backtrack a bit.[5] *Tactical* decisions are often short-term actions that solve an immediate, acute problem that may require a longer-term solution at a later date.[6] Think of it as the equivalent of securing something with duct tape, knowing it isn't a permanent fix.

The more tactical a decision, the closer it is to design. The more strategic a decision, the more it veers into architecture. How many people does it take to make the decision? What are their roles? Anything that can be decided by a developer or two is a prime example of a tactical decision. If a decision requires weeks of meetings up to and including the CTO, it's strategic.

Another way of looking at the difference is to ask yourself: what is the cost of change? Refactoring a method is mechanical and can be done by your IDE. Changing the signature on a widely used service is not. Refactoring a method is tactical; refactoring a monolith into microservices is strategic. Naming a method is tactical,[7] but naming an external service used by countless customers is strategic. Throughout any project, you will face countless tactical and strategic choices, and you will often have to make decisions with incomplete information and insufficient time.

Quality Attributes

As discussed earlier, an architect's focus is broader than the feature and functionality of the application; they have to think about the quality attributes necessary to deliver the complete solution. These are sometimes referred to as nonfunctional requirements, the architectural characteristics, quality goals, constraints, quality of service goals, the architecturally significant requirements or, more colloquially, the "illities."[8] One of your most important jobs as an architect is to understand the quality attributes that matter most to your application.

Nonfunctional Requirements

Many of you may be familiar with the idea of nonfunctional requirements and may wonder why we don't use that common term. Our choice boils down to this realization over many years: when you talk to a stakeholder about *nonfunctional* things, they often stop listening as they (consciously or not) think, "I don't want any of that nonfunctional stuff, focus on my business requirements."

Stakeholders are often laser-focused on what they need the application to do, which is important; you need to meet those needs. But you also need to see the bigger picture, the considerations most stakeholders aren't thinking about. Using the *quality attributes* changes the tenor of the conversation. Most customers are delighted to discuss quality.

You are likely familiar with several quality attributes such as maintainability, scalability, reliability, security, deployability, simplicity, usability, compatibility, fault tolerance—the list is long.[9] Which quality attributes matter most to you? It depends![10] The type of applications you build determine which of these are most important. For example, if you are writing a device driver, performance may be far more important than maintainability, while an ecommerce platform will prioritize availability and scalability.

To make matters more complicated, you can't turn every knob up to 11 either. In fact, some quality attributes have inverse relationships with each other: maximizing one will minimize another. Security and usability can sometimes be at odds with each other. You could design a system that is completely secure, not connect it to the network, run it in a locked room, and better yet, not have a login screen. While secure, it certainly isn't very usable.

Your challenge is to identify the quality attributes, then orient them in the correct tension with one another. Additionally, you often have to convince your stakeholders of the value of some of the less visible quality attributes.

Identifying Quality Attributes

How do you know what quality attributes matter on a project? Certain words and phrases should stand out to you that signal quality attributes. For example, consider the following Concert Comparison kata.

A concert ticketing site with big acts and high volume needs an elastic solution to sell tickets:

- Users: Thousands of concurrent users, bursts of up to 10,000/second when tickets go on sale.
- Requirements:
    - Allow concurrent ticket buying.
    - Do not sell a seat more than once!
    - Shoppers can see an overview of remaining seats.
- Additional context:
    - Consider an implementation in both space-based and microservices architecture style.
    - Identify the trade-offs for each solution.

Reading through these requirements, a few things should catch your eye. Having thousands of concurrent users with huge bursts when tickets go on sale implies a highly scalable solution that is also very resilient.[11] And it can likely scale down to near zero after the initial surge of activity.

Obviously, you can't sell the same seat twice, meaning you'll need to come up with a way of locking a seat but in a manner that doesn't allow individual users to block out dozens or hundreds of seats. Oh, and unstated here, but what is your solution to weed out bots?

Once you have an idea of which quality attributes matter most for your situation, take time to rank them. Ordering can be done with a mind map, a table, or even

just a numbered list. Once you have your list ready, it's time to get feedback. Share your artifact with interested stakeholders, consider their feedback, and iterate.

Gaining Stakeholder Alignment

Speaking of interested stakeholders, how do you get a stakeholder to understand the importance of a quality attribute? This is where your influencing skills can come in handy (see Chapter 13 for more on practicing influence and managing stakeholders). While explaining the importance of security to a decision maker may be straightforward, helping them understand the value in maintainability or simplicity requires you to flex your influence muscles.

You can start by outlining the benefits of quality attributes by using terms and examples that are relevant to them. Be careful you aren't framing things purely in a technical manner; while important to you, customers, for example, really shouldn't concern themselves with purely technical choices.[12] Find common ground and have a conversation with them about their concerns. While you may not think their concerns are important, hear them out: there is almost always a nugget to mine, and it's important to consider their concerns. At the end of the day, a conversation needs to take place.

It is also important for you to find the influential people in your organization, which often has little to do with the organizational chart. As an architect, you won't always have a direct line of sight to the decision maker. You may have to work through someone else to get what you want accomplished. Approach as an equal, and rely on the strength of your ideas and your reputation. Find common ground and wield the power of reciprocity.

> A reform often advances most rapidly by indirection.
>
> Frances Willard, American temperance activist and suffragist

The Power of Indirection

Nate here. Years ago, I tried to introduce Git into an organization. We put together a very compelling argument (as a gist no less) outlining why we wanted to replace our existing version control tool. Our presentation earned the go-ahead from our architecture group; however, once we tried to convince the team that managed the existing tool, we were shut down. Hard. We were informed that we already *had* a tool and that they weren't open to anything new.

We took a step back and recruited a more influential ally, the person who acted as the right hand of the CTO.[13] After explaining our approach, he had a powerful insight: don't introduce Git as a *replacement* but rather as a *complement* to our existing toolkit. He positioned Git as a solution to our code collaboration problem, a tool that would enable us to more easily share artifacts. This indirect approach was enough to deactivate the negative sentiment from the team that supported the existing tool; after all, it was additive.[14] We quickly spun up a pilot, discovered multiple opportunities for reuse, and had an enterprise-wide solution set up within a few months.

The lesson? The direct path often leads to the most resistance. Consider an alternative approach or framing when introducing new ideas, technologies, and tools.

Architectural Styles

After identifying the key quality attributes for your application, you can analyze and choose the proper architectural style. An architectural style describes the overall structure of the code as well as how it interacts with the datastore. You can choose from any number of styles for an application, and as you can probably guess, there's no-one-size-fits-all solution.

Broadly speaking, architectural styles can be categorized into two main types: monolithic (a single deployable unit) or distributed (multiple deployable units). While monoliths are often derided as being unmaintainable, the fault lies with a poorly structured system, not the deployment topology. Maintainable modular monoliths are possible, and without the proper discipline and attention to technical debt, heavily distributed systems can be extremely challenging to evolve.

Monolithic styles include the big ball of mud, layered, microkernel, and pipeline. Distributed styles include space based, service oriented, microservices, and event driven. A full accounting of the various architectural styles is beyond the scope of this book.[15] However, it is important to understand that each architectural style has different strengths and weaknesses across the various architectural characteristics. For example, a microservices architecture gives you excellent elasticity and scalability but comes with additional network hops, which can hurt performance, and increased complexity and cost compared to other options. Microservices also require monitoring and observability, and it can be challenging to debug and trace errors.

Choosing the correct style is all about trade-offs. If your application *needs* scalability and elasticity, microservices can be an excellent choice. But if those aren't key characteristics, another style is probably a better option. The right approach for one application could be wrong for another. Again, there's no way to search the internet for "What architectural style should I use?"[16] You have to look at the trade-offs. Identifying the proper architectural style for a given situation is one of the most important things an architect does.

The Agile Architect

Architecture is often defined as the decisions that are hard to change later, but change is the one constant in technology. Does that mean architecture is inherently antithetical to Agile projects? Some organizations (proudly) proclaim that they are Agile and as such don't have architects. Regardless of titles, you have people making architectural decisions on your projects. Hopefully, they're making good ones: you'll know in a year or two if they did.

Architecture and agility can absolutely coexist, though some, perhaps most, architectural activities won't complete within an iteration boundary.[17] Creating an agile architecture may require you to change your assumptions a bit, though. Instead of trying to create an architecture that can effortlessly handle future requirements, what if you designed your architecture expecting things to change? That idea is one of the core tenets of evolutionary architecture.

> An evolutionary architecture supports guided, incremental change across multiple dimensions.
>
> *Building Evolutionary Architectures, 2nd ed.*, by Neal Ford et al.

Evolutionary architectures allow you to change incrementally as required by shifting business needs, priorities, and technological change. The key word here is *incremental*. Agile software development is ultimately about nested feedback loops. Rather than spend months working on a bunch of features only to discover you've missed the mark, regular demonstrations provide opportunities for key stakeholders to react and adjust to actual working software. Hypothesis-driven development is one method that can help you gather data and proceed accordingly.

Hypothesis-driven development utilizes a structure like this:

- We believe <*this change*>
- Will result in <*this outcome*>
- We will know we have succeeded when <*we see X change in this metric*>

For example:

- We believe adding a distributed cache
- Will result in faster startup times
- We will know we have succeeded if startup time is less than 15 seconds

Decisions based on data are better than arguing for an hour and defaulting to the loudest voice in the room.

A Rose by Any Other Name…

Titles can be a funny thing in many organizations, with roles and responsibilities varying widely across companies. "Architect" is no different and may in fact be worse. Some companies will award the title of architect to only the select group of people who report to the VP of architecture. Regardless of title, it is quite common for senior developers to be actively involved in decisions that are squarely in the architect's domain.

Architects are often spread thin and may delegate certain architectural tasks to developers. This not only spreads the workload but also provides an excellent growth opportunity for anyone who aspires to one day be an architect.

Of course, some companies don't have the title of architect at all. It bears repeating: regardless of titles, you have people making architectural decisions on your projects. Hopefully, they're making good ones; you'll know in a year or two if they did.

Fitness Functions

Many architects and organizations invest heavily in the process of creating the appropriate architecture for a project. However, they don't often give much thought to how to *maintain* that architecture over time. Sadly, the second law of thermodynamics applies to software, and without effort to counteract disorder, your architecture will devolve.[18]

From the first commit, your architecture is changing, and while those steeped in its development may intimately understand the nuance of the key architectural considerations, new team members may not have that background. If you're not

diligent, your customers will be complaining about performance or scalability or some other vital quality attribute.

The Performance Problem

Nate here. Early in my architectural career, I was tasked with fixing a performance problem on an ancient VB6 application. Being new to the project (and the role!), I asked when the performance problem first started, hoping we could tie it to a recent update. When I was told the problem started sometime in the last three years, I knew we had a lot of work in front of us. And I started to question my desire to be an architect.

In retrospect, diagnosing the problem didn't have to be so challenging. Had the team created a fitness function that tested performance on a regular basis, it would have discovered the problem sooner, probably before customers even noticed.

You could hope that your application somehow avoids devolving into a big ball of mud.[19] Or you could be more proactive. You could leverage fitness functions to continuously test your architecture, alerting you when something goes out of band. Essentially, a fitness function measures how closely an architecture comes to meeting the objectives. The concept of fitness functions comes from evolutionary computing. Algorithms are mutated with the results tested—is this mutation a success? Applied to architecture, fitness functions allow you to identify and maintain the key architectural characteristics of your application.

Fitness functions are tests you create to ensure that a refactoring or new feature doesn't violate your architecture. You can think of fitness functions as a to-do list for developers from architects as well as lightweight, low-ceremony, governance. Essentially, fitness functions are a set of tests you execute to validate your architecture. For example, you could write tests that ensure the following:

- Service calls respond within an average of 100 milliseconds.
- Cyclomatic complexity shall not exceed 5.
- Average response times as the number of users and requests increases are reasonable.

Ideally, your fitness functions are all automated and run within your CI/CD pipelines. However, some tests may be manual depending on what you need to test. As an architect, you will identify fitness functions early in a project lifecycle; however, they will evolve and change over time. You should periodically review them to ensure they are still relevant. You may discover a better way to test them as well.

Architectural Diagrams

Ultimately, like everything else in software engineering, architecture is about communication, and one of the ways you will express yourself is via models. (See Chapter 5 for more about modeling.) There is no shortage of architectural diagrams you could utilize, but you need to know when to use which one and with which audience. An architecture diagram that's perfect for a developer will be incomprehensible for the VP of engineering.

Before creating any diagram, consider your audience. Is this model for a developer? A fellow architect? A business partner? Sometimes you'll build a diagram solely for yourself, to help decompose a problem or just explore a possible solution. Diagrams can provide context, explain and manage complexity, and identify quality attributes.

From component diagrams to deployment diagrams to sequence diagrams, there is no shortage of options at your disposal. Crafting the right picture at the right moment is an important skill for architects to master. When in doubt, ask yourself whether a given diagram will help you tell the story. If so, by all means build it. If not, don't be afraid to skip it.

Architectural Decision Records

On any software project, you will make a plethora of decisions. Some are small, like naming a variable, while others are more consequential, like choosing a cloud provider.[20] Inevitably, you will encounter a fork in the road forcing you to make a tough call regarding how your project should proceed. Architects often have to make decisions with insufficient time and inadequate information. After a few hours or days of deliberation, your team will make a choice and soldier on. But did you take any time to write down *why* you made the choice you made? The second law of software architecture states: why is more important than how.

Sure, you know what led to that decision, at least in the moment. But what about the person who comes after you? What about you in a few short weeks when today's crisis dominates your thinking? Architectural decision records (ADRs) turn organizational knowledge into a sequential log that future team members can read to understand how your project arrived at the current day.

ADRs were introduced by Michael Nygard as a way to record the motivation behind decisions. They are not meant to be epic tomes, but they should include enough detail to explain the "why." While you can create them in nearly any word processing tool, using plain text like Markdown or AsciiDoc is very common.[21]

Several templates are available on the web, but the basic outline of an ADR includes the following:

Title
: Description of the decision prefixed with a three-digit sequential ID. The ID allows you to easily sort the ADRs in order, allowing you to quickly see the path to today. Use descriptive names.

Status
: While different organizations may use different statuses, you will typically see the following:

  - Request for comment: This ADR is in active review by the team and relevant stakeholders.
  - Proposed: This ADR is a work in progress awaiting approval.
  - Accepted: This ADR has been reviewed and adopted.
  - Superseded: ADRs are immutable, but a later ADR might make an earlier ADR obsolete. The superseded ADR can be updated to point to the overriding ADR.

Context
: Every decision involves constraints and unique circumstances. For example, your CIO may have a set of preferred vendors, which may artificially limit your choices. Do not skimp on the context.

Options
: What options did you consider? If you eliminated any from consideration, why?

Decision
: What did you decide to do? Why did you decide to do it? What options did you reject and why? Again, err on the side of too much information.

Consequences
: Every decision you make has consequences (positive and negative). What are the consequences of this ADR? Again, err on the side of too much information.

Governance
: How will you ensure that the decision is actually followed? What fitness functions will you create to enforce this ADR?

Notes
: This section includes other information as needed as well as metadata about the ADR such as author(s), various dates (accepted, superseded, etc.), and approvers.

Real-world examples can be extremely helpful but also hard to come by; after all, most companies aren't going to publish the inner workings of their software projects on the internet. Thankfully, the results of the first Architectural Katas live training are an invaluable resource for the aspiring architect. Groups of three to five built out the architecture for a health food startup, including ADRs. Their work is available on GitHub. Here are three examples of ADRs: ArchColider, Miyagi's Little Forests, and Jiakaturi.

It is very common for ADRs to include additional diagrams or other supporting material. It may be tempting to omit things that are "common knowledge," but eventually someone will read your ADRs with completely fresh eyes. What would you need in order to understand the decision if it was your first day on the project?

ADRs are often stored in version control, and some organizations have built simple tooling around their ADRs. Others rely on a wiki, but whatever you choose, make sure it can be searched. Odds are you won't remember exactly which ADR covered that tricky problem with the network settings. When in doubt, keep things simple: software projects are hard enough already; don't overcomplicate things needlessly.

Wrapping Up

Architecture is a massive topic taking up several books, videos and podcasts, and countless hours of practice. Many developers aspire to be architects, and for many it is the pinnacle of their career. But being an architect is far more complicated than choosing a frontend library and drawing on the whiteboard.

Architecture requires you to see the bigger picture of the application, identify the key quality attributes, and navigate the trade-offs of decision after decision, all while navigating the politics of your organization communicating up, down, and across the org chart. Being an architect is a challenging but very rewarding path. Hopefully, these guideposts will help you navigate that journey should you choose to make it.

Putting It into Practice

There's an old joke about a tourist in New York City asking someone carrying a violin case how to get to Carnegie Hall, to which the musician replies, "Practice." Architecture is no different: if you want to improve, you need to architect a lot of systems! But few architects get to work on hundreds of systems in a career, prompting Ted Neward to ask: "So how are we supposed to get great architects, if they only get the chance to architect fewer than a half-dozen times in their career?" Ted's astute observation inspired him to create a set of architectural katas covering a variety of domains, with common constraints and just enough detail to allow ample room for interpretation.

You can use architectural katas in many ways.[22] You could pick one or let fate decide and spend an hour or two working through it on your own or with a small group. What questions would you need answered to create an architecture? What quality attributes matter most for your kata? What is the most complex or risky part of the application? What scares you most as an architect? What architectural style would you recommend?

Put your solution aside for a few days or even a few weeks. Take another look. What would you change? Did you miss anything the first time? Would you advise a different approach now? Perform an informal architectural review. What questions do your colleagues have? What are the gaps in your proposed solution? Every month or two, tackle a different kata or continue to refine your previous solution.

Keen on putting your architectural chops to a sterner test? Get a team together and register for an upcoming live Architectural Katas session on the O'Reilly learning platform. You will get a chance to practice your craft on a real problem in a fun and safe environment.

Katas also make for an excellent interview technique. Rather than asking questions about the quantity of piano tuners in a given metropolitan area or testing someone's ability to memorize computer science facts, walk through a kata with them. The questions they ask illustrate how they think and approach problems, and the inevitable collaboration will give you an excellent sense of what it would be like to work with this person on a real project.

Additional Resources

- *Thinking Architecturally* by Nathaniel Schutta (O'Reilly, 2018)
- *Head First Software Architecture* by Raju Gandhi et al. (O'Reilly, 2024)
- *Fundamentals of Software Architectures* by Mark Richards and Neal Ford (O'Reilly, 2020)
- *How to Win Friends and Influence People* by Dale Carnegie (Simon & Schuster, 1936)
- *Building Evolutionary Architectures* , 2nd Edition, by Neal Ford et al. (O'Reilly, 2022)
- "Code as Design" by Jack W. Reeves

- *Influence, New and Expanded: The Psychology of Persuasion* by Robert B. Cialdini (Harper Business, 2021)
- The C4 model for visualizing software architecture
- *Communication Patterns* by Jacqui Read (O'Reilly, 2023)
- *Creating Software with Modern Diagramming Techniques* by Ashley Peacock (Pragmatic Programmers, 2023)
- *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd Edition, by Martin Fowler (Addison-Wesley Professional, 2003)

[1] Sometimes there are too many answers.

[2] While you cannot predict the future, good architectural decisions can make it simpler to change things in the future by carefully considering how the application is likely to evolve.

[3] Your authors may even wear "It Depends" T-shirts to save time in meetings.

[4] And, for that matter, every decision you make in life.

[5] Take the time it takes so it takes less time.

[6] Don't be surprised when that short-term fix is still running years later.

[7] Which isn't to diminish its importance or its ability to spark a spirited debate.

[8] The running joke is that simply adding the suffix -illity to any word results in something an architect cares about.

[9] More architects should strive for simplicity.

[10] We weren't kidding about "it depends" being the answer to every question in architecture.

[11] Can you say Taylor Swift?

[12] And frankly some of the worst customers to work with are those who think they know more about software than you do.

[13] No, that isn't actually a title on the org chart, but it was clear this person was respected by leadership.

[14] It also helped that the proposal was introduced by someone with more organizational clout.

[15] For a complete discussion, see *Fundamentals of Software Architecture* by Mark Richards and Neal Ford (O'Reilly, 2022).

[16] It depends.

[17] If you've ever worked with procurement or legal, you quickly realize you are not in control of the schedule.

[18] In a nutshell, the second law of thermodynamics establishes that, without intervention, entropy within an isolated system will never decrease. In other words, a teenager's bedroom.

[19] Hope is not a strategy. But it is what rebellions are built on.

[20] Not to shortchange the importance of well-named variables, mind you.

[21] Using plain text also aids in searchability, which is very important.

[22] Even as a party game. With the right audience, at least.