Chapter 20. Analyzing Architecture Risk

Every architecture has risk associated with it, whether it be risk involving availability, scalability, or data integrity. Analyzing architecture risk is one of the key activities of architecture. By continually analyzing risk, the architect can address deficiencies within the architecture and take corrective action to mitigate the risk. In this chapter we introduce some of the key techniques and practices for qualifying risk, creating risk assessments, and identifying risk through an activity called *risk storming*.

Risk Matrix

The first issue that arises when assessing architecture risk is determining whether the risk should be classified as low, medium, or high. Too much subjectiveness usually enters into this classification, creating confusion about which parts of the architecture are really high risk versus medium risk. Fortunately, there is a risk matrix architects can leverage to help reduce the level of subjectiveness and qualify the risk associated with a particular area of the architecture.

The architecture risk matrix (illustrated in Figure 20-1) uses two dimensions to qualify risk: the overall impact of the risk and the likelihood of that risk occurring. Each dimensions has a low (1), medium (2), and high (3) rating. These numbers are multiplied together within each grid of the matrix, providing an objective numerical number representing that risk. Numbers 1 and 2 are considered low risk (green), numbers 3 and 4 are considered medium risk (yellow), and numbers 6 through 9 are considered high risk (red).

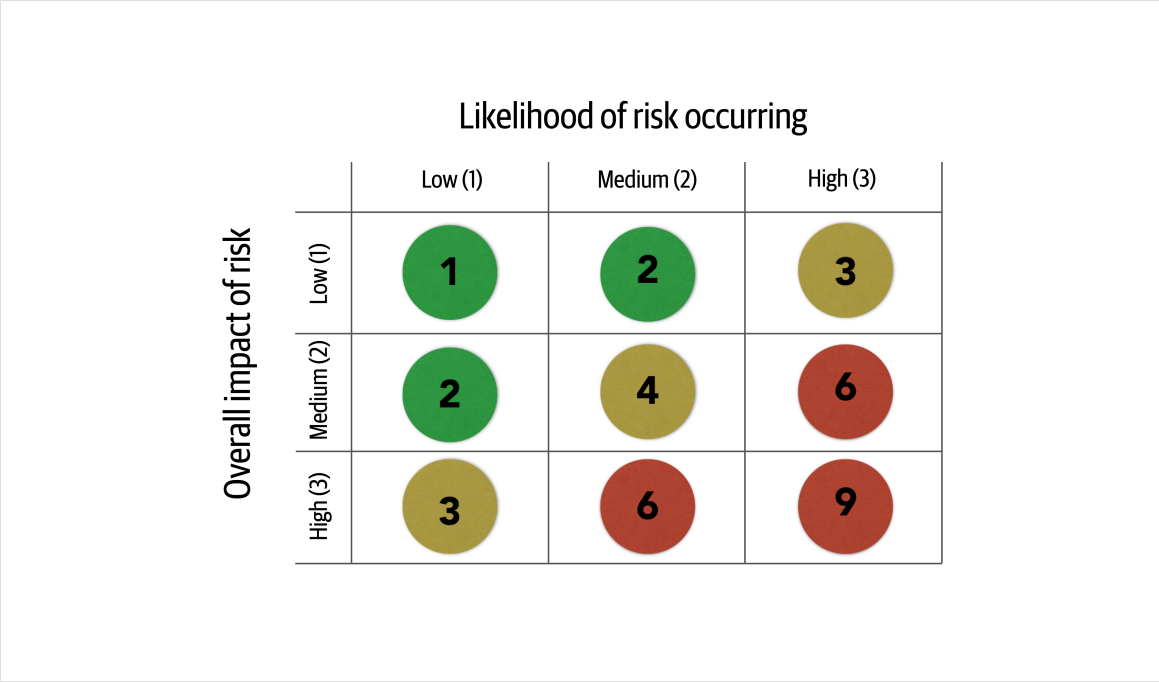

Figure 20-1. Matrix for determining architecture risk

To see how the risk matrix can be used, suppose there is a concern about availability with regard to a primary central database used in the application. First, consider the impact dimension—what is the overall impact if the database goes down or becomes unavailable? Here, an architect might deem that high risk, making that risk either a 3 (medium), 6 (high), or 9 (high). However, after applying the second dimension (likelihood of risk occurring), the architect realizes that the database is on highly available servers in a clustered configuration, so the likelihood is low that the database would become unavailable. Therefore, the intersection between the high impact and low likelihood gives an overall risk rating of 3 (medium risk).

> **TIP**
>
> When leveraging the risk matrix to qualify the risk, consider the impact dimension first and the likelihood dimension second.

Risk Assessments

The risk matrix described in the previous section can be used to build what is called a *risk assessment*. A risk assessment is a summarized report of the overall risk of an architecture with respect to some sort of contextual and meaningful assessment criteria.

Risk assessments can vary greatly, but in general they contain the risk (qualified from the risk matrix) of some *assessment criteria* based on services or domain areas of an application. This basic risk assessment report format is illustrated in Figure 20-2, where light gray (1-2) is low risk, medium gray (3-4) is medium risk, and dark gray (6-9) is high risk. Usually these are color-coded as green (low), yellow (medium), and red (high), but shading can be useful for black-and-white rendering and for color blindness.

Risk criteria	Customer registration	Catalog checkout	Order fulfillment	Order shipment	**Total risk**
Scalability	2	6	1	2	11
Availability	3	4	2	1	10
Performance	4	2	3	6	15
Security	6	3	1	1	11
Data integrity	9	6	1	1	17
Total risk	24	21	8	11	

Figure 20-2. Example of a standard risk assessment

The quantified risk from the risk matrix can be accumulated by the risk criteria and also by the service or domain area. For example, notice in Figure 20-2 that the accumulated risk for data integrity is the highest risk area at a total of 17, whereas the accumulated risk for Availability is only 10 (the least amount of risk). The relative risk of each domain area can also be

determined by the example risk assessment. Here, customer registration carries the highest area of risk, whereas order fulfillment carries the lowest risk. These relative numbers can then be tracked to demonstrate either improvements or degradation of risk within a particular risk category or domain area.

Although the risk assessment example in Figure 20-2 contains all the risk analysis results, rarely is it presented as such. Filtering is essential for visually indicating a particular message within a given context. For example, suppose an architect is in a meeting for the purpose of presenting areas of the system that are high risk. Rather than presenting the risk assessment as illustrated in Figure 20-2, filtering can be used to only show the high risk areas (shown in Figure 20-3), improving the overall signal-to-noise ratio and presenting a clear picture of the state of the system (good or bad).

Risk criteria	Customer registration	Catalog checkout	Order fulfillment	Order shipment	Total risk
Scalability		6			6
Availability					0
Performance				6	6
Security	6				6
Data integrity	9	6			15
Total risk	15	12	0	6	

Figure 20-3. Filtering the risk assessment to only high risk

Another issue with Figure 20-2 is that this assessment report only shows a snapshot in time; it does not show whether things are improving or getting worse. In other words, Figure 20-2 does not show the direction of risk. Rendering the direction of risk presents somewhat of an issue. If an up or down arrow were to be used to indicate direction, what would an up arrow

mean? Are things getting better or worse? We've spent years asking people if an up arrow meant things were getting better or worse, and almost 50% of people asked said that the up arrow meant things were progressively getting worse, whereas almost 50% said an up arrow indicated things were getting better. The same is true for left and right arrows. For this reason, when using arrows to indicate direction, a key must be used. However, we've also found this doesn't work either. Once the user scrolls beyond the key, confusion happens once again.

We usually use the universal direction symbol of a plus (+) and minus (-) sign next to the risk rating to indicate direction, as illustrated in Figure 20-4. Notice in Figure 20-4 that although performance for customer registration is medium (4), the direction is a minus sign (red), indicating that it is progressively getting worse and heading toward high risk. On the other hand, notice that scalability of catalog checkout is high (6) with a plus sign (green), showing that it is improving. Risk ratings without a plus or minus sign indicate that the risk is stable and neither getting better nor worse.

Risk criteria	Customer registration	Catalog checkout	Order fulfillment	Order shipment	**Total risk**
Scalability	2	6 +	1	2	11
Availability	3	4	2 -	1	10
Performance	4 -	2 +	3 -	6 +	15
Security	6 -	3	1	1	11
Data integrity	9 +	6 -	1 -	1	17
Total risk	24	21	8	11	

Figure 20-4. Showing direction of risk with plus and minus signs

Occasionally, even the plus and minus signs can be confusing to some people. Another technique for indicating direction is to leverage an arrow along with the risk rating number it is trending toward. This technique, as

illustrated in Figure 20-5, does not require a key because the direction is clear. Furthermore, the use of colors (red arrow for worse, green arrow for better) makes it even more clear where the risk is heading.

Risk criteria	Customer registration	Catalog checkout	Order fulfillment	Order shipment	Total risk
Scalability	2	6 ↑4	1	2	11
Availability	3	4	2 ↓3	1	10
Performance	4 ↓6	2 ↑1	3 ↓4	6 ↑4	15
Security	6 ↓9	3	1	1	11
Data integrity	9 ↑6	6 ↓9	1 ↓2	1	17
Total risk	24	21	8	11	

Figure 20-5. Showing direction of risk with arrows and numbers

The direction of risk can be determined by using continuous measurements through fitness functions described earlier in the book. By objectively analyzing each risk criteria, trends can be observed, providing the direction of each risk criteria.

Risk Storming

No architect can single-handedly determine the overall risk of a system. The reason for this is two-fold. First, a single architect might miss or overlook a risk area, and very few architects have full knowledge of every part of the system. This is where *risk storming* can help.

Risk storming is a collaborative exercise used to determine architectural risk within a specific dimension. Common dimensions (areas of risk) include unproven technology, performance, scalability, availability (including transitive dependencies), data loss, single points of failure, and security. While most risk storming efforts involve multiple architects, it is

wise to include senior developers and tech leads as well. Not only will they provide an implementation perspective to the architectural risk, but involving developers helps them gain a better understanding of the architecture.

The risk storming effort involves both an individual part and a collaborative part. In the individual part, all participants individually (without collaboration) assign risk to areas of the architecture using the risk matrix described in the previous section. This noncollaborative part of risk storming is essential so that participants don’t influence or direct attention away from particular areas of the architecture. In the collaborative part of risk storming, all participants work together to gain consensus on risk areas, discuss risk, and form solutions for mitigating the risk.

An architecture diagram is used for both parts of the risk storming effort. For holistic risk assessments, usually a comprehensive architecture diagram is used, whereas risk storming within specific areas of the application would use a contextual architecture diagram. It is the responsibility of the architect conducting the risk storming effort to make sure these diagrams are up to date and available to all participants.

Figure 20-6 shows an example architecture we’ll use to illustrate the risk storming process. In this architecture, an Elastic Load Balancer fronts each EC2 instance containing the web servers (Nginx) and application services. The application services make calls to a MySQL database, a Redis cache, and a MongoDB database for logging. They also make calls to the Push Expansion Servers. The expansion servers, in turn, all interface with the MySQL database, Redis cache, and MongoDB logging facility.

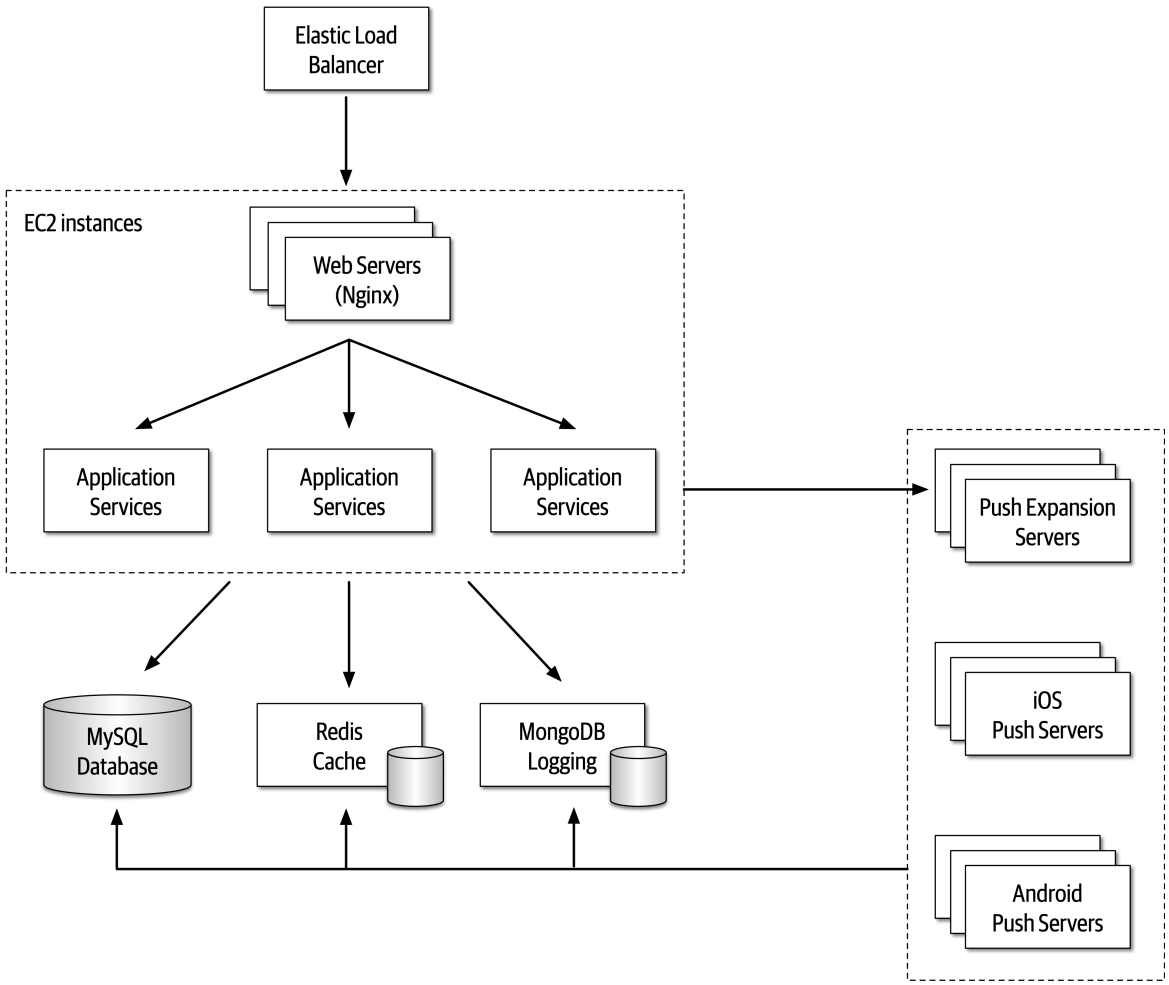

Figure 20-6. Architecture diagram for risk storming example

Risk storming is broken down into three primary activities:

1. Identification
2. Consensus
3. Mitigation

Identification is always an individual, noncollaborative activity, whereas consensus and mitigation are always collaborative and involve all participants working together in the same room (at least virtually). Each of these primary activities is discussed in detail in the following sections.

Identification

The *identification* activity of risk storming involves each participant individually identifying areas of risk within the architecture. The following steps describe the identification part of the risk storming effort:

1. The architect conducting the risk storming sends out an invitation to all participants one to two days prior to the collaborative part of the effort. The invitation contains the architecture diagram (or the location of where to find it), the risk storming dimension (area of risk being analyzed for that particular risk storming effort), the date when the collaborative part of risk storming will take place, and the location.

2. Using the risk matrix described in the first section of this chapter, participants individually analyze the architecture and classify the risk as low (1-2), medium (3-4), or high (6-9).

3. Participants prepare small Post-it notes with corresponding colors (green, yellow, and red) and write down the corresponding risk number (found on the risk matrix).

Most risk storming efforts only involve analyzing one particular dimension (such as performance), but there might be times, due to the availability of staff or timing issues, when multiple dimensions are analyzed within a single risk storming effort (such as performance, scalability, and data loss). When multiple dimensions are analyzed within a single risk storming effort, the participants write the dimension next to the risk number on the Post-it notes so that everyone is aware of the specific dimension. For example, suppose three participants found risk within the central database. All three identified the risk as high (6), but one participant found risk with respect to availability, whereas two participants found risk with respect to performance. These two dimensions would be discussed separately.

> **TIP**
>
> Whenever possible, restrict risk storming efforts to a single dimension. This allows participants to focus their attention to that specific dimension and avoids confusion about multiple risk areas being identified for the same area of the architecture.

Consensus

The *consensus* activity in the risk storming effort is highly collaborative with the goal of gaining consensus among all participants regarding the risk within the architecture. This activity is most effective when a large, printed version of the architecture diagram is available and posted on the wall. In lieu of a large printed version, an electronic version can be displayed on a large screen.

Upon arrival at the risk storming session, participants begin placing their Post-it notes on the architecture diagram in the area where they individually found risk. If an electronic version is used, the architect conducting the risk storming session queries every participant and electronically places the risk on the diagram in the area of the architecture where the risk was identified (see Figure 20-7).

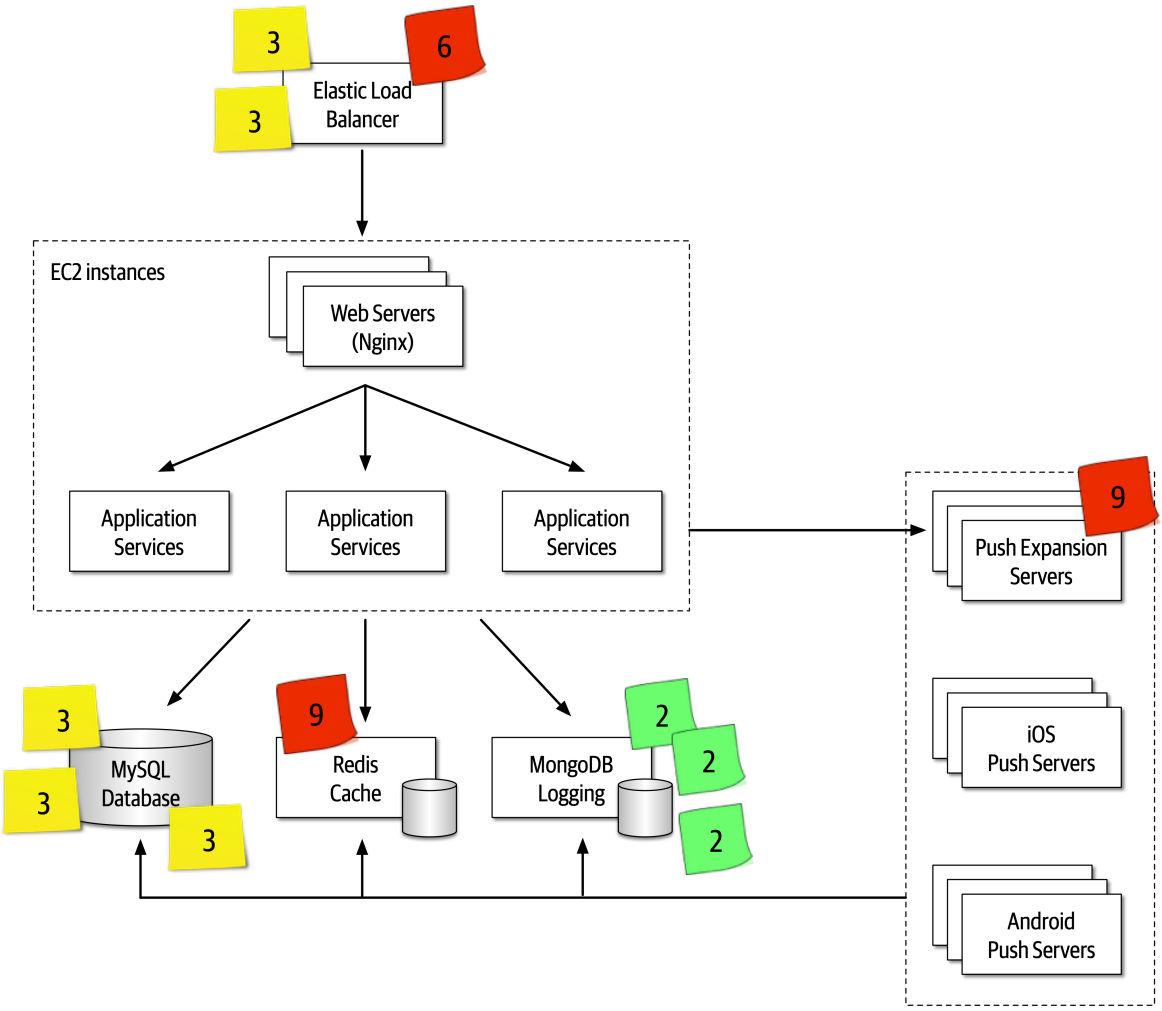

Figure 20-7. Initial identification of risk areas

Once all of the Post-it notes are in place, the collaborative part of risk storming can begin. The goal of this activity of risk storming is to analyze the risk areas as a team and gain consensus in terms of the risk qualification. Notice several areas of risk were identified in the architecture, illustrated in Figure 20-7:

1. Two participants individually identified the Elastic Load Balancer as medium risk (3), whereas one participant identified it as high risk (6).

2. One participant individually identified the Push Expansion Servers as high risk (9).

3. Three participants individually identified the MySQL database as medium risk (3).
4. One participant individually identified the Redis cache as high risk (9).
5. Three participants identified MongoDB logging as low risk (2).
6. All other areas of the architecture were not deemed to carry any risk, hence there are no Post-it notes on any other areas of the architecture.

Items 3 and 5 in the prior list do not need further discussion in this activity since all participants agreed on the level and qualification of risk. However, notice there was a difference of opinion in item 1 in the list, and items 2 and 4 only had a single participant identifying the risk. These items need to be discussed during this activity.

Item 1 in the list showed that two participants individually identified the Elastic Load Balancer as medium risk (3), whereas one participant identified it as high risk (6). In this case the other two participants ask the third participant why they identified the risk as high. Suppose the third participant says that they assigned the risk as high because if the Elastic Load Balancer goes down, the entire system cannot be accessed. While this is true and in fact does bring the overall impact rating to high, the other two participants convince the third participant that there is low risk of this happening. After much discussion, the third participant agrees, bringing that risk level down to a medium (3). However, the first and second participants might not have seen a particular aspect of risk in the Elastic Load Balancer that the third did, hence the need for collaboration within this activity of risk storming.

Case in point, consider item 2 in the prior list where one participant individually identified the Push Expansion Servers as high risk (9), whereas no other participant identified them as any risk at all. In this case, all other participants ask the participant who identified the risk why they rated it as high. That participant then says that they have had bad experiences with the

Push Expansion Servers continually going down under high load, something this particular architecture has. This example shows the value of risk storming—without that participant's involvement, no one would have seen the high risk (until well into production of course!).

Item 4 in the list is an interesting case. One participant identified the Redis cache as high risk (9), whereas no other participant saw that cache as any risk in the architecture. The other participants ask what the rationale is for the high risk in that area, and the one participant responds with, "What is a Redis cache?" In this case, Redis was unknown to the participant, hence the high risk in that area.

> **TIP**
>
> For unproven or unknown technologies, always assign the highest risk rating (9) since the risk matrix cannot be used for this dimension.

The example of item 4 in the list illustrates why it is wise (and important) to bring developers into risk storming sessions. Not only can developers learn more about the architecture, but the fact that one participant (who was in this case a developer on the team) didn't know a given technology provides the architect with valuable information regarding overall risk.

This process continues until all participants agree on the risk areas identified. Once all the Post-it notes are consolidated, this activity ends, and the next one can begin. The final outcome of this activity is shown in Figure 20-8.

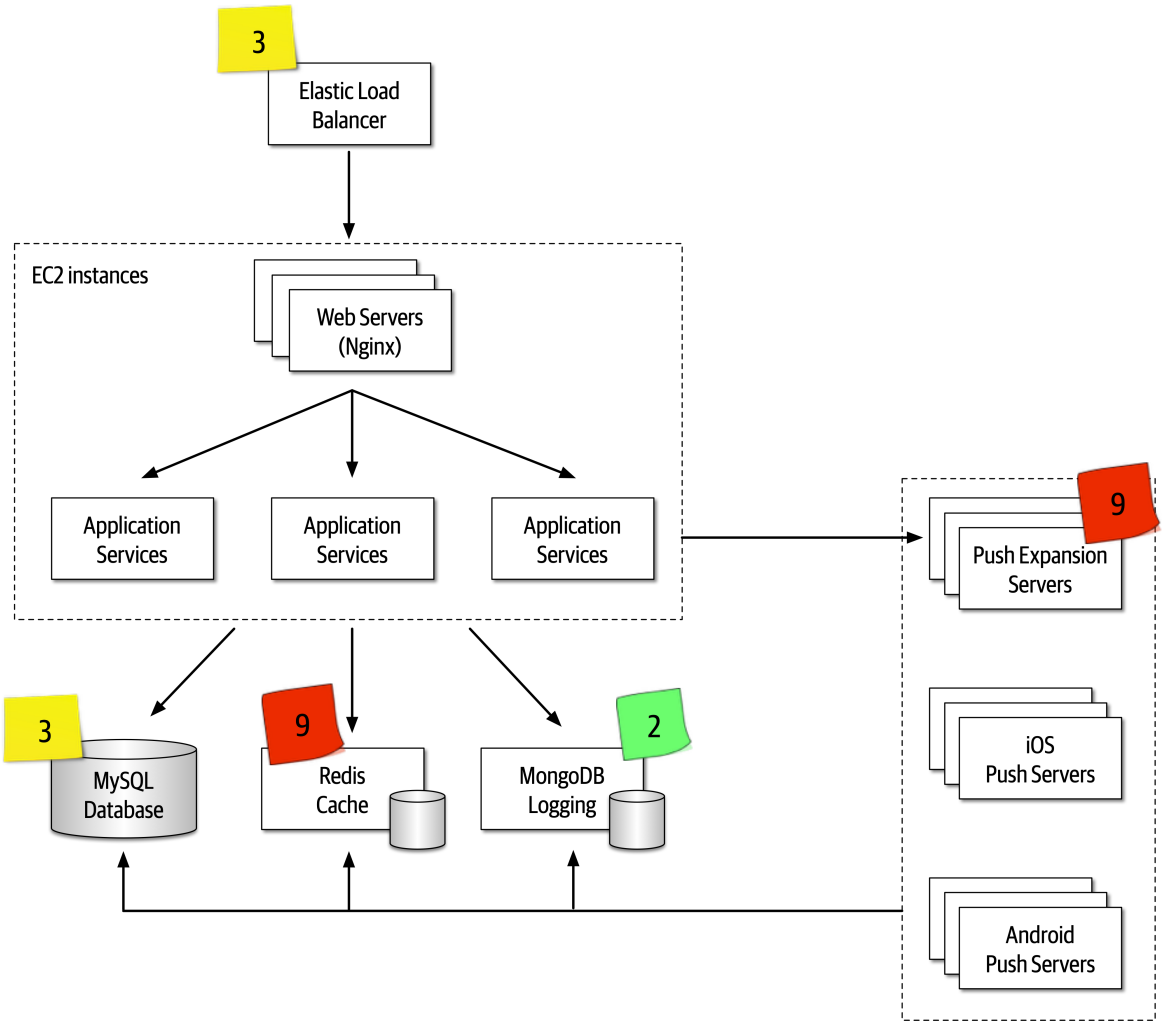

Figure 20-8. Consensus of risk areas

Mitigation

Once all participants agree on the qualification of the risk areas of the architecture, the final and most important activity occurs—*risk mitigation*. Mitigating risk within an architecture usually involves changes or enhancements to certain areas of the architecture that otherwise might have been deemed perfect the way they were.

This activity, which is also usually collaborative, seeks ways to reduce or eliminate the risk identified in the first activity. There may be cases where the original architecture needs to be completely changed based on the identification of risk, whereas others might be a straightforward architecture

refactoring, such as adding a queue for back pressure to reduce a throughput bottleneck issue.

Regardless of the changes required in the architecture, this activity usually incurs additional cost. For that reason, key stakeholders typically decide whether the cost outweighs the risk. For example, suppose that through a risk storming session the central database was identified as being medium risk (4) with regard to overall system availability. In this case, the participants agreed that clustering the database, combined with breaking the single database into separate physical databases, would mitigate that risk. However, while risk would be significantly reduced, this solution would cost $20,000. The architect would then conduct a meeting with the key business stakeholder to discuss this trade-off. During this negotiation, the business owner decides that the price tag is too high and that the cost does not outweigh the risk. Rather than giving up, the architect then suggests a different approach—what about skipping the clustering and splitting the database into two parts? The cost in this case is reduced to $8,000 while still mitigating most of the risk. In this case, the stakeholder agrees to the solution.

The previous scenario shows the impact risk storming can have not only on the overall architecture, but also with regard to negotiations between architects and business stakeholders. Risk storming, combined with the risk assessments described at the start of this chapter, provide an excellent vehicle for identifying and tracking risk, improving the architecture, and handling negotiations between key stakeholders.

Agile Story Risk Analysis

Risk storming can be used for other aspects of software development besides just architecture. For example, we've leveraged risk storming for determining overall risk of user story completion within a given Agile iteration (and consequently the overall risk assessment of that iteration) during story grooming. Using the risk matrix, user story risk can be identified by the first dimension (the overall impact if the story is not

completed within the iteration) and the second dimension (the likelihood that the story will not be completed). By utilizing the same architecture risk matrix for stories, teams can identify stories of high risk, track those carefully, and prioritize them.

Risk Storming Examples

To illustrate the power of risk storming and how it can improve the overall architecture of a system, consider the example of a call center system to support nurses advising patients on various health conditions. The requirements for such a system are as follows:

- The system will use a third-party diagnostics engine that serves up questions and guides the nurses or patients regarding their medical issues.
- Patients can either call in using the call center to speak to a nurse or choose to use a self-service website that accesses the diagnostic engine directly, bypassing the nurses.
- The system must support 250 concurrent nurses nationwide and up to hundreds of thousands of concurrent self-service patients nationwide.
- Nurses can access patients' medical records through a medical records exchange, but patients cannot access their own medical records.
- The system must be HIPAA compliant with regard to the medical records. This means that it is essential that no one but nurses have access to medical records.
- Outbreaks and high volume during cold and flu season need to be addressed in the system.
- Call routing to nurses is based on the nurse's profile (such as bilingual needs).

- The third-party diagnostic engine can handle about 500 requests a second.

The architect of the system created the high-level architecture illustrated in Figure 20-9. In this architecture there are three separate web-based user interfaces: one for self-service, one for nurses receiving calls, and one for administrative staff to add and maintain the nursing profile and configuration settings. The call center portion of the system consists of a call accepter which receives calls and the call router which routes calls to the next available nurse based on their profile (notice how the call router accesses the central database to get nurse profile information). Central to this architecture is a diagnostics system API gateway, which performs security checks and directs the request to the appropriate backend service.

There are four main services in this system: a case management service, a nurse profile management service, an interface to the medical records exchange, and the external third-party diagnostics engine. All communications are using REST with the exception of proprietary protocols to the external systems and call center services.

The architect has reviewed this architecture numerous times and believes it is ready for implementation. As a self-assessment, study the requirements and the architecture diagram in Figure 20-9 and try to determine the level of risk within this architecture in terms of availability, elasticity, and security. After determining the level of risk, then determine what changes would be needed in the architecture to mitigate that risk. The sections that follow contain scenarios that can be used as a comparison.

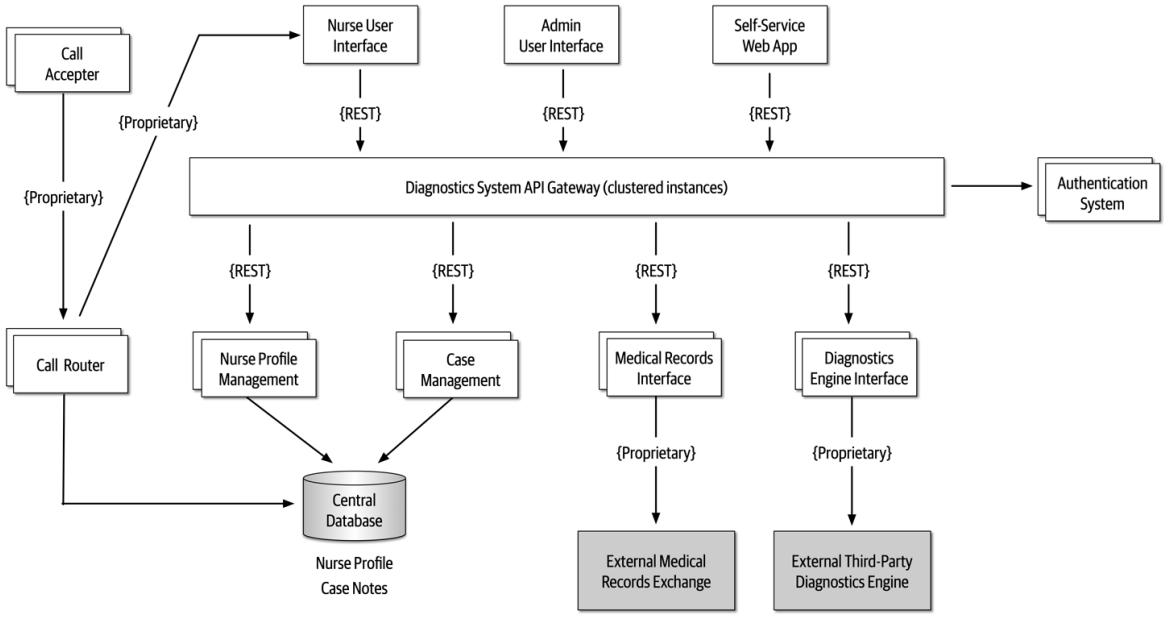

Figure 20-9. High-level architecture for nurse diagnostics system example

Availability

During the first risk storming exercise, the architect chose to focus on availability first since system availability is critical for the success of this system. After the risk storming identification and collaboration activities, the participants came up with the following risk areas using the risk matrix (as illustrated in Figure 20-10):

- The use of a central database was identified as high risk (6) due to high impact (3) and medium likelihood (2).
- The diagnostics engine availability was identified as high risk (9) due to high impact (3) and unknown likelihood (3).
- The medical records exchange availability was identified as low risk (2) since it is not a required component for the system to run.
- Other parts of the system were not deemed as risk for availability due to multiple instances of each service and clustering of the API gateway.

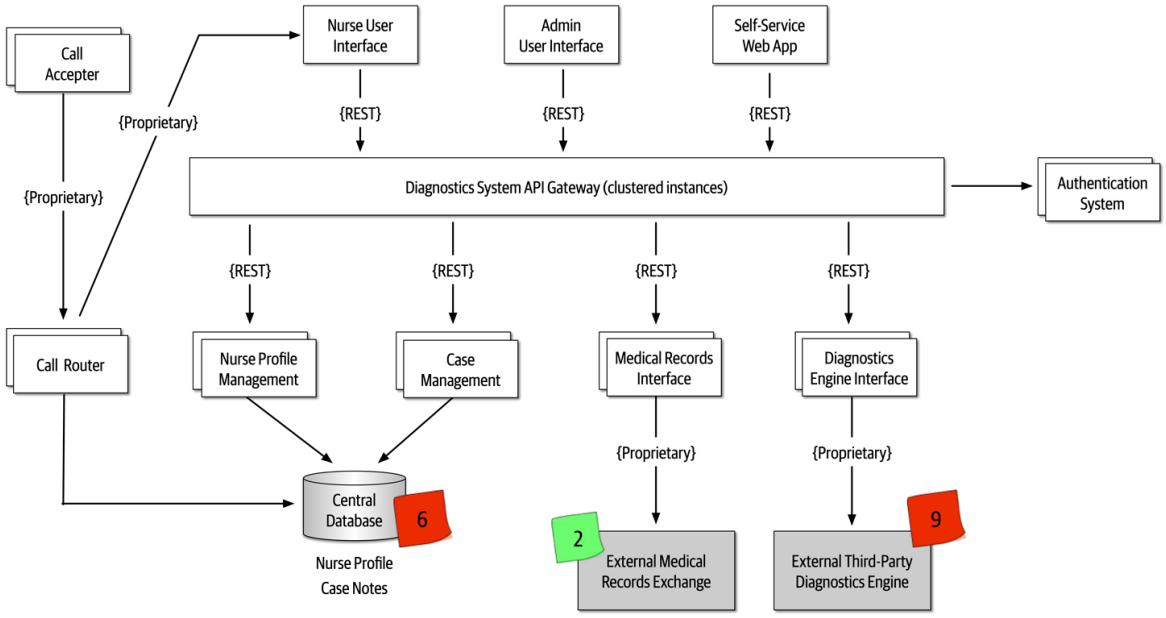

Figure 20-10. Availability risk areas

During the risk storming effort, all participants agreed that while nurses can manually write down case notes if the database went down, the call router could not function if the database were not available. To mitigate the database risk, participants chose to break apart the single physical database into two separate databases: one clustered database containing the nurse profile information, and one single instance database for the case notes. Not only did this architecture change address the concerns about availability of the database, but it also helped secure the case notes from admin access. Another option to mitigate this risk would have been to cache the nurse profile information in the call router. However, because the implementation of the call router was unknown and may be a third-party product, the participants went with the database approach.

Mitigating the risk of availability of the external systems (diagnostics engine and medical records exchange) is much harder to manage due to the lack of control of these systems. One way to mitigate this sort of availability risk is to research if there is a published service-level agreement (SLA) or service-level objective (SLO) for each of these systems. An SLA is usually a contractual agreement and is legally binding, whereas an SLO is usually not. Based on research, the architect found that the SLA for the diagnostics engine is guaranteed to be 99.99% available (that’s 52.60

minutes of downtime per year), and the medical records exchange is guaranteed at 99.9% availability (that's 8.77 hours of downtime per year). Based on the relative risk, this information was enough to remove the identified risk.

The corresponding changes to the architecture after this risk storming session are illustrated in Figure 20-11. Notice that two databases are now used, and also the SLAs are published on the architecture diagram.

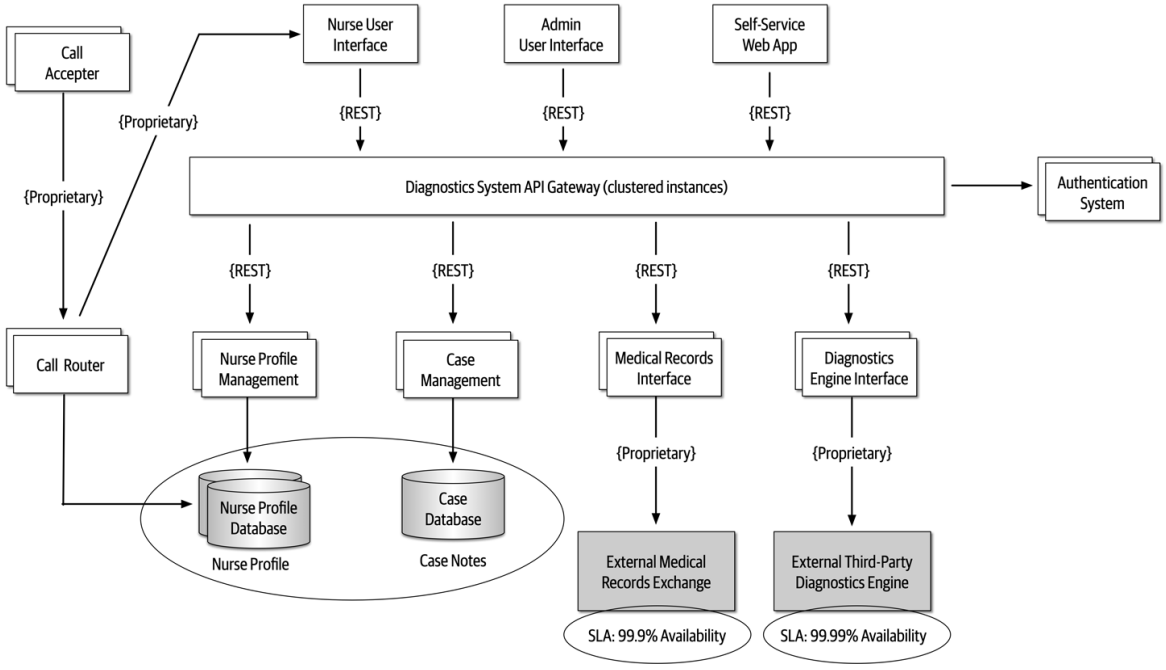

Figure 20-11. Architecture modifications to address availability risk

Elasticity

On the second risk storming exercise, the architect chose to focus on elasticity—spikes in user load (otherwise known as variable scalability). Although there are only 250 nurses (which provides an automatic governor for most of the services), the self-service portion of the system can access the diagnostics engine as well as nurses, significantly increasing the number of requests to the diagnostics interface. Participants were concerned about outbreaks and flu season, when anticipated load on the system would significantly increase.

During the risk storming session, the participants all identified the diagnostics engine interface as high risk (9). With only 500 requests per second, the participants calculated that there was no way the diagnostics engine interface could keep up with the anticipated throughput, particularly with the current architecture utilizing REST as the interface protocol.

One way to mitigate this risk is to leverage asynchronous queues (messaging) between the API gateway and the diagnostics engine interface to provide a back-pressure point if calls to the diagnostics engine get backed up. While this is a good practice, it still doesn't mitigate the risk, because nurses (as well as self-service patients) would be waiting too long for responses from the diagnostics engine, and those requests would likely time out. Leveraging what is known as the Ambulance Pattern would give nurses a higher priority over self-service. Therefore two message channels would be needed. While this technique helps mitigate the risk, it still doesn't address the wait times. The participants decided that in addition to the queuing technique to provide back-pressure, caching the particular diagnostics questions related to an outbreak would remove outbreak and flu calls from ever having to reach the diagnostics engine interface.

The corresponding architecture changes are illustrated in Figure 20-12. Notice that in addition to two queue channels (one for the nurses and one for self-service patients), there is a new service called the *Diagnostics Outbreak Cache Server* that handles all requests related to a particular outbreak or flu-related question. With this architecture in place, the limiting factor was removed (calls to the diagnostics engine), allowing for tens of thousands of concurrent requests. Without a risk storming effort, this risk might not have been identified until an outbreak or flu season happened.

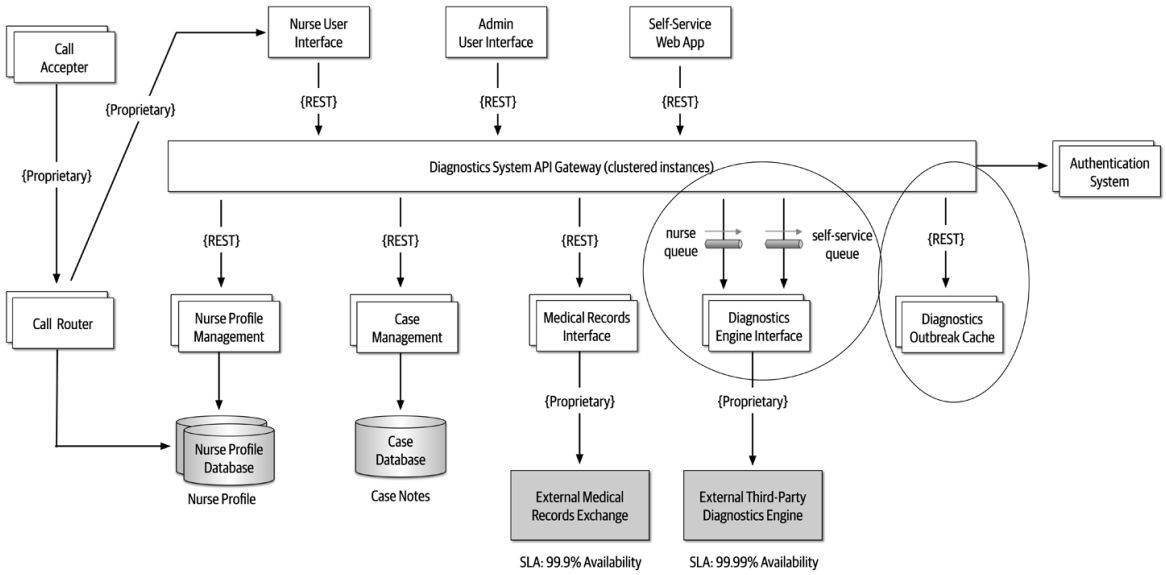

Figure 20-12. Architecture modifications to address elasticity risk

Security

Encouraged by the results and success of the first two risk storming efforts, the architect decides to hold a final risk storming session on another important architecture characteristic that must be supported in the system to ensure its success—security. Due to HIPAA regulatory requirements, access to medical records via the medical record exchange interface must be secure, allowing only nurses to access medical records if needed. The architect believes this is not a problem due to security checks in the API gateway (authentication and authorization) but is curious whether the participants find any other elements of security risk.

During the risk storming, the participants all identified the Diagnostics System API gateway as a high security risk (6). The rationale for this high rating was the high impact of admin staff or self-service patients accessing medical records (3) combined with medium likelihood (2). Likelihood of risk occurring was not rated high because of the security checks for each API call, but still rated medium because all calls (self-service, admin, and nurses) are going through the same API gateway. The architect, who only rated the risk as low (2), was convinced during the risk storming consensus activity that the risk was in fact high and needed mitigation.

The participants all agreed that having separate API gateways for each type of user (admin, self-service/diagnostics, and nurses) would prevent calls from either the admin web user interface or the self-service web user interface from ever reaching the medical records exchange interface. The architect agreed, creating the final architecture, as illustrated in Figure 20-13.

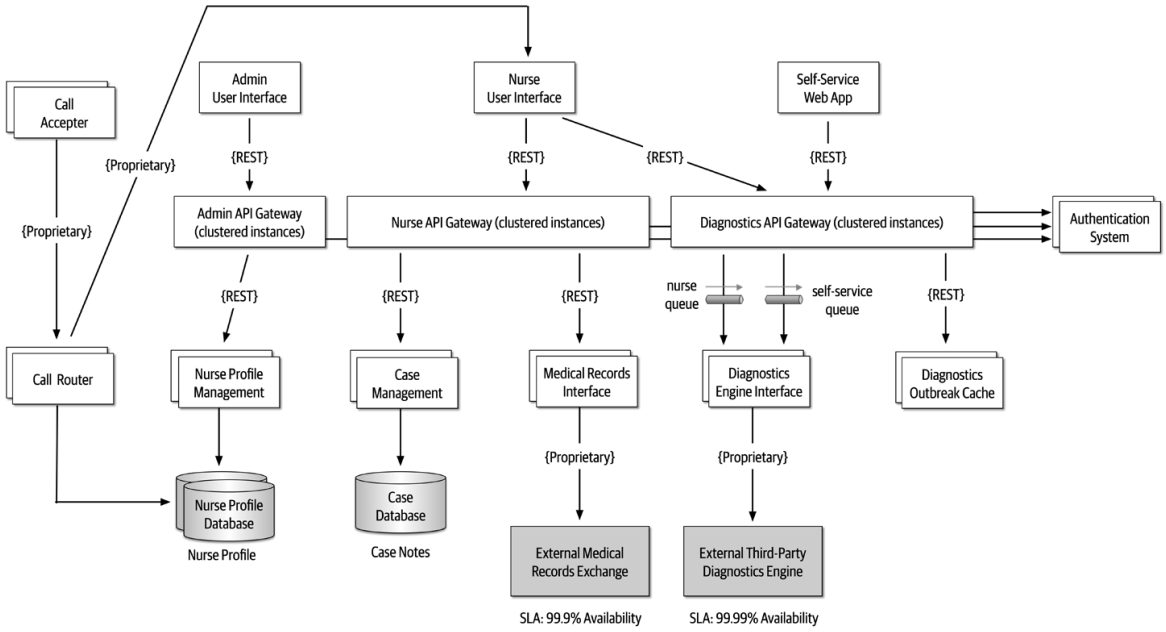

Figure 20-13. Final architecture modifications to address security risk

The prior scenario illustrates the power of risk storming. By collaborating with other architects, developers, and key stakeholders on dimensions of risk that are vital to the success of the system, risk areas are identified that would otherwise have gone unnoticed. Compare figures Figure 20-9 and Figure 20-13 and notice the significant difference in the architecture prior to risk storming and then after risk storming. Those significant changes address availability concerns, elasticity concerns, and security concerns within the architecture.

Risk storming is not a one-time process. Rather, it is a continuous process through the life of any system to catch and mitigate risk areas before they happen in production. How often the risk storming effort happens depends on many factors, including frequency of change, architecture refactoring efforts, and the incremental development of the architecture. It is typical to

undergo a risk storming effort on some particular dimension after a major feature is added or at the end of every iteration.

Chapter 21. Diagramming and Presenting Architecture

Newly minted architects often comment on how surprised they are at how varied the job is outside of technical knowledge and experience, which enabled their move into the architect role to begin with. In particular, effective communication becomes critical to an architect's success. No matter how brilliant an architect's technical ideas, if they can't convince managers to fund them and developers to build them, their brilliance will never manifest.

Diagramming and presenting architectures are two critical soft skills for architects. While entire books exist about each topic, we'll hit some particular highlights for each.

These two topics appear together because they have a few similar characteristics: each forms an important visual representation of an architecture vision, presented using different media. However, representational consistency is a concept that ties both together.

When visually describing an architecture, the creator often must show different views of the architecture. For example, the architect will likely show an overview of the entire architecture topology, then drill into individual parts to delve into design details. However, if the architect shows a portion without indicating where it lies within the overall architecture, it confuses viewers. *Representational consistency* is the practice of always showing the relationship between parts of an architecture, either in diagrams or presentations, before changing views.

For example, if an architect wanted to describe the details of how the plug-ins relate to one another in the Silicon Sandwiches solution, the architecture would show the entire topology, then drill into the plug-in structure,

showing the viewers the relationship between them; an example of this appears in Figure 21-1.

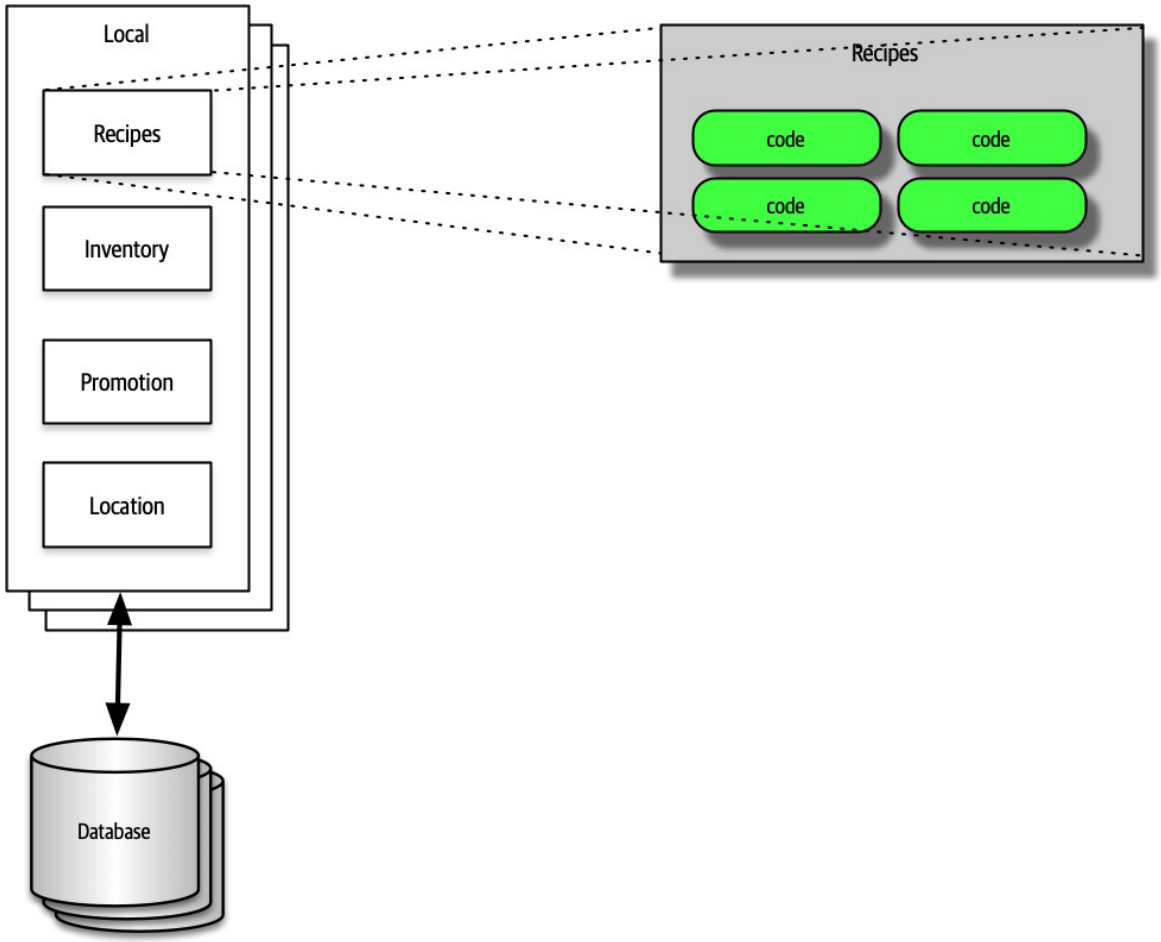

Figure 21-1. Using representational consistency to indicate context in a larger diagram

Careful use of representational consistency ensures that viewers understand the scope of items being presented, eliminating a common source of confusion.

Diagramming

The topology of architecture is always of interest to architects and developers because it captures how the structure fits together and forms a valuable shared understanding across the team. Therefore, architects should hone their diagramming skills to razor sharpness.

Tools

The current generation of diagramming tools for architects is extremely powerful, and an architect should learn their tool of choice deeply. However, before going to a nice tool, don't neglect low-fidelity artifacts, especially early in the design process. Building very ephemeral design artifacts early prevents architects from becoming overly attached to what they have created, an anti-pattern we named the *Irrational Artifact Attachment* anti-pattern.

IRRATIONAL ARTIFACT ATTACHMENT

…is the proportional relationship between a person's irrational attachment to some artifact and how long it took to produce. If an architect creates a beautiful diagram using some tool like Visio that takes two hours, they have an irrational attachment to that artifact that's roughly proportional to the amount of time invested, which also means they will be more attached to a four-hour diagram than a two-hour one.

One of the benefits to the low-ritual approach used in Agile software development revolves around creating just-in-time artifacts, with as little ceremony as possible (this helps explain the dedication of lots of agilists to index cards and sticky notes). Using low-tech tools lets team members throw away what's not right, freeing them to experiment and allow the true nature of the artifact emerge through revision, collaboration, and discussion.

An architect's favorite variation on the cell phone photo of a whiteboard (along with the inevitable "Do Not Erase!" imperative) uses a tablet attached to an overhead projector rather than a whiteboard. This offers several advantages. First, the tablet has an unlimited canvas and can fit as many drawings that a team might need. Second, it allows copy/paste "what if" scenarios that obscure the original when done on a whiteboard. Third, images captured on a tablet are already digitized and don't have the inevitable glare associated with cell phone photos of whiteboards.

Eventually, an architect needs to create nice diagrams in a fancy tool, but make sure the team has iterated on the design sufficiently to invest time in capturing something.

Powerful tools exist to create diagrams on every platform. While we don't necessarily advocate one over another (we quite happily used OmniGraffle for all the diagrams in this book), architects should look for at least this baseline of features:

Layers

> Drawing tools often support layers, which architects should learn well. A layer allows the drawer to link a group of items together logically to enable hiding/showing individual layers. Using layers, an architect can build a comprehensive diagram but hide overwhelming details when they aren't necessary. Using layers also allows architects to incrementally build pictures for presentations later (see "Incremental Builds").

Stencils/templates

> Stencils allow an architect to build up a library of common visual components, often composites of other basic shapes. For example, throughout this book, readers have seen standard pictures of things like microservices, which exist as a single item in the authors' stencil. Building a stencil for common patterns and artifacts within an organization creates consistency within architecture diagrams and allows the architect to build new diagrams quickly.

Magnets

> Many drawing tools offer assistance when drawing lines between shapes. Magnets represent the places on those shapes where lines snap to connect automatically, providing automatic alignment and other visual niceties. Some tools allow the architect to add more magnets or create their own to customize how the connections look within their diagrams.

In addition to these specific helpful features, the tool should, of course, support lines, colors, and other visual artifacts, as well as the ability to export in a wide variety of formats.

Diagramming Standards: UML, C4, and ArchiMate

Several formal standards exist for technical diagrams in software.

UML

Unified Modeling Language (UML) was a standard that unified three competing design philosophies that coexisted in the 1980s. It was supposed to be the best of all worlds but, like many things designed by committee, failed to create much impact outside organizations that mandated its use.

Architects and developers still use UML class and sequence diagrams to communicate structure and workflow, but most of the other UML diagram types have fallen into disuse.

C4

C4 is a diagramming technique developed by Simon Brown to address deficiencies in UML and modernize its approach. The four C's in C4 are as follows:

Context

> Represents the entire context of the system, including the roles of users and external dependencies.

Container

> The physical (and often logical) deployment boundaries and containers within the architecture. This view forms a good meeting point for operations and architects.

Component

> The component view of the system; this most neatly aligns with an architect's view of the system.

Class

> C4 uses the same style of class diagrams from UML, which are effective, so there is no need to replace them.

If a company seeks to standardize on a diagramming technique, C4 offers a good alternative. However, like all technical diagramming tools, it suffers from an inability to express every kind of design an architecture might undertake. C4 is best suited for monolithic architectures where the container and component relationships may differ, and it's less suited to distributed architectures, such as microservices.

ArchiMate

ArchiMate (an amalgam of Arch*itecture-Ani*mate) is an open source enterprise architecture modeling language to support the description, analysis, and visualization of architecture within and across business domains. ArchiMate is a technical standard from The Open Group, and it offers a lighter-weight modeling language for enterprise ecosystems. The goal of ArchiMate is to be "as small as possible," not to cover every edge case scenario. As such, it has become a popular choice among many architects.

Diagram Guidelines

Regardless of whether an architect uses their own modeling language or one of the formal ones, they should build their own style when creating diagrams and should feel free to borrow from representations they think are particularly effective. Here are some general guidelines to use when creating technical diagrams.

Titles

Make sure all the elements of the diagram have titles or are well known to the audience. Use rotation and other effects to make titles "sticky" to the thing they associate with and to make efficient use of space.

Lines

Lines should be thick enough to see well. If lines indicate information flow, then use arrows to indicate directional or two-way traffic. Different types of arrowheads might suggest different semantics, but architects should be consistent.

Generally, one of the few standards that exists in architecture diagrams is that solid lines tend to indicate synchronous communication and dotted lines indicate asynchronous communication.

Shapes

While the formal modeling languages described all have standard shapes, no pervasive standard shapes exist across the software development world. Thus, each architect tends to make their own standard set of shapes, sometimes spreading those across an organization to create a standard language.

We tend to use three-dimensional boxes to indicate deployable artifacts and rectangles to indicate containership, but we don't have any particular key beyond that.

Labels

Architects should label each item in a diagram, especially if there is any chance of ambiguity for the readers.

Color

Architects often don't use color enough—for many years, books were out of necessity printed in black and white, so architects and developers became accustomed to monochrome drawings. While we still favor monochrome, we use color when it helps distinguish one artifact from another. For example, when discussing microservices communication strategies in "Communication", we used color to indicate that two difference microservices participate in the coordination, not two instances of the same service, as reproduced in Figure 21-2.

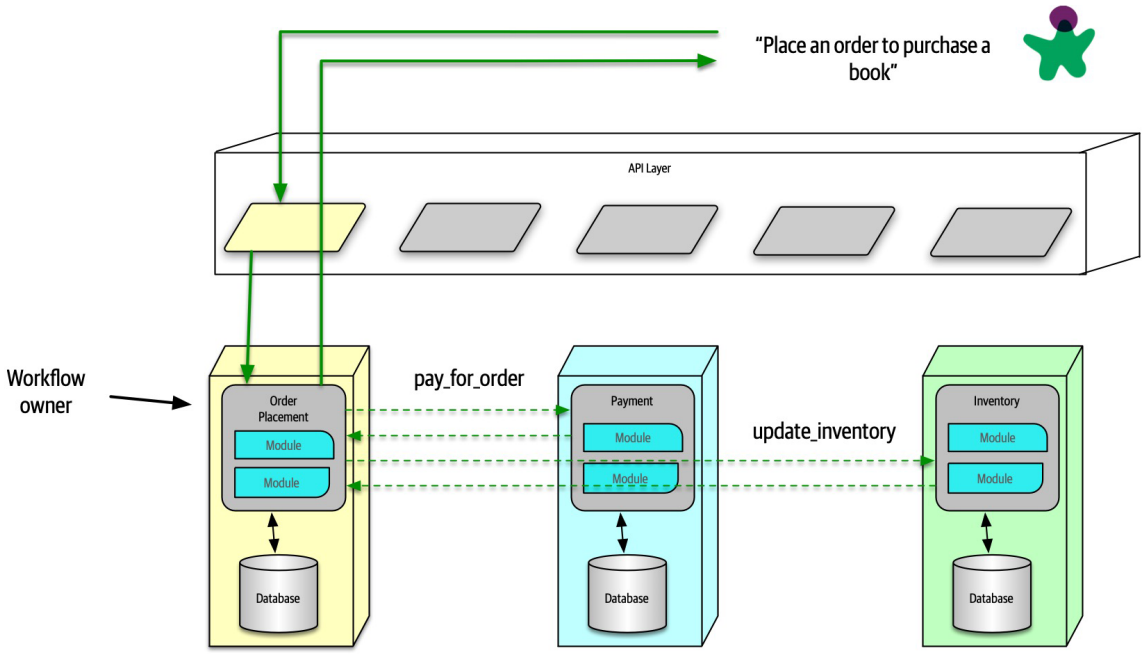

Figure 21-2. Reproduction of microservices communication example showing different services in unique colors

Keys

If shapes are ambiguous for any reason, include a key on the diagram clearly indicating what each shape represents. Nothing is worse than a diagram that leads to misinterpretation, which is worse than no diagram.

Presenting

The other soft skill required by modern architects is the ability to conduct effective presentations using tools like PowerPoint and Keynote. These tools are the lingua franca of modern organizations, and people throughout the organization expect competent use of these tools. Unfortunately, unlike word processors and spreadsheets, no one seems to spend much time studying how to use these tools well.

Neal, one of the coauthors of this book, wrote a book several years ago entitled *Presentation Patterns* (Addison-Wesley Professional), about taking the patterns/anti-patterns approach common in the software world and applying it to technical presentations.

Presentation Patterns makes an important observation about the fundamental difference between creating a document versus a presentation to make a case for something—*time*. In a presentation, the presenter controls how quickly an idea is unfolding, whereas the reader of a document controls that. Thus, one of the most important skills an architect can learn in their presentation tool of choice is how to manipulate time.

Manipulating Time

Presentation tools offer two ways to manipulate time on slides: transitions and animations. Transitions move from slide to slide, and animations allow the designer to create movement within a slide. Typically, presentation tools allow just one transition per slide but a host of animations for each element: build in (appearance), build out (disappearance), and actions (such as movement, scale, and other dynamic behavior).

While tools offer a variety of splashy effects like dropping anvils, architects use transition and animations to hide the boundaries between slides. One common anti-pattern called out in *Presentation Patterns* named Cookie-Cutter states that ideas don't have a predetermined word count, and accordingly, designers shouldn't artificially pad content to make it appear to fill a slide. Similarly, many ideas are bigger than a single slide. Using subtle combinations of transitions and animations such as dissolve allows presenters to hide individual slide boundaries, stitching together a set of slides to tell a single story. To indicate the end of a thought, presenters should use a distinctly different transition (such as door or cube) to provide a visual clue that they are moving to a different topic.

Incremental Builds

The *Presentation Patterns* book calls out the Bullet-Riddled Corpse as a common anti-pattern of corporate presentations, where every slide is essentially the speaker's notes, projected for all to see. Most readers have the excruciating experience of watching a slide full of text appear during a presentation, then reading the entire thing (because no one can resist

reading it all as soon as it appears), only to sit for the next 10 minutes while the presenter slowly reads the bullets to the audience. No wonder so many corporate presentations are dull!

When presenting, the speaker has two information channels: verbal and visual. By placing too much text on the slides and then saying essentially the same words, the presenter is overloading one information channel and starving the other. The better solution to this problem is to use incremental builds for slides, building up (hopefully graphical) information as needed rather than all at once.

For example, say that an architect creates a presentation explaining the problems using feature branching and wants to talk about the negative consequences of keeping branches alive too long. Consider the graphical slide shown in Figure 21-3.

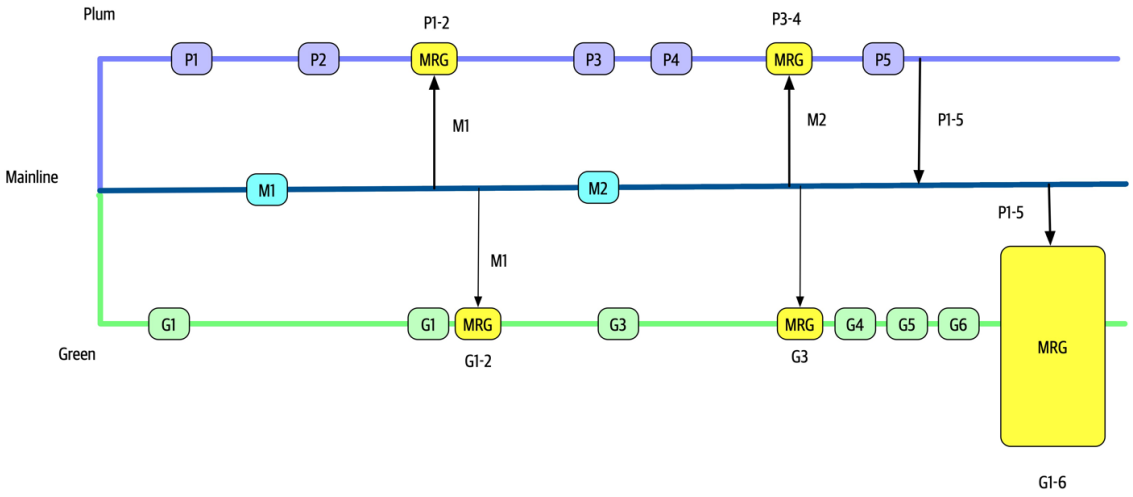

Figure 21-3. Bad version of a slide showing a negative anti-pattern

In Figure 21-3, if the presenter shows the entire slide right away, the audience can see that something bad happens toward the end, but they have to wait for the exposition to get to that point.

Instead, the architect should use the same image but obscure parts of it when showing the slide (using a borderless white box) and expose a portion at a time (by performing a build out on the covering box), as shown in Figure 21-4.

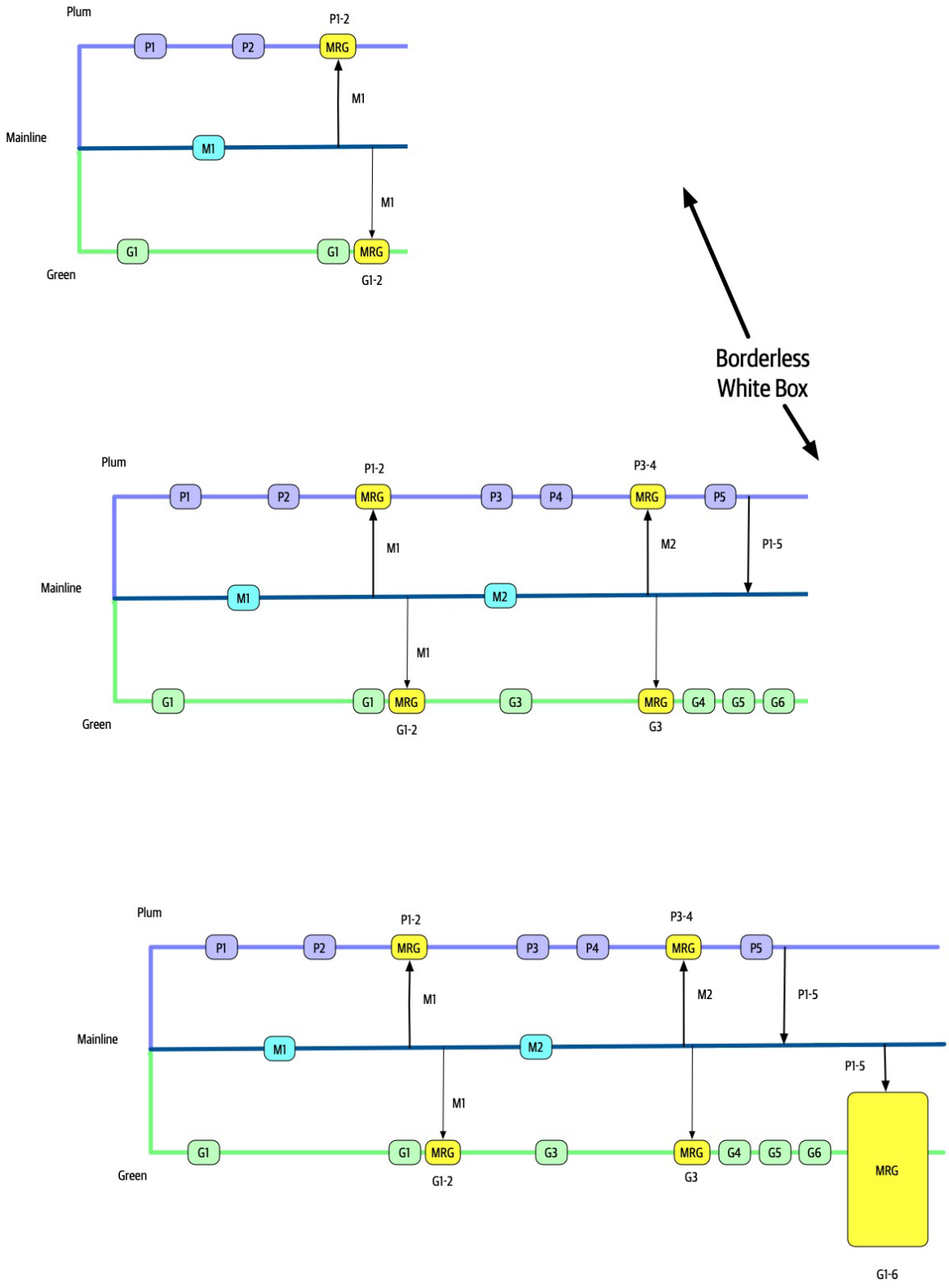

Figure 21-4. A better, incremental version that maintains suspense

In Figure 21-4, the presenter still has a fighting chance of keeping some suspense alive, making the talk inherently more interesting.

Using animations and transitions in conjunction with incremental builds allows the presenter to make more compelling, entertaining presentations.

Infodecks Versus Presentations

Some architects build slide decks in tools like PowerPoint and Keynote but never actually present them. Rather, they are emailed around like a magazine article, and each individual reads them at their own pace. *Infodecks* are slide decks that are not meant to be projected but rather summarize information graphically, essentially using a presentation tool as a desktop publishing package.

The difference between these two media is comprehensiveness of content and use of transitions and animations. If someone is going to flip through the deck like a magazine article, the author of the slides does not need to add any time elements. The other key difference between infodecks and presentations is the amount of material. Because infodecks are meant to be standalone, they contain all the information the creator wants to convey. When doing a presentation, the slides are (purposefully) meant to be half of the presentation, the other half being the person standing there talking!

Slides Are Half of the Story

A common mistake that presenters make is building the entire content of the presentation into the slides. However, if the slides are comprehensive, the presenter should spare everyone the time of sitting through a presentation and just email it to everyone as a deck! Presenters make the mistake of adding too much material to slides when they can make important points more powerfully. Remember, presenters have two information channels, so using them strategically can add more punch to the message. A great example of that is the strategic use of invisibility.

Invisibility

Invisibility is a simple pattern where the presenter inserts a blank black slide within a presentation to refocus attention solely on the speaker. If someone has two information channels (slides and speaker) and turns one of them off (the slides), it automatically adds more emphasis to the speaker. Thus, if a presenter wants to make a point, insert a blank slide—everyone in the room will focus their attention back on the speaker because they are now the only interesting thing in the room to look at.

Learning the basics of a presentation tool and a few techniques to make presentations better is a great addition to the skill set of architects. If an architect has a great idea but can’t figure out a way to present it effectively, they will never get a chance to realize that vision. Architecture requires collaboration; to get collaborators, architects must convince people to sign on to their vision. The modern corporate soapboxes are presentation tools, so it’s worth learning to use them well.

Chapter 22. Making Teams Effective

In addition to creating a technical architecture and making architecture decisions, a software architect is also responsible for guiding the development team through the implementation of the architecture. Software architects who do this well create effective development teams that work closely together to solve problems and create winning solutions. While this may sound obvious, too many times we've seen architects ignore development teams and work in siloed environments to create an architecture. This architecture then gets handed it off to a development team which then struggles to implement the architecture correctly. Being able to make teams productive is one of the ways effective and successful software architects differentiate themselves from other software architects. In this chapter we introduce some basic techniques an architect can leverage to make development teams effective.

Team Boundaries

It's been our experience that a software architect can significantly influence the success or failure of a development team. Teams that feel left out of the loop or estranged from software architects (and also the architecture) often do not have the right level of guidance and right level of knowledge about various constraints on the system, and consequently do not implement the architecture correctly.

One of the roles of a software architect is to create and communicate the constraints, or the box, in which developers can implement the architecture. Architects can create boundaries that are too tight, too loose, or just right. These boundaries are illustrated in Figure 22-1. The impact of having too

tight or too loose of a boundary has a direct impact on the teams' ability to successfully implement the architecture.

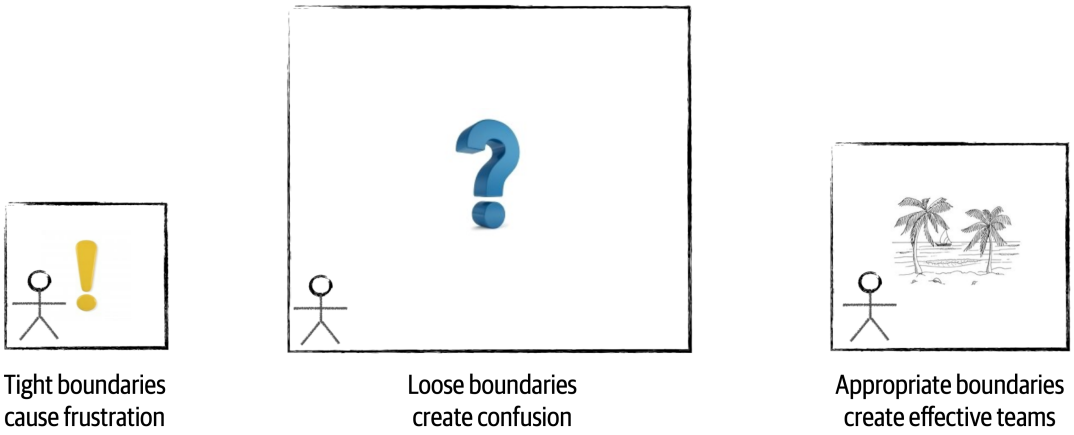

Figure 22-1. Boundary types created by a software architect

Architects that create too many constraints form a tight box around the development teams, preventing access to many of the tools, libraries, and practices that are required to implement the system effectively. This causes frustration within the team, usually resulting in developers leaving the project for happier and healthier environments.

The opposite can also happen. A software architect can create constraints that are too loose (or no constraints at all), leaving all of the important architecture decisions to the development team. In this scenario, which is just as bad as tight constraints, the team essentially takes on the role of a software architect, performing proof of concepts and battling over design decisions without the proper level of guidance, resulting in unproductiveness, confusion, and frustration.

An effective software architect strives to provide the right level of guidance and constraints so that the team has the correct tools and libraries in place to effectively implement the architecture. The rest of this chapter is devoted to how to create these effective boundaries.

Architect Personalities

There are three basic types of architect personalities: a *control freak architect* (Figure 22-2), an *armchair architect* (Figure 22-3), and an *effective architect* (Figure 22-5). Each personality matches a particular boundary type discussed in the prior section on team boundaries: control freak architects produce tight boundaries, armchair architects produce loose boundaries, and effective architects produce just the right kinds of boundaries.

Control Freak

Figure 22-2. Control freak architect (iStockPhoto)

The control freak architect tries to control every detailed aspect of the software development process. Every decision a control freak architect makes is usually too fine-grained and too low-level, resulting in too many constraints on the development team.

Control freak architects produce the tight boundaries discussed in the prior section. A control freak architect might restrict the development team from downloading any useful open source or third-party libraries and instead insist that the teams write everything from scratch using the language API. Control freak architects might also place tight restrictions on naming conventions, class design, method length, and so on. They might even go so far as to write pseudocode for the development teams. Essentially, control freak architects steal the art of programming away from the developers, resulting in frustration and a lack of respect for the architect.

It is very easy to become a control freak architect, particularly when transitioning from developer to architect. An architect's role is to create the building blocks of the application (the components) and determine the interactions between those components. The developer's role in this effort is to then take those components and determine how they will be implemented using class diagrams and design patterns. However, in the transaction from developer to architect, it is all too tempting to want to create the class diagrams and design patterns as well since that was the newly minted architect's prior role.

For example, suppose an architect creates a component (building block of the architecture) to manage reference data within the system. Reference data consists of static name-value pair data used on the website, as well as things like product codes and warehouse codes (static data used throughout the system). The architect's role is to identify the component (in this case, `Reference Manager`), determine the core set of operations for that component (for example, `GetData`, `SetData`, `ReloadCache`, `NotifyOnUpdate`, and so on), and which components need to interact with the `Reference Manager`. The control freak architect might think that the best way to *implement* this component is through a parallel loader pattern leveraging an internal cache, with a particular data structure for that cache. While this might be an effective design, it's not the only design. More importantly, it's no longer the architect's role to come up with this internal design for the `Reference Manager`—it's the role of the developer.

As we'll talk about in "How Much Control?", sometimes an architect needs to play the role of a control freak, depending on the complexity of the project and the skill level on the team. However, in most cases a control freak architect disrupts the development team, doesn't provide the right level of guidance, gets in the way, and is ineffective at leading the team through the implementation of the architecture.

Armchair Architect

Figure 22-3. Armchair architect (iStockPhoto)

The armchair architect is the type of architect who hasn't coded in a very long time (if at all) and doesn't take the implementation details into account when creating an architecture. They are typically disconnected from the development teams, never around, or simply move from project to project once the initial architecture diagrams are completed.

In some cases the armchair architect is simply in way over their head in terms of the technology or business domain and therefore cannot possibly lead or guide teams from a technical or business problem standpoint. For

example, what do developers do? Why, they code, of course. Writing program code is really hard to fake; either a developer writes software code, or they don't. However, what does an architect do? No one knows! Most architects draw lots of lines and boxes—but how detailed should an architect be in those diagrams? Here's a dirty little secret about architecture—it's really easy to fake it as an architect!

Suppose an armchair architect is in way over their head or doesn't have the time to architect an appropriate solution for a stock trading system. In that case the architecture diagram might look like the one illustrated in Figure 22-4. There's nothing wrong with this architecture—it's just too high level to be of any use to anyone.

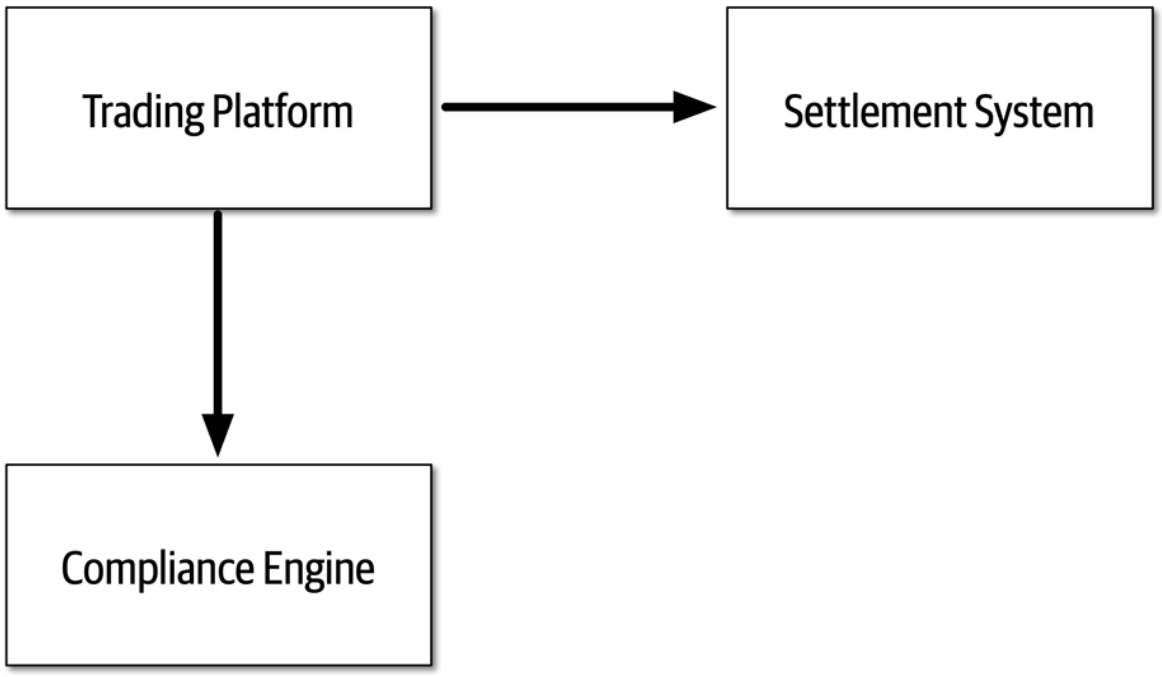

Figure 22-4. Trading system architecture created by an armchair architect

Armchair architects create loose boundaries around development teams, as discussed in the prior section. In this scenario, development teams end up taking on the role of architect, essentially doing the work an architect is supposed to be doing. Team velocity and productivity suffer as a result, and teams get confused about how the system should work.

Like the control freak architect, it is all too easy to become an armchair architect. The biggest indicator that an architect might be falling into the

armchair architect personality is not having enough time to spend with the development teams implementing the architecture (or choosing not to spend time with the development teams). Development teams need an architect's support and guidance, and they need the architect available for answering technical or business-related questions when they arise. Other indicators of an armchair architect are following:

- Not fully understanding the business domain, business problem, or technology used
- Not enough hands-on experience developing software
- Not considering the implications associated with the implementation of the architecture solution

In some cases it is not the intention of an architect to become an armchair architect, but rather it just "happens" by being spread too thin between projects or development teams and loosing touch with technology or the business domain. An architect can avoid this personality by getting more involved in the technology being used on the project and understanding the business problem and business domain.

Effective Architect

Figure 22-5. Effective software architect (iStockPhoto)

An effective software architect produces the appropriate constraints and boundaries on the team, ensuring that the team members are working well together and have the right level of guidance on the team. The effective architect also ensures that the team has the correct and appropriate tools and technologies in place. In addition, they remove any roadblocks that may be in the way of the development teams reaching their goals.

While this sounds obvious and easy, it is not. There is an art to becoming an effective leader on the development team. Becoming an effective software architect requires working closely and collaborating with the team, and gaining the respect of the team as well. We'll be looking at other ways of becoming an effective software architect in later chapters in this part of the book. But for now, we'll introduce some guidelines for knowing how much control an effective architect should exert on a development team.

How Much Control?

Becoming an effective software architect is knowing how much control to exert on a given development team. This concept is known as Elastic Leadership and is widely evangelized by author and consultant Roy Osherove. We're going to deviate a bit from the work Osherove has done in this area and focus on specific factors for software architecture.

Knowing how much an effective software architect should be a control freak and how much they should be an armchair architect involves five main factors. These factors also determine how many teams (or projects) a software architect can manage at once:

Team familiarity

> How well do the team members know each other? Have they worked together before on a project? Generally, the better team members know each other, the less control is needed because team members start to become self-organizing. Conversely, the newer the team members, the more control needed to help facilitate collaboration among team members and reduce cliques within the team.

Team size

> How big is the team? (We consider more than 12 developers on the same team to be a big team, and 4 or fewer to be a small team.) The larger the team, the more control is needed. The smaller the team, less control is needed. This is discussed in more detail in "Team Warning Signs".

Overall experience

> How many team members are senior? How many are junior? Is it a mixed team of junior and senior developers? How well do they know the technology and business domain? Teams with lots of junior developers require more control and mentoring, whereas teams with more senior developers require less control. In the latter cases, the architect moves from the role of a mentor to that of a facilitator.

Project complexity

> Is the project highly complex or just a simple website? Highly complex projects require the architect to be more available to the team and to assist with issues that arise, hence more control is needed on the team. Relatively simple projects are straightforward and hence do not require much control.

Project duration

> Is the project short (two months), long (two years), or average duration (six months)? The shorter the duration, the less control is needed; conversely, the longer the project, the more control is needed.

While most of the factors make sense with regard to more or less control, the project duration factor may not appear to make sense. As indicated in the prior list, the shorter the project duration, the less control is needed; the longer the project duration, the more control is needed. Intuitively this might seem reversed, but that is not the case. Consider a quick two-month project. Two months is not a lot of time to qualify requirements, experiment, develop code, test every scenario, and release into production. In this case the architect should act more as an armchair architect, as the development team already has a keen sense of urgency. A control freak architect would just get in the way and likely delay the project. Conversely, think of a project duration of two years. In this scenario the developers are relaxed, not thinking in terms of urgency, and likely planning vacations and taking long lunches. More control is needed by the architect to ensure the project moves along in a timely fashion and that complex tasks are accomplished first.

It is typical within most projects that these factors are utilized to determine the level of control at the start of a project; but as the system continues to evolve, the level of control changes. Therefore, we advise that these factors continually be analyzed throughout the life cycle of a project to determine how much control to exert on the development team.

To illustrate how each of these factors can be used to determine the level of control an architect should have on a team, assume a fixed scale of 20 points for each factor. Minus values point more toward being an armchair architect (less control and involvement), whereas plus values point more toward being a control freak architect (more control and involvement). This scale is illustrated in Figure 22-6.

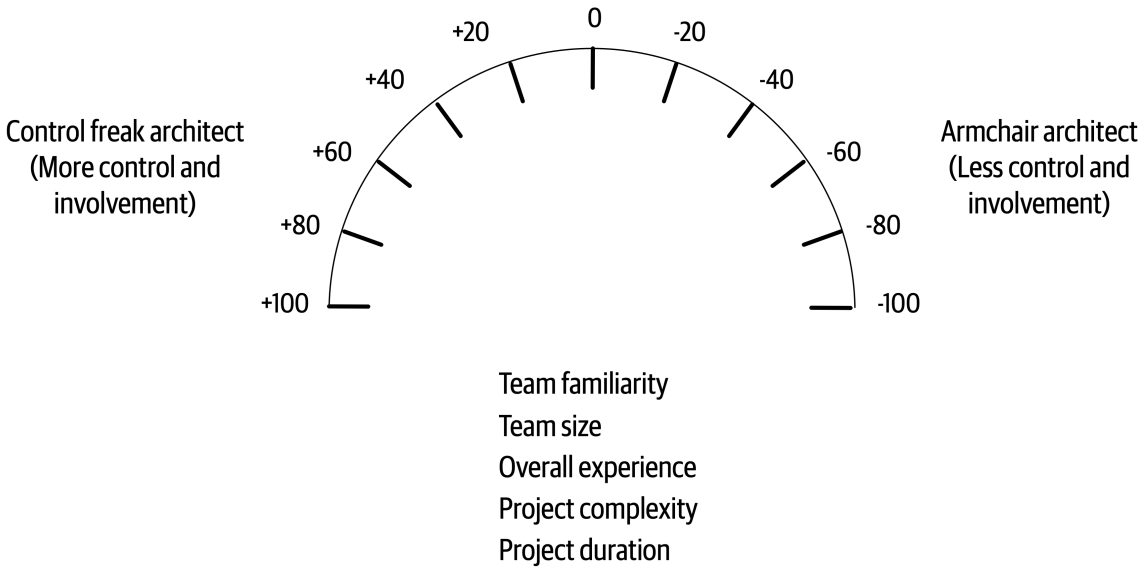

Figure 22-6. Scale for the amount of control

Applying this sort of scaling is not exact, of course, but it does help in determining the relative control to exert on a team. For example, consider the project scenario shown in Table 22-1 and Figure 22-7. As shown in the table, the factors point to either a control freak (+20) or an armchair architect (-20). These factors add up and to an accumulated score of -60, indicating that the architect should play more of an armchair architect role and not get in the team's way.

Table 22-1. Scenario 1 example for amount of control

Factor	Value	Rating	Personality
Team familiarity	New team members	+20	Control freak
Team size	Small (4 members)	-20	Armchair architect
Overall experience	All experienced	-20	Armchair architect
Project complexity	Relatively simple	-20	Armchair architect
Project duration	2 months	-20	Armchair architect
Accumulated score		-60	Armchair architect

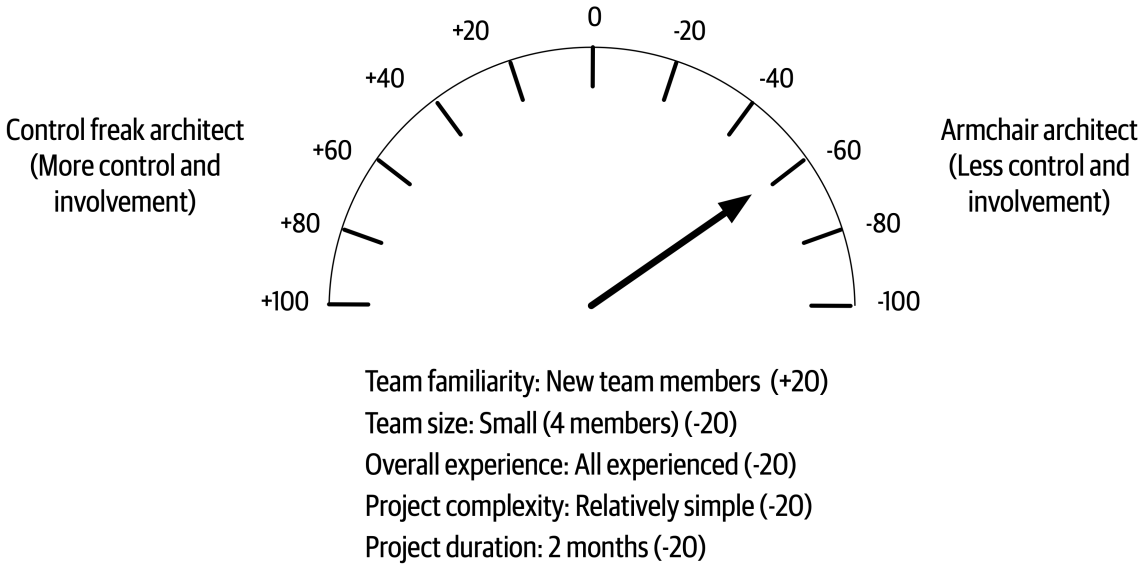

Figure 22-7. Amount of control for scenario 1

In scenario 1, these factors are all taken into account to demonstrate that an effective software architect should initially play the role of facilitator and not get too involved in the day-to-day interactions with the team. The architect will be needed for answering questions and to make sure the team is on track, but for the most part the architect should be largely hands-off and let the experienced team do what they know best—develop software quickly.

Consider another type of scenario described in Table 22-2 and illustrated in Figure 22-8, where the team members know each other well, but the team is large (12 team members) and consists mostly of junior (inexperienced) developers. The project is relatively complex with a duration of six months. In this case, the accumulated score comes out to -20, indicating that the effective architect should be involved in the day-to-day activities within the team and take on a mentoring and coaching role, but not so much as to disrupt the team.

Table 22-2. Scenario 2 example for amount of control

Factor	Value	Rating	Personality
Team familiarity	Know each other well	-20	Armchair architect
Team size	Large (12 members)	+20	Control freak
Overall experience	Mostly junior	+20	Control freak
Project complexity	High complexity	+20	Control freak
Project duration	6 months	-20	Armchair architect
Accumulated score		-20	Control freak

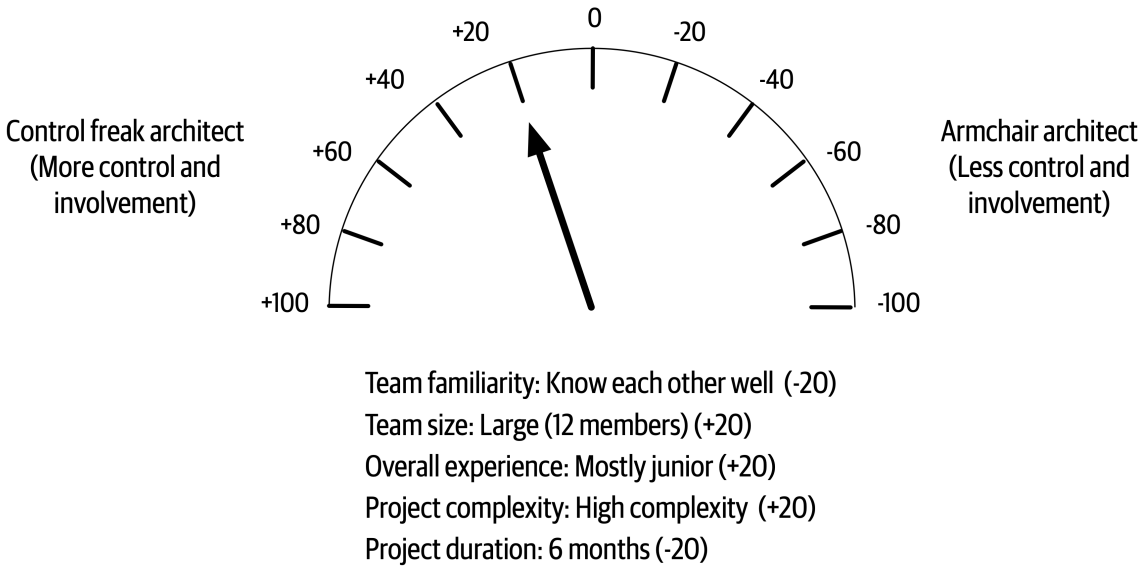

Figure 22-8. Amount of control for scenario 2

It is difficult to objectify these factors, as some of them (such as the overall team experience) might be more weighted than others. In these cases the metrics can easily be weighted or modified to suit any particular scenario or condition. Regardless, the primary message here is that the amount of control and involvement a software architect has on the team varies by these five main factors and that by taking these factors into account, an architect can gauge what sort of control to exert on the team and what the box in which development teams can work in should look like (tight boundaries and constraints or loose ones).

Team Warning Signs

As indicated in the prior section, team size is one of the factors that influence the amount of control an architect should exert on a development team. The larger a team, the more control needed; the smaller the team, the less control needed. Three factors come into play when considering the most effective development team size:

- Process loss
- Pluralistic ignorance

- Diffusion of responsibility

Process loss, otherwise known as Brook's law, was originally coined by Fred Brooks in his book *The Mythical Man Month* (Addison-Wesley). The basic idea of process loss is that the more people you add to a project, the more time the project will take. As illustrated in Figure 22-9, the *group potential* is defined by the collective efforts of everyone on the team. However, with any team, the *actual productivity* will always be less than the group potential, the difference being the *process loss* of the team.

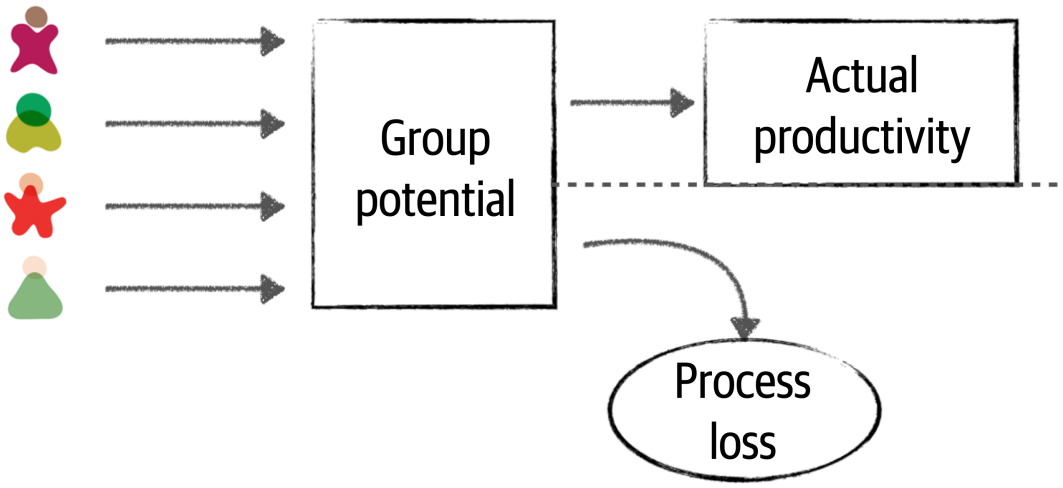

Figure 22-9. Team size impacts actual productivity (Brook's law)

An effective software architect will observe the development team and look for process loss. Process loss is a good factor in determining the correct team size for a particular project or effort. One indication of process loss is frequent merge conflicts when pushing code to a repository. This is an indication that team members are possibly stepping on each other's toes and working on the same code. Looking for areas of parallelism within the team and having team members working on separate services or areas of the application is one way to avoid process loss. Anytime a new team member comes on board a project, if there aren't areas for creating parallel work streams, an effective architect will question the reason why a new team member was added to the team and demonstrate to the project manager the negative impact that additional person will have on the team.

Pluralistic ignorance also occurs as the team size gets too big. Pluralistic ignorance is when everyone agrees to (but privately rejects) a norm because they think they are missing something obvious. For example, suppose on a large team the majority agree that using messaging between two remote services is the best solution. However, one person on the team thinks this is a silly idea because of a secure firewall between the two services. However, rather than speak up, that person also agrees to the use of messaging (but privately rejects the idea) because they are afraid that they are either missing something obvious or afraid they might be seen as a fool if they were to speak up. In this case, the person rejecting the norm was correct—messaging would not work because of a secure firewall between the two remote services. Had they spoken up (and had the team size been smaller), the original solution would have been challenged and another protocol (such as REST) used instead, which would be a better solution in this case.

The concept of pluralistic ignorance was made famous by the Danish children's story "The Emperor's New Clothes", by Hans Christian Andersen. In the story, the king is convinced that his new clothes are invisible to anyone unworthy to actually see them. He struts around totally nude, asking all of his subjects how they like his new clothes. All the subjects, afraid of being considered stupid or unworthy, respond to the king that his new clothes are the best thing ever. This folly continues until a child finally calls out to the king that he isn't wearing any clothes at all.

An effective software architect should continually observe facial expressions and body language during any sort of collaborative meeting or discussion and act as a facilitator if they sense an occurrence of pluralistic ignorance. In this case, the effective architect might interrupt and ask the person what they think about the proposed solution and be on their side and support them when they speak up.

The third factor that indicates appropriate team size is called *diffusion of responsibility*. Diffusion of responsibility is based on the fact that as team size increases, it has a negative impact on communication. Confusion about who is responsible for what on the team and things getting dropped are good signs of a team that is too large.

Look at the picture in Figure 22-10. What do you observe?

Figure 22-10. Diffusion of responsibility

This picture shows someone standing next to a broken-down car on the side of a small country road. In this scenario, how many people might stop and ask the motorist if everything is OK? Because it's a small road in a small community, probably everyone who passes by. However, how many times have motorists been stuck on the side of a busy highway in the middle of a large city and had thousands of cars simply drive by without anyone stopping and asking if everything is OK? All the time. This is a good example of the diffusion of responsibility. As cities get busier and more crowded, people assume the motorist has already called or help is on the way due to the large number of people witnessing the event. However, in most of these cases help is not on the way, and the motorist is stuck with a dead or forgotten cell phone, unable to call for help.

An effective architect not only helps guide the development team through the implementation of the architecture, but also ensures that the team is healthy, happy, and working together to achieve a common goal. Looking for these three warning signs and consequently helping to correct them helps to ensure an effective development team.

Leveraging Checklists

Airline pilots use checklists on every flight—even the most experienced, seasoned veteran pilots. Pilots have checklists for takeoff, landing, and thousands of other situations, both common and unusual edge cases. They use checklists because one missed aircraft setting (such as setting the flaps to 10 degrees) or procedure (such as gaining clearance into a terminal control area) can mean the difference between a safe flight and a disastrous one.

Dr. Atul Gawande wrote an excellent book called *The Checklist Manifesto* (Picador), in which he describes the power of checklists for surgical procedures. Alarmed at the high rate of staph infections in hospitals, Dr. Gawande created surgical checklists to attempt to reduce this rate. In the book he demonstrates that staph infection rates in hospitals using the checklists went down to near zero, while staph infection rates in control hospitals not using the checklists continued to rise.

Checklists work. They provide an excellent vehicle for making sure everything is covered and addressed. If checklists work so well, then why doesn't the software development industry leverage them? We firmly believe through personal experience that checklists make a big difference in the effectiveness of development teams. However, there are caveats to this claim. First, most software developers are not flying airliners or performing open heart surgery. In other words, software developers don't require checklists for everything. The key to making teams effective is knowing when to leverage checklists and when not to.

Consider the checklist shown in Figure 22-11 for creating a new database table.

Done	Task description
☐	Determine database column field names and types
☐	Fill out database table request form
☐	Obtain permission for new database table
☐	Submit request form to database group
☐	Verify table once created

Figure 22-11. Example of a bad checklist

This is not a checklist, but a set of procedural steps, and as such should not be in a checklist. For example, the database table cannot be verified if the form has not yet been submitted! Any processes that have a procedural flow of dependent tasks should not be in a checklist. Simple, well-known processes that are executed frequently without error also do not need a checklist.

Processes that are good candidates for checklists are those that don't have any procedural order or dependent tasks, as well as those that tend to be error-prone or have steps that are frequently missed or skipped. The key to making checklists effective is to not go overboard making everything a checklist. Architects find that checklists do, in fact, make development teams more effective, and as such start to make everything a checklist, invoking what is known as the *law of diminishing returns*. The more checklists an architect creates, the less chance developers will use them. Another key success factor when creating checklists is to make them as small as possible while still capturing all the necessary steps within a process. Developers generally will not follow checklists that are too big. Seek items that can be performed through automation and remove those from the checklist.

> **TIP**
>
> Don't worry about stating the obvious in a checklist. It's the obvious stuff that's usually skipped or missed.

Three key checklists that we've found to be most effective are a *developer code completion* checklist, a *unit and functional testing* checklist, and a *software release* checklist. Each checklist is discussed in the following sections.

THE HAWTHORNE EFFECT

One of the issues associated with introducing checklists to a development team is making developers actually use them. It's all too common for some developers to run out of time and simply mark all the items in a particular checklist as completed without having actually performed the tasks.

One of the ways of addressing this issue is by talking with the team about the importance of using checklists and how checklists can make a difference in the team. Have team members read *The Checklist Manifesto* by Atul Gawande to fully understand the power of a checklist, and make sure each team member understands the reasoning behind each checklist and why it is being used. Having developers collaborate on what should and shouldn't be on a checklist also helps.

When all else fails, architects can invoke what is known as the Hawthorne effect. The Hawthorne effect essentially means that if people know they are being observed or monitored, their behavior changes, and generally they will do the right thing. Examples include highly visible cameras in and around buildings that actually don't work or aren't really recording anything (this is very common!) and website monitoring software (how many of those reports are actually viewed?).

The Hawthorne effect can be used to govern the use of checklists as well. An architect can let the team know that the use of checklists is critical to the team's effectiveness, and as a result, all checklists will be verified to make sure the task was actually performed, when in fact the architect is only occasionally spot-checking the checklists for correctness. By leveraging the Hawthorne effect, developers will be much less likely to skip items or mark them as completed when in fact the task was not done.

Developer Code Completion Checklist

The developer code completion checklist is an effective tool to use, particularly when a software developer states that they are “done” with the code. It also is useful for defining what is known as the “definition of done.” If everything in the checklist is completed, then the developer can claim they are actually done with the code.

Here are some of the things to include in a developer code completion checklist:

- Coding and formatting standards not included in automated tools
- Frequently overlooked items (such as absorbed exceptions)
- Project-specific standards
- Special team instructions or procedures

Figure 22-12 illustrates an example of a developer code completion checklist.

Done	Task description
☐	Run code cleanup and code formatting
☐	Execute custom source validation tool
☐	Verify the audit log is written for all updates
☐	Make sure there are no absorbed exceptions
☐	Check for hardcoded values and convert to constants
☐	Verify that only public methods are calling setFailure()
☐	Include @ServiceEntrypoint on service API class

Figure 22-12. Example of a developer code completion checklist

Notice the obvious tasks “Run code cleanup and code formatting” and “Make sure there are no absorbed exceptions” in the checklist. How may times has a developer been in a hurry either at the end of the day or at the end of an iteration and forgotten to run code cleanup and formatting from the IDE? Plenty of times. In *The Checklist Manifesto*, Gawande found this

same phenomenon with respect to surgical procedures—the obvious ones were often the ones that were usually missed.

Notice also the project-specific tasks in items 2, 3, 6, and 7. While these are good items to have in a checklist, an architect should always review the checklist to see if any items can be automated or written as plug-in for a code validation checker. For example, while “Include @ServiceEntrypoint on service API class” might not be able to have an automated check, the “Verify that only public methods are calling setFailure()” certainly can (this is a straightforward automated check with any sort of code crawling tool). Checking for areas of automation helps reduce both the size and the noise within a checklist, making it more effective.

Unit and Functional Testing Checklist

Perhaps one of the most effective checklists is a unit and functional testing checklist. This checklist contains some of the more unusual and edge-case tests that software developers tend to forget to test. Whenever someone from QA finds an issue with the code based on a particular test case, that test case should be added to this checklist.

This particular checklist is usually one of the largest ones due to all the types of tests that can be run against code. The purpose of this checklist is to ensure the most complete coding possible so that when the developer is done with the checklist, the code is essentially production ready.

Here are some of the items found in a typical unit and functional testing checklist:

- Special characters in text and numeric fields
- Minimum and maximum value ranges
- Unusual and extreme test cases
- Missing fields

Like the developer code completion checklist, any items that can be written as automated tests should be removed from the checklist. For example, suppose there is an item in the checklist for a stock trading application to test for negative shares (such as a BUY for –1,000 shares of Apple [AAPL]). If this check is automated through a unit or functional test within the test suite, then the item should be removed from the checklist.

Developers sometimes don't know where to start when writing unit tests or how many unit tests to write. This checklist provides a way of making sure general or specific test scenarios are included in the process of developing the software. This checklist is also effective in bridging the gap between developers and testers in environments that have these activities performed by separate teams. The more development teams perform complete testing, the easier the job of the testing teams, allowing the testing teams to focus on certain business scenarios not covered in the checklists.

Software Release Checklist

Releasing software into production is perhaps one of the most error-prone aspects of the software development life cycle, and as such makes for a great checklist. This checklist helps avoid failed builds and failed deployments, and it significantly reduces the amount of risk associated with releasing software.

The software release checklist is usually the most volatile of the checklists in that it continually changes to address new errors and circumstances each time a deployment fails or has issues.

Here are some of the items typically included within the software release checklist:

- Configuration changes in servers or external configuration servers
- Third-party libraries added to the project (JAR, DLL, etc.)
- Database updates and corresponding database migration scripts

Anytime a build or deployment fails, the architect should analyze the root cause of the failure and add a corresponding entry to the software release checklist. This way the item will be verified on the next build or deployment, preventing that issue from happening again.

Providing Guidance

A software architect can also make teams effective by providing guidance through the use of design principles. This also helps form the box (constraints), as described in the first section of this chapter, that developers can work in to implement the architecture. Effectively communicating these design principles is one of the keys to creating a successful team.

To illustrate this point, consider providing guidance to a development team regarding the use of what is typically called the *layered stack*—the collection of third-party libraries (such as JAR files, and DLLs) that make up the application. Development teams usually have lots of questions regarding the layered stack, including whether they can make their own decisions about various libraries, which ones are OK, and which ones are not.

Using this example, an effective software architect can provide guidance to the development team by first having the developer answer the following questions:

1. Are there any overlaps between the proposed library and existing functionality within the system?
2. What is the justification for the proposed library?

The first question guides developers to looking at the existing libraries to see if the functionality provided by the new library can be satisfied through an existing library or existing functionality. It has been our experience that developers sometimes ignore this activity, creating lots of duplicate functionality, particularly in large projects with large teams.

The second question prompts the developer into questioning why the new library or functionality is truly needed. Here, an effective software architect will ask for both a technical justification as well as a business justification as to why the additional library is needed. This can be a powerful technique to create awareness within the development team of the need for business justifications.

THE IMPACT OF BUSINESS JUSTIFICATIONS

One of your authors (Mark) was the lead architect on a particularly complex Java-based project with a large development team. One of the team members was particularly obsessed with the Scala programming language and desperately wanted to use it on the project. This desire for the use of Scala ended up becoming so disruptive that several key team members informed Mark that they were planning on leaving the project and moving on to other, "less toxic," environments. Mark convinced the two key team members to hold off on their decision for a bit and had a discussion with the Scala enthusiast. Mark told the Scala enthusiast that he would support the use of Scala within the project, but the Scala enthusiast would have to provide a business justification for the use of Scala because of the training costs and rewriting effort involved. The Scala enthusiast was ecstatic and said he would get right on it, and he left the meeting yelling, "Thank you—you're the best!"

The next day the Scala enthusiast came into the office completely transformed. He immediately approached Mark and asked to speak with him. They both went into the conference room, and the Scala enthusiast immediately (and humbly) said, "Thank you." The Scala enthusiast explained to Mark that he could come up with all the technical reasons in the world to use Scala, but none of those technical advantages had any sort of business value in terms of the architecture characteristics needed ("-ilities"): cost, budget, and timeline. In fact, the Scala enthusiast realized that the increase in cost, budget, and timeline would provide no benefit whatsoever.

Realizing what a disruption he was, the Scala enthusiast quickly transformed himself into one of the best and most helpful members on the team, all because of being asked to provide a business justification for something he wanted on the project. This increased awareness of justifications not only made him a better software developer, but also made for a stronger and healthier team.

As a postscript, the two key developers stayed on the project until the very end.

Continuing with the example of governing the layered stack, another effective technique of communicating design principles is through graphical explanations about what the development team can make decisions on and what they can't. The illustration in Figure 22-13 is an example of what this graphic (as well as the guidance) might look like for controlling the layered stack.

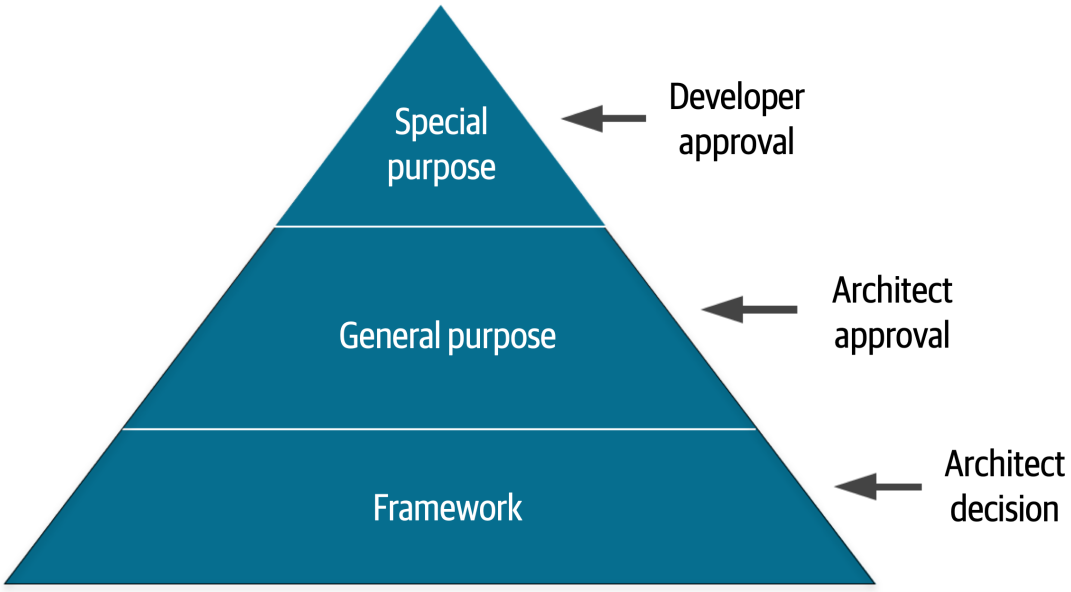

Figure 22-13. Providing guidance for the layered stack

In Figure 22-13, an architect would provide examples of what each category of the third-party library would contain and then what the guidance is (the design principle) in terms of what the developers can and can't do (the box described in the first section of the chapter). For example, here are the three categories defined for any third-party library:

Special purpose

> These are specific libraries used for things like PDF rendering, bar code scanning, and circumstances that do not warrant writing custom software.

General purpose

> These libraries are wrappers on top of the language API, and they include things like Apache Commons, and Guava for Java.

Framework

> These libraries are used for things like persistence (such as Hibernate) and inversion of control (such as Spring). In other words, these libraries make up an entire layer or structure of the application and are highly invasive.

Once categorized (the preceding categories are only an example—there can be many more defined), the architect then creates the box around this design principle. Notice in the example illustrated in Figure 22-13 that for this particular application or project, the architect has specified that for special-purpose libraries, the developer can make the decision and does not need to consult the architect for that library. However, notice that for general purpose, the architect has indicated that the developer can undergo overlap analysis and justification to make the recommendation, but that category of library requires architect approval. Finally, for framework libraries, that is an architect decision—in other words, the development teams shouldn't even undergo analysis for these types of libraries; the architect has decided to take on that responsibility for those types of libraries.

Summary

Making development teams effective is hard work. It requires lots of experience and practice, as well as strong people skills (which we will discuss in subsequent chapters in this book). That said, the simple techniques described in this chapter about elastic leadership, leveraging checklists, and providing guidance through effectively communicating design principles do, in fact, work, and have proven effective in making development teams work smarter and more effectively.

One might question the role of an architect for such activities, instead assigning the effort of making teams effective to the development manager or project manager. We strongly disagree with this premise. A software architect not only provides technical guidance to the team, but also leads the team through the implementation of the architecture. The close collaborative relationship between a software architect and a development team allows the architect to observe the team dynamics and hence facilitate changes to make the team more effective. This is exactly what differentiates a technical architect from an effective software architect.

Chapter 23. Negotiation and Leadership Skills

Negotiation and leadership skills are hard skills to obtain. It takes many years of learning, practice, and “lessons learned” experiences to gain the necessary skills to become an effective software architect. Knowing that this book cannot make an architect an expert in negotiation and leadership overnight, the techniques introduced in this chapter provide a good starting point for gaining these important skills.

Negotiation and Facilitation

In the beginning of this book, we listed the core expectations of an architect, the last being the expectation that a software architect must understand the political climate of the enterprise and be able to navigate the politics. The reason for this key expectation is that almost every decision a software architect makes will be challenged. Decisions will be challenged by developers who think they know more than the architect and hence have a better approach. Decisions will be challenged by other architects within the organization who think they have a better idea or way of approaching the problem. Finally, decisions will be challenged by stakeholders who will argue that the decision is too expensive or will take too much time.

Consider the decision of an architect to use database clustering and federation (using separate physical domain-scoped database instances) to mitigate risk with regard to overall availability within a system. While this is a sound solution to the issue of database availability, it is also a costly decision. In this example, the architect must negotiate with key business stakeholders (those paying for the system) to come to an agreement about the trade-off between availability and cost.

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

Negotiating with Business Stakeholders

Consider the following real-world scenario (scenario 1) involving a key business stakeholder and lead architect:

Scenario 1

> The senior vice president project sponsor is insistent that the new trading system must support five nines of availability (99.999%). However, the lead architect is convinced, based on research, calculations, and knowledge of the business domain and technology, that three nines of availability (99.9%) would be sufficient. The problem is, the project sponsor does not like to be wrong or corrected and really hates people who are condescending. The sponsor isn't overly technical (but thinks they are) and as a result tends to get involved in the nonfunctional aspects of projects. The architect must convince the project sponsor through negotiation that three nines (99.9%) of availability would be enough.

In this sort of negotiation, the software architect must be careful to not be too egotistical and forceful in their analysis, but also make sure they are not missing anything that might prove them wrong during the negotiation. There are several key negotiation techniques an architect can use to help with this sort of stakeholder negotiation.

TIP

Leverage the use of grammar and buzzwords to better understand the situation.

Phases such as “we must have zero downtime” and “I needed those features yesterday” are generally meaningless but nevertheless provide valuable information to the architect about to enter into a negotiation. For example, when the project sponsor is asked when a particular feature is needed and responds, “I needed it yesterday,” that is an indication to the software architect that time to market is important to that stakeholder. Similarly, the phrase “this system must be lightning fast” means performance is a big concern. The phase “zero downtime” means that availability is critical in the application. An effective software architect will leverage this sort of nonsense grammar to better understand the real concerns and consequently leverage that use of grammar during a negotiation.

Consider scenario 1 described previously. Here, the key project sponsor wants five nines of availability. Leveraging this technique tells the architect that availability is very important. This leads to a second negotiation technique:

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

The phrase “five nines” is grammar that indicates high availability. However, what exactly is five nines of availability? Researching this ahead of time and gathering knowledge prior to the negotiation yields the information shown in Table 23-1.

Table 23-1. 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 is 5 minutes and 35 seconds of downtime per year, or 1 second a day of unplanned downtime. Quite ambitious, but also quite costly and unnecessary for the prior example. Putting things in hours and minutes (or in this case, seconds) is a much better way to have the conversation than sticking with the nines vernacular.

Negotiating scenario 1 would include validating the stakeholder’s concerns (“I understand that availability is very important for this system”) and then bringing the negotiation from the nines vernacular to one of reasonable hours and minutes of unplanned downtime. Three nines (which the architect deemed good enough) averages 86 seconds of unplanned downtime per day—certainly a reasonable number given the context of the global trading system described in the scenario. The architect can always resort to this tip:

> **TIP**
>
> When all else fails, state things in terms of 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 such as, "That's going to cost a lot of money" or "We don't have time for that." Money and time (effort involved) are certainly key factors in any negotiation but should be used as a last resort so that other justifications and rationalizations that matter more be tried first. Once an agreement is reached, then cost and time can be considered if they are important attributes to the negotiation.

Another important negotiation technique to always remember is the following, particularly in situations as described in scenario 1:

> **TIP**
>
> Leverage the "divide and conquer" rule to qualify demands or requirements.

The ancient Chinese warrior Sun Tzu wrote in *The Art of War*, "If his forces are united, separate them." This same divide-and-conquer tactic can be applied by an architect during negotiations as well. Consider scenario 1 previously described. In this case, the project sponsor is insisting on five nines (99.999%) of availability for the new trading system. However, does the entire system need five nines of availability? Qualifying the requirement to the specific area of the system actually requiring five nines of availability reduces the scope of difficult (and costly) requirements and the scope of the negotiation as well.

Negotiating with Other Architects

Consider the following actual scenario (scenario 2) between a lead architect and another architect on the same project:

Scenario 2

> The lead architect on a project believes that asynchronous messaging would be the right approach for communication between a group of

services to increase both performance and scalability. However, the other architect on the project once again strongly disagrees and insists that REST would be a better choice, because REST is always faster than messaging and can scale just as well ("see for yourself by Googling it!"). This is not the first heated debate between the two architects, nor will it be the last. The lead architect must convince the other architect that messaging is the right solution.

In this scenario, the lead architect can certainly tell the other architect that their opinion doesn't matter and ignore it based on the lead architect's seniority on the project. However, this will only lead to further animosity between the two architects and create an unhealthy and noncollaborative relationship, and consequently will end up having a negative impact on the development team. The following technique will help in these types of situations:

> **TIP**
>
> Always remember that *demonstration defeats discussion*.

Rather than arguing with another architect over the use of REST versus messaging, the lead architect should demonstrate to the other architect how messaging would be a better choice in their specific environment. Every environment is different, which is why simply Googling it will never yield the correct answer. By running a comparison between the two options in a production-like environment and showing the other architect the results, the argument would likely be avoided.

Another key negotiation technique that works in these situations is as follows:

> **TIP**
>
> Avoid being too argumentative or letting things get too personal in a negotiation—calm leadership combined with clear and concise reasoning will always win a negotiation.

This technique is a very powerful tool when dealing with adversarial relationships like the one described in scenario 2. Once things get too personal or argumentative, the best thing to do is stop the negotiation and reengage at a later time when both parties have calmed down. Arguments will happen between architects; however, approaching these situations with calm leadership will usually force the other person to back down when things get too heated.

Negotiating with Developers

Effective software architects don't leverage their title as architect to tell developers what to do. Rather, they work with development teams to gain respect so that when a request is made of the development team, it doesn't end up in an argument or resentment. Working with development teams can be difficult at times. In many cases development teams feel disconnected from the architecture (and also the architect), and as a result feel left out of the loop with regard to decisions the architect makes. This is a classic example of the *Ivory Tower* architecture anti-pattern. Ivory tower architects are ones who simply dictate from on high, telling development teams what to do without regard to their opinion or concerns. This usually leads to a loss of respect for the architect and an eventual breakdown of the team dynamics. One negotiation technique that can help address this situation is to always provide a justification:

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

By providing a reason why something needs to be done, developers will more likely agree with the request. For example, consider the following conversation between an architect and a developer with regard to making a simple query within a traditional n-tiered layered architecture:

***Architect**: "You must go through the business layer to make that call."*
***Developer**: "No. It's much faster just to call the database directly."*

There are several things wrong with this conversation. First, notice the use of the words "you must." This type of commanding voice is not only demeaning, but is one of the worst ways to begin a negotiation or conversation. Also notice that the developer responded to the architect's demand with a reason to counter the demand (going through the business layer will be slower and take more time). Now consider an alternative approach to this demand:

***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 the performance issues for simple queries?"*

Notice here the architect is providing the justification for the demand that all requests need to go through the business layer of the application. Providing the justification or reason first is always a good approach. Most of the time, once a person hears something they disagree with, they stop listening. By stating the reason first, the architect is sure that the justification will be heard. Also notice the architect removed the personal nature of this demand. By not saying "you must" or "you need to," the architect effectively turned the demand into a simple statement of fact ("this means…"). Now take a look at the developer's response. Notice the conversation shifted from disagreeing with the layered architecture restrictions to a question about increasing performance for simple calls. Now the architect and developer 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 when negotiating with a developer or trying to convince them to accept a particular design or architecture decision they disagree with is to have the developer arrive at the solution on their own. This creates a win-win situation where the architect cannot lose. For example, suppose an architect is choosing between two frameworks, Framework X and Framework Y. The architect sees that Framework Y doesn't satisfy the security requirements for the system and so naturally chooses Framework X. A developer on the team strongly disagrees and insists that Framework Y would still be the better choice. Rather than argue the matter, the architect tells the developer that the team will use Framework Y if the developer can show how to address the security requirements if Framework Y is used. One of two things will happen:

1. The developer will fail trying to demonstrate that Framework Y will satisfy the security requirements and will understand firsthand that the framework cannot be used. By having the developer arrive at the solution on their own, the architect automatically gets buy-in and agreement for the decision to use Framework X by essentially making it the developer's decision. This is a win.
2. The developer finds a way to address the security requirements with Framework Y and demonstrates this to the architect. This is a win as well. In this case the architect missed something in Framework Y, and it also ended up being a better framework over the other one.

TIP

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

It's really through collaboration with the development team that the architect is able to gain the respect of the team and form better solutions. The more developers respect an architect, the easier it will be for the architect to negotiate with those developers.

The Software Architect as a Leader

A software architect is also a leader, one who can guide a development team through the implementation of the architecture. We maintain that about 50% of being an effective software architect is having good people skills, facilitation skills, and leadership skills. In this section we discuss several key leadership techniques that an effective software architect can leverage to lead development teams.

The 4 C's of Architecture

Each day things seem to be getting more and more complex, whether it be increased complexity in business processes or increased complexity of systems and even architecture. Complexity exists within architecture as well as software development, and always will. Some architectures are very complex, such as ones supporting 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."

One of the traps many architects fall into is adding unnecessary complexity to solutions, diagrams, and documentation. Architects (as well as developers) seem to love complexity. To quote Neal:

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

Consider the diagram in Figure 23-1 illustrating the major information flows for the backend processing systems at a very large global bank. Is this necessarily complex? No one knows the answer to this question because the architect has made it complex. This sort of complexity is called *accidental complexity*—in other words, "we have made a problem hard." Architects sometimes do this to prove their worth when things seem too simple or to guarantee that they are always kept in the loop on discussions and decisions that are made regarding the business or architecture. Other architects do this to maintain job security. Whatever the reason, introducing accidental

complexity into something that is not complex is one of the best ways to become an ineffective leader as an architect.

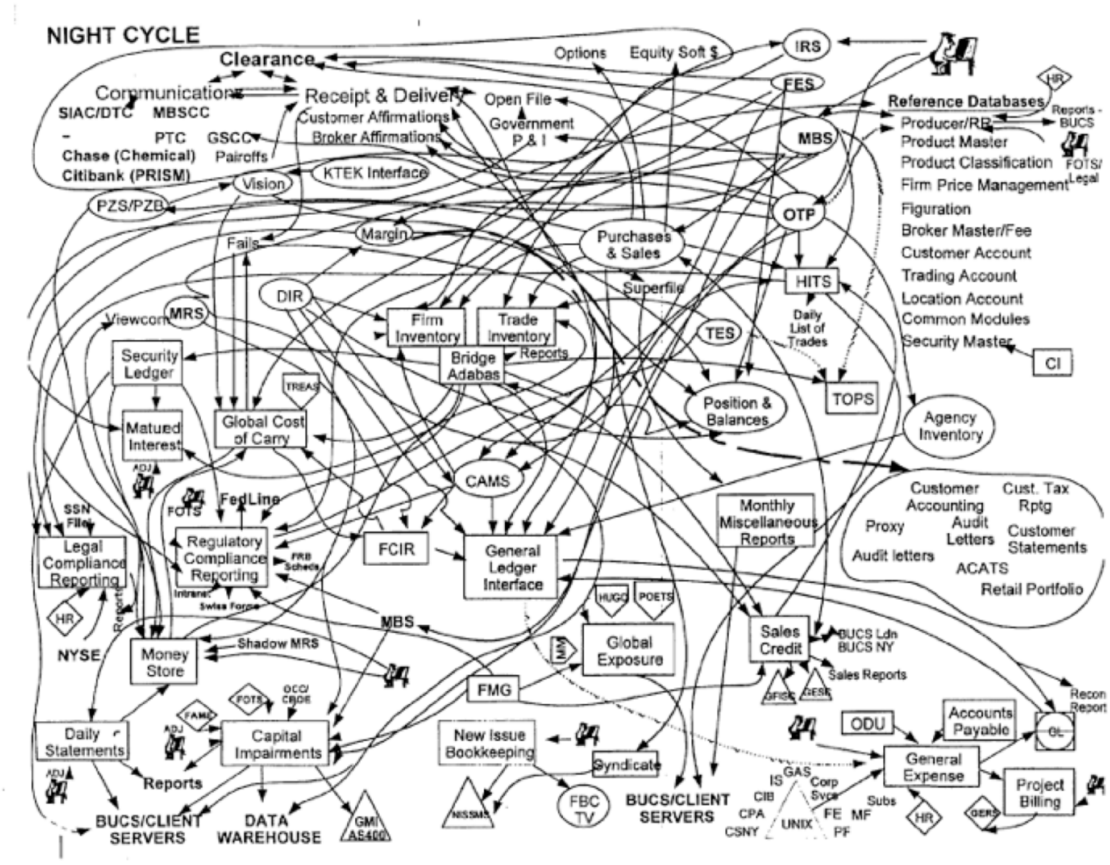

Figure 23-1. Introducing accidental complexity into a problem

An effective way of avoiding accidental complexity is what we call the *4 C's* of architecture: *communication*, *collaboration*, *clarity*, and *conciseness*. These factors (illustrated in Figure 23-2) all work together to create an effective communicator and collaborator on the team.

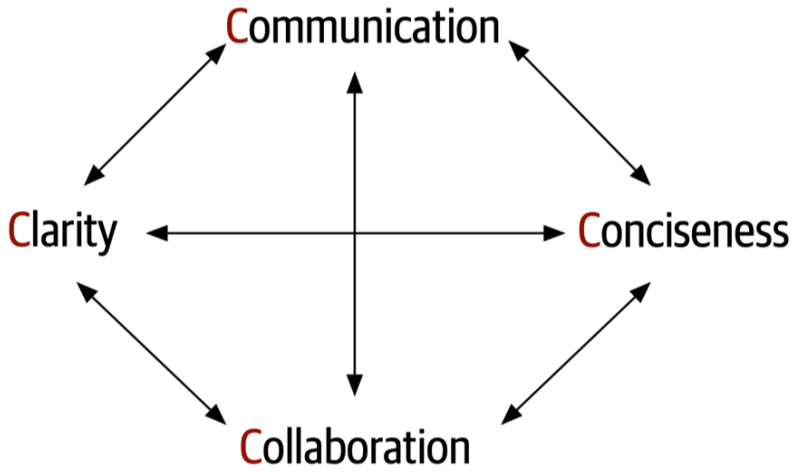

Figure 23-2. The 4 C's of architecture

As a leader, facilitator, and negotiator, is it vital that a software architect be able to effectively communicate in a clear and concise manner. It is equally important that an architect also be able to collaborate with developers, business stakeholders, and other architects to discuss and form solutions together. Focusing on the 4 C's of architecture helps an architect gain the respect of the team and become the go-to person on the project that everyone comes to not only for questions, but also for advice, mentoring, coaching, and leadership.

Be Pragmatic, Yet Visionary

An effective software architect must be pragmatic, yet visionary. Doing this is not as easy as it sounds and takes a fairly high level of maturity and significant practice to accomplish. To better understand this statement, consider the definition of a visionary:

Visionary

> *Thinking about or planning the future with imagination or wisdom.*

Being a visionary means applying strategic thinking to a problem, which is exactly what an architect is supposed to do. Architecture is about planning for the future and making sure the architectural vitality (how valid an

architecture is) remains that way for a long time. However, too many times, architects become too theoretical in their planning and designs, creating solutions that become too difficult to understand or even implement. Now consider the definition of being pragmatic:

Pragmatic

> *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 and realistic solutions. Being pragmatic is taking all of the following factors and constraints into account when creating an architectural solution:

- Budget constraints and other cost-based factors
- Time constraints and other time-based factors
- Skill set and skill level of the development team
- Trade-offs and implications associated with an architecture decision
- Technical limitations of a proposed architectural design or solution

A good software architect is one that strives to find an appropriate balance between being pragmatic while still applying imagination and wisdom to solving problems (see Figure 23-3). For example, consider the situation where an architect is faced with a difficult problem dealing with elasticity (unknown sudden and significant increases in concurrent user load). A visionary might come up with an elaborate way to deal with this through the use of a complex data mesh, which is a collection of distributed, domain-based databases. In theory this might be a good approach, 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 really solve the problem?

A pragmatic architect would first look at what the limiting factor is when needing high levels of elasticity. Is it the database that’s the bottleneck?

Maybe it’s a bottleneck with respect to some of the services invoked or other external sources needed. Finding and isolating the bottleneck would be a first practical approach to the problem. In fact, even if it is the database, could some of the data needed be cached so that the database need not be accessed at all?

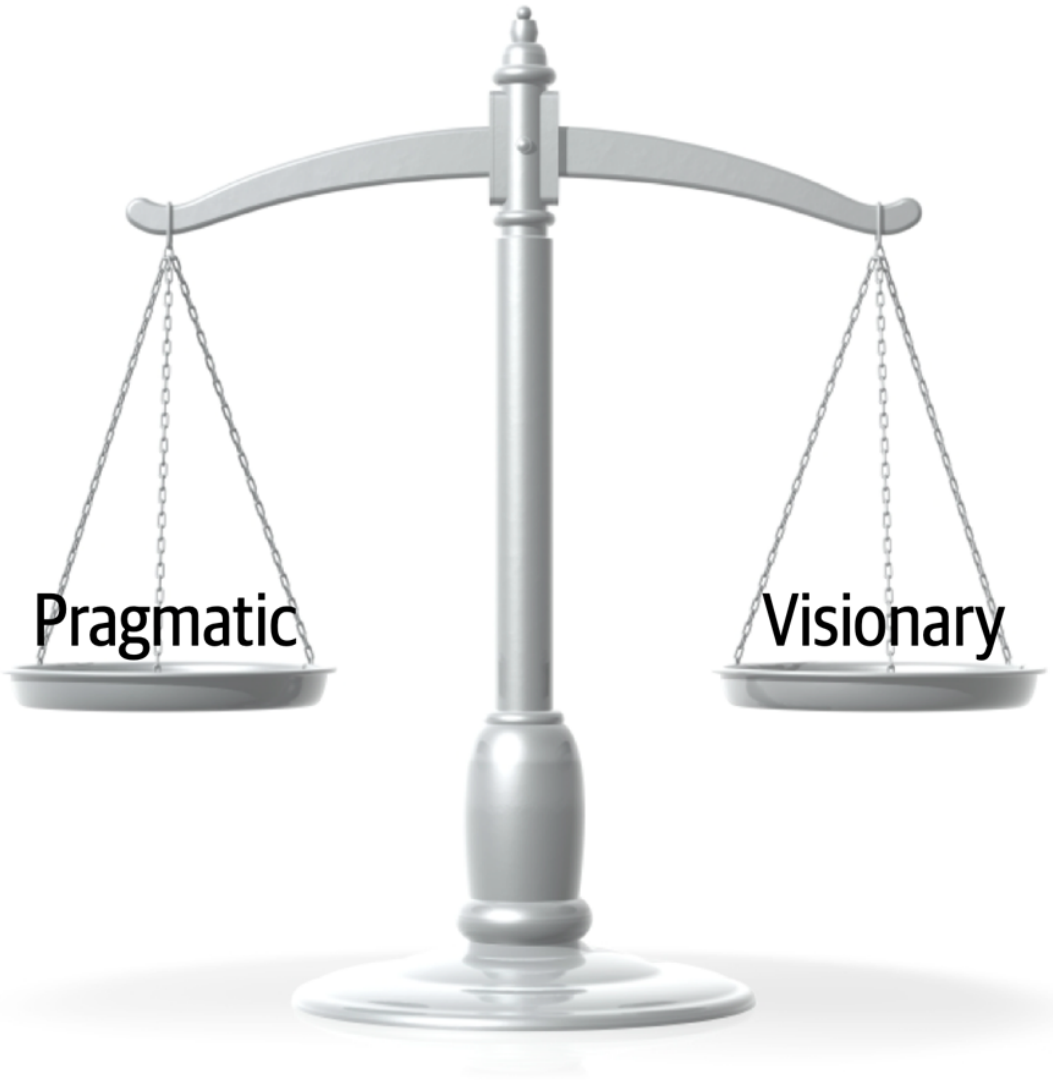

Figure 23-3. Good architects find the balance between being pragmatic, yet visionary

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

Leading Teams by Example

Bad software architects leverage their title to get people to do what they want them to do. Effective software architects get people to do things by not leveraging their title as architect, but rather by leading through example, not by title. 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 and a sergeant during a military battle. The high-ranking captain, who is largely removed from the troops, commands all of the troops to move forward during the battle to take a particularly difficult hill. However, rather than listen to the high-ranking captain, the soldiers, full of doubt, look over to the lower-ranking sergeant for whether they should take the hill or not. The sergeant looks at 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—it has to do with *technical knowledge*. While having technical knowledge is certainly necessary for solving a problem, it's only a part of the overall equation for solving any problem. Suppose, for example, an architect is holding a 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 with, "Well, *that's* a dumb idea." Not only will that developer not make any more suggestions, but none of the other developers will dare say anything. The architect in this case has effectively shut down the entire team from collaborating on the solution.

Gaining respect and leading teams is about basic people skills. Consider the following dialogue between an architect and a customer, client, or development team with regard to a performance issue in the application:

***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."*

By using the words "what you need to do is…" or "you must," the architect is forcing their opinion onto the developer and essentially shutting down collaboration. This is a good example of using communication, not collaboration. Now consider the revised dialogue:

***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…"*

Notice the use of the words "have you considered…" or "what about…" in the dialogue. By asking the question, it puts control back on the developer or client, creating a collaborative conversation where both the architect and developer are working together to form a solution. The use of grammar is vitally important when trying to build a collaborative environment. Being a leader as an architect is not only being able to collaborate with others to create an architecture, but also to help promote collaboration among the team by acting as a facilitator. As an architect, try to observe team dynamics and notice when situations like the first dialogue occurs. By taking team members aside and coaching them on the use of grammar as a means of collaboration, not only will this create better team dynamics, but it will also help create respect among the team members.

Another basic people skills technique that can help build respect and healthy relationships between an architect and the development team is to always try to use the person's name during a conversation or negotiation. Not only do people like hearing their name during a conversation, it also helps breed familiarity. Practice remembering people's names, and use them frequently. Given that names are sometimes hard to pronounce, make sure

to get the pronunciation correct, then practice that pronunciation until it is perfect. Whenever we ask someone's name, we repeat it to the person and ask if that's the correct way to pronounce it. If it's not correct, we repeat this process until we get it right.

If an architect meets someone for the first time or only occasionally, always shake the person's hand and make eye contact. A handshake is an important people skill that goes back to medieval times. The physical bond that occurs during a simple handshake lets both people know they are friends, not foes, and forms a bond between the two people. However, while very basic, it is sometimes hard to get a simple handshake right.

When shaking someone's hand, give a firm (but not overpowering) handshake while looking 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 hold on to the handshake too long. A simple two- to three-second, firm handshake is all that is needed to start off a conversation or to greet someone. There is also the issue of going overboard with the handshake technique and making the other person uncomfortable enough to not want to communicate or collaborate with you. For example, imagine a software architect who comes in every morning and starts shaking everyone's hand. Not only is this a little weird, it creates an uncomfortable situation. However, imagine a software architect who must meet with the head of operations monthly. This is the perfect opportunity to stand up, say "Hello Ruth, nice seeing you again," and give a quick, firm handshake. Knowing when to do a handshake and when not to is part of the complex art of people skills.

A software architect as a leader, facilitator, and negotiator should be careful to preserve the boundaries that exist between people at all levels. The handshake, as described previously, is an effective and professional technique of forming a physical bond with the person you are communicating or collaborating with. However, while a handshake is good, a hug in a professional setting, regardless of the environment, is not. An architect might think that it exemplifies more physical connection and bonding, but all it does is sometimes make the other person at work more

uncomfortable and, more importantly, can lead to potential harassment issues within the workplace. Skip the hugs all together, regardless of the professional environment, and stick with the handshake instead (unless of course everyone in the company hugs each other, which would just be… weird).

Sometimes it's best to turn a request into a favor as a way of getting someone to do something they otherwise might not want to do. In general, people do not like to be told what to do, but for the most part, people want to help others. This is basic human nature. 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, to provide better fault tolerance and scalability in the website. It shouldn't take too long."*
> ***Developer**: "No way, man. Way too busy this iteration for that. Sorry, can't do it."*
> ***Architect**: "Listen, this is important and needs to be done this iteration."*
> ***Developer**: "Sorry, no can do. Maybe one of the other developers can do it. I'm just too busy."*

Notice the developer's response. It is an immediate rejection of the task, even though the architect justified it through better fault tolerance and scalability. In this case, notice that the architect is *telling* the developer to do something they are simply too busy to do. Also notice the 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**: "(Pause)…I'm really busy this iteration, but I guess so. 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, notice the use of the person's name repeatedly throughout the conversation. Using the person's name makes the conversation more of a personal, familiar nature rather than an impersonal professional demand. Second, notice the architect admits they are in a "real bind" and that splitting the services would really "help them out a lot." This technique does not always work, but playing off of basic human nature of helping each other has a better probability of success over the first conversation. Try this technique the next time you face this sort of situation and see the results. In most cases, the results will be much more positive than telling someone what to do.

To lead a team and become an effective leader, a software architect should try to become the go-to person on the team—the person developers go to for their 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. When a software architect observes someone struggling with a technical issue, they should step in and offer help or guidance. The same is true for nontechnical situations as well. Suppose an architect observes a team member that comes into work looking sort of depressed and bothered—clearly something is up. In this circumstance, an effective software architect would notice the situation and offer to talk—something like, "Hey, Antonio, I'm heading over to get some coffee. Why don't we head over together?" and then during the walk ask if everything is OK. This at least provides an opening for more of a personal discussion; and at it's best, a chance to mentor and coach at a more personal level.

However, an effective leader will also recognize times to not be too pushy and will back off by reading various verbal signs and facial expressions.

Another technique to start gaining respect as a leader and become the go-to person on the team is to host periodic brown-bag lunches to talk about a specific technique or technology. Everyone reading this book has a particular skill or knowledge that others don't have. By hosting a periodic brown-bag lunch session, the architect not only is able to exhibit their technical prowess, but also practice speaking skills and mentoring skills. For example, host a lunch session on a review of design patterns or the latest features of the programming language release. Not only does this provide valuable information to developers, but it also starts identifying you as a leader and mentor on the team.

Integrating with the Development Team

An architect's calendar is usually filled with meetings, with most of those meetings overlapping with other meetings, such as the calendar shown in Figure 23-4. If this is what a software architect's calendar looks like, then when does the architect have the time to integrate with the development team, help guide and mentor them, and be available for questions or concerns when they come up? Unfortunately, meetings are a necessary evil within the information technology world. They happen frequently, and will always happen.

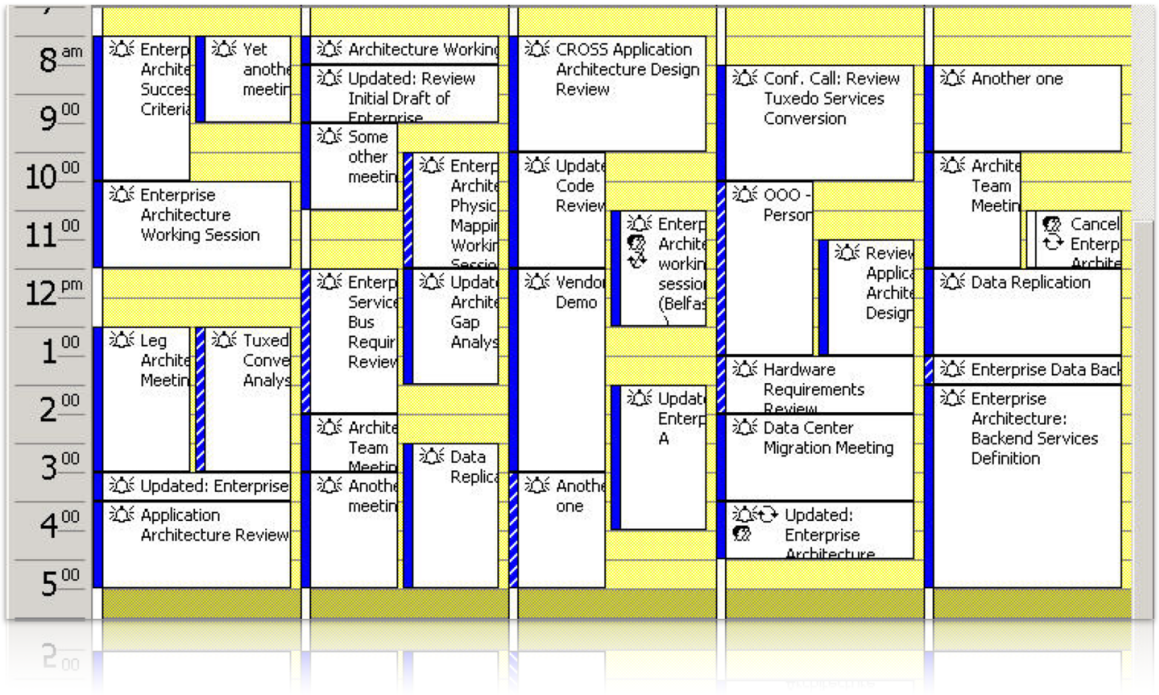

Figure 23-4. A typical calendar of a software architect

The key to being an effective software architect is making more time for the development team, and this means controlling meetings. There are two types of meetings an architect can be involved in: those imposed upon (the architect is invited to a meeting), and those imposed by (the architect is calling the meeting). These meeting types are illustrated in Figure 23-5.

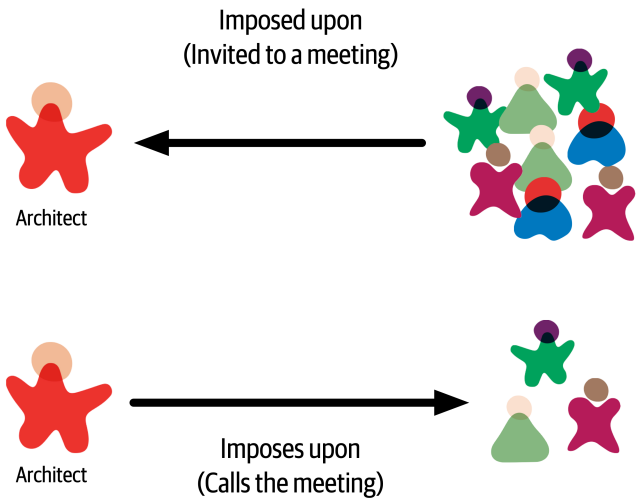

Figure 23-5. Meeting types

Imposed upon meetings are the hardest to control. Due to the number of stakeholders a software architect must communicate and collaborate with, architects are invited to almost every meeting that gets scheduled. When invited to a meeting, an effective software architect should always ask the meeting organizer why they are needed in that meeting. Many times architects get invited to meetings simply to keep them in the loop on the information being discussed. That's what meeting notes are for. By asking why, an architect can start to qualify which meetings they should attend and which ones they can skip. Another related technique to help reduce the number of meetings an architect is involved in is to ask for the meeting agenda before accepting a meeting invite. The meeting organizer may feel that the architect is necessary, but by looking at the agenda, the architect can qualify whether they really need to be in the meeting or not. Also, many times it is not necessary to attend the entire meeting. By reviewing the agenda, an architect can optimize their time by either showing up when relevant information is being discussed or leaving after the relevant discussion is over. Don't waste time in a meeting if you can be spending that time working with the development team.

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

Another effective technique to keep a development team on track and to gain their respect is to take one for the team when developers are invited to a meeting as well. Rather than having the tech lead attend the meeting, go in their place, particularly if both the tech lead and architect are invited to a meeting. This keeps a development team focused on the task at hand rather than continually attending meetings as well. While deflecting meetings away from useful team members increases the time an architect is in meetings, it does increase the development team's productivity.

Meetings that an architect imposes upon others (the architect calls the meeting) are also a necessity at times but should be kept to an absolute minimum. These are the kinds of meetings an architect has control over. An effective software architect will always ask whether the meeting they are calling is more important than the work they are pulling their team members away from. Many times an email is all that is required to communicate some important information, which saves everyone tons of wasted time. When calling a meeting as an architect, always set an agenda and stick to it. Too often, meetings an architect calls get derailed due to some other issue, and that other issue may not be relevant to everyone else in the meeting. Also, as an architect, pay close attention to developer flow and be sure not to disrupt it by calling a meeting. 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, a developer might be working on a particularly difficult algorithm or piece of code, and literally hours go by while it seems only minutes have passed. When calling a meeting, an architect should always try to schedule meetings either first thing in the morning, right after lunch, or toward the end of the day, but not during the day when most developers experience flow state.

Aside from managing meetings, another thing an effective software architect can do to integrate better with the development team is to sit with that team. Sitting in a cubicle away from the team sends the message that the architect is special, and those physical walls surrounding the cubicle are a distinct message that the architect is not to be bothered or disturbed. Sitting alongside a development team sends the message that the architect is an integral part of the team and is available for questions or concerns as they arise. By physically showing that they are part of the development team, the architect gains more respect and is better able to help guide and mentor the team.

Sometimes it is not possible for an architect to sit with a development team. In these cases the best thing an architect can do is continually walk around and be seen. Architects that are stuck on a different floor or always in their

offices or cubicles and never seen cannot possibly help guide the development team through the implementation of the architecture. Block off time in the morning, after lunch, or late in the day and make the time to converse with the development team, help with issues, answer questions, and do basic coaching and mentoring. Development teams 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 the way to get more coffee is an excellent way of keeping communication open and available with business and other key stakeholders.

Summary

The negotiation and leadership tips presented and discussed in this chapter are meant to help the software architect form a better collaborative relationship with the development team and other stakeholders. These are necessary skills an architect must have in order to become an effective software architect. While the tips we presented in this chapter are good tips for starting the journey into becoming more of an effective leader, perhaps the best tip of all is from a quote from Theodore Roosevelt, the 26th US president:

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

Chapter 24. Developing a Career Path

Becoming an architect takes time and effort, but based on the many reasons we've outlined throughout this book, managing a career path after becoming an architect is equally tricky. While we can't chart a specific career path for you, we can point you to some practices that we have seen work well.

An architect must continue to learn throughout their career. The technology world changes at a dizzying pace. One of Neal's former coworkers was a world-renowned expert in Clipper. He lamented that he couldn't take the enormous body of (now useless) Clipper knowledge and replace it with something else. He also speculated (and this is still an open question): has any group in history learned and thrown away so much detailed knowledge within their lifetimes as software developers?

Each architect should keep an eye out for relevant resources, both technology and business, and add them to their personal stockpile. Unfortunately, resources come and go all too quickly, which is why we don't list any in this book. Talking to colleagues or experts about what resources they use to keep current is one good way of seeking out the latest newsfeeds, websites, and groups that are active in a particular area of interest. Architects should also build into their day some time to maintain breadth utilizing those resources.

The 20-Minute Rule

As illustrated in Figure 2-7, technology breadth is more important to architects than depth. However, maintaining breadth takes time and effort, something architects should build into their day. But how in the world does anyone have the time to actually go to various websites to read articles,

watch presentations, and listen to podcasts? The answer is…not many do. Developers and architects alike struggle with the balance of working a regular job, spending time with the family, being available for our children, carving out personal time for interests and hobbies, and trying to develop careers, while at the same time trying to keep up with the latest trends and buzzwords.

One technique we use to maintain this balance is something we call the *20-minute rule*. The idea of this technique, as illustrated in Figure 24-1, is to devote *at least* 20 minutes a day to your career as an architect by learning something new or diving deeper into a specific topic. Figure 24-1 illustrates examples of some of the types of resources to spend 20 minutes a day on, such as InfoQ, DZone Refcardz, and the ThoughtWorks Technology Radar. Spend that minimum of 20 minutes to Google some unfamiliar buzzwords ("the things you don't know you don't know" from Chapter 2) to learn a little about them, promoting that knowledge into the "things you know you don't know." Or maybe spend the 20 minutes going deeper into a particular topic to gain a little more knowledge about it. The point of this technique is to be able to carve out some time for developing a career as an architect and continuously gaining technical breadth.

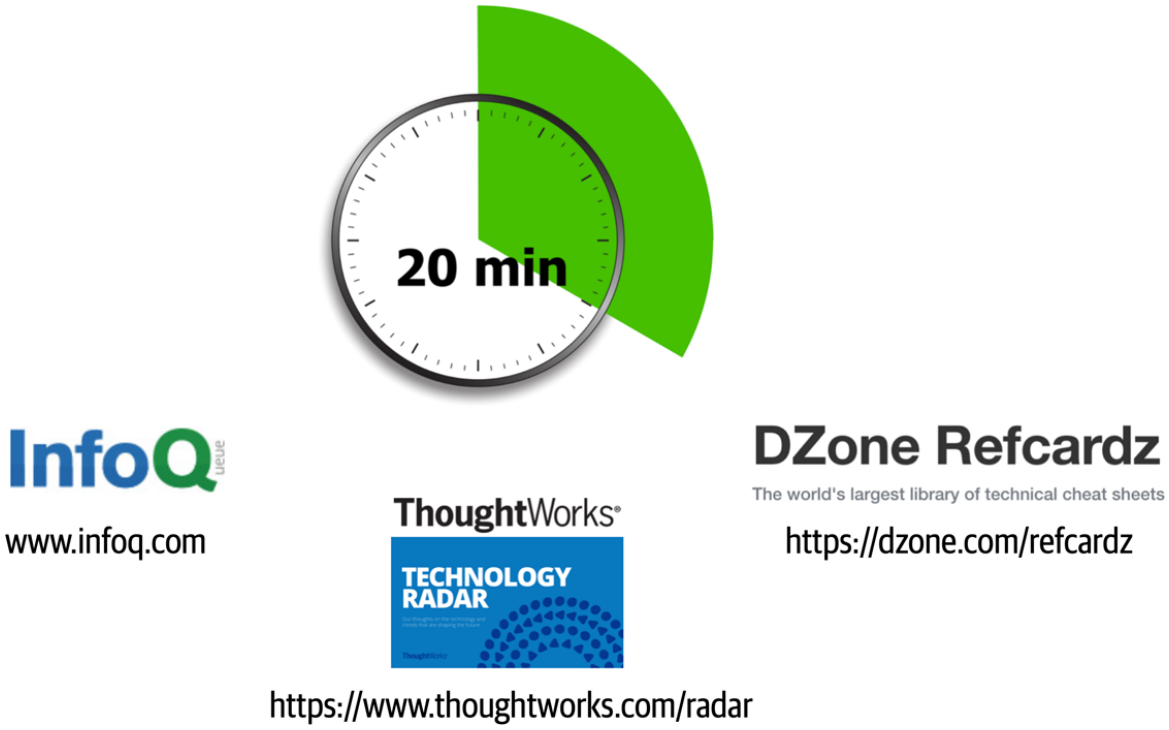

Figure 24-1. The 20-minute rule

Many architects embrace this concept and plan to spend 20 minutes at lunch or in the evening after work to do this. What we have experienced is that this rarely works. Lunchtime gets shorter and shorter, becoming more of a catch-up time at work rather than a time to take a break and eat. Evenings are even worse—situations change, plans get made, family time becomes more important, and the 20-minute rule never happens.

We strongly recommend leveraging the 20-minute rule first thing in the morning, as the day is starting. However, there is a caveat to this advice as well. For example, what is the first thing an architect does after getting to work in the morning? Well, the very first thing the architect does is to get that wonderful cup of coffee or tea. OK, in that case, what is the second thing every architect does after getting that necessary coffee or tea—check email. Once an architect checks email, diversion happens, email responses are written, and the day is over. Therefore, our strong recommendation is to invoke the 20-minute rule first thing in the morning, right after grabbing that cup of coffee or tea and before checking email. Go in to work a little early. Doing this will increase an architect's technical breadth and help develop the knowledge required to become an effective software architect.

Developing a Personal Radar

For most of the ’90s and the beginning of the ’00s, Neal was the CTO of a small training and consulting company. When he started there, the primary platform was Clipper, which was a rapid-application development tool for building DOS applications atop dBASE files. Until one day it vanished. The company had noticed the rise of Windows, but the business market was still DOS…until it abruptly wasn’t. That lesson left a lasting impression: ignore the march of technology at your peril.

It also taught an important lesson about technology bubbles. When heavily invested in a technology, a developer lives in a memetic bubble, which also serves as an echo chamber. Bubbles created by vendors are particularly dangerous, because developers never hear honest appraisals from within the bubble. But the biggest danger of Bubble Living comes when it starts collapsing, which developers never notice from the inside until it’s too late.

What they lacked was a technology radar: a living document to assess the risks and rewards of existing and nascent technologies. The radar concept comes from ThoughtWorks; first, we’ll describe how this concept came to be and then how to use it to create a personal radar.

The ThoughtWorks Technology Radar

The ThoughtWorks Technology Advisory Board (TAB) is a group of senior technology leaders within ThoughtWorks, created to assist the CTO, Dr. Rebecca Parsons, in deciding technology directions and strategies for the company and its clients. This group meets face-to-face twice a year. One of the outcomes of the face to face meeting was the Technology Radar. Over time, it gradually grew into the biannual Technology Radar.

The TAB gradually settled into a twice-a-year rhythm of Radar production. Then, as often happens, unexpected side effects occurred. At some of the conferences Neal spoke at, attendees sought him out and thanked him for helping produce the Radar and said that their company had started producing their own version of it.

Neal also realized that this was the answer to a pervasive question at conference speaker panels everywhere: “How do you (the speakers) keep up with technology? How do you figure out what things to pursue next?” The answer, of course, is that they all have some form of internal radar.

Parts

The ThoughtWorks Radar consists of four quadrants that attempt to cover most of the software development landscape:

Tools

: Tools in the software development space, everything from developers tools like IDEs to enterprise-grade integration tools

Languages and frameworks

: Computer languages, libraries, and frameworks, typically open source

Techniques

: Any practice that assists software development overall; this may include software development processes, engineering practices, and advice

Platforms

: Technology platforms, including databases, cloud vendors, and operating systems

Rings

The Radar has four rings, listed here from outer to inner:

Hold

: The original intent of the hold ring was “hold off for now,” to represent technologies that were too new to reasonably assess yet—technologies that were getting lots of buzz but weren’t yet proven. The hold ring has evolved into indicating “don’t start anything new with this technology.” There’s no harm in using it on existing projects, but developers should think twice about using it for new development.

Assess

The assess ring indicates that a technology is worth exploring with the goal of understanding how it will affect an organization. Architects should invest some effort (such as development spikes, research projects, and conference sessions) to see if it will have an impact on the organization. For example, many large companies visibly went through this phase when formulating a mobile strategy.

Trial

The trial ring is for technologies worth pursuing; it is important to understand how to build up this capability. Now is the time to pilot a low-risk project so that architects and developers can really understand the technology.

Adopt

For technologies in the adopt ring, ThoughtWorks feels strongly that the industry should adopt those items.

An example view of the Radar appears in Figure 24-2.

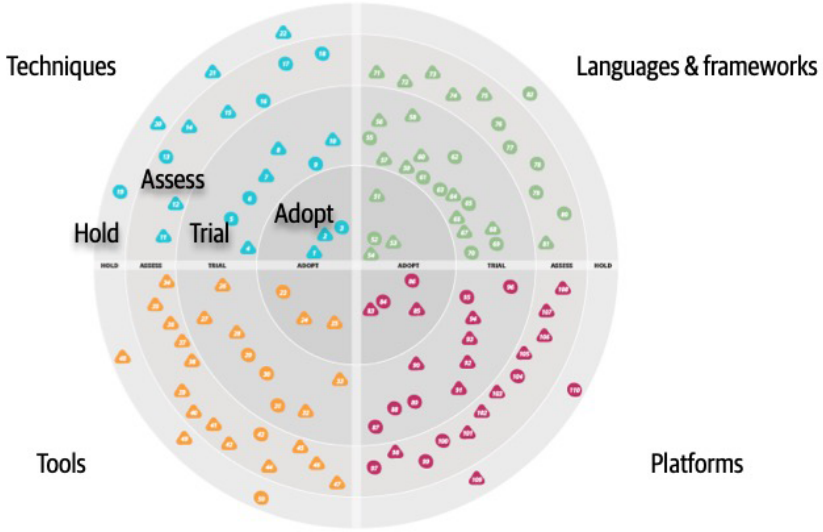

Figure 24-2. A sample ThoughtWorks Technology Radar

In Figure 24-2, each blip represents a different technology or technique, with associated short write-ups.

While ThoughtWorks uses the radar to broadcast their opinions about the software world, many developers and architects also use it as a way of structuring their technology assessment process. Architects can use the tool described in "Open Source Visualization Bits" to build the same visuals used by ThoughtWorks as a way to organize their thinking about what to invest time in.

When using the radar for personal use, we suggest altering the meanings of the quadrants to the following:

Hold

> An architect can include not only technologies and techniques to avoid, but also habits they are trying to break. For example, an architect from the .NET world may be accustomed to reading the latest news/gossip on forums about team internals. While entertaining, it may be a low-value information stream. Placing that in hold forms a reminder for an architect to avoid problematic things.

Assess

> Architects should use *assess* for promising technologies that they have heard good things about but haven't had time to assess for themselves yet—see "Using Social Media". This ring forms a staging area for more serious research at some time in the future.

Trial

> The *trial* ring indicates active research and development, such as an architect performing spike experiments within a larger code base. This ring represents technologies worth spending time on to understand more deeply so that an architect can perform an effective trade-off analysis.

Adopt

The *adopt* ring represents the new things an architect is most excited about and best practices for solving particular problems.

It is dangerous to adopt a laissez-faire attitude toward a technology portfolio. Most technologists pick technologies on a more or less ad hoc basis, based on what's cool or what your employer is driving. Creating a technology radar helps an architect formalize their thinking about technology and balance opposing decision criteria (such as the "more cool" technology factor and being less likely to get a new job versus a huge job market but with less interesting work). Architects should treat their technology portfolio like a financial portfolio: in many ways, they are the same thing. What does a financial planner tell people about their portfolio? Diversify!

Architects should choose some technologies and/or skills that are widely in demand and track that demand. But they might also want to try some technology gambits, like open source or mobile development. Anecdotes abound about developers who freed themselves from cubicle-dwelling servitude by working late at night on open source projects that became popular, purchasable, and eventually, career destinations. This is yet another reason to focus on breadth rather than depth.

Architects should set aside time to broaden their technology portfolio, and building a radar provides a good scaffolding. However, the exercise is more important than the outcome. Creating the visualization provides an excuse to think about these things, and, for busy architects, finding an excuse to carve out time in a busy schedule is the only way this kind of thinking can occur.

Open Source Visualization Bits

By popular demand, ThoughtWorks released a tool in November 2016 to assist technologists in building their own radar visualization. When ThoughtWorks does this exercise for companies, they capture the output of the meeting in a spreadsheet, with a page for each quadrant. The ThoughtWorks Build Your Own Radar tool uses a Google spreadsheet as

input and generates the radar visualization using an HTML 5 canvas. Thus, while the important part of the exercise is the conversations it generates, it also generates useful visualizations.

Using Social Media

Where can an architect find new technologies and techniques to put in the assess ring of their radar? In Andrew McAfee's book *Enterprise 2.0* (Harvard Business Review Press), he makes an interesting observation about social media and social networks in general.

When thinking about a person's network of contact between people, three categories exist, as illustrated in Figure 24-3.

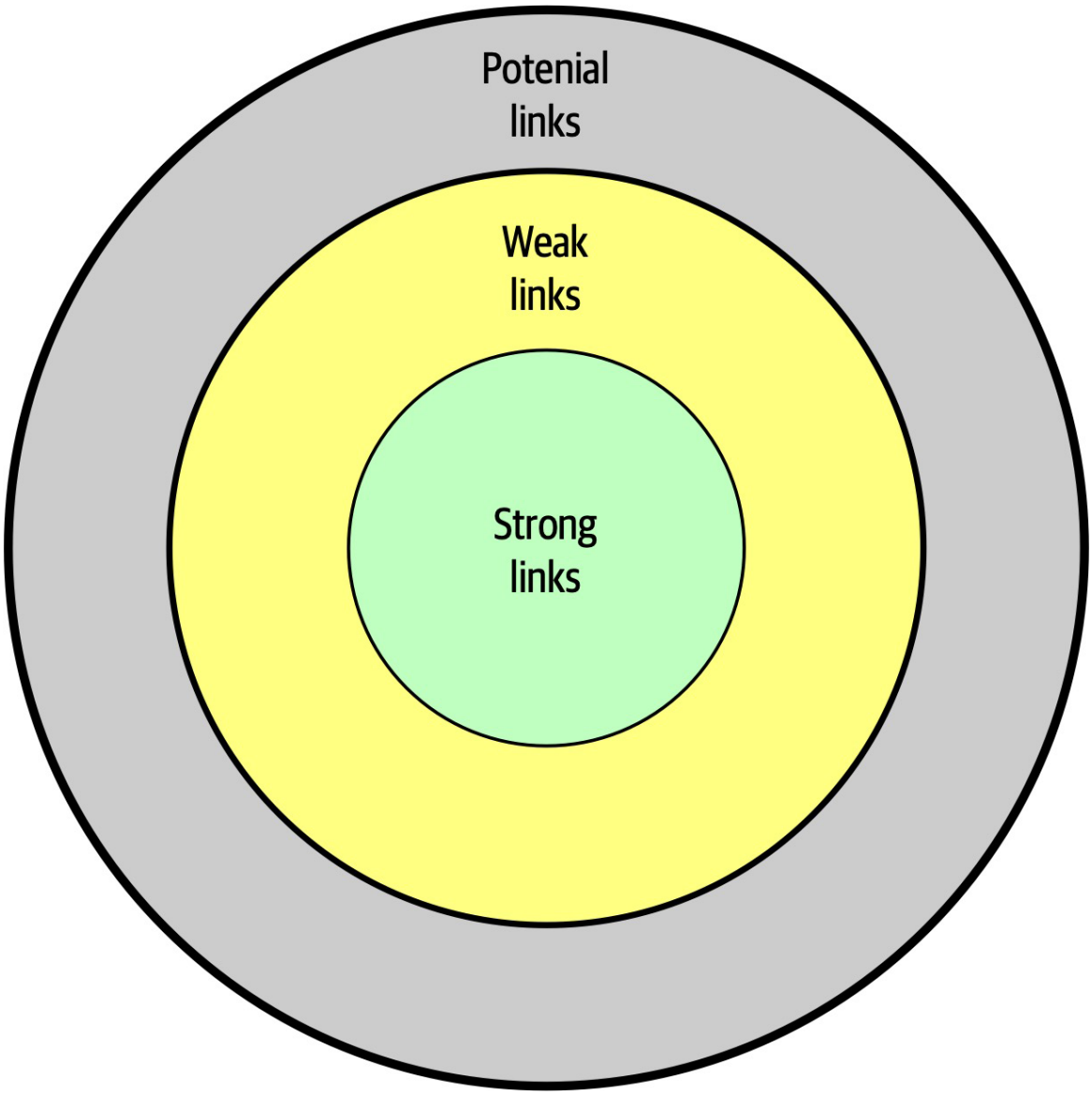

Figure 24-3. Social circles of a person's relationships

In Figure 24-3, *strong links* represent family members, coworkers, and other people whom a person regularly contacts. One litmus test for how close these connections are: they can tell you what a person in their strong links had for lunch at least one day last week. *Weak links* are casual acquaintances, distant relatives, and other people seen only a few times a year. Before social media, it was difficult to keep up with this circle of people. Finally, *potential links* represent people you haven't met yet.

McAfee's most interesting observation about these connections was that someone's next job is more likely to come from a weak link than a strong

one. Strongly linked people know everything within the strongly linked group—these are people who see each other all the time. Weak links, on the other hand, offer advice from outside someone's normal experience, including new job offers.

Using the characteristics of social networks, architects can utilize social media to enhance their technical breadth. Using social media like Twitter professionally, architects should find technologists whose advice they respect and follow them on social media. This allows an architect to build a network on new, interesting technologies to assess and keep up with the rapid changes in the technology world.

Parting Words of Advice

> *How do we get great designers? Great designers design, of course.*
>
> —Fred Brooks

> *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 Neward

Practice is the proven way to build skills and become better at anything in life…including architecture. We encourage new and existing architects to keep honing their skills, both for individual technology breadth but also for the craft of designing architecture.

To that end, check out the architecture katas on the companion website for the book. Modeled after the katas used as examples here, we encourage architects to use these as a way to practice building skills in architecture.

A common question we get about katas: is there an answer guide somewhere? Unfortunately such an answer key does not exist. To quote your author, Neal:

> *There are not right or wrong answers in architecture—only trade-offs.*

When we started using the architecture katas exercise during live training classes, we initially kept the drawings the students produced with the goal of creating an answer repository. We quickly gave up, though, because we realized that we had incomplete artifacts. In other words, the teams had captured the topology and explained their decisions in class but didn't have the time to create architecture decision records. While how they implemented their solutions was interesting, the why was much more interesting because it contains the trade-offs they considered in making that decision. Keeping just the how was only half of the story.

So, our last parting works of advice: always learn, always practice, and *go do some architecture*!

Appendix A. Self-Assessment Questions

Chapter 1: Introduction

1. What are the four dimensions that define software architecture?
2. What is the difference between an architecture decision and a design principle?
3. List the eight core expectations of a software architect.
4. What is the First Law of Software Architecture?

Chapter 2: Architectural Thinking

1. Describe the traditional approach of architecture versus development and explain why that approach no longer works.
2. List the three levels of knowledge in the knowledge triangle and provide an example of each.
3. Why is it more important for an architect to focus on technical breadth rather than technical depth?
4. What are some of the ways of maintaining your technical depth and remaining hands-on as an architect?

Chapter 3: Modularity

1. What is meant by the term *connascence*?
2. What is the difference between static and dynamic connascence?
3. What does *connascence of type* mean? Is it static or dynamic connascence?
4. What is the strongest form of connascence?
5. What is the weakest form of connascence?
6. Which is preferred within a code base—static or dynamic connascence?

Chapter 4: Architecture Characteristics Defined

1. What three criteria must an attribute meet to be considered an architecture characteristic?
2. What is the difference between an implicit characteristic and an explicit one? Provide an example of each.
3. Provide an example of an operational characteristic.
4. Provide an example of a structural characteristic.
5. Provide an example of a cross-cutting characteristic.
6. Which architecture characteristic is more important to strive for—availability or performance?

Chapter 5: Identifying Architecture Characteristics

1. Give a reason why it is a good practice to limit the number of characteristics (“-ilities”) an architecture should support.
2. True or false: most architecture characteristics come from business requirements and user stories.
3. If a business stakeholder states that time-to-market (i.e., getting new features and bug fixes pushed out to users as fast as possible) is the most important business concern, which architecture characteristics would the architecture need to support?
4. What is the difference between scalability and elasticity?
5. You find out that your company is about to undergo several major acquisitions to significantly increase its customer base. Which architectural characteristics should you be worried about?

Chapter 6: Measuring and Governing Architecture Characteristics

1. Why is cyclomatic complexity such an important metric to analyze for architecture?
2. What is an architecture fitness function? How can they be used to analyze an architecture?
3. Provide an example of an architecture fitness function to measure the scalability of an architecture.
4. What is the most important criteria for an architecture characteristic to allow architects and developers to create fitness functions?

Chapter 7: Scope of Architecture Characteristics

1. What is an architectural quantum, and why is it important to architecture?
2. Assume a system consisting of a single user interface with four independently deployed services, each containing its own separate database. Would this system have a single quantum or four quanta? Why?
3. Assume a system with an administration portion managing static reference data (such as the product catalog, and warehouse information) and a customer-facing portion managing the placement of orders. How many quanta should this system be and why? If you envision multiple quanta, could the admin quantum and customer-facing quantum share a database? If so, in which quantum would the database need to reside?

Chapter 8: Component-Based Thinking

1. We define the term *component* as a building block of an application—something the application does. A component usually consist of a group of classes or source files. How are components typically manifested within an application or service?
2. What is the difference between technical partitioning and domain partitioning? Provide an example of each.
3. What is the advantage of domain partitioning?
4. Under what circumstances would technical partitioning be a better choice over domain partitioning?
5. What is the entity trap? Why is it not a good approach for component identification?
6. When might you choose the workflow approach over the Actor/Actions approach when identifying core components?

Chapter 9: Architecture Styles

1. List the eight fallacies of distributed computing.
2. Name three challenges that distributed architectures have that monolithic architectures don’t.
3. What is stamp coupling?
4. What are some ways of addressing stamp coupling?

Chapter 10: Layered Architecture Style

1. What is the difference between an open layer and a closed layer?
2. Describe the layers of isolation concept and what the benefits are of this concept.
3. What is the architecture sinkhole anti-pattern?
4. What are some of the main architecture characteristics that would drive you to use a layered architecture?
5. Why isn’t testability well supported in the layered architecture style?
6. Why isn’t agility well supported in the layered architecture style?

Chapter 11: Pipeline Architecture

1. Can pipes be bidirectional in a pipeline architecture?
2. Name the four types of filters and their purpose.
3. Can a filter send data out through multiple pipes?
4. Is the pipeline architecture style technically partitioned or domain partitioned?
5. In what way does the pipeline architecture support modularity?
6. Provide two examples of the pipeline architecture style.

Chapter 12: Microkernel Architecture

1. What is another name for the microkernel architecture style?
2. Under what situations is it OK for plug-in components to be dependent on other plug-in components?
3. What are some of the tools and frameworks that can be used to manage plug-ins?
4. What would you do if you had a third-party plug-in that didn’t conform to the standard plug-in contract in the core system?
5. Provide two examples of the microkernel architecture style.
6. Is the microkernel architecture style technically partitioned or domain partitioned?
7. Why is the microkernel architecture always a single architecture quantum?
8. What is domain/architecture isomorphism?

Chapter 13: Service-Based Architecture

1. How many services are there in a typical service-based architecture?
2. Do you have to break apart a database in service-based architecture?
3. Under what circumstances might you want to break apart a database?
4. What technique can you use to manage database changes within a service-based architecture?
5. Do domain services require a container (such as Docker) to run?
6. Which architecture characteristics are well supported by the service-based architecture style?
7. Why isn’t elasticity well supported in a service-based architecture?
8. How can you increase the number of architecture quanta in a service-based architecture?

Chapter 14: Event-Driven Architecture Style

1. What are the primary differences between the broker and mediator topologies?
2. For better workflow control, would you use the mediator or broker topology?
3. Does the broker topology usually leverage a publish-and-subscribe model with topics or a point-to-point model with queues?
4. Name two primary advantage of asynchronous communications.
5. Give an example of a typical request within the request-based model.
6. Give an example of a typical request in an event-based model.
7. What is the difference between an initiating event and a processing event in event-driven architecture?
8. What are some of the techniques for preventing data loss when sending and receiving messages from a queue?
9. What are three main driving architecture characteristics for using event-driven architecture?
10. What are some of the architecture characteristics that are not well supported in event-driven architecture?

Chapter 15: Space-Based Architecture

1. Where does space-based architecture get its name from?
2. What is a primary aspect of space-based architecture that differentiates it from other architecture styles?
3. Name the four components that make up the virtualized middleware within a space-based architecture.
4. What is the role of the messaging grid?
5. What is the role of a data writer in space-based architecture?
6. Under what conditions would a service need to access data through the data reader?
7. Does a small cache size increase or decrease the chances for a data collision?
8. What is the difference between a replicated cache and a distributed cache? Which one is typically used in space-based architecture?
9. List three of the most strongly supported architecture characteristics in space-based architecture.
10. Why does testability rate so low for space-based architecture?

Chapter 16: Orchestration-Driven Service-Oriented Architecture

1. What was the main driving force behind service-oriented architecture?
2. What are the four primary service types within a service-oriented architecture?
3. List some of the factors that led to the downfall of service-oriented architecture.
4. Is service-oriented architecture technically partitioned or domain partitioned?
5. How is domain reuse addressed in SOA? How is operational reuse addressed?

Chapter 17: Microservices Architecture

1. Why is the bounded context concept so critical for microservices architecture?
2. What are three ways of determining if you have the right level of granularity in a microservice?
3. What functionality might be contained within a sidecar?
4. What is the difference between orchestration and choreography? Which does microservices support? Is one communication style easier in microservices?
5. What is a saga in microservices?
6. Why are agility, testability, and deployability so well supported in microservices?
7. What are two reasons performance is usually an issue in microservices?
8. Is microservices a domain-partitioned architecture or a technically partitioned one?
9. Describe a topology where a microservices ecosystem might be only a single quantum.
10. How was domain reuse addressed in microservices? How was operational reuse addressed?

Chapter 18: Choosing the Appropriate Architecture Style

1. In what way does the data architecture (structure of the logical and physical data models) influence the choice of architecture style?
2. How does it influence your choice of architecture style to use?
3. Delineate the steps an architect uses to determine style of architecture, data partitioning, and communication styles.
4. What factor leads an architect toward a distributed architecture?

Chapter 19: Architecture Decisions

1. What is the covering your assets anti-pattern?
2. What are some techniques for avoiding the email-driven architecture anti-pattern?
3. What are the five factors Michael Nygard defines for identifying something as architecturally significant?
4. What are the five basic sections of an architecture decision record?
5. In which section of an ADR do you typically add the justification for an architecture decision?
6. Assuming you don’t need a separate Alternatives section, in which section of an ADR would you list the alternatives to your proposed solution?
7. What are three basic criteria in which you would mark the status of an ADR as Proposed?

Chapter 20: Analyzing Architecture Risk

1. What are the two dimensions of the risk assessment matrix?
2. What are some ways to show direction of particular risk within a risk assessment? Can you think of other ways to indicate whether risk is getting better or worse?
3. Why is it necessary for risk storming to be a collaborative exercise?
4. Why is it necessary for the identification activity within risk storming to be an individual activity and not a collaborative one?
5. What would you do if three participants identified risk as high (6) for a particular area of the architecture, but another participant identified it as only medium (3)?
6. What risk rating (1-9) would you assign to unproven or unknown technologies?

Chapter 21: Diagramming and Presenting Architecture

1. What is irrational artifact attachment, and why is it significant with respect to documenting and diagramming architecture?
2. What do the 4 C’s refer to in the C4 modeling technique?
3. When diagramming architecture, what do dotted lines between components mean?
4. What is the bullet-riddled corpse anti-pattern? How can you avoid this anti-pattern when creating presentations?
5. What are the two primary information channels a presenter has when giving a presentation?

Chapter 22: Making Teams Effective

1. What are three types of architecture personalities? What type of boundary does each personality create?
2. What are the five factors that go into determining the level of control you should exhibit on the team?
3. What are three warning signs you can look at to determine if your team is getting too big?
4. List three basic checklists that would be good for a development team.

Chapter 23: Negotiation and Leadership Skills

1. Why is negotiation so important as an architect?
2. Name some negotiation techniques when a business stakeholder insists on five nines of availability, but only three nines are really needed.
3. What can you derive from a business stakeholder telling you “I needed it yesterday”?
4. Why is it important to save a discussion about time and cost for last in a negotiation?
5. What is the divide-and-conquer rule? How can it be applied when negotiating architecture characteristics with a business stakeholder? Provide an example.
6. List the 4 C’s of architecture.
7. Explain why it is important for an architect to be both pragmatic and visionary.
8. What are some techniques for managing and reducing the number of meetings you are invited to?

Chapter 24: Developing a Career Path

1. What is the 20-minute rule, and when is it best to apply it?
2. What are the four rings in the ThoughtWorks technology radar, and what do they mean? How can they be applied to your radar?
3. Describe the difference between depth and breadth of knowledge as it applies to software architects. Which should architects aspire to maximize?

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

About the Authors

Mark Richards is an experienced hands-on software architect involved in the architecture, design, and implementation of microservices and other distributed architectures. He is the founder of *DeveloperToArchitect.com*, a website devoted to assisting developers in the journey from developer to a software architect.

Neal Ford is director, software architect, and meme wrangler at ThoughtWorks, a global IT consultancy with an exclusive focus on end-to-end software development and delivery. Before joining ThoughtWorks, Neal was the chief technology officer at The DSW Group, Ltd., a nationally recognized training and development firm.

Colophon

The animal on the cover of *Fundamentals of Software Engineering* is the red-fan parrot (*Deroptyus accipitrinus*), a native to South America where it is known by several names such as *loro cacique* in Spanish, or *anacã*, *papagaio-de-coleira*, and *vanaquiá* in Portugese. This New World bird makes its home up in the canopies and tree holes of the Amazon rainforest, where it feeds on the fruits of the *Cecropia* tree, aptly known as "snake fingers," as well as the hard fruits of various palm trees.

As the only member of the genus *Deroptyus*, the red-fan parrot is distinguished by the deep red feathers that cover its nape. Its name comes from the fact that those feathers will "fan" out when it feels excited or threatened and reveal the brilliant blue that highlights each tip. The head is topped by a white crown and yellow eyes, with brown cheeks that are streaked in white. The parrot's breast and belly are covered in the same red feathers dipped in blue, in contrast with the layered bright green feathers on its back.

Between December and January, the red-fan parrot will find its lifelong mate and then begin laying 2-4 eggs a year. During the 28 days in which the female is incubating the eggs, the male will provide her with care and support. After about 10 weeks, the young are ready to start fledging in the wild and begin their 40-year life span in the world's largest tropical rainforest.

While the red-fan parrot's current conservation status is designated as of Least Concern, many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *Lydekker's Royal Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.