Chapter 25. Negotiation and Leadership Skills

Negotiation and leadership skills are hard skills, obtained only through many years of learning, practice, and mistakes—but they are critical in becoming an effective software architect. This book can't make anyone an expert in negotiation and leadership overnight, but the techniques we introduce in this chapter are a good starting point. To dive deeper into this topic, we suggest Tanya Reilly's book *The Staff Engineer's Path: A Guide for Individual Contributors* (O'Reilly, 2022) and the classic negotiation book *Getting to Yes: Negotiating Agreement Without Giving In* (Penguin Books, 2011) by Roger Fisher, William L. Ury, and Bruce Patton.

Negotiation and Facilitation

In Chapter 1, we listed core expectations for software architects. The last expectation we discussed was understanding and being able to navigate your organization's office politics. The reason this is such an important expectation is that almost every decision you make as a software architect will be challenged: by developers who think they know more, by other architects who think they have a better idea or way of approaching the problem, and by stakeholders who think your solution is too expensive or will take too much time.

Negotiation is one of an architect's most important skills. Effective software architects understand the politics of the organization, have strong negotiation and facilitation skills, and can overcome disagreements to create solutions that all stakeholders agree on.

Negotiating with Business Stakeholders

Consider the following scenario in which you, as the lead architect, must negotiate with a key business stakeholder:

Scenario 1

> Parker, the senior vice president and product sponsor for this project, insists that the new global trading system must support "five nines" of availability (99.999%). However, based on the amount of time between global markets when trading doesn't occur (two hours), you know that three nines of availability (99.9%) would be sufficient to meet the project requirements. The problem is, you've seen that Parker does not like to be wrong and doesn't respond well to being corrected, especially if they perceive it as condescending. Parker isn't technically knowledgeable, but thinks they are, so they tend to get involved in the non-functional aspects of projects. As the architect, your goal is to convince Parker, through negotiation, that three nines (99.9%) of availability would be enough.

You'll have to be careful to not be too egotistical and forceful in your analysis, but you also need to make sure you're not missing anything that might prove you wrong during the negotiation. There are several key negotiation techniques you can use to help with this sort of stakeholder negotiation. The first is:

Tip

Pay attention to the buzzwords and jargon people use, even if they seem meaningless. They often contain clues that can help you better understand and negotiate the situation.

Corporate buzzwords are generally meaningless, but can nevertheless provide valuable information when you're about to enter into negotiations. Say you ask when a particular feature is needed and Parker responds, "I needed it yesterday." You can't literally provide that, but it should tell you that time to market is important to this stakeholder. Similarly, the phrase "this system must be lightning fast" means performance is a big concern, and "zero downtime," rather than being literal, means that availability is critical in this application. Effective software architects read between the lines of such exaggerated statements to identify the stakeholder's real concerns. This leads to a second negotiation technique:

Tip

Gather as much information as possible *before* entering into a negotiation.

Parker's use of the phrase "five nines" indicates that they want the system to have high availability (specifically, 99.999% of uptime). However, it might be the case that Parker is unaware of what "five nines" of availability actually means in terms of amount of downtime per year. Table 25-1 shows the downtimes based on the number of nines of availability.

Table 25-1. The nines of availability

Percentage uptime	Downtime per year (per day)
90.0% (one nine)	36 days 12 hrs (2.4 hrs)
99.0% (two nines)	87 hrs 46 min (14 min)
99.9% (three nines)	8 hrs 46 min (86 sec)
99.99% (four nines)	52 min 33 sec (7 sec)
99.999% (five nines)	5 min 35 sec (1 sec)
99.9999% (six nines)	31.5 sec (86 ms)

"Five nines" of availability comes out to 5 minutes and 35 seconds of downtime per year, or an average of 1 second a day of unplanned downtime. That's ambitious, costly, and, as it turns out, unnecessary for Scenario 1 based on the amount of time between global trading when trading doesn't occur. Stating these goals in hours and minutes (or, in this case, seconds) is a much better way to have the conversation than sticking with the "nines" vernacular because it brings some actual metrics and quantified numbers into the discussion.

Negotiating Scenario 1 would include validating Parker's concerns ("I understand that availability is very important for this system"), then steering the negotiation from the nines vernacular to one of reasonable hours and minutes of unplanned downtime. Three nines (which you deem good enough) comes to an average of 86 seconds of unplanned downtime per day—certainly a reasonable number, given the context of this global trading system. Stating it this way might be enough to convince Parker that this amount of unplanned downtime is satisfactory. However, if it doesn't, here's another tip:

Tip

When all else fails, state things in terms of *qualified* cost and time.

We recommend saving this negotiation tactic for last. We've seen too many negotiations start off on the wrong foot due to opening statements like "That's going to cost a lot of money" or "We don't have time for that." Money and time (which includes the effort involved) are certainly key factors in any negotiation, but try other justifications and rationalizations that matter more before you bring them up. They should be a last resort. Once you reach an agreement with the stakeholder, you can consider cost and time if they are important.

Another important negotiation technique for such situations:

Tip

When you need to qualify demands or requirements, "divide and conquer."

In *The Art of War*, the ancient Chinese warrior Sun Tzu writes of one's opponent, "If his forces are united, separate them." You can use this divide-and-conquer tactic during negotiations as well. In Scenario 1, Parker is insisting on five nines (99.999%) of availability for the new trading system. But does the *entire system* needs five nines of availability? You can qualify the requirement, narrowing it down to any specific area(s) of the system that genuinely need five nines of availability. This reduces the scope of difficult (and costly) requirements, not to mention the scope of the negotiation.

Negotiating with Other Architects

It's not unusual for architects to have to negotiate with other architects on a project. Consider Scenario 2, where two architects disagree on which protocol to use:

Scenario 2

> You're the senior of two architects on a project, and you believe that asynchronous messaging would be the right approach for communication between a group of services, to increase both performance and scalability. However, Addison, the other architect on the project, strongly disagrees. They insist that REST would be a better choice, because it's always faster than messaging and can scale just as well. They cite their research, which consists of a Google search and the output of a prompt to a popular generative AI tool. This is not the first heated debate you've had with Addison, nor will it be the last. You want to convince them that messaging is the right solution.

Now, as the senior architect, you certainly could tell Addison that their opinion doesn't matter and ignore it. However, this will only intensify the animosity between you, and an unhealthy, noncollaborative relationship between the project's two architects would almost certainly have a negative impact on the development team.

Tip

Always remember that *demonstration defeats discussion*.

Rather than arguing over REST versus messaging, you should *demonstrate* to Addison why messaging would be a better choice in this specific environment. Because every environment is different, simply Googling it rarely yields the correct answer. But if you compare the two options in a production-like environment and show Addison the results, you might be able to avoid an argument entirely.

Tip

Avoid being overly argumentative or letting things get too personal. Calm leadership, combined with clear and concise reasoning, will almost always win a negotiation.

This is a very powerful technique for dealing with adversarial relationships. Once things get personal or heated, the best thing to do is stop the negotiation. Re-engage later, when both parties have calmed down. Architects argue from time to time, but if you stay calm and project leadership, the other person will usually back down.

Negotiating with Developers

Effective software architects gain the team's respect through working together. That way, when they request something of the development team, it's far less likely to spark an argument or resentment.

As you saw in Chapter 24, working with development teams can be difficult at times. When development teams feel disconnected from the architecture (or the architect), they often feel left out of decisions. This is a classic example of the *Ivory Tower* architecture antipattern. Ivory tower architects dictate from on high, giving development teams orders without regard for their opinions or concerns. This usually leads to the team losing respect for the architect and can eventually cause the team's dynamics to break down altogether. One negotiation technique that can help address this situation is always providing a justification for your decisions.

Tip

When convincing developers to adopt an architecture decision or to do a specific task, provide a justification rather than "dictating from on high."

If you provide a reason *why* something needs to be done, developers are more likely to agree. For example, consider the following conversation between an architect and a developer about making a simple query within a traditional n-tiered layered architecture:

> **Architect**: "You must go through the Business layer to make that call."
> **Developer**: "I disagree. It's much faster just to call the database directly."

There are several things wrong with this conversation. First, notice the architect's use of the words "you must." Not only is this type of commanding voice demeaning, it's one of the worst ways to begin a negotiation (or a conversation). The developer's response includes a reason: going through the Business layer will be slower and take more time.

Now consider an alternative approach:

> **Architect**: “Since change control is most important to us, we have formed a closed-layered architecture. This means all calls to the database need to come from the Business layer.”
> **Developer**: “OK, I get it—but in that case, how am I going to deal with these performance issues for simple queries?”

Here, the architect is *justifying* the demand that all calls go through the Business layer. Providing the justification *first* is always a good approach. Most people tend to stop listening as soon as they hear something they disagree with. Stating the reason before the demand ensures that the developer will hear the architect’s justification.

The architect has also taken a less personal approach to phrasing this demand. By saying “this means” rather than “you must,” they’ve turned the demand into a simple statement of fact. Now take a look at the developer’s response: instead of disagreeing with the layered-architecture restrictions, the developer asks a question about improving performance for simple calls. Now the two can engage in a collaborative conversation to find ways to make simple queries faster while still preserving the closed layers in the architecture.

Another effective negotiation tactic is to have the developer arrive at the solution on their own. For example, suppose that you, as an architect, are choosing between two frameworks, Framework X and Framework Y. Framework Y doesn’t satisfy the security requirements for the system, so you naturally choose Framework X. A developer on your team strongly disagrees, insisting that Framework Y would still be the better choice. Rather than argue the matter, you tell the developer that if they can show you how Framework Y addresses the security concerns, the team will use Framework Y.

One of two things will happen next. Option 1 is that the developer tries to demonstrate that Framework Y satisfies the security requirements, but fails. In failing, they come to understand firsthand why the team can’t use this framework. And because they arrive at the solution on their own, you automatically get their buy-in for the decision to use Framework X. You’ve essentially made it the developer’s decision. This is a win. Option 2 is that the developer finds a way to address the security requirements with Framework Y and demonstrates it to you. This is a win as well. In this case, you missed something in your assessment of Framework Y, and now you have a better solution to the problem (and the developer still feels involved in the decision).

Tip

If a developer disagrees with a decision you make, have them arrive at the solution on their own.

Developers are smart people with useful knowledge. Collaborating with the development team is how architects gain their respect—and their assistance in finding better solutions. The more developers respect an architect, the easier it will be for the architect to negotiate with them.

The Software Architect as a Leader

A software architect is also a leader, guiding a development team as they implement the architecture. We maintain that about 50% of being an effective software architect is having good people skills, including facilitation and

leadership. In this section, we discuss leadership techniques for software architects.

The 4 Cs of Architecture

We've said many times that change is a constant—and that's never more true than when the change in question involves things getting more and more complex, whether it's business processes, technology, or even architecture itself. As you saw in Part III of this book, some architectures are *very* complex. If an architecture needs to support six nines of availability (99.9999%), that's equivalent to unplanned downtime of about 86 milliseconds a day, or 31.5 seconds of downtime per *year*. This sort of complexity is known as *essential complexity*—in other words, "we have a hard problem."

It's easy for architects (and developers) to fall into the trap of adding unnecessary complexity to solutions, diagrams, and documentation. To quote Neal:

> Developers are drawn to complexity like moths to a flame—frequently with the same result.

Figure 25-1 diagrams the major information flows of the backend processing systems at a very large global bank. Is this system *necessarily* complex—in other words, does it *have* to be complex? No one knows, because the architect has *made* it complex. This sort of complexity is called *accidental complexity*: in short, "we have made a problem hard." Architects sometimes add complexity to prove their worth when things seem too simple, or to guarantee that they are always kept in the loop on decisions or even to maintain job security. Whatever the reason, introducing accidental complexity into something that does not have to be complex is one of the best ways to lose respect and become an ineffective leader and architect.

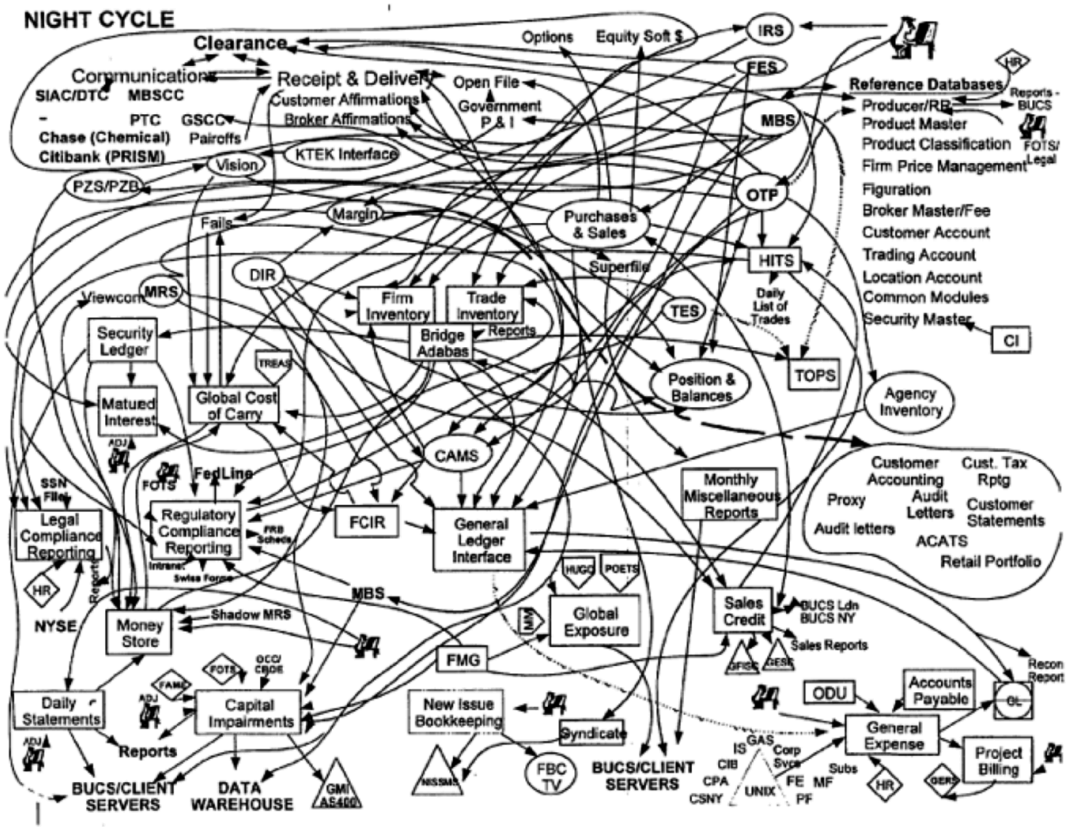

Figure 25-1. Introducing accidental complexity into a problem

To avoid accidental complexity, we use what we call the *4 Cs* of architecture leadership: *communication*, *collaboration, clear*, and *concise*. These factors, as shown in Figure 25-2 (not to be confused with the 4 Cs within the C4 Model of diagramming), work together to create an effective communicator and collaborator on the team.

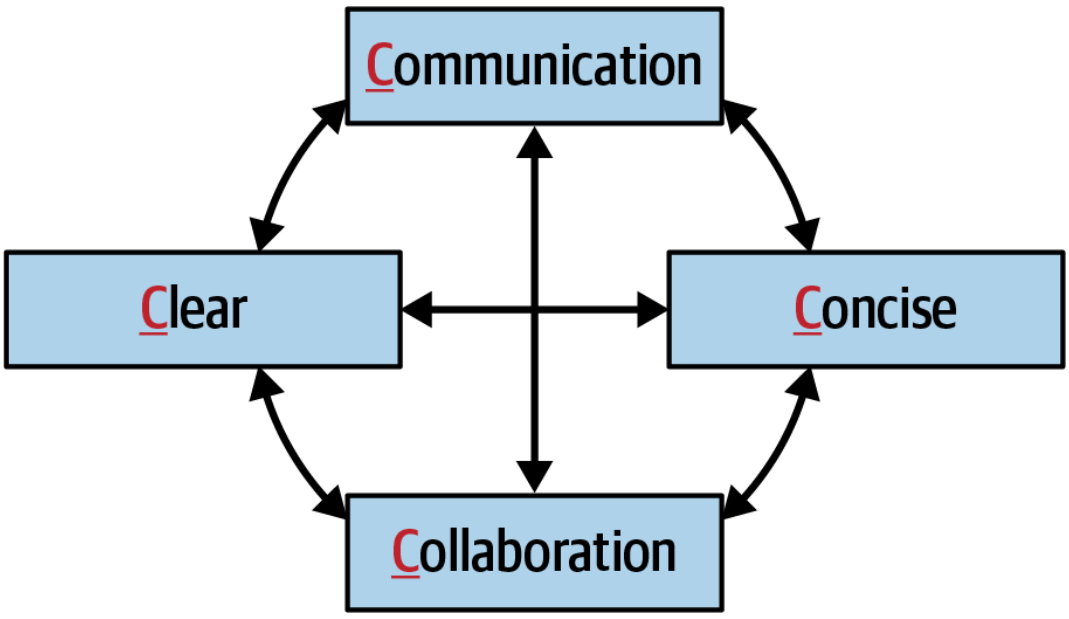

Figure 25-2. The 4 Cs of architecture

As a leader, facilitator, and negotiator, it's vital for a software architect to be able to *communicate clearly* and *concisely*. It is equally important to be able to *collaborate* with developers, business stakeholders, and other architects. Focusing on the 4 Cs helps an architect gain the team's respect and helps make them the go-to person on the project for questions, advice, mentoring, coaching, and leadership.

Be Pragmatic, Yet Visionary

An effective software architect must be pragmatic, yet visionary. Striving for this balance is not as easy as it sounds; it takes a fairly high level of maturity and significant practice to accomplish.

A *visionary* is someone who thinks about or plans the future with imagination or wisdom. Being a visionary means applying strategic thinking to a problem, which is exactly what architects do. Architects plan for the future and make sure the architecture they build remains vital (valid and useful) for a long time. However, architects often become too theoretical in their planning and designs, creating solutions that are too difficult to implement—or even understand.

Now consider the other side of the coin, which is being *pragmatic*. Being pragmatic means dealing with things sensibly and realistically in a way that is based on *practical* rather than *theoretical* considerations. While architects need to be visionaries, they also need to apply practical, realistic solutions. Being pragmatic when creating an architectural solution means accounting for:

- Budget constraints and other cost-based factors
- Time constraints and other time-based factors
- The development team's skill set and skill level

- The trade-offs and implications of each architecture decision
- The technical limitations of any proposed design or solution

Good software architects strive to balance pragmatism with imagination and wisdom in solving problems (see Figure 25-3).

For example, consider an architect faced with sudden, significant increases in concurrent user load for reasons that are unclear. A visionary might come up with an elaborate way to improve the system's elasticity by breaking apart the databases and building a complex *data mesh*: a collection of distributed, domain-partitioned databases used to separate analytical data concerns from transactional ones. In theory, this approach *might* be possible, but being pragmatic means applying reason and practicality to the solution. For example, has the company ever used a data mesh before? What are the trade-offs of using a data mesh? Would this solution really solve the problem?

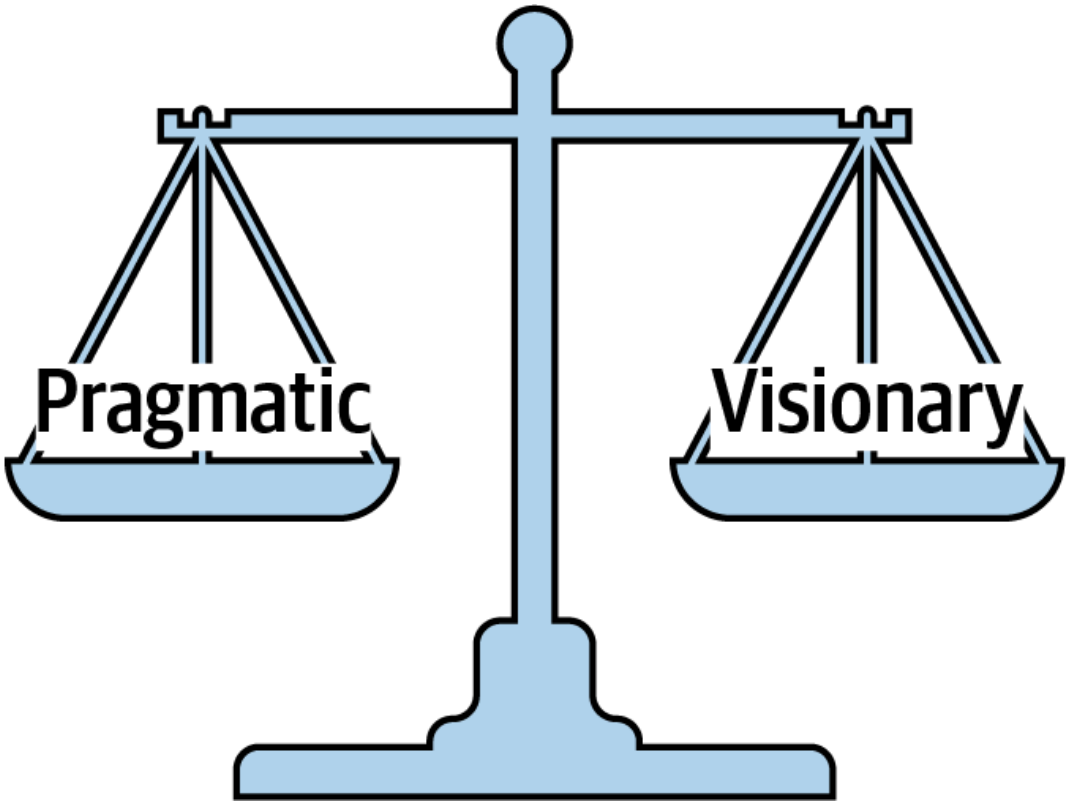

Figure 25-3. Good architects find the balance between being pragmatic and being visionary

Maintaining a good balance between being pragmatic and being visionary is an excellent way to gain respect as an architect. Business stakeholders appreciate visionary solutions that fit within a set of constraints, and developers appreciate having a practical (rather than theoretical) solution to implement.

A pragmatic architect would first look at what factors are limiting the system's elasticity. Finding and isolating any bottlenecks would be a practical first approach to this problem. Is the database creating a bottleneck with respect to some of the services invoked or other external sources needed? If so, could some of the data be cached to reduce calls to the database?

Leading Teams by Example

Bad software architects "pull rank," using their title to get people to do things. Effective software architects get people to do things by leading through example.

Again, this is all about gaining the respect of development teams, business stakeholders, and other people throughout the organization (such as the head of operations, development managers, and product owners).

The classic "lead by example, not by title" story involves a captain in command of a sergeant during a military battle. The captain, who is largely removed from the troops, commands all of the troops forward to take a particularly difficult hill. The soldiers, full of doubt, look for confirmation from the lower-ranking sergeant. The sergeant, understanding the situation, nods his head slightly, and the soldiers immediately move forward with confidence to take the hill.

The moral of this story is that rank and title mean very little when it comes to leading people. The computer scientist Gerald Weinberg is famous for saying, "No matter what the problem is, it's a people problem." Most people think that solving technical issues has nothing to do with people skills and everything to do with *technical knowledge*. While technical knowledge is certainly necessary, it's only a part of solving any problem.

Suppose, for example, that an architect is meeting with a team of developers to solve an issue that's come up in production. One of the developers makes a suggestion, and the architect responds, "Well, *that's* a dumb idea." Not only will that developer not make any more suggestions, but now none of the other developers dare to say anything. The architect has just shut down collaboration across the entire team.

Let's look at another snippet of dialogue between an architect and a developer:

> **Developer**: "So how are we going to solve this performance problem?"
> **Architect**: "What you need to do is use a cache. That would fix the problem."
> **Developer**: "Don't tell me what to do."
> **Architect**: "What I'm telling you is that it would fix the problem."

This is a good example of communicating without *collaborating*. By using the words "what you need to do is," the architect is shutting down collaboration. Now consider a revised approach:

> **Developer**: "So how are we going to solve this performance problem?"
> **Architect**: "Have you considered using a cache? That might fix the problem."
> **Developer**: "Hmmm, no, we didn't think about that. What are your thoughts?"
> **Architect**: "Well, if we put a cache here…"

The words "have you considered" turn the command into a question, placing control back in the developer's hands so that they can have a collaborative conversation with the architect. How you use language is vitally important to building a collaborative environment.

Leading collaboration as an architect isn't just about how you personally collaborate with others—it's also about facilitating collaboration among the team members. Try to observe team dynamics and notice situations like the one in this dialogue. If you witness a team member using demanding or condescending language, take them aside and coach them on using collaborative language. Not only will this create better team dynamics, it will also help the team members respect one another.

If you need to get someone to do something they otherwise might not want to do, sometimes it's best to turn a request into a favor. In general, human beings dislike being told what to do, but want to help others. Consider the following conversation between an architect and developer regarding an architecture refactoring effort during a busy iteration:

> **Architect**: "I'm going to need you to split the payment service into five different services, with each service containing the functionality for each type of payment we accept, such as store credit, credit card, PayPal, gift card, and reward points. That will provide better fault tolerance and scalability in the website. It shouldn't take too long."
> **Developer**: "Sorry, I'm way too busy this iteration for that. I really can't do it."
> **Architect**: "Listen, this is important and it needs to be done this iteration."
> **Developer**: "Sorry, I can't. Maybe one of the other developers can do it. I'm just too busy."

The developer immediately rejects the task, even though the architect has justified it as providing better fault tolerance and scalability. The architect is *telling* the developer to do something they are simply too busy to do—and their demand doesn't even include the person's name! Now consider the technique of turning the request into a favor:

> **Architect**: "Hi, Sridhar. Listen, I'm in a real bind. I really need to have the payment service split into separate services for each payment type to get better fault tolerance and scalability, and I waited too long to do it. Is there any way you can squeeze this into this iteration? It would really help me out."
> **Developer *(pausing)***: "I'm really busy this iteration, but I guess I'll see what I can do."
> **Architect**: "Thanks, Sridhar, I really appreciate the help. I owe you one."
> **Developer**: "No worries. I'll see that it gets done this iteration."

First, using the person's name makes the conversation more personal and familiar, rather than an impersonal professional demand. Using a person's name and proper pronouns during conversations or negotiations can help build respect and healthy relationships. Not only do people like hearing their own names, but it creates a sense of familiarity as well. Practice remembering people's names by using them frequently. If you find a name hard to pronounce, research the correct pronunciation, then practice it until it is perfect. When someone tells you their name, we like to repeat it back and ask if we're pronouncing it correctly. If it's not correct, we repeat this process until we get it right.

Second, the architect in the prior conversation admits they are in a "real bind" and that splitting the services would really "help them out a lot." This doesn't always work, but playing off the basic human urge to help others has a better probability of success than the first conversation. Try it the next time you face this sort of situation.

Another effective leadership technique when you meet someone for the first time or greet someone you only see occasionally is to always shake the person's hand and make eye contact. The handshake is an important people skill that goes back to medieval times. It lets both people know they are friends, not foes, and forms a bond between them. That said, be aware of the cultural aspect of handshakes. For example, in the US, the UK, Europe, and Australia, a handshake is an acceptable means of greeting someone, whereas other cultures use different greetings (such

as Japan, where it's about bowing and the timing of the bow). Nevertheless, sometimes it can be hard to get a simple handshake right.

A handshake should be firm, but not overpowering. Look the person in the eye; looking away while shaking someone's hand is a sign of disrespect, and most people will notice that. Also, don't keep the handshake going too long. Two or three seconds are all you need. Don't go overboard with handshakes either. If you come to the office every morning and start shaking everyone's hand, it's weird enough to make people uncomfortable. However, a monthly meeting with the head of operations is the perfect opportunity to stand up, say "Hello, Ruth, nice seeing you again," and give Ruth a quick, firm handshake. Knowing when to shake hands and when not to is part of the complex art of people skills.

As a leader, be careful to respect and preserve personal boundaries between people at all levels. The handshake is a professional way of forming a physical bond. Since that's a good thing, some people assume that it's even better and more bonding to hug someone. It's not. Hugging in a professional setting, regardless of the environment, can make people uncomfortable and could potentially even become a form of workplace harassment. The same holds true for general conduct in the workplace or when traveling with a coworker. Skip the hugs, adhere to commonsense professional conduct, and stick with handshakes.

A good leader should become the "go-to person" on the team—the person developers go to with questions and problems. An effective software architect will seize the opportunity and take the initiative to lead the team, regardless of their title or role on the team. If someone is struggling with a technical issue, take initiative: step in and offer help or guidance. The same is true for nontechnical situations. If a team member comes into work looking sort of depressed and bothered, as though something is definitely up, an effective software architect might offer to talk. "Hey, Antonio, I'm heading over to get some coffee. Why don't we head over together?" Then during the walk, they can ask if everything is OK. This provides an opening for a more personal discussion and maybe even, in the best case, a chance to mentor and coach. However, an effective leader will also pay attention to verbal and nonverbal signs (like facial expressions and body language) and recognize when it's time to back off.

Another way to establish yourself as the go-to person on the team is to host periodic brown-bag "lunch and learn" sessions about a specific technique or technology, like design patterns or some new features of the latest programming-language release. If you are reading this book, you have some particular skill or knowledge that others don't have. Hosting these events isn't just about providing valuable information to developers or exhibiting your technical prowess—it's also an opportunity to practice mentoring and public-speaking skills, and identifies you as a leader and mentor.

Integrating with the Development Team

An architect's calendar is usually filled with overlapping meetings, like the calendar shown in Figure 25-4. So how do architects find the time to integrate with the team, guide and mentor them, and be available for questions or concerns?

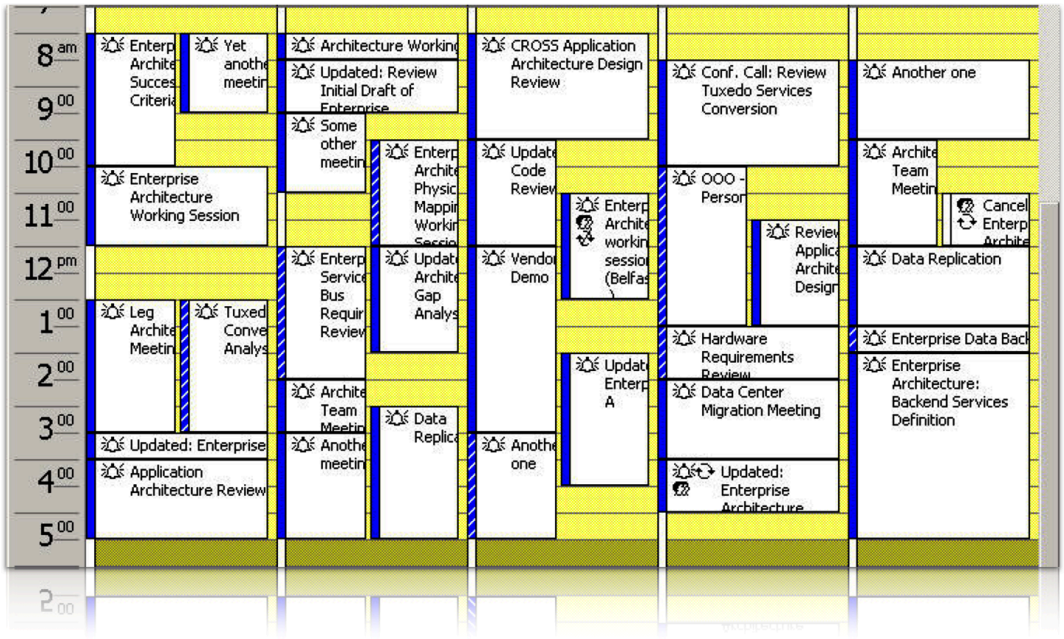

Figure 25-4. A typical calendar of a software architect

Unfortunately, frequent meetings are a necessary evil. The key is controlling them so you can make time for the team. As Figure 25-5 shows, there are two types of meetings: those *imposed upon* you (someone invites you to a meeting), and those *you impose* (you call the meeting).

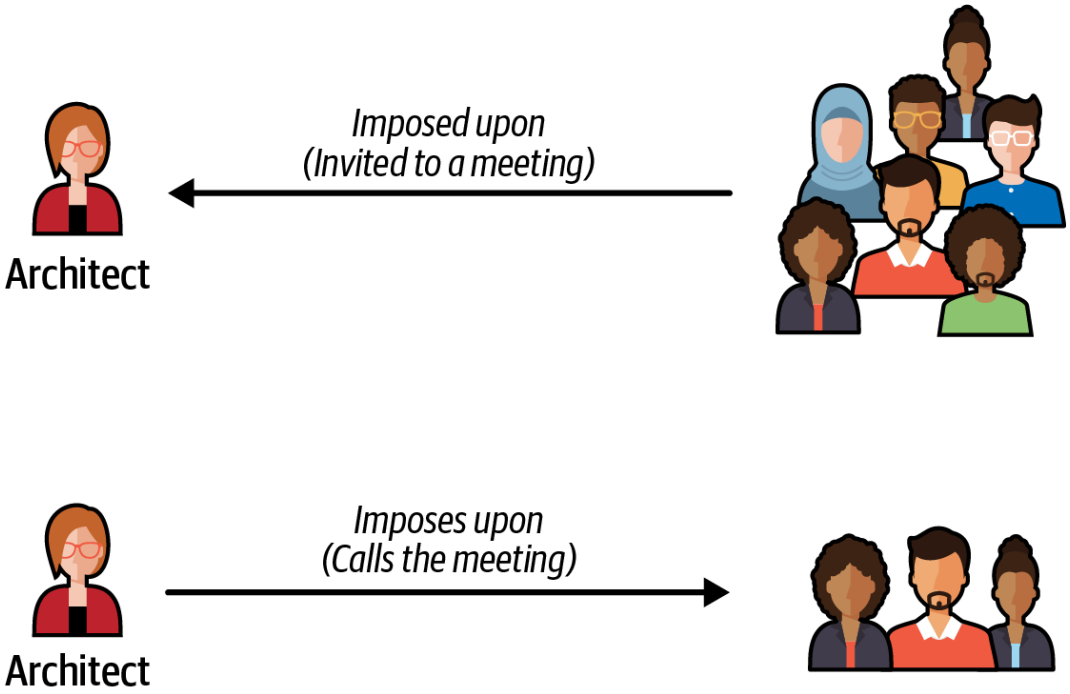

Figure 25-5. Meeting types

The meetings others call are the hardest to control. Because software architects must communicate and collaborate with lots of different stakeholders, they are invited to almost every meeting—even if their presence isn't really needed. When invited to a meeting, ask the organizer why you're needed. If inviting you is just a way to keep you in the loop, that's what meeting notes are for. Asking *why* you should be at a meeting can help you decide which meetings to attend and which you can skip. Looking at the meeting agenda is also useful for this. Also, are you needed at the *whole* meeting, or just the part that will discuss a particular topic?

Could you leave after that agenda item? Don’t waste time in a meeting that you could be spending helping the development team solve problems.

Tip

Ask for the meeting agenda ahead of time to help qualify if you are really needed at the meeting or not.

When developers, or the tech lead, are invited to a meeting, consider going in their place, particularly if both you and the tech lead are invited, so the team can stay focused on the tasks at hand. While deflecting meetings away from useful team members might increase the time *you* spend in meetings, it increases the development team’s productivity and their respect for you.

As an architect you will sometimes have to impose meetings on others, but since this is where you have control, keep them to an absolute minimum. Set an agenda and stick to it. Don’t let some other issue that isn’t relevant to everyone disrupt the meeting. Ask yourself whether the meeting you are calling is more important than the work it will pull the team away from. For example, if it’s a matter of communicating some important information, could you simply send an email rather than call a meeting? However, if you do need to schedule a meeting with a development team, try to schedule meetings first thing in the morning, right after lunch, or toward the end of the day so as to minimize the disruption to the developers during central work hours.

Developer Flow State

Flow is a state of mind developers frequently get into where the brain gets 100% engaged in a particular problem, allowing full attention and maximum creativity. For example, when a developer might be working on a particularly difficult algorithm or piece of code, hours can feel like minutes. Pay close attention to your team’s *productivity flow* and be sure not to disrupt it. You can read more about flow state in the book *Flow: The Psychology of Optimal Experience* by Mihaly Csikszentmihalyi (Harper Perennial, 2008).

Another good way to integrate with the development team, if you work on-site, is to sit with them. Sitting in a cubicle away from the team sends the message “I am special and should not be disturbed.” Sitting alongside the team sends the message “I’m an integral part of the team and available for questions or concerns.” When you’re on-site but can’t sit with your development team, the best thing to do is walk around and be seen as much as possible. An architect who’s never visible, stuck on a different floor or always in their office, cannot possibly guide the team. Block off time in the morning, after lunch, or late in the day to converse, help with issues, answer questions, and do basic coaching and mentoring. Developers appreciate this type of communication and will respect you for making time for them during the day. The same holds true for other stakeholders: stopping in to say hi to the head of operations while on a coffee run is an excellent way to keep the lines of communication open.

Sometimes, such as if you work in a remote environment, it’s not possible to sit with the development team or walk around and be seen. This makes collaboration much more difficult. For more information on managing remote teams, we highly recommend Jacqui Read’s book *Communication Patterns* (O’Reilly, 2023), Part 4 of which is entirely devoted to remote teams.

Summary

The negotiation and leadership tips we present in this chapter are meant to help software architects form better collaborative relationships with the development team and other stakeholders. These are necessary skills. But don’t just take it from us; take it from Theodore Roosevelt, the 26th president of the United States:

> The most important single ingredient in the formula of success is knowing how to get along with people.
>
> Theodore Roosevelt