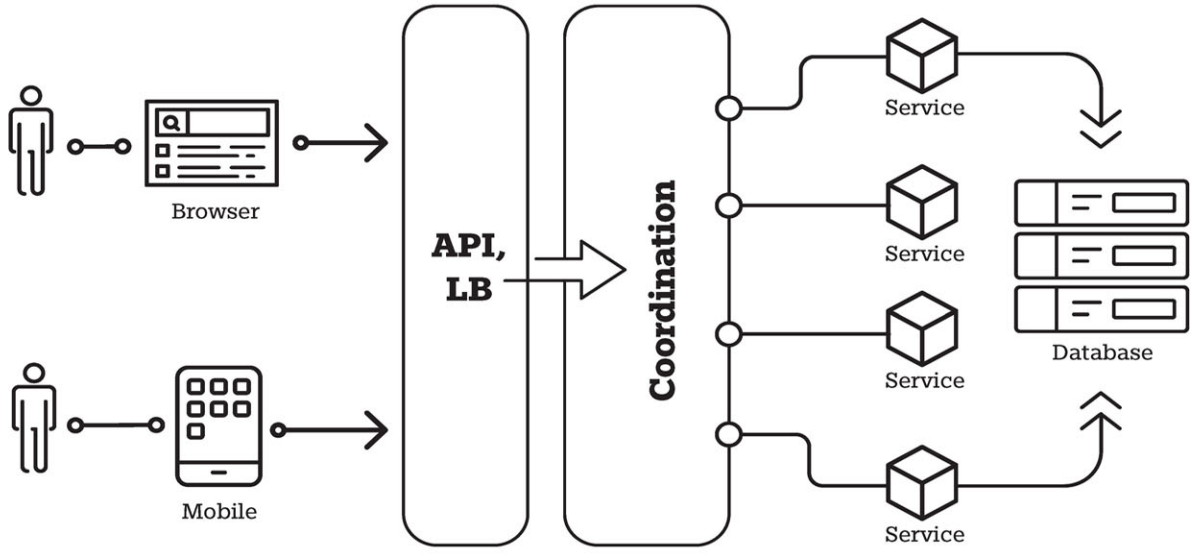
SOFTWARE ARCHITECTURE and DECISION-MAKING

Leveraging Leadership, Technology, and Product Management to Build Great Products



SRINATH PERERA

FREE SAMPLE CHAPTER |

SOFTWARE ARCHITECTURE AND DECISION-MAKING

This page intentionally left blank

SOFTWARE ARCHITECTURE AND DECISION-MAKING

LEVERAGING LEADERSHIP, TECHNOLOGY, AND PRODUCT MANAGEMENT TO BUILD GREAT PRODUCTS

Srinath Perera

Addison-Wesley

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2023946912

Hoboken, NJ

Cover image: wowomnom/Shutterstock; AVA AVA/Shutterstock

ISBN-13: 978-0-13-824973-1
ISBN-10: 0-13-824973-3

$PrintCode

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

This page intentionally left blank

This book is dedicated to my family: Miyuru, Basilu, and Nithika. Your presence brings color and purpose to my life. And to my parents, whose unwavering love and trust continue to astonish and uplift me.

I also extend my heartfelt gratitude to Frank for your steadfast support throughout this journey to its realization. Your insights were invaluable in shaping the core idea of this book.

This page intentionally left blank

Contents

About the Author

I started my architecting journey as an Apache open-source developer and have continued that for 20 years. I learned a lot by watching and later participating in architecture discussions in developer lists for those open-source projects, which is a great place for an aspiring architect to start.

I have played a major role in the architecture of Apache Axis2, Apache Airavata, WSO2 CEP (Siddhi), and WSO2 Choreo. I have designed two SOAP engines and worked closely with four. I was (and continue to be) a committer (a developer who can commit to a code base) for Apache Axis, Axis2, Apache Geronimo, and Apache Airavata.

I joined WSO2 in 2009. WSO2 products are used by many Fortune 500 companies such as airlines, banks, governments, and so on. At WSO2, I played an architecture review role for 10+ projects and 100+ releases. I reviewed hundreds of customer-solution architectures and deployments and sat in on thousands of architecture reviews.

At WSO2, when we faced a problem that could not be resolved by the immediate team, we set up a war room, where a hand-picked team restlessly attacked the problem. I have been in many war rooms and have led several, which have made me painfully aware of mistakes made in the software architecture. I've had a front row seat to world-class technical leadership and have also built many systems and learned from mistakes.

Later switching to analytics and AI-related topics, I co-designed WSO2 Siddhi and envisioned and shaped the AI features in WSO2 Choreo. Throughout this time, I published 40+ peer-reviewed research articles, which have been referenced by thousands of other research publications.

I hope you enjoy this book. Given the central role software plays in the world today, I will be content if this book helps make you a better software architect, knowing that I have contributed to better software that will be the lifeblood of the world for many years.

This page intentionally left blank

2

Understanding Systems, Design, and Architecture

What Is Software Architecture?

Software architecture is a plan to build a software system. This plan usually involves two things: defining a system as a set of components and specifying how those components work together. In complex systems, this decomposition happens recursively, where the architect breaks down each component into smaller components and defines their behaviors.

There are good plans and bad plans. The same goes for software architecture. What are the goals of good software architecture?

The overarching goal of software systems (hence, for software architecture) is to build systems that meet quality standards and that provide the highest return on investment (ROI) in the long run or within a defined period of time. *Long run* is the operative ideal here. For example, if we do not invest long term and build a crappy product, we end up with unhappy users, ultimately losing their revenue or spending too much money trying to make them happy. Cheaper in the short run is often more expensive in the long run. You pay now, or you pay later. On the other hand, adding a critical new feature, thus spending more money, can give us more revenue, improving the ROI.

Three kinds of uncertainty complicate architecting. First, we have a partial understanding of our users and what they want. Second, we have a limited comprehension of how our systems behave, especially in complicated and new situations. Third, we fail to recognize that as use

cases and users evolve, their requirements change. Therefore, we want our architectures to be easy to understand and flexible, enabling us to handle surprises.

When reaching for the best architecture, we can employ several best practices or tactics:

- Making decisions as late as possible. For example, when we start the design, we know the least about a system and the problems we are supposed to solve. If we can postpone solving some problems, we have an opportunity to learn more. This approach lets us make better design decisions.
- Envisioning a design that is easy to understand and to change. Problems change over time, and we may face a few surprises. Because a system goes through frequent changes over its lifetime, good architecture makes those changes easier.
- Saying no to features as much as possible. Many system features are rarely used, although they were often deemed important at design time. Knowing the audience, implementing only necessary features, and communicating why not may save us money in the long run.

Many argue that these tactics are always good, even considering them to be goals of the architecture. I disagree. To me, even those tactics incur costs; thus, everything is relative, and these tactics are useful only if they help us achieve the overarching goal. Let's look at a few examples.

If we know that a use case will significantly change in the future, forcing us to rewrite the system from scratch, we should not invest in making the current design extensible.

When writing a cloud app, we have two choices: We can choose a single cloud, taking advantage of its unique strengths for our application, or we can make the application portable across several cloud providers. Choosing only one cloud vendor makes development easier, but there is a chance that we may have to rewrite the system or parts of the system if porting to a different cloud. The cost of making the current version portable may be much higher than rewriting it, only if required.

You may have a great architecture that makes the system easy to change; however, it involves complicated concepts that will be hard for the current team to handle. (It may have been an adverse call forcing us to choose this architecture.) You fight with the army you have, at least until you have resources to enlist a better army.

You may be a startup trying to get your MVP out. Then, chances are you will rewrite everything once you are successful. You would not care for scale or even ease of change at this stage.

Uncertainties create risks, but the remedies have their costs too, which also create risks. Architecting is balancing those risks. Context is king, and it is hard to architect with evergreen rules and to get them right. To quote a *Star Trek: Discovery* episode:

> *Universal law is for lackeys. Context is for kings.*

Understanding all types of risks, planning around those uncertainties, communicating the plan, enlisting people, and managing risk along the way define *leadership*. This book discusses the leadership process in software architecting.

How to Design a System

As discussed, design requires holistic decisions that keep the big picture and final goal in focus. Although there are many good tactics such as creating designs that are easy to change, we need to ensure the medicine is not worse than the problem. Gregor Hohpe explains that architecture creates options, just like financing alternatives, which is an apt analogy because having options also incurs costs. Sometimes, it makes sense to have options, and sometimes it does not. An architect must have options and the wisdom to recognize when not to use them.

System design is like war: You must know your enemy (the problem, how often it changes, etc.) and your strengths and weaknesses of the team and yourself. Also, you must play the odds. For example, if the cost of being portable is 50% more, but there is only a 10% chance that you will have to move to a different cloud, then the expected cost of being portable early is 150%. However, if we choose to not be portable, then there is a 10% chance that we will have to port later. Let's assume that to port later we have to do 250% more work. Then, the expected cost of not being portable is (100% + 250 * 10%) = 125%, which suggests that we should go with the nonportable choice.

Chapter 1 discussed the concept of business context, which includes not only TOGAF's business architecture but also other factors like project timelines, team skills, and competitive threats. It's this business context along with user experience that add complexity to software architecture. Any given situation will present various trade-offs in terms of costs, which could include time, complexity, required skills, and benefits like performance, stability, and speed to market. The relative importance of these costs and benefits varies depending on the business context and user experience. Making these trade-offs and arriving at the right technical decision require sound judgment.

This book proposes five questions and seven principles that help us understand the context, acting as guideposts for good judgment. Part II and Part III discuss knowledge and explain how it relates to judgment.

As a reminder, the five questions include

- When is the best time to market?
- What is the skill level of the team?
- What is our system's performance sensitivity?
- When can we rewrite the system?
- What are the hard problems?

The seven principles are

- Drive everything from the user's journey.
- Use an iterative thin slice strategy.
- On each iteration, add the most value for the least effort to support more users.
- Make decisions and absorb the risks.
- Design deeply things that are hard to change but implement them slowly.
- Eliminate the unknowns and learn from the evidence by working on hard problems early and in parallel.
- Understand the trade-offs between cohesion and flexibility in the software architecture.

The five questions are designed to help us understand the terrain (or the business context). Starting with time, team, and performance requirements, the effects of the first two questions are well understood. Performance comes next because it determines how much precision we need in the design. Many find the fourth question regarding rewrites odd, but I believe there is a natural second phase to all projects, where we can rewrite the system. This question is crucial because it defines the first phase and lets us defer some hard but not immediate problems uncovered by the fifth question to the second phase. I believe this question significantly clarifies our scope.

Once we understand the terrain, the seven principles are about what to implement, when, and how. The first principle tells us we should look at everything from the vantage point of the user and choose only things that are useful for their journey. The second and third principles say that we should iterate, starting with a thin slide, exploring the design space, and getting user feedback. Next is the most important takeaway, making decisions and absorbing the risks. The fifth and sixth principles are about going into depth, and the seventh principle reminds us that most good architectural practices come with a cost, and this principle balances advantages and costs.

Five Questions

Good questions make us think, uncover details, and transform our understanding. I have found them to be a great tool when designing. These questions have helped me scope a system and dig into it. They are designed to ground us in concrete situations and to avoid grasping for ideals that often cause projects to fail.

Question 1: When Is the Best Time to Market?

The business, not the architect, decides the project timing. However, this is the first question that we, as architects, need to ask. Time can be our enemy because, often, even skills and money can't change the timeline of the product.

Time to market is everything, and our designs must incorporate these realities. When the deadlines are strict, we can design with the knowledge that we will be able to rewrite the system beyond the deadline.

My usual experience is that although the time-to-market deadlines are often not negotiable, features that should go into that version are often flexible. My recommendation is to work with a UX designer and product manager to understand the minimal features you must incorporate into the design and do it as fast as possible, using the most straightforward approach.

Question 2: What Is the Skill Level of the Team?

Leadership is about working with your team. Some teams are so good they might handle the system without any help from you at all. However, leadership is needed when we work with less-than-perfect teams.

> *You go to war with the army you have, not the army you might want or wish to have at a later time.*
> *—Donald Rumsfeld*

Take a hard, realistic look at your team. Your team may be veteran superstars, handpicked and employed by you over the years, fresh hires, or some mix of these. You must pick an architecture your team can manage. For example, do not pick an event-driven architecture or CQRS (Command and Query Responsibility Segregation)-based server unless you have a few people who have done this before. Those kinds of architectures have high costs in understandability and debugging challenges, and they most likely will cost much more in the long run unless the team understands their finer details.

What if you feel in your gut that CQRS is the right solution, but you do not have experts on board to achieve that architecture? I recommend designing the current version using a simple architecture and starting a proof of concept (PoC). That way, you can try out the CQRS in the background with the person who is most likely to handle it and with the hope that, in the second version, you will undertake CQRS.

You might be thinking, "Can't I train the team?" Yes, in some cases. For example, you might hire an expert who works shoulder to shoulder with the team for a few months. However, a deeper understanding of most complex systems takes time. My experience is giving someone a fingertip feel for performance, keeping in mind that the ability to handle details like concurrency or an LMAX disrupter takes at least a year, maybe two.[1]

Similarly, you should pick the programming language also based on the team. Programmers have many skills acquired around a given programming language, and it is often tough to change.

1. https://lmax-exchange.github.io/disruptor/

Certain abilities, like security and user experience, are essential. So, the leader must find a way to cover these areas. Doing so could involve hiring a consultant or the leader personally stepping in to support the team and provide guidance.

Rarely, the right choice could be to refuse to build certain software with a novice team. Instead, sometimes (in a startup, for instance) you can build a limited version (e.g., that scales less), which can then be used to justify more investment later. Even then, you need to make things very clear for people who are making the investment and make sure they know the risks.

Question 3: What Is Our System's Performance Sensitivity?

If a system operates close to the performance limits of a naive architecture, we say that the system is *performance sensitive*. Architectural considerations change significantly between systems that are sensitive and insensitive to performance.

The performance sensitivity of the system tells how much leeway you have and how much precision you need. Achieving a higher precision is like walking a tightrope; it is exponentially hard and needs experienced developers. Thus, performance-sensitive systems need exotic techniques, careful design, greater creativity, continuous performance measurements, and a feedback cycle. We need to test and identify unknowns through experiments as soon as possible. We must have a thin slice working end to end and invest early on in the system to collect detailed metrics on its mechanics. We discuss this style of design in Chapter 3. All this adds complexity and cost.

We can design performance-insensitive systems using a fingertip feel for the performance and simple architectural choices. We discuss this approach in detail in Chapter 3. Hence, the answer to this question significantly affects our architectural choices.

Note that many systems are performance insensitive. For example, using open-source service frameworks such as a Spring Boot and a database, you can easily implement a service that handles a few hundred requests per second. Even 50 requests per second are 4.32 million requests per day. With most businesses, if you are getting that many requests, chances are that you are already successful and can afford to write the second and third versions of the system. Most systems never need to exceed this limit.

Here's a second follow-up question: When we go beyond the limit of trivial implementation (e.g., 50 requests per second), will we have enough money to rewrite the system? Always ask this question: If we have that many requests, will we have enough money to rewrite the system? If the answer is yes, you can start with a simpler design and wait.

A much trickier scenario is if use cases require operating with latency bounds. We discuss this topic in Chapter 3. However, naive architecture can support (in most cases) expectations of latency of less than a few seconds (e.g., 1–10 seconds).

Question 4: When Can We Rewrite the System?

The fourth question helps us accept that we will rewrite the system eventually. For example, if you are a startup, do not try to build the architecture that you will need when you have a few billion users and hundreds of millions of dollars in revenue. When you get there, you will have enough money to rewrite the system several times over. Most successful systems have been rewritten many times over.

The common objection is that it would waste money to redo the system. Yes, it will cost, but to believe that you can think through all the details of a system as it will be in three to five years down the line is arrogance. There is so much uncertainty along the way. Chances are your system will not work for the first few moderate trials and will take longer to deliver.

Instead, be humble. Make the system work for the first 10,000 to 50,000 users, learn from them, and rewrite when the time is appropriate. Often, that time is not that far into the future. This approach helps us to be lean and simple, focusing on a few key problems, yet solving those systems properly. Do Things that Don't Scale!!.

Also, with the new IDEs, it is comparatively easy to refactor and redesign logic into a new structure. My belief is that we should plan to rewrite beyond key milestones (e.g., startup PoC to first serious funding round or beyond a million users). Having accepted that we will rewrite, we often realize that many features or guarantees can be done in the next rewrite.

Question 5: What Are the Hard Problems?

With the line of thinking I am proposing, it is easy to forget hard problems or push them out to the future. This question guards against such procrastination. But, sometimes, the hard problem is unrelated to the software, which is someone else's problem.

Most systems are part of a competitive landscape. We must, therefore, ask this question: What is our competitive advantage? If the competitive advantage is in the software, we have to work hard to achieve that. By definition, good competitive advantages are difficult. Otherwise, your competition would have already accomplished that or will do it once they figure it out. We can't achieve sustainable competitive advantages by doing as little as possible.

If your hard problems do not give you competitive advantages, then there is a good chance you can learn about hard problems from others. Likely, others have done it before, which can save you a lot of time and money. If a hard problem provides a competitive advantage, you must invest your time and energy in solving it. You need to invest in those as PoCs, independently of the system's design.

You need to start this process as early as possible. To do so, we should first ask the question: What is the minimal implementation that tests the idea? Then we should conduct a PoC to test it. We should bring the PoCs into the system after eliminating uncertainties in the simplest way possible.

In summary, we need to identify hard problems and handle them differently. Postponing them is not advantageous. We should identify problems that need long-term work and start fixing them early on, giving us time to get them right.

Seven Principles: The Overarching Concepts

Several overarching concepts (tactics) will help us achieve good software architecture. However, they may not always help, so we must evaluate them against the end goal of the system and use what is helpful.

Principle 1: Drive Everything from the User's Journey

The user journey defines what they can and will do with the system. It is not what is written down in the requirement specification. It is everything that can happen. The user journey, however, is never fully defined. It evolves as the user evolves and includes almost unlimited possibilities. For example, if we consider a bookstore, the user journey is what people do when they come in, and that is never fully defined. Do users want to search by the number of pages in the book? Do they want a specific author, or are they looking for a specific topic? Perhaps the user journey in this instance is to look only at journals.

We must strive to understand the user journey in as much detail as possible, covering the most important scenarios. Doing so provides a basis for building great UX and stops us from building unnecessary features.

UX makes or breaks a system. To provide a vivid example, "The Secret Startup That Saved the Worst Website in America," by Robinson Meyer explains how bad user experiences at Healthcare.gov almost broke the Affordable Care Act (ACA).[2] Many users gave up when registering, even though the alternative was not being able to go to a hospital when needed—the UX stopped even desperate users! UX alone does not make our systems successful, but without a good UX, our users won't have a chance.

The greatest source of errors in our architectures is unused or rarely used features, wasting time and money spent on them. The first step in reducing such features is to understand the user journey and evaluate everything in terms of the feature's utility to the user and the cost of forgoing the feature. We should build things that add value, not things that are easy regardless of the value.

Most systems have multiple groups of users who are interested in different parts of the user journey. We can never support all the users in all aspects of the user journey. We must choose one or the other. We have to make those choices deliberately and continuously. We return to

2. https://www.theatlantic.com/technology/archive/2015/07/the-secret-startup-saved-healthcare-gov-the-worst-website-in-america/397784/

this topic in the second principle. Furthermore, when we make a decision about the architecture, we need to consider these additional questions:

- How does this affect the user journey?
- How much value does it add?
- Is there something else we can do that adds more value?

Principle 2: Use an Iterative Thin Slice Strategy

Premature optimization is the root of all evil. —Donald Knuth

There are two ways to build systems. The first approach is to build all the parts and then integrate them. In my experience, most problems surface in the integration step, often adding months, if not years, to the project. The second approach creates a thin slice of the system that goes end to end and is useful at each step, using the most simple architectural choices. Then we identify bottlenecks and improve those, add new features, and replace anything only later, implementing complex architectural choices as needed.

When we are writing a basic application, this means getting the main path working as soon as possible, not worrying about the performance in the first round, then profiling the system and improving it to handle bottlenecks. With advanced compilers like JIT (Just In Time) that do many optimizations, it is tough to guess what parts need special handling. It is better to write simple code and optimize it only if and when needed.

Using this approach with a distributed app is a bit harder; however, the same idea works. Start with the most straightforward architecture and iteratively improve it. This also means integrating and merging new code as soon as possible. In other words, do small commits.

The Wright brothers are a great example of the power of this approach. Working with limited funds to build an airplane, they competed against well-funded professionals. Their competitors focused on creating the best design, building the plane, and then flying it. Opponents thought (perhaps, arrogantly) that they could think through all contingencies and build a plane that would fly on the first run. However, every time it failed to fly, it *wrecked* the prototype, setting them back months.

In contrast, the Wright brothers used an iterative *thin slice* strategy. They focused on first building a glider that worked, one that could land successfully, and then preserving the prototype. This strategy enabled them to do many more test flights. They perfected the glider and figured out how to control it. Then they added propellers and engines, gradually converting the glider into an airplane. This approach allowed them to learn, to tinker, and to experiment without months of setbacks at each failure.

An iterative thin slice strategy creates a powerful feedback cycle. This thin slice approach enabled the Wright brothers to improve gradually while in competition with much greater brain power and millions of dollars.

Unless you have a specific reason, always start with simple architectural choices. Measure the system, find the bottlenecks, and improve the system later; choose complex architectures only if needed. (Parts II and III describe some default choices and more complex selections for many situations.)

When undertaking the thin slice strategy, I have seen that simple architectures are enough to support systems over the years. A great example comes from threading models where the request per thread (with a pool) is inefficient, and nonblocking architectures can do much better. However, the resulting code from nonblocking models is harder to read, and it is not easy to find people experienced in writing this code. For many use cases, a simple request per thread model is sufficient throughout its life cycle. Let's keep our systems as simple as possible, starting unambiguously and then gradually adding complexity.

Another advantage of the thin slice strategy is that it forces everyone to integrate code early, fixing any misunderstandings about design before they come to a head and become overwhelming. This strategy works because it rapidly creates a working system, unlocking feedback, and enables us to uncover integration problems early. This approach gives us the time and the opportunity to improve and fix any problem we might encounter.

Principle 3: On Each Iteration, Add the Most Value for the Least Effort to Support More Users

As discussed, when designing the software architecture, we want to use an iterative approach that starts with limited features and then gets user feedback to improve the system. On each iteration, we want to add the most value for the least amount of effort. This means avoiding features that have little value, delaying less value-adding features to later iterations. It is important to note that most systems have many different user groups, and certain features add unique value for different users.

The user journey provides a powerful lens for making feature-related decisions. In most products, many users do only a few critical things. Find those and optimize for them. Doing this is the secret behind Apple's legendary UX. The podcast "Inside the Apple Factory: Software Design in the Age of Steve Jobs" describes Apple's approach in detail.[3] At Apple, about one-third of most teams are UX experts, so their UX quality is not an accident; they invest in it. Also, at Apple, any feature starts with the product lead (or product manager) and UX experts who then do mockups and iterations for stakeholders until the design is perfect. The code comes later.

Investing in such a process early on removes a lot of future changes and also provides a strong basis for accepting or rejecting future feature requests. Consequently, features won't be what is easy to implement but what is required by the end user.

3. https://www.youtube.com/watch?v=kI2Flp4oK-g

The first step for this principle is defining value. This step can mean supporting the most number of users, users who bring the most revenue, or users who can give the product the most exposure. We may even use different value criteria at different stages of the product. Examine the user journey to identify features that would add the most value by focusing on user groups that bring in the most value. In line with this principle, the following are concepts I try to follow:

- Principle 3.1: It is impossible to thoroughly think through how users will use your product, so embrace a minimum viable product (MVP). The idea is to identify a few use cases, do only features that support those cases, get feedback, and shape the product based on the feedback and experience from the MVP.
- Principle 3.2: Do as few features as possible. When in doubt (e.g., when the team disagrees), leave it out. Many features are never used, so you might develop an extension point instead.
- Principle 3.3: Wait for someone to ask for the feature. If the feature is not a deal-breaker, wait until three people ask for it before focusing on implementation.
- Principle 3.4: Have the courage to stand your ground if the features the customer requests adversely affect the product. Focus on the bigger picture and try to find another way to handle the problem.

  Remember the quote often attributed to Henry Ford: "If I had asked people what they wanted, they would have said faster horses." Also remember that *you are the expert*. You are supposed to lead. It is the leader's job to do what is right, not what is popular. Users will thank you later (fourth principle).
- Principle 3.5: Look out for Google envy. Do not overengineer. We all like shiny designs. It is easy to bring features and solutions into your architecture that you will never need. For features such as quality of service (QOS) improvements, scale, and performance limitations, wait until those requirements are imminent. Also, approach the product with the mindset that you will rewrite it. Implement what you want now.[4]
- Principle 3.6: When possible, use middleware tools or cloud services. For example, consider authentication and authorization. If you decide to implement these, it will create a lot of feature requirements in the future. For instance, you will need a user registration flow, password recovery, and attack detection. Using an identity and access management (IAM) tool supports all those features, and IAM will continue to evolve its product as requirements change. The same idea applies to message brokers, workflow systems, payment systems, and so forth.
- Principle 3.7: Interfaces and other abstractions are techniques for creating options and delaying decisions. Use them carefully. Like financial options, software options also

4. For details, see "You Are Not Google" at https://blog.bradfieldcs.com/you-are-not-google-84912cf44afb.

have costs. Learn to be mindful of them. Know that this presents a trade-off, thus a judgment call and, hence, a leader's responsibility. For example, a common mistake, or anti-pattern, is too many abstraction layers, which creates a terrible performance impact when we ignore the cost of abstractions.

This UX approach should go beyond UIs. We must use the same approach with APIs and internal and external messages because these formats are hard to change later. Create those APIs and message formats, iterate, and get feedback. Remember, we must design deeply but implement as little as possible.

There is one exception to implementing features as late as possible. This minimal approach does not work with features you'll need as a competitive advantage or for security. You must invest in them independently of the design process. The sixth principle addresses these unknowns.

Principle 4: Make Decisions and Absorb the Risks

The most senior technical person in the project (whom I call the chief architect) must make decisions and absorb the risks. Any project faces many uncertainties; for example, how much load and latency limits should the first version of the system have? The reality is that, often, nobody knows that number. We often ask customers, and they do not know it either. However, someone has to put down the numbers so that the team can go ahead and hit the target date. Without a target, the team can lose much time in indecision.

Richard Rumelt's book *Good Strategy, Bad Strategy* (Profile Books, 2011) provides a great example of this principle. When beginning to design a moon rover, nobody knew the moon's surface. The team designing the first such vehicle was stuck. Phyllis Buwalda, director of NASA's Future Mission Space Studies team, wrote a specification for the moon's surface based on the toughest desert on Earth. She understood that unless she took the risk of specifying the target, much time would go to waste. By writing the specification, she absorbed the uncertainty on her shoulders, thus enabling the team to make real progress.

Similarly, the chief architect must collect the required data, perform the necessary experiments, and yet, at the end, understand the unresolvable uncertainties (such as how much load the system will get) and make decisions that set concrete targets. Leaders must remove ambiguity and create targets that are solvable.

Principle 5: Design Deeply Things That Are Hard to Change but Implement Them Slowly

In my opinion, this fifth principle is the crux of designing software systems. We should design deeply but implement slowly. Let's explore what this means.

I usually advocate simple designs and adding complexity only when needed. However, some parts of the design are hard to change, such as

- APIs exposed directly to customers
- APIs of highly shared services
- Database schemas (if we deploy a product that uses a database in the customer premises)
- Shared data, objects, and message formats
- Technology frameworks

When designing, we need to expend significant energy in designing parts like APIs and database schemas. These designs must go through a lot of reviews and iterations before putting them out to the customer. For example, with APIs, even if we version those that are exposed to our customers, old releases hang around for a long time. They are hard to change, even beyond a rewrite. APIs of shared services are also difficult to change because that would require coordinated releases.

To understand what is hard to change, we must design the system deeply. At the design level, we need to dive thoroughly into creating a design that can potentially solve the entire problem and even create PoCs as needed. Having a potential design opens our eyes to possible surprises and enables us to learn more from evidence as they come up. Designing early and deeply lets us start discussions and build consensus from the start, which often consumes a lot of time.

When designing deeply, know that it's impossible to go deep into every aspect of the software due to limited time and resources. For any part of the system that we can change and evolve without affecting the rest of the system and for those that do not contain significant unknowns, we can defer the details to a later date. Doing this properly requires judgment. Unless we do this, however, we will drown in the details.

For example, writing a service is a well-understood problem, but unless we see the need for that service to handle complexity (e.g., large throughput, large messages), we can defer the implementation details after defining the APIs. In general, if an API or interface hides the implementation details and that is understood, we can delay the implementation design. Thus, designing deeply should focus on APIs, interfaces, and their interactions. We must, however, realize that the current design will be based on our incomplete understanding of the problem and will evolve over time.

The deep design does not imply an urgency to implement it. Doing things slowly lets us implement things with more understanding and helps avoid future changes. Bring about things only when your user journey analysis indicates that they are necessary and that they add significant value. Designing deeply, implementing slowly, and using the judgment required to do this efficiently and decisively are hallmarks of a great architect.

Principle 6: Eliminate the Unknowns and Learn from the Evidence by Working on Hard Problems Early and in Parallel

Detect unknowns early and systematically eliminate them rather than trusting your luck. Often this effort requires experiments to resolve them, which is one of the chief architect's key responsibilities. Resolving unknowns requires trial and error, which usually takes time. Proactively exploring unknowns gives us enough time to inspect those problems and find the right solutions. This foresight differentiates a great architect from a good one.

Kelly Johnson, the aircraft designer, offers a great example. Designing aircraft for Defense Advanced Research Projects Agency (DARPA), his team built the first aircraft that goes three times faster than sound (Mach 3). Wind tunnels at that time could not simulate wing design at this speed. Kelly found a simple solution: He collected data by borrowing 400 missiles, mounted different wing designs on them, and conducted experiments.

Experiments are a crucial tool in any designer's arsenal. Because it is much easier to do experiments with software than with an aircraft; we have little excuse for not doing them. One of my advisors used to say never argue or analyze something that you can check with fifteen minutes of code.

This principle also ties in with the deep design that allows us to proactively identify unknowns beyond what is apparent at first glance. If we believe a certain part of the design is unknown and risky, we need to dig into that part early to give us time to resolve the unknown.

There is a second related point. With software, it is easy to rerun something. Yet, we do not want to build monitoring into the system and are bad at collecting enough data to understand what is really happening. Ironically, because it is easy to collect the data, we never collect it. Yet, complex problems and situations do not happen often and are hard to recreate. Unless we collect data, it is hard to learn from these situations, robbing us of an opportunity to fix bugs and to deeply understand the system.

In contrast, designers in many other disciplines such as vehicle design, aeronautics, and medicine have only a few experiments about a particular topic at their disposal. Hence, they collect a lot of data and usually know much more about their systems than software professionals do.

We should add monitoring into our systems early and take the time to instrument it. For example, we can measure operating system telematics, queue sizes, selected traces, timed breakdowns, and throughput at different places in our system. Also, because it is not practical to comb through the data daily, we should automate the analysis process as much as possible. Careful monitoring enables us to learn a lot from every situation.

Monitoring has a minor performance penalty. Yet, in the long run, we will save money by building better systems. This kind of monitoring is essential for the feedback loop if we operate within tight performance constraints.

Principle 7: Understand the Trade-offs Between Cohesion and Flexibility in the Software Architecture

As budding architects, we learned about the principles of flexibility and cohesion in the architecture. Venkat Subramaniam's talks are a great source for understanding these principles.[5] However, most of these principles have costs too. Hence, software architecture must be evaluated in its context, which we explored in the five questions, but sometimes we have to break the principles to create the best architecture.

Flexibility refers to the ability of the system to change. As mentioned, flexibility also costs and can be more expensive. For example, as we discussed earlier in this chapter, flexibility to run on multiple clouds can, on average, be more expensive than building for one cloud and redesigning if and when it's needed.

Cohesion broadly means that architectural concepts are applied throughout the system. A common thing to check is whether the system reuses its components or services everywhere. An ideal system should be composed of services or components that handle one aspect (e.g., only logging, security, messaging, registry, mediation, or analytics), and all parts of the system must reuse those aspects when needed without reimplementing them. If you need configuration parsing, use configuration parsing components. If you need logs, use the logging component. This extends the DRY principle (Don't Repeat Yourself) from code to architecture.

In modern architectures, this reuse can happen at the library level (same process) or at the service level. Unfortunately, trying to enforce this principle too rigidly can lead to problems. For example, asking every service to call a configuration service or query builder service can be too much (but not always). Sometimes, bringing in a component can also be too complicated because it brings in other dependent components in turn. Simple features can cascade into significant changes. I saw an example of this, where adding mediation dependency to an identity server added hundreds of new dependencies.

The most unfortunate use of cohesion happens as follows: We detect some aspects of one service that can be reused by another service and ask the first team to refactor and create a new service or a component. The second team incorporates this service into their system. This kind of refactoring, which forces close communication between multiple teams, should be done only when it is absolutely necessary.

Usually, it is not worth doing this to reduce duplication slightly. I have done this and paid the price. With hindsight, I am now willing to live with some level of duplication and inconsistencies when fixing those results in significant complexity. The cure, sometimes, can be worse than the disease.

It is useful to think about architecture as a way to build systems that are cheaper in the long run and tactics as tools in your toolbox. We use tools only when they make sense. In the next section, we look at a sample system to explore how to use these questions and principles.

---

5. See http://alex-ii.github.io/notes/2017/12/09/core_design_principles.html.

Designing for an Online Bookstore

As a running example, let's consider an online bookshop, where users can search for rare books, order them, make payments, and then track the order until it is delivered. It also includes returns and any after-sales services. This example will show us how to use the concepts mentioned in this chapter in real use cases.

As previously noted, a solid design process begins with an understanding of the business context. Take, for example, a bookstore, which can range from being quite simple (like a friendly neighborhood bookstore that announces new arrivals via WhatsApp) to extremely complex (like Amazon). These differences are shaped by the unique business context.

It falls to leadership to ensure the business context isn't lost when faced with making tough choices or trade-offs at the information system or technology levels. We've already discussed five questions that aid in understanding the business and technical context, as well as seven principles for iteratively improving the system's design in the realm of software architecture.

Furthermore, as we've previously discussed, our conversation primarily centers on the design layer of the information system.

First, let's consider the business context. We have six months to take the product to market with an average team. Our initial goal is to establish the product in the market. We do not know how much load we can expect. However, the back-of-a-napkin calculator shows us 50–100 TPS (Transactions per Second) throughput, which means that the business will be in good shape. It is fair to assume that we can rewrite the system at that point.

A developer cannot make this decision; one of the leaders has to make it, which is an example of the fourth principle. The two unknowns are transaction processing at scale and book recommendations. We are able to differentiate the first problem because we need only 50–100 TPS. We need to start exploring the recommendations soon because this is unknown to the team.

As per the first principle, we should start by understanding the user journey. My recommendation is to start with a UX design. In my experience, writing a requirement specification does not work well because neither designers, developers, nor users can see the fine points in the design without experiencing the system. We need an iterative approach. A mocked UX lets everyone experience the system and iterate it.

As we alluded to previously, the design has many levels of recursive abstractions. A typical system would have a macro-level architecture that describes different services, data stores, and other middleware and how they relate to each other. Then, each service would have an architecture that describes different components and how they relate to each other, and each component would have an architecture on the code level and how those code segments relate to each other. This book focuses primarily on the first two levels. We discuss how to architect the overall system in Chapters 5–10 and how to architect individual services in Chapter 11.

Having narrowed down the UX, we should focus on the macro architecture. Figure 2.1 shows a typical macro architecture for the bookstore. Typical software architecture in the 2020s would use databases to store the state: a set of (stateless) services that handle business logic. Those services are used in one of three ways: a single-page application (SPA) running in the browser, a mobile app, or direct API calls. (Chapters 5–10 discuss these in more detail.)

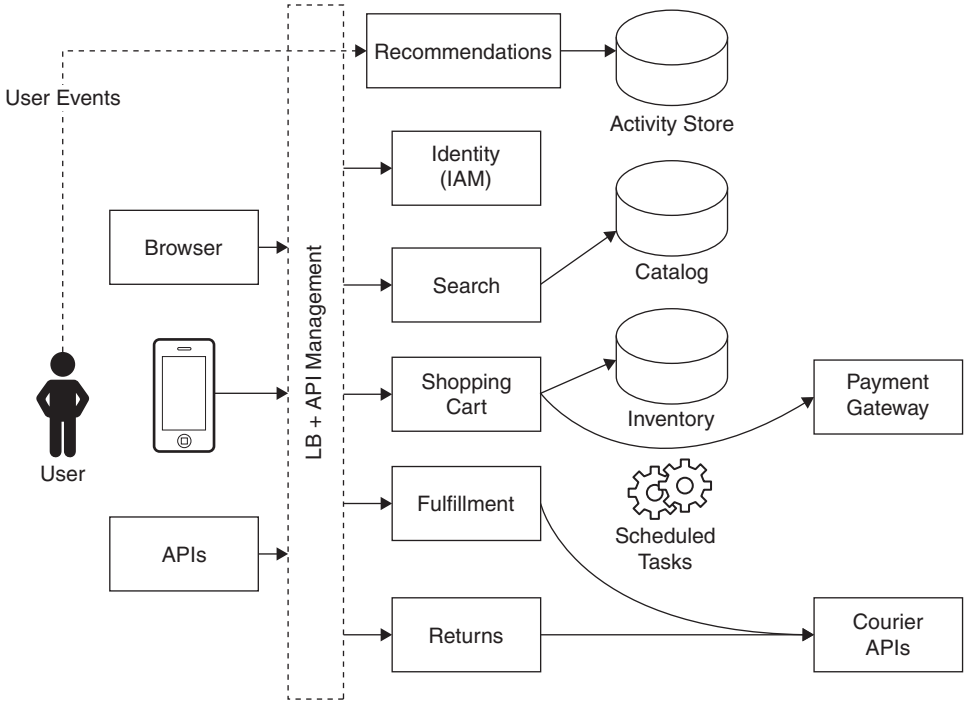


Figure 2.1
A typical macro architecture for the online bookstore.

Services are loosely coupled with other services. If we use microservices concepts in our macro architecture, each service can be developed, released, and deployed independently. Identifying services given a problem is called *service decomposition*, which is a crucial skill of an architect. Chapter 5 discusses it in more detail under SOA.

Once we have identified services, the next step is to identify interservice interactions and to define message formats (APIs) for those interactions. At this point, our "do as little as possible" approach applies some friction.

It is hard to change the message formats or API of a widely used service later. We must spend time thinking through these interactions and develop a mature set of APIs. We can use user interactions identified in our UX design in this phase. Following the fifth principle, at this point, we should design deeply, define message formats, and think through immediate and long-term use cases. As part of thinking deeply, we should also define the database schema.

When we deeply design both schemas and APIs, it clarifies most of the design. It is a good idea to take a lot of feedback and discussion about APIs and databases to ensure we get them right.

As mentioned, we implement slowly, learning and revising the design as we go on. A far-reaching API design gives us a broad and balanced understanding of the system. Public APIs need extra care, however. If well-defined message standards exist, we should adopt them as much as possible. For example, using JWT (JSON Web Tokens) tokens for authentication saves us the need to define a token format and also gives us the flexibility to change our identity server later.

Once we have a design, we should plan the implementation. As principles 2 and 3 mention, we should first identify a thin slice and get that working. This could be the ability to see a book, select it, and order it. Each iteration after that should create features to maximize the value they add. For example, iterations can add search, shopping cart, returns, recommendations, and so on to our online bookstore.

In parallel, as per principle 6, we need to start exploring hard problems such as recommendations and even scalable transaction processing. The reason is that we need time to get them right.

After identifying the abstract architecture, while implementing iterations, we should design our services. We can do this by deciding which parts to develop, which parts to reuse, and how to implement them. Here are some examples:

- We can implement each service using tools like Spring Boot and MySQL. For services such as IAM and payment APIs, we can use either an off-the-shelf middleware or an SaaS (Software as a Service) solution.
- We can implement fulfillment and return services using a message queue or a workflow system, due to their asynchronous and long-running nature.

The final choice needs to factor in considerations such as time to market, required performance, and the experience of the team. My recommendation is to start simple and add complexity as needed unless you have prior experience in building similar systems.

At some point in the middle of development, we should take the product to customers. This point is called *minimum viable product* or *minimum lovable product*. It can start with friendly users and expand to more and more users.

At each step, we should strive to learn. Although we have a design, we can modify it if our learning suggests changes. Note that this process continues as long as the system is live.

From the viewpoint of TOGAF's three layers, the majority of the architecture we've talked about falls into the category of information systems architecture. Yet, most decisions are influenced by the business context, which expands upon TOGAF's business architecture. We delve into technology architecture only when we discuss specific technologies, like Spring Boot or MySQL, and that's mainly done as examples or to illustrate complexity.

Designing for the Cloud

Several exciting possibilities open up if we are designing the system for the cloud. We have two choices:

- Shallow cloud integration: We write our services, pack them as containers, and run them in the cloud using cloud services for databases and storage only. Such a design architecturally behaves similarly to on-premises systems.
- Deep cloud integration: We build the system using the cloud as much as possible, replacing all services with serverless functions and as much functionality with cloud and SaaS services.

Figure 2.2 shows an example of architecture for a bookshop that uses the cloud as much as possible.

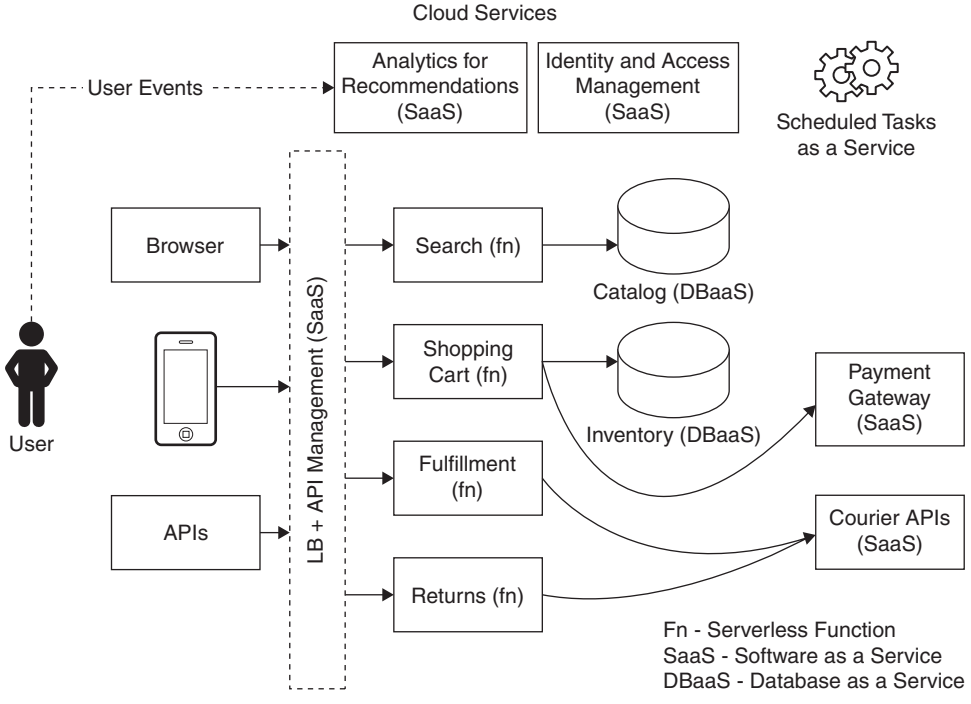


Figure 2.2
Software architecture for a bookshop that uses the cloud.

Choosing such an architecture yields several advantages. First, it provides a faster time to market because it

- Needs less coding and configuration
- Avoids boilerplate code such as security logic with configurations

- Replaces functionality with API calls
- Focuses on business logic instead of plumbing
- Provides HA, scalability, and DevOps out of the box

Second, it lowers platform costs by providing a true, pay-as-you-go model while eliminating idle time costs. Most applications have variable loads. However, according to the central limit theorem, when many of those variable loads are aggregated, the resulting workload has a predictable distribution curve even if the individual workloads are not predictable. Consequently, cloud platforms can operate with fewer resources despite the additional overhead. As a result, cloud providers enjoy substantial savings through economies of scale, which they can pass on to the users.

Third, cloud platforms lower development costs by taking over DevOps and monitoring costs. Cloud platforms can deliver DevOps and monitoring for a fraction of the cost, using economies of scale, tools, and optimized operations. They can also pass on some of those savings to the end user, creating a win-win situation for both.

Fourth, cloud architectures provide predictable costs, tying costs to the amount of work the system receives, thus reducing capital expenditures and the risks of operating the system. Usually, doing more work brings organizations more money; hence, having cost tied to future revenues is a welcome development. Because cloud platforms are metered with fine granularity, they provide greater insights into managing costs.

Cloud-based architectures also have disadvantages:

- A deep cloud integration invariably creates a lock-in restriction, making it hard and expensive to move away from a cloud provider after going to production.
- Using the cloud requires the team to learn new programming models. Furthermore, cloud platforms are opinionated, forcing programmers to follow preset patterns, allowing users little or no leverage to get those fixed if cloud features do not fit well with their requirements.
- The cloud can be more expensive than other options if the system receives a significant load around the clock.

An architect must balance these pros and cons and decide on which approach to use, based on the key questions we discussed in this chapter. Sometimes, it may be more economical to accept the lock-in and commit to rewriting if we have to move out of the cloud.

Next, we move on to Part II, where we discuss performance and UX concepts, which are key tools for good design. However, if you do not want to delve into technical details, you can skip to Part III, where we discuss macro-level and micro-level design.

Summary

Following are key takeaways from this chapter:

- Software architecture is a plan to build a software system.
- The overarching goal of creating a software system (hence, for software architecture) is to meet quality standards and ones that are more economical in the long run.
- Although there are essential tactics (e.g., making code easy to change, avoiding lock-in), we should evaluate each of those tactics not as something that stands alone but as something that is a part of the whole. For example, sometimes, it might make sense to accept lock-in and go to the cloud if in the long run it is cheaper to rewrite the system than switching to a new cloud provider.
- The best design depends on the context. Hence, it is a matter of judgment.
- We discussed five questions and seven principles that can help us make the right judgment calls when designing and implementing software systems. We saw those questions and principles in action by way of an example.

This page intentionally left blank

Index

Numbers

A

B

C

D

E

F

G

H

I

J

K-L

M

N

O

P

Q-R

S

T

U-V

W

X-Y-Z