Chapter 26. Architectural Intersections

So far in this book, we've shown how to identify the critical characteristics an architecture must support, how to select the most appropriate architectural style for those characteristics and for the business problem, how to make effective architecture decisions, and how to lead and guide development teams through implementing the architecture. However, for an architecture to work, it must also be aligned with other facets of the technical and business environment. We call these alignments the *intersections of architecture*.

In this chapter, we discuss several important intersections that arise when creating or validating a software architecture:

Implementation

> Is the implementation aligned with architectural concerns surrounding operational characteristics, architectural constraints, and the internal structure of the architecture?

Infrastructure

> Do the infrastructure and the way the architecture is deployed align with the operational concerns of the architecture, such as scalability, responsiveness, fault tolerance, and availability?

Data topologies

> One widely ignored alignment is that between the intersection of architecture and data topologies and data type. The data topology (monolithic, domain databases, and database per service) must properly align with the architectural style for the system to work.

Engineering practices

> Does the way the development team creates, maintains, and tests software match the corresponding architecture? Does the deployment pipeline match the architectural style?

Team topologies

> The way teams are organized can significantly impact the architecture, and vice versa. If the team structure is not properly aligned with the architecture, development teams will usually struggle, finding even the simplest of changes challenging.

Systems integration

> What other systems or services does the architecture need to communicate with? Not paying attention to this particular intersection can have devastating results in terms of maintenance, reliability, and operational characteristics such as scalability, responsiveness, and availability.

The enterprise

> Is the architecture aligned with the frameworks, practices, guiding principles, and standards across the organization and the enterprise?

The business environment

> Is the architecture properly aligned with the business environment and the problem domain? Too often architects ignore this important intersection, and as a result the architecture fails to meet the goals or needs of the business.

Generative AI

> How does the increasing use of large language models (LLMs) impact the architecture? This intersection is quickly becoming an important one as more companies leverage generative AI within their systems.

The following sections describe these intersections in more detail.

Architecture and Implementation

The First Law of Software Architecture also happens to be software architects' most common response to any question: "It depends." Perhaps the second most common response is, "That's an implementation detail." When a software architecture fails to achieve its goals, this second response is often to blame.

For an architecture to work properly, its *implementation*—that is, its source code—must be aligned with its design to address three things: the architecture's operational concerns (such as fault tolerance, responsiveness, scalability, and so on), internal structure, and constraints. This section addresses each of the three in turn.

Operational Concerns

Operational concerns are the architectural characteristics we focused on in Part I of the book, which form the foundation of any software architecture, and which the architecture must support in order to solve the business problem at hand. Architectural characteristics are what drive architectural decisions (as discussed in Chapter 21).

So what does it mean for the architecture and the implementation to fall out of alignment in how they address the system's operational concerns? Let's say you're an architect working on a new order-entry system that needs to support anywhere from several thousand to half a million concurrent customers. You choose a microservices architecture style, based on the star ratings found in "Style Characteristics" in Chapter 18, which is appropriate based on this system's high scalability and elasticity needs. However, during *implementation*, the development team observes that, because the services' bounded contexts are so tightly formed (see "Bounded Context" in Chapter 18), the `Order Placement` service cannot access the inventory database directly. Instead, it must synchronously call the `Inventory` service to get the current inventory for any item a customer is interested in buying. Not only does this synchronous call tightly couple the two services, it greatly slows the system's responsiveness.

The development team decides to use an *in-memory replicated cache* between these services, as shown in Figure 26-1: data resides in the *internal memory* of

each service instance, and is always kept in sync behind the scenes through caching. Caching products for this purpose include Apache Ignite and Hazelcast. Here, the `Inventory` service would have a writable in-memory cache containing the item IDs and current inventory counts; each instance of the `Order Placement` service would have a replicated read-only version of that cache in its internal memory. Using an in-memory replicated cache decouples the services and *significantly* improves responsiveness.

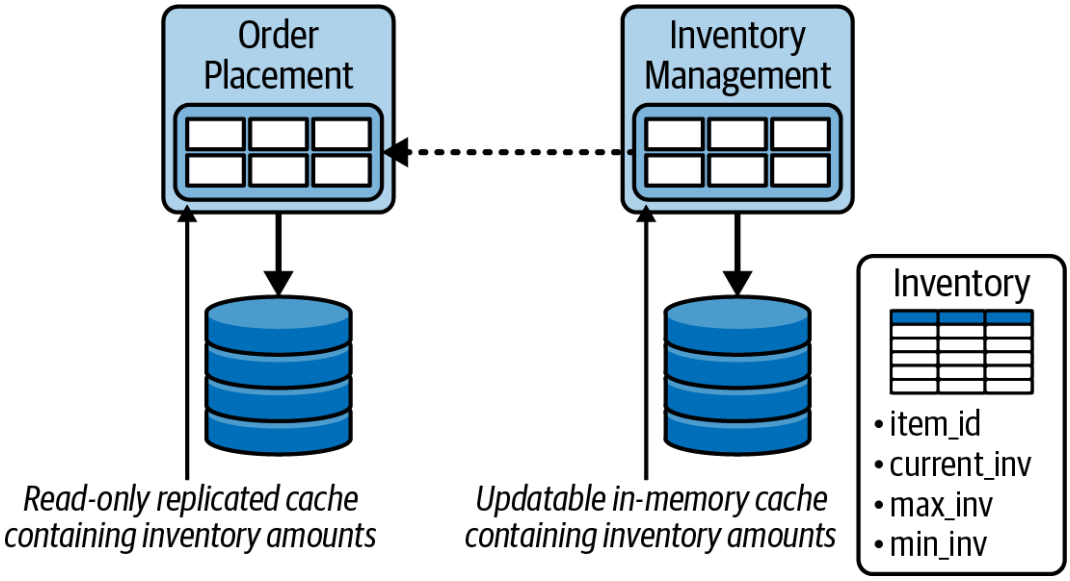

Figure 26-1. The development team decided to use in-memory replicated caching between services, resulting in out-of-memory conditions when scaling the services

After production release, as the number of concurrent users grows, more instances of each service are needed to handle the load. The system crashes when the load reaches about 80,000 concurrent customers because the internal cache's memory requirements are too high, causing out-of-memory conditions in all of the virtual machines.

In this scenario, the architecture and its implementation are misaligned. While the architecture focuses on supporting high levels of *scalability* and *elasticity*, the implementation focuses on *responsiveness* and *service decoupling*. Both teams made good decisions, but in service of different goals.

Structural Integrity

You learned in Chapter 8 that logical components are the building blocks of any system and form its *logical architecture*. They are usually represented through the directory structures in the source-code repository (or namespaces, depending on the programming language). Since the logical architecture describes how the system works and what parts of the system interact with other parts, it's critical that the structure of the source code matches that of the logical architecture.

Without proper guidance, knowledge, and governance, it's easy for developers to ignore the system's logical architecture and start creating directory structures and namespaces as they please, without regard for their impact on the system's integrity. This misalignment results in architectures that are difficult to maintain, test, and deploy, and as a result become less reliable and harder to evolve or adapt to new features, such as the logical architecture illustrated in Figure 26-2.

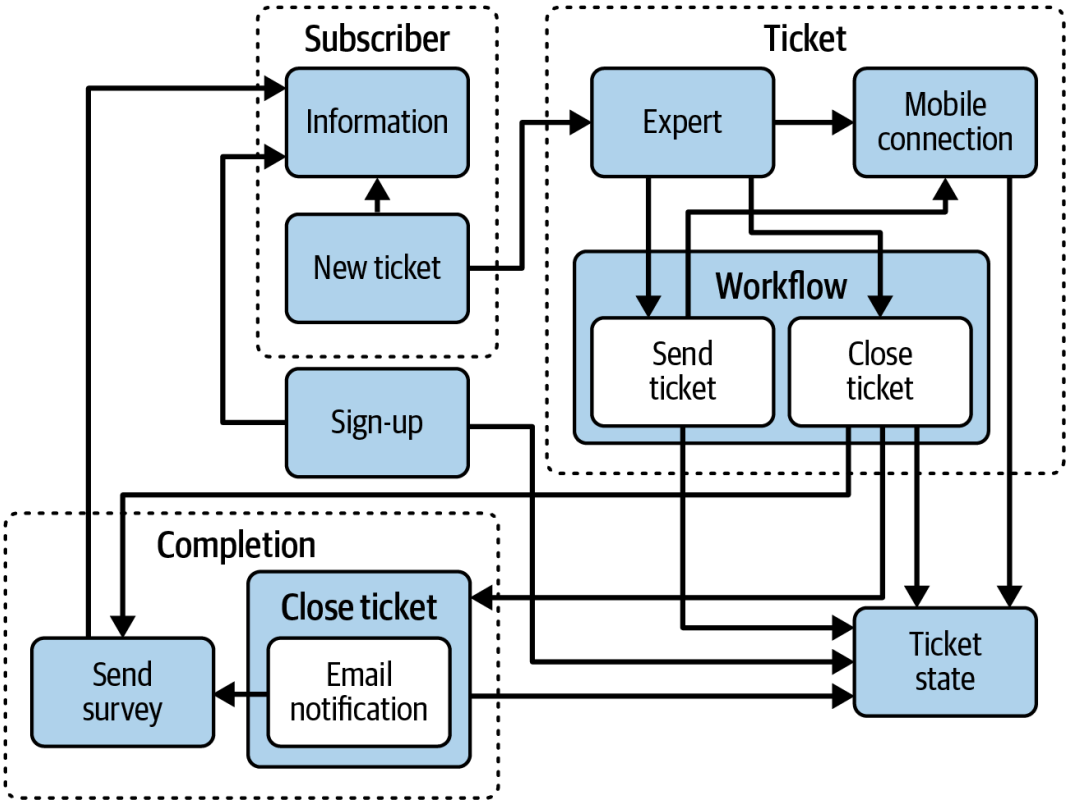

Figure 26-2. An example of an internal logical architecture that lacks governance and alignment

To ensure that the structure of the source code matches the logical architecture, we recommend using automated governance tools, such as ArchUnit for the Java platform, ArchUnitNet and NetArchTest for the .NET platform, PyTestArch for Python, or TSArch for TypeScript and JavaScript. These automated tools, coupled with good communication and collaboration between the architect and the development team, create implementations that are properly aligned with the architecture, as shown in Figure 26-3.

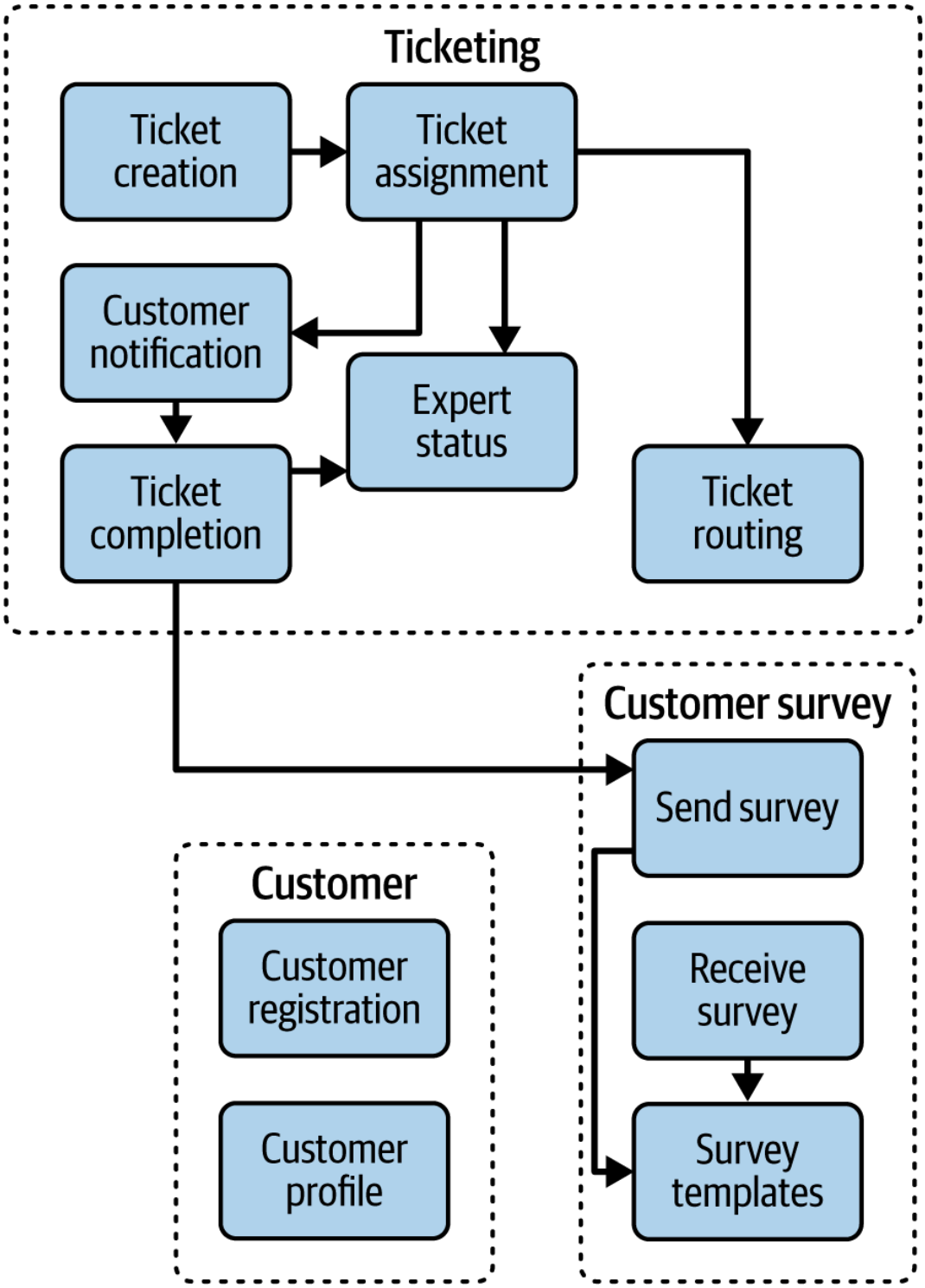

Figure 26-3. An example of an internal logical architecture based on proper governance and alignment

Compare Figure 26-2 with Figure 26-3. Notice how the architecture in Figure 26-3 is much more maintainable, testable, deployable, reliable, adaptable, and extensible than that of the misaligned architecture shown in Figure 26-2. This comparison illustrates the importance of this type of implementation alignment.

Architectural Constraints

A *constraint* is a governing rule or principle describing some sort of restriction within the architecture (such as limiting communications to only REST or using a specific type of database) required to achieve its goals. If the system's implementation doesn't adhere to its constraints, the architecture will fail. Thus,

part of a software architect's job is to identify and communicate the *constraints* of an architecture.

To illustrate what we mean by this particular intersection, consider a business trying to release a new system with a very limited budget and a tight deadline. This business expects lots of structural changes to the database and needs to get those changes done as quickly as possible. In this case, a traditional layered architecture (see Chapter 10) would be an excellent fit due to its simplicity, cost-effectiveness, and technical partitioning. Because this style separates the layers, database changes can be isolated to only one layer, making them easier and faster.

For the layered architecture to work in this particular business problem, the architect would need to define the following constraints:

- All database logic must reside in the Persistence layer.
- The Presentation layer cannot access the Persistence layer directly, but must go through all of the layers, even for simple queries.

These constraints are necessary to prevent spreading the database logic throughout the entire architecture, and so that changes to the physical database structure (such as dropping a table or changing a column name) won't affect any code outside the Persistence layer.

Now let's say the UI developers decide it's faster to call the database directly and implement the architecture that way instead. In addition, the backend developers realize it would be much easier to maintain and test the code if the Business logic and database logic are together, so they also ignore the constraints and couple these concerns in the business layer of the architecture. This implementation is not aligned with the constraints of the architecture, which means that database changes will impact all of the code in every layer, take too long, and the system will not meet the business goals.

Architectural tools are also useful for governing architectural constraints.

Architecture and Infrastructure

The scope of software architecture has grown over the last couple of decades to encompass more and more responsibility and perspective. Around the mid-2000s, the typical relationship between architecture and operations was contractual and formal, with lots of bureaucracy. Most companies outsourced operations to a third party to avoid the complexity of hosting their own operations, with SLAs for uptime, scale, responsiveness, and other important characteristics. Today, however, architecture styles such as microservices freely leverage characteristics that used to be solely operational. For example, elastic scale was once built into space-based architectures (see Chapter 16), but today, microservices handles it less painfully through tighter collaboration between architects and DevOps.

History: How Pets.com Gave Us Elastic Scale

People often assume that our current technical capabilities (like elastic scale) are just invented one day by some clever developer. In reality, though, the best ideas are often born of hard lessons. Pets.com is an early example. This ecommerce site appeared around 1998, hoping to become the Amazon.com of pet supplies. Its

brilliant marketing department created a compelling mascot: a sock puppet with a microphone that said irreverent things. The mascot became a superstar, appearing in public at parades and national sporting events.

Unfortunately, Pets.com's management apparently spent all the money on the mascot, not on infrastructure. Once orders started pouring in, they weren't prepared. The website was slow, transactions were lost, deliveries delayed…it was pretty much the worst-case scenario. Shortly after a disastrous Christmas rush, Pets.com closed down, selling its only remaining valuable asset—the mascot.

What Pets.com needed was *elastic scale*: the ability to spin up more instances of resources when they became needed. Cloud providers now offer this feature as a commodity, but early ecommerce companies had to manage their own infrastructure, and many fell victim to a previously unheard-of phenomenon: too much success can kill a business. The fall of Pets.com, and other similar horror stories, led architects to pay more attention to such intersections when creating software architectures.

The intersection of architecture and infrastructure is important because it facilitates operational architectural characteristics. Just because an *architecture* can support high scalability doesn't mean it will—if the corresponding *infrastructure* doesn't support it, it won't (as demonstrated by Pets.com). At client sites, we've all too often witnessed architects and developers being blamed for architectural failures that were really caused by a misalignment between architecture and infrastructure.

In most cases, this misalignment occurs because of a lack of communication and collaboration between the architect and those responsible for the infrastructure and operations. Architects often fail to realize infrastructure's influence on characteristics such as scalability, responsiveness, fault tolerance, performance, availability, elasticity, and so on. This misalignment gave rise to the field of DevOps.

For many years, many companies considered operations to be separate from software development, often outsourcing them to a third-party company as a cost-saving measure. In the 1990s and 2000s, many architectures were designed defensively around the assumption that operations would be outsourced and thus outside of architects' control. (For a good example of this, see Chapter 16.) However, in the mid-2000s, companies started experimenting with new forms of architecture that combined many operational concerns. For example, older architecture styles such as orchestration-driven SOA required elaborate tools and frameworks to support capabilities like scalability and elasticity, which greatly complicated implementation. So architects built architectures that could handle scale, performance, elasticity, and a host of other capabilities *internally*. The side effect was that these architectures were vastly more complex.

The creators of the microservices architecture style realized that operational concerns are better handled by operations. By creating a collaborative relationship between architecture and operations, the architects realized they could simplify their designs and rely on the operations people to handle the things they handle best. They teamed up with operations to create microservices, and to lay the foundations of what would become the DevOps movement. While DevOps has helped, this intersection of architecture and infrastructure is nevertheless still problematic for most companies.

While infrastructure is less of a concern, cloud environments can still get misaligned with an architecture. For example, deploying services across regions or even availability zones can decrease or even cancel out the performance and data integrity benefits of in-memory replicated caches and distributed caches.

Similarly, co-locating services, containers, or even Kubernetes Pods on the same virtual machine will significantly increase performance, but it will also adversely impact scalability, fault tolerance, availability, and elasticity.

Aligning architecture with infrastructure involves close communication and collaboration between architects and infrastructure team members, or even adopting DevOps practices, so that all stakeholders understand the critical operational concerns. Only then can architects truly realize the operational benefits of the architecture they selected—the ones that led us to grant those wonderful five-star ratings.

Architecture and Data Topologies

The intersection of architecture and data topologies is often overlooked. Choosing the wrong database type or topology can harm an architecture and negate its best architectural characteristics. For example, monolithic databases, while providing good data consistency and transactional support, can detract from scalability and fault tolerance. Similarly, distributed database topologies, while good at scalability and change control, can decrease a system's data integrity, data consistency, and performance.

The following sections describe the intersection of architecture and data topologies.

Database Topology

A database's *topology*, as we discussed in Chapter 15, refers to how its physical databases are configured within the architecture. Three basic topologies include a monolithic database, distributed domain-based databases, or the distributed database-per-service topology. Figure 26-4 illustrates these basic database topology types.

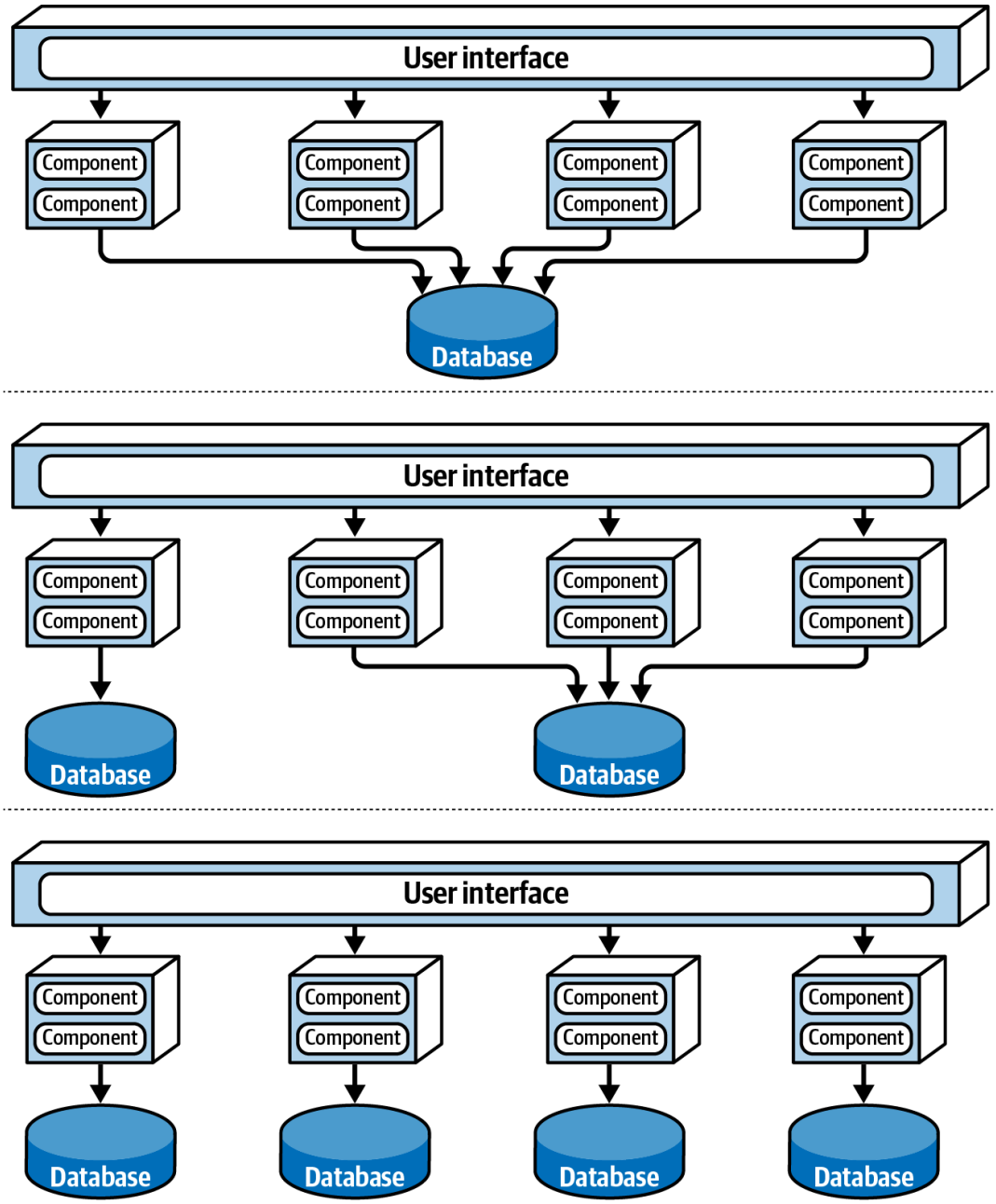

Figure 26-4. Common types of database topologies

The database topology must be aligned with the architecture in order to work correctly. For example, microservices architectures typically use a database-per-service pattern to maintain a strict bounded context. Without this proper alignment, it would be extremely challenging for architects to control change, and the system's operational characteristics, such as fault tolerance, scalability, elasticity, maintainability, testability, and deployability, would all suffer. That said, some styles, such as the service-based architecture (see Chapter 14), are more flexible with regard to the physical database topology.

Architectural Characteristics

In Part II of this book, we showed that every architectural style has its superpowers (rated at 4 to 5 stars) and its weaknesses (1 to 2 stars). So do database types—and it's important to align a system's architectural superpowers with the corresponding superpowers of its database type. In our book *Software Architecture: The Hard Parts*, your authors rated the characteristics of six different database types: relational, key-value, document, columnar, graph, and NoSQL. You might recall that scalability and elasticity are superpowers for

microservices, event-driven, and space-based architectures. They are also superpowers for key-value and columnar databases, which means that these database types are good choices to amplify those architectural characteristics.

Data Structure

The structure of data being stored and accessed is also a factor in this intersection. If the *data structure* is relational—that is, built upon a hierarchy of interdependent relationships—then a relational database will align well. However, storing key-value pairs in a relational database is a misalignment that can lead to inefficiencies in both the database and the architecture. Not all data carries with it the same structure, so keep an eye on this. Some data may be relational, other data may be document based (particularly when storing JSON-based event or request payloads), and still other data may be key-value driven. Given the potential diversity of data structures within any given architecture, we recommend leveraging polyglot databases whenever feasible.

Read/Write Priority

If the business problem involves high read or write volumes, that's important information for aligning the database topology with the architecture. For example, if the architecture requires high write volumes as a priority over infrequent reads, then a columnar database would be a good fit. However, if the reverse is true and high read volumes are the priority, then a key-value database, document database, or graph database would be more appropriate. If the system prioritizes reads and writes about equally, then relational and NewSQL databases would be good choices. Misaligning this factor can lead to poorly performing systems.

Architecture and Engineering Practices

In the late 20th century, dozens of software-development methodologies became popular, including Waterfall and many flavors of Agile (such as Scrum, Extreme Programming, Lean, and Crystal). At the time, most architects believed that none of this affected software architecture, treating development as an entirely separate process. However, over the last several years, advances in engineering have thrust process concerns upon software architecture. It's useful to separate software-development *processes* from *engineering practices*. By *processes*, we mean how teams are formed and managed, how meetings are conducted and workflows organized—in short, the mechanics of how people organize and interact. *Engineering practices*, on the other hand, refer to the process-agnostic techniques and tools teams use to develop and release software. For example, Extreme Programming (XP), continuous integration (CI), continuous delivery (CD), and test-driven development (TDD) are all proven engineering *practices* that don't rely on a particular process. Thus, the term *software engineering* encompasses both software development and these practices.

Focusing on engineering practices is important. Software development lacks many of the features of more mature engineering disciplines. For example, civil engineers can predict structural change with much more accuracy than software engineers can predict similar aspects of software structure. This means that the Achilles' heel of software development is estimation: how much time, how many resources, how much money? Part of what contributes to this difficulty is that traditional practices of estimation don't accommodate the exploratory nature of

software development and the unknowns that typically arise when developing software.

While process is mostly separate from architecture, iterative processes fit its nature better. Trying to build a modern system like microservices using an antiquated process like Waterfall will create a great deal of friction. One aspect of architecture where Agile methodologies really shine is in migrating from one architectural style to another. Agile methodologies support such changes better than planning-heavy processes because they have tight feedback loops and encourage techniques like the Strangler Pattern and feature toggles.

Architects often also serve as project technical leaders, which means determining what engineering practices the team uses. Just like carefully considering the problem domain before choosing an architecture, architects must also ensure that their architectural style and engineering practices mesh. For example, the microservices architectural philosophy assumes that teams will automate things like machine provisioning, testing, and deployment. Trying to build a microservices architecture with an antiquated operations group, manual processes, and little testing would likely lead to failure. Just as different problem domains lend themselves toward certain architectural styles, so do different engineering practices.

The evolution of thought continues, from Extreme Programming to continuous delivery and beyond, as advances in engineering practices make new architecture capabilities possible. Neal's book, *Building Evolutionary Architectures*, highlights new ways to think about the intersection of engineering practices and architecture that can improve how we automate architectural governance. The book offers an important new nomenclature and way of thinking about architectural characteristics, and covers techniques for building architectures that change gracefully over time.

In the software development world, nothing remains static. Architects might design a system to meet certain criteria, but to ensure that their designs survive both implementation and the inevitable march of change, what we need is an *evolutionary architecture*.

Building Evolutionary Architectures introduces the concept of using *architectural fitness functions* to protect (and govern) architectural characteristics as change occurs over time. Recall from Chapter 6 that architectural fitness functions are a way to get an objective integrity assessment of some architectural characteristic(s). This assessment may include a variety of mechanisms, such as metrics, unit tests, monitors, and chaos engineering.

To see how fitness functions can be used to help align this intersection, consider the business problem of needing a fast time to market. Time to market translates to *agility*—the system's ability to respond quickly to change. Agility is a composite architectural characteristic consisting of maintainability, testability, and deployability (see Chapter 6). All three of these architectural characteristics are influenced by engineering practices and procedures, and as such can be measured and tracked through fitness functions. For example, both microservices and service-based architectures support high levels of agility. However, if the engineering practices surrounding these architectural characteristics are not aligned with the architecture, the system won't meet those agility goals and requirements. Fitness functions can be used to help identify a misalignment, prompting the architect to take action to realign the engineering practices with the architecture (or vice versa).

Architecture and Team Topologies

As we've discussed throughout Part II of the book, team topologies can have a direct impact on software architecture, and vice versa. This alignment is so important that we've included a section on team topologies for each architecture style we introduce in this book.

One of the most basic ways team topologies can align with architecture is by type of partitioning. Like architectures, teams can be domain partitioned or technically partitioned. Domain-partitioned teams are organized by domain area and are typically cross-functional, with specialization throughout the team. For example, a particular domain-partitioned team might focus on the customer-facing portion of a system, and as such would be responsible for the end-to-end processing of customer-related functionality, from the UI to the database. Technically partitioned teams, in contrast, each focus on one particular technical function of the architecture and are usually organized by technical categories: for example, UI teams, backend-processing teams, shared-services teams, and database teams would align with the layered architecture style very nicely. Alternatively, these teams might be technically partitioned into business-function teams and data-synchronization teams, which would align well with the space-based architectural style.

Understanding how teams are organized is vital for ensuring the success of the system. If the organization's team topology is misaligned with the architecture, the teams will struggle to implement and maintain the architecture, which will be unlikely to meet its business goals.

Architecture and Systems Integration

Systems rarely live in isolation. Most require additional processing and data from other systems—and that brings us to the intersection of architecture and systems integration. When one system needs to communicate with another system to perform additional processing or retrieve data, its architect faces a host of challenges and implications. For example, is the system that is being called available? Does it scale and perform to the same level as the requirements of the calling system?

When architects don't focus enough on systems integration, the systems' static and dynamic coupling often results in architectures that can't scale, aren't responsive, and lack agility. When integrating with other systems, consider which communication protocols to use, what types of contracts to have between systems, whether the systems' architectural characteristics are compatible, and if the integration preserves each system's architectural quantum.

Architecture and the Enterprise

Every enterprise has a set of standards and guiding principles. By *enterprise*, we mean the collection of all systems and products within a company (or a department or division within the company). For example, many companies force certain security standards, practices, or procedures on architectural solutions, regardless of the type of system. Enterprise standards can also involve platforms, technologies, documentation standards, and diagramming standards, to name a few. Be aware of enterprise-level standards and practices, and ensure the architecture is properly aligned with them.

We have experienced many situations where the architect ignored enterprise-level practices, standards, and procedures. The usual result has been that the architectural solution, however technically effective, is deemed a failed “one-off” solution and scrapped. We can’t stress enough the importance of aligning the architecture with enterprise practices to ensure its success.

Architecture and the Business Environment

The business environment has a significant and *direct* influence on the architecture of its systems (and vice versa), and the business environment never stops changing. Is the company undergoing severe cost-cutting measures to stay afloat, or aggressively expanding? Is the business pivoting and repositioning every quarter to find its niche in a highly volatile and competitive market, or does it find itself in a position of stability? An effective software architect understands the position and direction of the company, and aligns the architectures of critical systems to match the business environment.

We call this alignment *domain-to-architecture isomorphism*. For example, companies undergoing extreme cost-cutting measures would not align well with microservices or space-based architectures, which are very costly to create and maintain. Conversely, businesses that are aggressively expanding through mergers and acquisitions would not be served well by monolithic architecture styles that lack the ability to evolve and adapt.

One issue architects typically face with this intersection is business change, particularly *unknown change*. Former US Secretary of Defense Donald Rumsfeld once famously said that:

> There are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns—the ones we don’t know we don’t know.

Many products and systems start with a list of *known unknowns*: things developers must learn about the domain and technology they know will change. However, these same systems also fall victim to *unknown unknowns*: things no one knew were going to crop up, yet have appeared unexpectedly. “Unknown unknowns” are the nemeses of software systems. This is why all “Big Design Up Front” software efforts suffer: architects cannot design for unknown unknowns. To quote Mark:

> All architectures become iterative because of *unknown unknowns*. Agile just recognizes this and does it sooner.

Planning for change in software architecture is *hard*. Evolutionary architecture practices help address an ever-changing business landscape, as does iterative architecture. Embracing architectural characteristics such as *portability*, *scalability*, *evolvability*, and *adaptability* also helps make a software architecture more flexible and adaptable to change.

Barry O’Reilly, an experienced software architect specializing in complexity theory and software design, came up with a new way of thinking about constant business change, called *residuality theory*. In his book *Residues: Time, Change, and Uncertainty in Software Architecture* (Leanpub, 2024), O’Reilly describes techniques for treating business change as *stressors*, and the corresponding

architectural changes as *residues*. His theory is that as the architect responds to change by applying more and more residues to the architecture, these residues will eventually start addressing *unknown* changes the architect cannot possibly predict, creating an architecture that has reached a critical state within complexity theory. It's an interesting theory, one we are watching closely.

Architecture and Generative AI

As we write the second edition of this book in early 2025, *generative artificial intelligence* (Gen AI) and large language models (LLMs) have infiltrated the world of software development and software design. Many companies are incorporating them into systems to perform tasks that previously were only performed manually by humans. Not surprisingly, Gen AI intersects with software architecture too: architects are incorporating LLMs into software architectures, and some are even using Gen AI tools to assist them in thinking through hard problems.

Incorporating Generative AI into Architecture

One approach we recommend for incorporating Gen AI into an architecture is to leverage *abstraction* and *modularity*. It's important to be able to replace one LLM with another quickly, and to allow for guardrails (rails) and evaluate results (evals) from various LLMs.

For example, suppose a job search company wants to leverage Gen AI to anonymize résumés, with the goal of reducing bias and placing focus on job seekers' skills rather than their demographics or other such factors. This task is normally done by humans, but could easily be done by an LLM. But are the LLM's results accurate? Does it remove too much information from the résumé? Does it keep too much demographic information in place? For this sort of system, it's vital to be able to gather samples and metrics and compare LLM engines. Tools such as Langfuse help create this kind of observability within the architecture.

Generative AI as an Architect Assistant

Given a proper prompt, an LLM (such as Copilot) can produce source code that saves developers a lot of time and effort. They're great for solving very specific, deterministic problems, such as "write source code in the C# programming language that generates a unique four-digit PIN number that has no repeating digits." But can LLM technology assist software architects with typical tasks? Here are some general examples of architecture-related prompts:

- Risk assessment: "Are there risk areas within this architecture?"
- Risk mitigation: "How should I address this risk?"
- Antipatterns: "Are there any common antipatterns in this architecture?"
- Decisions: "Should I use orchestration or choreography for this workflow?"

As of the time of the writing of this second edition (early 2025), we haven't had a tremendous amount of success with this endeavor. Asking an LLM whether microservices or space-based architecture would be most appropriate for a given situation rarely (if ever) yields the right answer. Why? Because, as we've demonstrated in this book, *everything in software architecture is a trade-off*.

LLMs are great for understanding *knowledge*, but to this day, they still lack the *wisdom* necessary to make appropriate decisions. That wisdom includes so much context that it's much faster for the architect to solve a business problem by themselves than to teach an LLM all about the problem and its extended environment and context. The fact that we've included eight other intersections to be concerned about should be evidence enough that this is a daunting task.

That said, we've seen some tools with promise. For instance, Thoughtworks Haiven can interpret an architecture diagram and fully describe a software architecture, saving the work of having to export a diagram into a machine-readable format such as XML and use it to prompt the LLM. Once they've imported that information, users can ask Haiven simple questions about the architecture, such as whether it can identify any bottlenecks or issues. Other efforts have included using an LLM to translate a PlantUML diagram or a pseudolanguage description of an architecture into executable ArchUnit code to govern the structure of a system. A lot of activity is happening in this area, so expect rapid change in the coming years in how Gen AI can assist architects.

Summary

Software architecture is a holistic activity that involves many facets of an organization. Effective software architects realize that creating and maintaining an architecture is much more than just selecting a particular architectural style and moving forward with implementation. It's also about making sure that the architecture is aligned with other aspects of the environment, and about using the communication and collaboration skills described in Part III to make that alignment happen.