Chapter 21. Architectural Decisions

One of the core expectations of an architect is to make architectural decisions. Architectural decisions usually involve the structure of the application or system, but they may involve technology decisions as well, particularly when those technology decisions impact architectural characteristics. Whatever the context, a good architectural decision is one that helps guide development teams in making the right technical choices. Making architectural decisions involves gathering enough relevant information, justifying the decision, documenting the decision, and effectively communicating that decision to the right stakeholders.

Architectural Decision Antipatterns

The programmer Andrew Koenig defines an *antipattern* as something that seems like a good idea when you begin, but leads you into trouble. Another definition of an antipattern is a repeatable process that produces negative results. The three most common architectural decision antipatterns that can (and usually do) emerge when an architect makes decisions are the Covering Your Assets antipattern, the Groundhog Day antipattern, and the Email-Driven Architecture antipattern. These three antipatterns usually follow a progressive flow: overcoming the Covering Your Assets antipattern leads to the Groundhog Day antipattern, and overcoming that leads to the Email-Driven Architecture antipattern. Making effective and accurate architectural decisions requires overcoming all three.

The Covering Your Assets Antipattern

The Covering Your Assets antipattern occurs when an architect avoids or defers making an architectural decision out of fear of making the wrong choice. There are two ways to overcome this. The first is to wait until the *last responsible moment* to make an important architectural decision: that is, when there's enough information to justify and validate the decision, but not so long that it holds up development teams or sends the architect into the *Analysis Paralysis* antipattern, where they are stuck forever in analyzing the decision. A good way to determine the last responsible moment is to ask when the cost of deferring the decision exceeds the risk associated with deciding. As illustrated in Figure 21-1, notice that in the early stages of the decision-making time scale, the cost (denoted by the solid line) is low because there's less time spent on making the decision, but the risk (denoted by the dotted line) is high because less is known about the problem or solution. Spending more time deferring the decision increases the cost, but also reduces risk because the architect can make a more complete analysis of the problem and possible alternatives. The time to make a decision is where these two factors intersect and the cost increase exceeds the reduced risk.

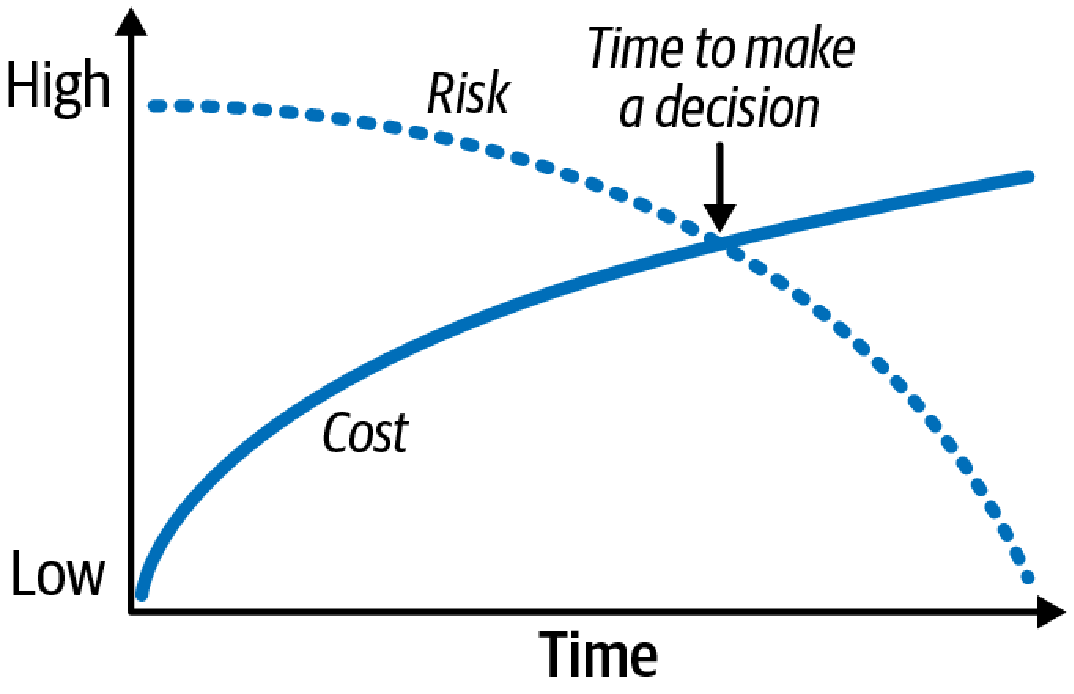

Figure 21-1. Last responsible moment

Another way to avoid this antipattern is to collaborate with development teams to ensure that the decision can be implemented as expected. This is vitally important because no architect can possibly know every single detail about any issue associated with a particular technology. By closely collaborating with development teams, the architect can respond quickly, gain more insight, and reduce the risk of the decision being the incorrect one.

To illustrate this point: suppose you, as the architect, decide that all product-related reference data (such as product description, weight, and dimensions) should be cached in all service instances needing that information. You decide that this will be done using a read-only replicated cache, with the primary cache owned by the `Catalog` service. (A *replicated*, or *in-memory*, cache means that if there are any changes to product information or new products added, the `Catalog` service updates its cache, which is then replicated to all other services requiring that data through a replicated cache product.) Your justification for this decision is to reduce coupling between the services and to share data effectively without having to make an interservice call. However, the development teams implementing this architectural decision find that, due to some services' scalability requirements, this decision would require more in-process memory than is available. Because you are closely collaborating with these teams, you quickly become aware of the issue and adjust your architectural decision accordingly.

Groundhog Day Antipattern

The Groundhog Day antipattern occurs when people don't know why an architect made a particular decision, so they keep discussing it over and over and over, never coming to a final resolution or agreement. It gets its name from the 1993 movie *Groundhog Day*, in which Bill Murray's character must relive February 2 over and over, every day.

This antipattern occurs because architects fail to justify their decision (or fail to justify it completely). It's important to provide both technical and business justifications for an architectural decision.

For example, say you decide to break apart a monolithic application into separate services. Your justification is to decouple the functional aspects of the application so that each part of the application uses fewer virtual machine resources and can be maintained and deployed separately. While this is a good technical justification, what's missing is the *business* justification—in other words, why should the business pay for this architectural refactoring? A good business justification for this decision might be to deliver new business functionality faster, thereby improving time to market. Another might be to reduce the costs associated with developing and releasing new features.

Providing the business value is vitally important when justifying an architectural decision. It is also a good litmus test for determining whether the architectural decision should be made in the first place. If it doesn't provide any business value, perhaps the architect should reconsider the decision.

Four of the most common business justifications are cost, time to market, user satisfaction, and strategic positioning. Consider what's important to the business stakeholders. Justifying a particular decision based on cost savings alone might not be the right call if the business stakeholders are more concerned about time to market.

Email-Driven Architecture Antipattern

Once an architect makes and fully justifies their decisions, another architecture antipattern often emerges: Email-Driven Architecture. This antipattern occurs when people lose or forget an architectural decision, or don't even know that it's been made and therefore cannot possibly implement it. Overcoming this antipattern is all about communicating architectural decisions effectively. Email is a great tool for communication, but it makes a poor document repository system.

Fortunately, an architect can easily avoid the Email-Driven Architecture antipattern by learning how to communicate architectural decisions effectively. First, be sure not to include the decision in the body of an email. Doing so creates multiple systems of record for that decision, because each email contains a copy of the decision rather than having it in only one place. Many such emails leave out important details about the decision (including the justification), creating the Groundhog Day antipattern all over again. Also, if that architectural decision is ever changed or superseded, it's difficult to know whether all the relevant people receive the revised decision.

A better approach is to mention only the nature and context of the decision in the body of the email and provide a link to the single system of record, where the architectural decision and corresponding details are stored (whether that's a link to a wiki page or a reference to a document in a filesystem).

Consider the following email about an architectural decision:

> "Hi, Sandra, I've made an important decision regarding communication between services that directly impacts you. Please see the decision using the following link…."

Notice that, in the phrasing of the beginning of the first sentence, the context is mentioned (communication between services), but not the actual decision itself. The second part of the first sentence is also important: if an architectural decision doesn't directly impact the person, then why bother that person with it? This is a great litmus test for determining which stakeholders (including developers) should be notified directly of an architectural decision. The second sentence in this example provides a link to the single location of the architectural decision, providing a single system of record for decisions.

Architectural Significance

Many architects believe that if a decision involves a specific technology, then it's not an architectural decision, but rather a technical decision. This is not always true. If an architect decides to use a particular technology because it directly supports a particular architecture characteristic (such as performance or scalability), then it's still an architectural decision.

Michael Nygard, a well-known software architect and author of the second edition of *Release It!* (Pragmatic Bookshelf, 2018), addresses the problem of what decisions an architect should be responsible for (and hence what constitutes an architectural decision) by coining the term *architecturally significant*. According to Nygard, architecturally significant decisions are those that affect a system's structure, non-functional characteristics, dependencies, interfaces, or construction techniques.

Structure, in this context, refers to decisions that affect the architecture patterns or styles being used. For example, a decision by an architect to share code between a set of microservices impacts the bounded context of the microservice, and thus affects the structure of the system.

The system's *non-functional characteristics* are the architectural characteristics that are important for the system being developed or maintained. For example, if a choice of technology affects performance, and performance is an important aspect of the application, that choice becomes an architectural decision, even though it may specify a particular product, framework, or technology.

Dependencies refer to the coupling points between components and/or services within the system. Dependencies can affect architecture characteristics such as scalability, modularity, agility, testability, reliability, and so on, so decisions about dependencies become architectural decisions.

Interfaces refer to how services and components are accessed and orchestrated: usually through a gateway, integration hub, service bus, adapter, or API proxy. Making decisions about interfaces usually involves defining contracts, including versioning and deprecation strategies. Interfaces impact others using the system and hence are architecturally significant.

Finally, *construction techniques* refer to decisions about platforms, frameworks, tools, and even processes that, although technical in nature, might impact some aspect of the architecture.

Architectural Decision Records

One of the most effective ways of documenting architectural decisions is through *Architectural Decision Records* (ADRs). Michael Nygard first evangelized ADRs in a 2011 blog post, and in 2017, Thoughtworks Technology Radar recommended the technique for widespread adoption.

An ADR consists of a short text file (usually one to two pages long) describing a specific architectural decision. While ADRs can be written using plain text or in a wiki page template, they are usually written in some sort of text document format, like AsciiDoc or Markdown.

Tooling is also available for managing ADRs. Nat Pryce, coauthor of *Growing Object-Oriented Software, Guided by Tests* (Addison-Wesley, 2009), has written an open source tool called ADR Tools that provides a command-line interface to manage ADRs, including numbering schemes, locations, and superseded logic. Micha Kops, a software engineer from Germany, provides some great examples of using ADR tools to manage architectural decision records.

Basic Structure

The basic structure of an ADR consists of five main sections: *Title*, *Status*, *Context*, *Decision*, and *Consequences*. We usually add two additional sections as part of the basic structure: *Compliance* and *Notes*. The Compliance section is a space to think about and document how the architectural decision will be governed and enforced (manually or through automated fitness functions). The Notes section is a space to include metadata about the decision, such as the author, who approved it, when it was created, and so on.

It's fine to extend this basic structure (as illustrated in Figure 21-2) to include any other needed section. Just keep the template consistent and concise. A good example of this might be to add an *Alternatives* section analyzing all the other possible solutions.

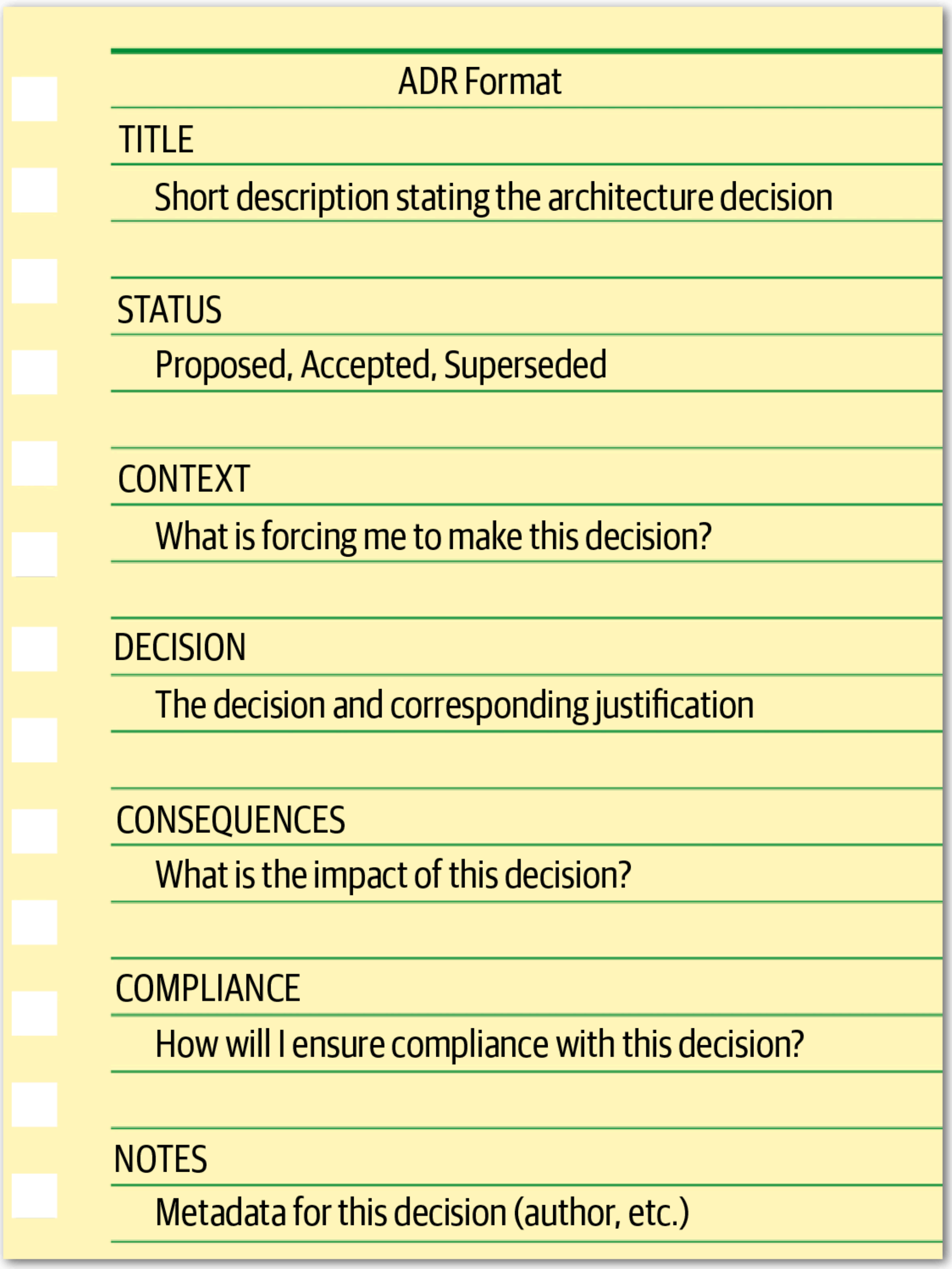
ADR Format
TITLE
Short description stating the architecture decision
STATUS
Proposed, Accepted, Superseded
CONTEXT
What is forcing me to make this decision?
DECISION
The decision and corresponding justification
CONSEQUENCES
What is the impact of this decision?
COMPLIANCE
How will I ensure compliance with this decision?
NOTES
Metadata for this decision (author, etc.)

Figure 21-2. Basic ADR structure

Title

ADR titles are usually numbered sequentially and contain a short phrase describing the architectural decision. For example, an ADR title describing a decision to use asynchronous messaging between the `Order` service and the `Payment` service might read: “42. Use of Asynchronous Messaging Between Order and Payment Services.” The title should be short and concise, but descriptive enough to remove any ambiguity about the nature and context of the decision.

Status

Every ADR has one of three statuses: *Proposed*, *Accepted*, or *Superseded*. The Proposed status means the decision must be approved by a higher-level decision maker or some sort of architectural governance body (such as an architecture review board). Accepted means the decision has been approved and is ready for implementation. Superseded means the decision has been changed and superseded by another ADR. The Superseded status always assumes that the prior ADR's status was Accepted; in other words, a Proposed ADR would never be superseded by another ADR; it would be modified until Accepted.

The Superseded status is a powerful way of keeping historical records of what decisions have been made, why they were made at the time, what the new decision is, and why it was changed. Usually, when an ADR has been superseded, it is marked with the number of the decision that superseded it. Similarly, the decision that supersedes another ADR is marked with the number of the ADR it superseded.

For example, let's say ADR 42 ("Use of Asynchronous Messaging Between Order and Payment Services") is in Approved status. Due to later changes to the implementation and location of the `Payment` service, you decide that REST should now be used between the two services. You therefore create a new ADR (number 68) to document this changed decision. The corresponding statuses look as follows:

> *ADR 42. Use of Asynchronous Messaging Between Order and Payment Services*
>
> Status: Superseded by 68
>
> *ADR 68. Use of REST Between Order and Payment Services*
>
> Status: Accepted, supersedes 42

The link and history trail between ADRs 42 and 68 lets you avoid the inevitable "what about using messaging?" question regarding ADR 68.

ADRs and Request for Comments (RFC)

Sending a draft ADR out or comments can help an architect validate their assumptions and assertions with a larger audience of stakeholders. An effective way of engaging developers and initiating collaboration is creating a new kind of status called *Request for Comments* (RFC) and specifying a deadline for reviewers to complete their feedback. Once that date is reached, the architect can analyze the comments, make any necessary adjustments to the decision, make the final decision, and set the status to Proposed (or Accepted, if the architect has authority to approve the decision).

An ADR with RFC status would look as follows:

STATUS
Request For Comments, Deadline 09 JAN 2026

Another significant aspect of the Status section of an ADR is that it forces the architect and their boss or lead architect to discuss the criteria for approving an architectural decision and determine whether the architect can do so on their own, or whether it must be approved through a higher-level architect, architecture review board, or some other governing body.

Three good starting places for these conversations are cost, cross-team impact, and security. Cost should include software purchase or licensing fees, additional hardware costs, and the overall level of effort to implement the architectural decision. To estimate this, multiply the estimated number of hours to implement the architectural decision by the company's standard *full-time equivalency* (FTE) rate. The project owner or project manager usually has the FTE amount. This conversation might lead everyone to agree that, for example, if the cost of the architectural decision exceeds a certain amount, then it must be set to a Proposed status and approved by someone else. If the architectural decision impacts other teams or systems or has any sort of security implications, then it must be approved by a higher-level governing body or lead architect.

Once the team establishes and agrees on the criteria and corresponding limits (such as "costs exceeding $5,000 must be approved by the architecture review board"), document it well so that all architects creating ADRs will know when they can and cannot approve their own architectural decisions.

Context

The Context section of an ADR specifies the forces at play. In other words, "What situation is forcing me to make this decision?" This section of the ADR allows the architect to describe the specific circumstances and concisely elaborate on the possible alternatives. If the architect is required to document the analysis of each alternative in detail, add an Alternatives section rather than including that analysis in the Context section.

The Context section also provides a place to document a specific area of the architecture itself. By describing the context, the architect is also describing the architecture. Using the example from the prior section, the Context section might read as follows: "The Order service must pass information to the Payment service to pay for an order currently being placed. This could be done using REST or asynchronous messaging." Notice that this concise statement specifies not only the scenario, but also the alternatives considered.

Decision

The Decision section of the ADR contains a description of the architectural decision, along with a full justification. Nygard recommends stating architectural decisions in an affirmative, commanding voice rather than a passive one. For example, the decision to use asynchronous messaging between services would read, "*We will use* asynchronous messaging between services." This is much better than "*I think* asynchronous messaging between services would be the best choice," which does not make clear what the decision is or if a decision has even been made—only the opinion of the architect.

One of the most powerful aspects of the Decision section of ADRs is that it lets the architect emphasize the justification for the decision. Understanding *why* a decision was made is far more important than understanding *how* something works. This helps developers and other stakeholders better understand the rationale behind a decision, and therefore makes them more likely to agree with it.

To illustrate this point, let's say you decide to use Google's Remote Procedure Call (gRPC) to communicate between two particular services to reduce network latency due to very high responsiveness needs. Several years later, a new architect on the team decides to use REST instead of gRPC to make communications between services more consistent. Because the new architect doesn't understand *why* gRPC was chosen in the first place, their decision ends up having a significant impact on latency, causing timeouts in upstream systems. If the new architect had access to an ADR, they would have understood that the original decision to use gRPC was meant to reduce latency (at the cost of tightly coupled services) and could have prevented this problem.

Consequences

Every decision an architect makes has some sort of impact, good or bad. The Consequences section of an ADR forces an architect to describe the overall impact of an architectural decision, allowing them to think about whether the negative impacts outweigh the benefits.

This section is also a good place to document the trade-off analysis performed during the decision-making process. For example, let's say you decide to use asynchronous (fire-and-forget) messaging for posting reviews on a website. Your justification for this decision is to improve responsiveness (from 3,100 milliseconds down to 25 milliseconds) because users wouldn't have to wait for the actual review to be posted—only for the message to be sent to a queue. One member of your development team argues that this is a bad idea due to the complexity of the error handling associated with an asynchronous request: "What happens if someone posts a review with some bad words?" What this team member doesn't know is that you discussed that very problem with the business stakeholders and other architects when analyzing the trade-offs of this decision, and decided together that it was better to improve responsiveness and deal with complex error handling rather than increase the wait time and provide feedback as to whether the review post was successful or not. If this decision was documented using an ADR, you could have provided this trade-off analysis in the Consequences section, preventing this sort of disagreement.

Compliance

The Compliance section is not one of the standard sections of an ADR, but it's one we highly recommend adding. The Compliance section states how the architectural decision will be measured and governed. Will the compliance check for this decision be manual, or can it be automated using a fitness function? If it can be automated, the architect can specify how the fitness function will be written, along with any other changes to the code base needed to measure this architectural decision for compliance.

For example, suppose you make the decision within a traditional n-tiered layered architecture (as illustrated in Figure 21-3) that all shared objects used by business objects in the Business layer must reside in the Shared Services layer to isolate and contain shared functionality.

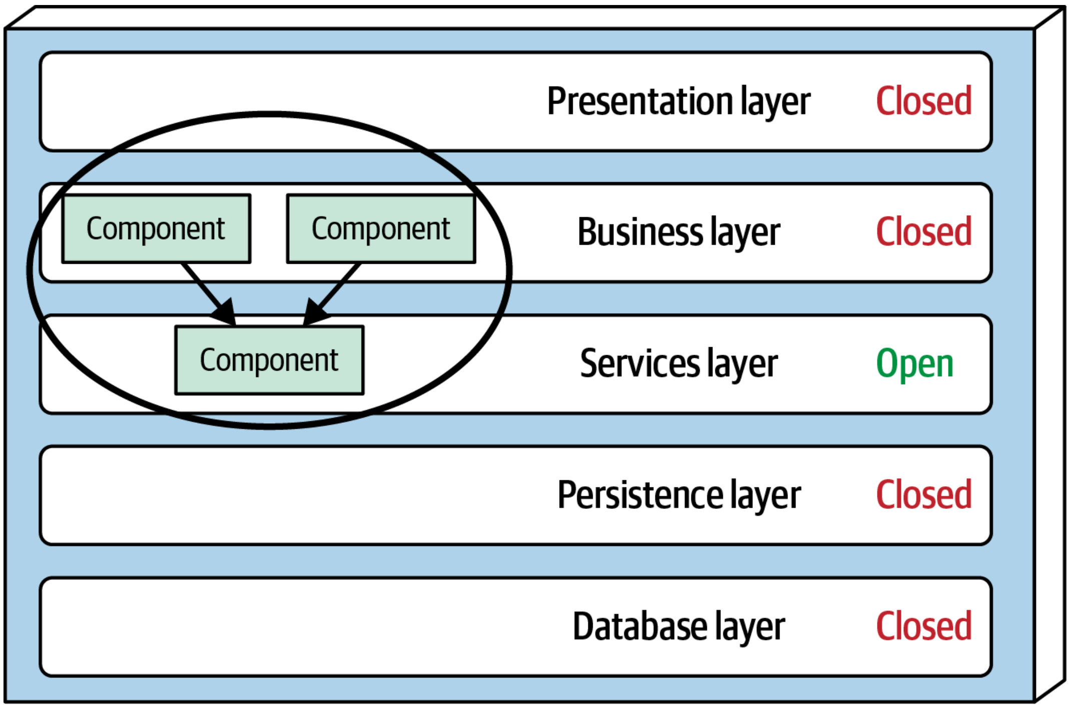

Figure 21-3. An example of an architectural decision

This architectural decision can be measured and governed using a host of automation tools, including ArchUnit in Java and NetArchTest in C#. With ArchUnit in Java, the automated fitness-function test for this architectural decision might look like this:

```
@Test
public void shared_services_should_reside_in_services_layer() {
    classes().that().areAnnotatedWith(SharedService.class)
        .should().resideInAPackage("..services..")
        .check(myClasses);
}
```

This automated fitness function would require writing new stories to create a Java annotation (`@SharedService`) and add it to all shared classes to support this method of governance.

Notes

Another section that is not part of a standard ADR but that we highly recommend adding is a Notes section. This section includes various metadata about the ADR:

- Original author
- Approval date
- Approved by
- Superseded date
- Last modified date
- Modified by
- Last modification

Even when storing ADRs in a version-control system (such as Git), it’s useful to have additional metadata beyond what the repository can support. We recommend adding this section regardless of how and where the architect stores ADRs.

Example

Our example of the Going, Going, Gone (GGG) auction system includes dozens of architectural decisions. Splitting up the bidder and auctioneer user interfaces, using a hybrid architecture consisting of event-driven and microservices, leveraging the Real-time Transport Protocol (RTP) for video capture, using a single API Gateway, and using separate queues for messaging are just a few of the architectural decisions an architect would make. Every architectural decision an architect makes, no matter how obvious, should be documented and justified.

Figure 21-4 illustrates one of the architectural decisions within the GGG auction system: using separate point-to-point queues between the bid capture, bid streamer, and bid tracker services rather than a single publish-and-subscribe topic (or even REST, for that matter).

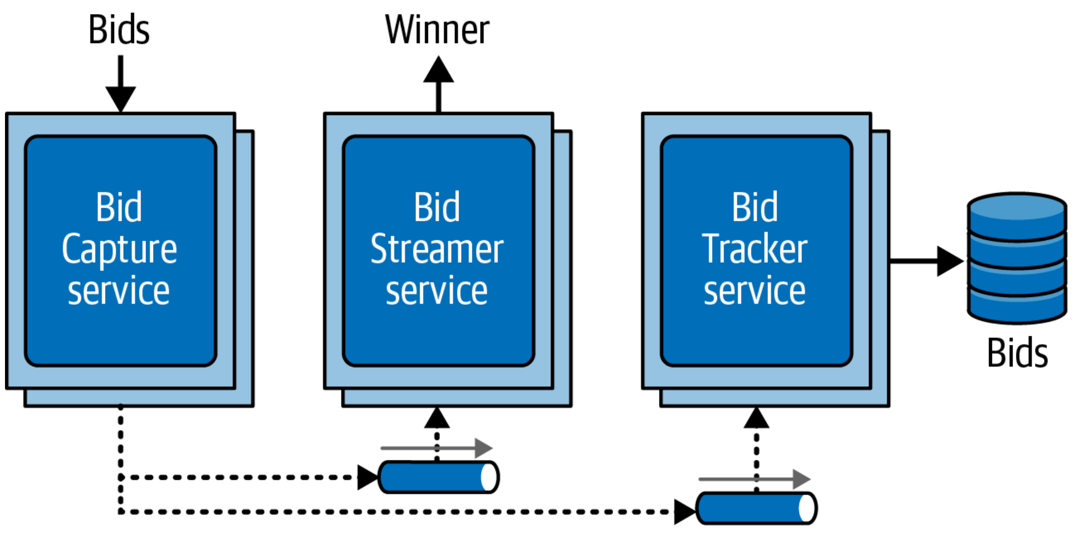

Figure 21-4. Use of pub/sub between services

Without an ADR to justify this decision, others involved with designing and developing this system might disagree and decide to implement it in a different way.

The following is an example of an ADR for this architectural decision:

ADR 76. Separate Queues for Bid Streamer and Bidder Tracker Services

STATUS
Accepted

CONTEXT
The `Bid Capture` service, upon receiving a bid, must forward that bid to the `Bid Streamer` service and the `Bidder Tracker` service. This could be done using a single topic (pub/sub), separate queues (point-to-point) for each service, or REST via the Online Auction API layer.

DECISION
We will use separate queues for the `Bid Streamer` and `Bidder Tracker` services.

The `Bid Capture` service does not need any information from the `Bid Streamer` service or `Bidder Tracker` service (communication is only one-way).

The `Bid Streamer` service must receive bids in the exact order they were accepted by the `Bid Capture` service. Using messaging and queues automatically guarantees the bid order for the stream by leveraging first-in, first out (FIFO) queues.

Multiple bids come in for the same amount (for example, "Do I hear a hundred?"). The `Bid Streamer` service only needs the first bid received for that amount, whereas the `Bidder Tracker` needs all bids received. Using a topic (pub/sub) would require the `Bid Streamer` to ignore bids that are the same as the prior amount, forcing the `Bid Streamer` to store shared state between instances.

The `Bid Streamer` service stores the bids for an item in an in-memory cache, whereas the `Bidder Tracker` stores bids in a database. The `Bidder Tracker` will therefore be slower and might require backpressure. Using a dedicated `Bidder Tracker` queue provides this dedicated backpressure point.

CONSEQUENCES
We will require clustering and high availability of the message queues.

This decision will require the `Bid Capture` service to send the same information to multiple queues.

Internal bid events will bypass security checks done in the API layer. *UPDATE: Upon review at the January 14, 2025, ARB meeting, the ARB decided that this was an acceptable trade-off and that no additional security checks are needed for bid events between these services.*

COMPLIANCE
We will use periodic manual code reviews to ensure that asynchronous pub/sub messaging is being used between the `Bid Capture` service, `Bid Streamer` service, and `Bidder Tracker` service.

NOTES
Author: Subashini Nadella

Approved: ARB Meeting Members, 14 JAN 2025

Last Updated: 14 JAN 2025

Storing ADRs

Once an architect creates an ADR, they need to store it somewhere. Regardless of where that is, each architectural decision should have its own file or wiki page. Some architects like to keep ADRs in the same Git repository as the source code, allowing the team to version and track ADRs as they would source code.

However, for larger organizations, we caution against this practice for several reasons. First, everyone who needs to see the architectural decision might not have access to the Git repository containing the ADRs. Second, the application's Git repository is not a good place to store ADRs that have a context outside of them (such as integration architectural decisions, enterprise architectural decisions, or decisions that are common to every application). For these reasons, we recommend storing ADRs in a dedicated ADR Git repository to which everyone has access, a wiki (using a wiki template), or a shared directory on a shared file server that can be accessed easily by a wiki or other document-rendering software.

Figure 21-5 shows what this directory structure (or wiki page navigation structure) might look like.

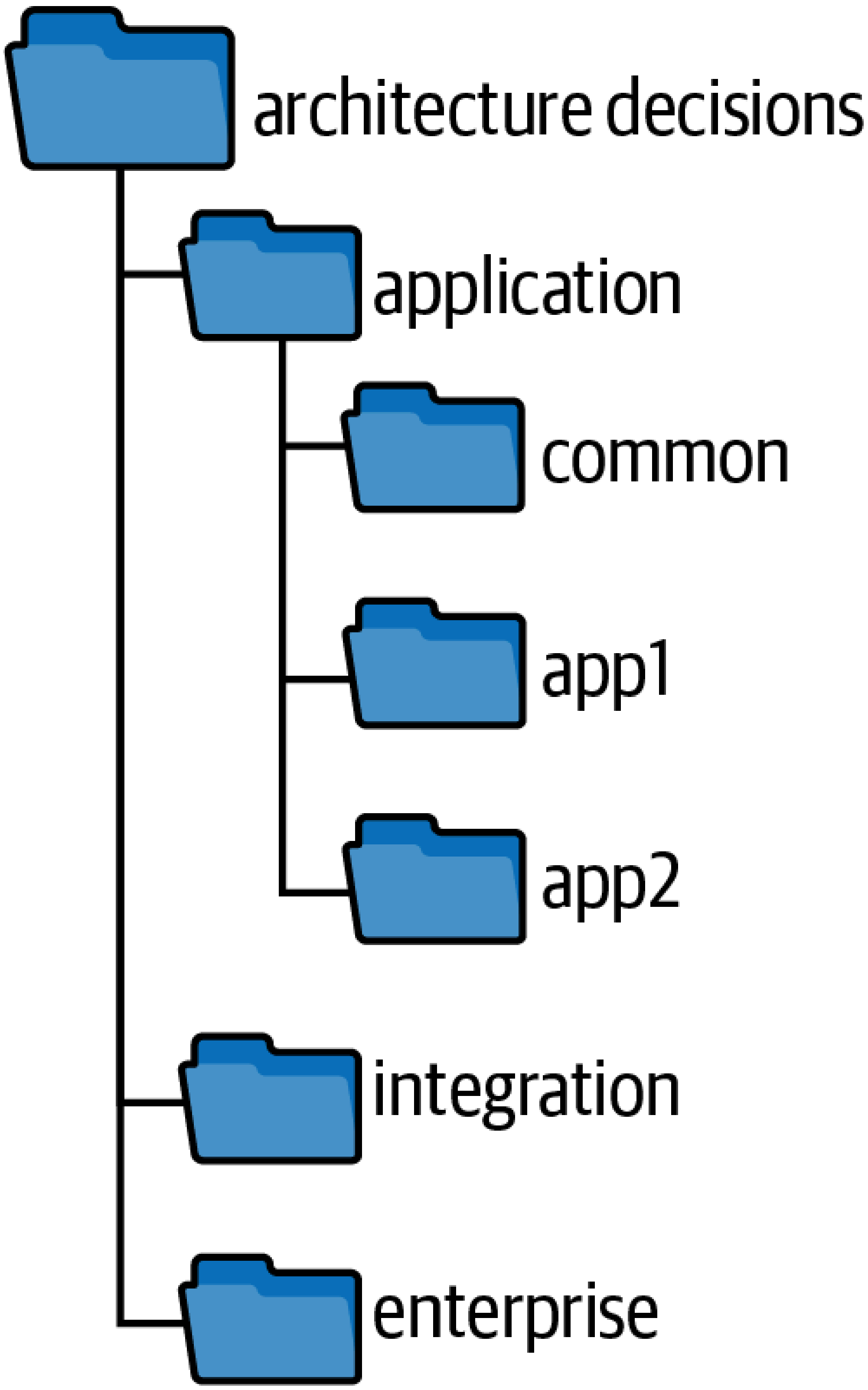

Figure 21-5. Example directory structure for storing ADRs

The *application* directory contains architectural decisions that are specific to some sort of application (or product) context. This directory is subdivided into further directories:

common

> The *common* subdirectory is for architectural decisions that apply to all applications, such as, “All framework-related classes will contain an annotation (@Framework in Java) or attribute ([Framework] in C#) identifying the class as belonging to the underlying framework code.”

application

> Subdirectories under the *application* directory correspond to the specific application or system context and contain architectural decisions specific to that application or system (in this example, the *app1* and *app2* applications).

integration

> The *integration* directory contains ADRs that involve communication between application, systems, or services.

enterprise

> Enterprise architecture ADRs are contained within the *enterprise* directory, indicating that these are global architectural decisions impacting all systems and applications. An example of an enterprise architecture ADR would be "All access to a system database will only be from the owning system," preventing databases from being shared across multiple systems.

When storing ADRs in a wiki, the same structure applies, with each directory structure representing a navigational landing page. Each ADR is represented as a single wiki page within each navigational landing page (application, integration, or enterprise).

The directory and landing-page names indicated in this section are only recommendations and examples. Choose whatever names fit the company's situation, as long as those names are consistent across teams.

ADRs as Documentation

Documenting software architecture has always been difficult. While some standards are emerging for diagramming architecture (such as software architect Simon Brown's C4 Model or The Open Group ArchiMate standard), no agreed-upon standard exists for documenting software architecture. That's where ADRs come in.

ADRs can be an effective means to document a software architecture. The Context section provides an excellent opportunity to describe the specific area of the system that requires an architectural decision to be made, as well as to describe the alternatives. More importantly, the Decision section describes the reasons why a particular decision is being made, which is by far the best form of architectural documentation. The Consequences section adds the final piece of the puzzle by describing the trade-off analysis for the decision—for example, the reasons (and trade-offs) for choosing performance over scalability.

Using ADRs for Standards

Very few developers like standards. Unfortunately, standards are sometimes more about control rather than providing a useful purpose. Using ADRs for standards can change this bad practice. For example, the Context section of an ADR describes the situation that is forcing the organization to adopt the particular standard. The Decision section of an ADR can thus indicate not only what the standard is, but more importantly, why it needs to exist.

This is a great way to qualify whether a particular standard should even exist in the first place. If an architect can't justify it, then perhaps it isn't a good standard to set and enforce. Furthermore, the more developers understand *why* a particular standard exists, the more likely they are to follow it (and, correspondingly, not challenge it). The Consequences section of an ADR is another great place to qualify whether a standard is valid: it requires the architect to think about and document the standard's implications and consequences, and whether the particular standard should be implemented or not.

Using ADRs with Existing Systems

Many architects question the usefulness of ADRs for existing systems. After all, architectural decisions have already been made and the system is in production. Do ADRs serve any real purpose at this point? In fact, they do. Remember, ADRs are more than just documentation—they help architects and developers understand *why* a decision was made and if it was the most appropriate one.

Start by writing some ADRs for the more significant architectural decisions made and questioning whether those decisions are the right ones or not. For example, maybe a group of services are sharing a single database. Why? Is there a good reason? Should the data be broken apart?

Part of the journey of incorporating ADRs into an existing system is doing a little investigative work to uncover these *why* questions. Unfortunately, the person who made the original decision might have left the company a long time ago, so no one knows the answer. In these cases it's up to the architect to identify and analyze the alternatives and the trade-offs of each option and attempt to validate (or invalidate) the existing decision. In either case, writing ADRs for these sorts of significant decisions starts to build up the justifications and rationales (and brain trust) for the system and can help identify architectural inefficiencies and incorrect system design.

Leveraging Generative AI and LLMs in Architectural Decisions

One of the many intriguing aspects of generative AI is whether architects can use it to help make and validate decisions. Should services use messaging, streaming, or event sourcing when sending downstream data? Should the database remain as a single monolith, or should it be broken down into domain databases? Should `Payment Processing` be deployed as a single service or broken apart into multiple services, one for each payment type?

Most architects already know the answer to these questions—it depends! Coming back to our First Law of Software Architecture, *everything in software architecture is a trade-off*. Decisions like these depend on a lot of factors, including the specific context in which the decision is applied. Every situation and every environment is different, which is why there are no "best practices" for these sorts of structural questions.

Most LLMs base their results largely on probability. In other words, what is the most probable answer given the context of the prompt, and what is the "best practice" for this problem? However, probability and "best practices" have no place in making architectural decisions. Answering architectural questions requires a careful analysis of the trade-offs involved, and applying a specific business and technical context to identify the most appropriate choice. For example, if the business is most concerned about time to market (getting changes and new features out to customers as fast as possible), then *maintainability* is going to be far more important than *performance*, and will drive most of the decision-making process toward optimizing maintainability.

Architectural decisions require translating business concerns (such as time to market or sustained growth) into architectural characteristics (such as maintainability, testability, deployability, and so on). This translation is not always obvious, and getting it right requires years of experience. Once completed, it serves as a basis for trade-off analysis. For example, deciding whether to have a single service for payment processing or a service per payment type boils down to a trade-off between maintainability and performance: a single service provides better performance, but multiple services provide better maintainability. If the business is primarily concerned about time to market, maintainability is far more important than performance, so separate services would be the appropriate choice for this specific context.

Because of the very specific and individualistic nature of analyzing trade-offs and business context, it is difficult for generative AI as it currently exists to arrive at the most appropriate architectural decision. The best-case scenario, based on recent experiments your authors have done, is to have a generative AI tool outline the possible trade-offs of a decision, to assist in identifying any missed trade-offs. While generative AI tools have plenty of *knowledge*, they lack the *wisdom* required to make the most appropriate architectural decision.