# STEP 7: Evolve Your Architecture

You have already learned that software architecture is not fixed forever. It changes along with the business, technology, and environment.

That is why it is important to design an architecture that can evolve without requiring a complete refactor.

Is it possible? Yes.

But it is not easy due to the various mistakes we make.

When starting in a new environment, the amount of information to learn about the business domain is relatively high. At the same time, we have to make most of our technical decisions, which can lead to a "big ball of mud" scenario.

We are also influenced by conferences, content creators, and success stories from FAANG companies. This often results in overcomplicating the architecture from the start, making it hard to maintain. Another issue is oversimplification—keeping the architecture too basic for too long instead of allowing it to evolve.

> *Choose an architecture based on your current needs and context, not wishful thinking.* —Me :)

To address this situation, it is worth taking the evolutionary approach. Apply patterns that will help you at that given moment. Think about the traffic you must handle in the next hours and days.

Observe and adapt. Evolution, not revolution.

In the following sections, we will explore the most common challenges faced during software evolution and learn how to prevent them using an evolutionary architecture approach. Additionally, we will leverage the power of architectural drivers and see how they can influence our decision-making process.

# Project paradox

When starting application development, our domain knowledge is minimal, and the number of decisions to be made is enormous. We need to consider business processes, libraries, frameworks, and infrastructure. Due to our limited knowledge, many of these initial decisions are likely to be incorrect.

Over time, our understanding grows, but the main problem is that most of the decisions have already been made. It is called Project paradox[1]. To address this, it is crucial to defer as many decisions as possible to later phases. This way, as our knowledge improves, we can make more accurate decisions.

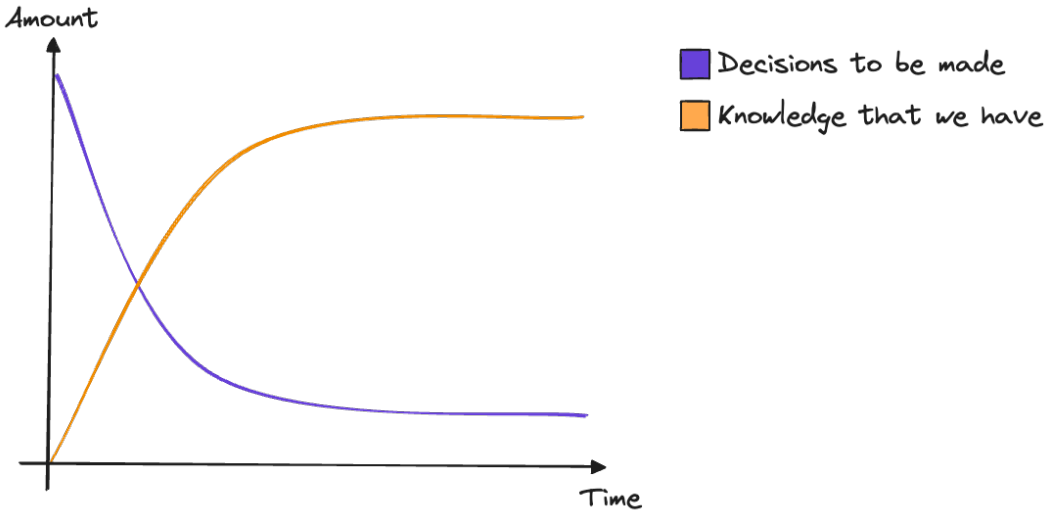


**Figure 152. Relation between the amount of knowledge and the number of decisions to be made**

This brings us to a crucial point: why not invest more time in understanding and planning what and why we want to build? By doing so, we can significantly improve the quality of our initial decisions, creating a safer environment and avoiding many potential problems that could arise from rushing through the early stages.

Feel free to use this diagram in conversations with business representatives and your development team. Many people find it eye-opening.

[1] https://beyond-agility.com/project-paradox/

# Common architectural pitfalls

Due to issues related to project paradox, I frequently encounter one of the following two problems when it comes to software architecture:

1. **Too complex from the start**. This is a common issue, occurring in about 50% of cases I faced. Teams often overcomplicate the system by implementing components that are not needed at the moment. For example, they might add caching when the database alone could handle reads, or choose microservices for a small system intended for a few thousand users. This results in a system with a high entry threshold. Lack of business knowledge leads to poor technical decisions.
2. **Too trivial for too long**. Based on my experience, this happens in about 30% of cases. Teams may start with a simple architecture (which is a good idea) but fail to evolve it as the system grows. This oversight leads to performance and maintenance issues over time. Lack of evolution often results in deferred problems.

Let's look closer at each of them.

## Too complex architecture from the start

Let's assume you have just joined a company as a software architect. Although you have made some architectural decisions in the past, this is the first time you are designing a system from scratch.

It is a great opportunity, and you want to build the application using the patterns and components you have always dreamed of. In the last few years, you have attended talks, meetups, and conferences where you learned about microservices, containers, cache, data streams, aggregates, and many more. It is time to put them into practice!

Other developers on your team also support this approach. Although there are some dissenting voices, you manage to persuade the majority. After several rounds of brainstorming, you collectively decide to:

- Go with microservices.

- Run them in Kubernetes.
- Add cache.
- Add data streaming.

You start the development and work heavily on features you have to deliver. The process takes twice as long as planned, but the results are impressive.

Each microservice scales effectively and is deployed to Kubernetes. Reads are optimized using cache, but optimization is not yet needed. You added a data streaming component, but in the end, you do not need to stream anything.

Now, you can serve millions of users! However, the reality is different. After several months, there are only 5,000 users who rarely use the application. Instead of having a simple setup that would be enough, you have to deal with a complex one.

There are a few problems:

1. When something does not work as expected, you have many places to check, due to the numerous components involved.
2. Anyone who joins your team has a very high entry threshold. It takes several months to start being productive.
3. Infrastructure costs are higher than alternative solutions. For example, I once consulted for a company whose infrastructure costs could be reduced by 90%, from $100,000 to $10,000.
4. Dealing with a distributed system and facing all problems related to it, such as network errors, higher latency, independent deployments, different versions, and general complexity.
5. Even if the boundaries for microservices were correctly set, they might change a lot in the first weeks and months. This means that you will need to get rid of some microservices, merge them together, or split them into separate ones. This is one of the main reasons why, in most cases, starting with microservices does not make much sense.

The most common reason for this decision is a fascination with new and trendy technologies. We hear about them at conferences and see how they solve problems for others, which leads us to want to adopt them ourselves.

However, applications and environments vary, and a solution that works for FAANG companies may not be effective for us due to differences in scale.

Often, the realization that the architecture is too complex comes too late to switch to something simpler. Due to this complexity, each change or extension becomes costly. Also, when it becomes apparent that a component is unnecessary and incurs high costs, removing it is not straightforward and can be time-consuming.

## Too trivial architecture for too long

Let's say you have just joined an e-commerce startup company as the CTO. This is your first time in such a critical role, and it is a fantastic challenge. You are determined to meet expectations and are committed to delivering the MVP (Minimum Viable Product) on time.

However, you face three major challenges:

1. The deadline is in four weeks.
2. Your budget is extremely limited.
3. Only you and one other developer are available for implementation.

You need to start the implementation quickly. There is no time to learn new technologies, so you choose from what you already know—whether that is a no-code, low-code, or heavily coded solution.

At this stage, all your design decisions are acceptable—after all, you need to test your product with real users as soon as possible. However, there is one catch: if the product succeeds and gathers more interest, you will need to start iteratively refactoring the application. And this is where the problem arises.

After a successful release, customers demand more features, which increases the workload. There is no time to bring in new people, and it takes ages to hire them.

In the MVP version, you implemented a table of `Products` and decided to add all the related information to it:

| Id | Name | Description | Price | AvailableAmount |
|---|---|---|---|---|
| 1 | Smartphone | A cutting-edge smartphone with top-tier features | 1999.99 | 146 |
| 2 | Microwave | High-performance microwave for efficient cooking | 299.00 | 83 |
| 3 | T-shirt | Comfortable and stylish t-shirt for everyday wear | 19.50 | 21 |

This shortcut created a technical debt. You knew that price and availability should not have been part of the `Products` table, but due to time pressure, you included them anyway. Together with your team, you decided to rework it in case the product was successful.

However, there is no time for that now. New requirements are emerging, and you need to extend the product with size and material information.

Size can have various values: for T-shirts, it is small (S) to extra extra large (XXL), while for microwaves or smartphones, customers are interested in dimensions like height, width, and depth.

Material is relevant when talking about smartphones (metal, glass, plastic) or T-shirts (cotton, silk, polyester), but it is less pertinent in the case of microwaves.

You add special logic to handle all these cases.

Several weeks later, your company decides to sell shoes. The size specifications for shoes differ from those for T-shirts or smartphones. In the EU, sizes might be 36, 42, or 46, while in the US, they could be 9, 10.5, or 12. Additional logic is implemented to accommodate these variations.

After a year, the `Products` table has grown to 85 columns, representing different characteristics and prices. The codebase has expanded enormously, with everything tightly coupled. There are dozens of rules to handle aspects like size correctly, and changes in one area often cause issues in multiple other areas.

Congrats! You have created another big ball of mud:

- There are extreme performance problems.
- New features and changes are released twice a year because each release costs tons of money (lots of manual testing).
- Bug fixes are overwhelming, as any change tends to introduce new bugs.

As a result, existing customers stop using the application in favor of better alternatives. New customers avoid it due to poor reviews, leading to financial losses. Despite your efforts to resolve the issues, it is too late—you have lost trust. Ultimately, this leads to the collapse of the company.

Initial success does not mean eternal glory, which is why evolving the architecture as the business grows is so important. Adapting to changing conditions is key to sustained success.

If caught early enough, the impact will not be as significant as in the case of overengineered architecture. However, the issue often only becomes obvious when performance starts to slow down dramatically or maintenance costs become very high.

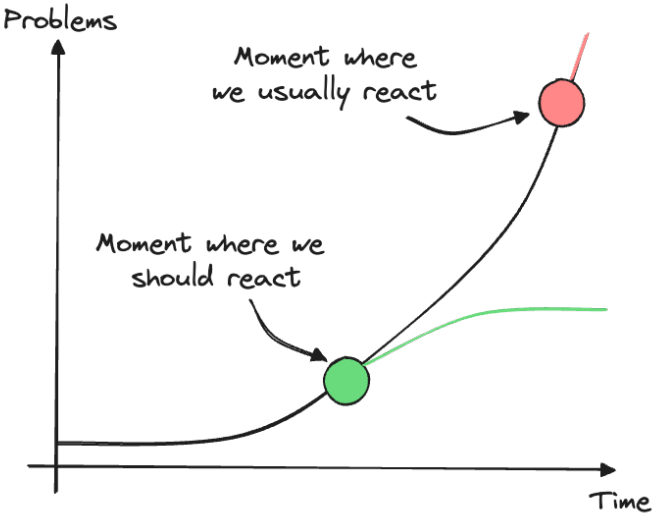


**Figure 153. Increasing problems of maintaining trivial architecture for too long**

When the issue is eventually noticed, companies often choose one of two popular solutions: starting over with a new technology (which usually is a terrible idea) or hiring a team of consultants who specialize in refactoring legacy applications (which is expensive but may be the only approach).

## Evolve together with your business

Instead of predicting the future of our application and setting up a bulletproof architecture, I would like to share an approach that allows us to adapt to current circumstances based on the application's evolution.

This approach involves continuously observing the environment around the application, which will change over time due to factors like increased usage, evolving functional requirements, shifting teams, and changes in the business structure.

To facilitate this evolution, I have outlined four steps that will help you make accurate decisions, depending on your specific case and the current state of the application.

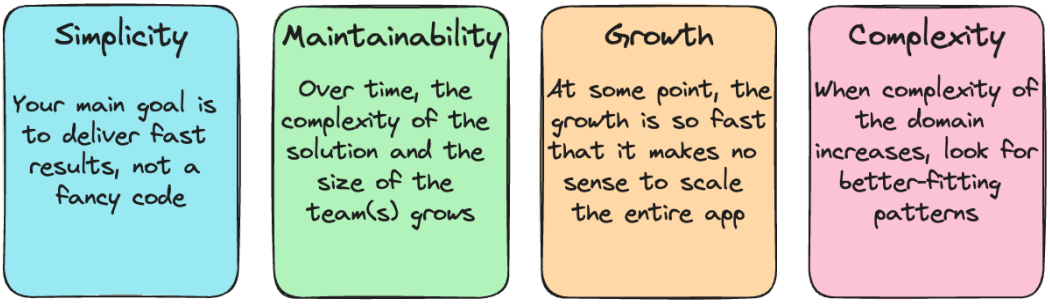


**Figure 154. Four steps of evolution**

Each step focuses on different aspects and can be viewed as a map. There is only one rule: the farther you go, the more complex the decisions you have to make.

I will practically guide you through each step of the evolution process using our case as an example. There is no one-size-fits-all solution, so use this as a model and feel free to adapt it to your own needs.

In the following sections, I will cover topics such as CQRS, transaction script, database partitioning, sharding and replicas, caching, aggregates, entities, and many others as they come along.

No more theory; let's get to work.

# First: Focus on simplicity

This step primarily applies to greenfield applications. It works great when you need to quickly deliver your product to market while laying the groundwork for future extensions without requiring complete refactoring.

A common criticism is that this step seems too naive and unlikely to work in "our case." However, we should remember that simple does not mean silly.

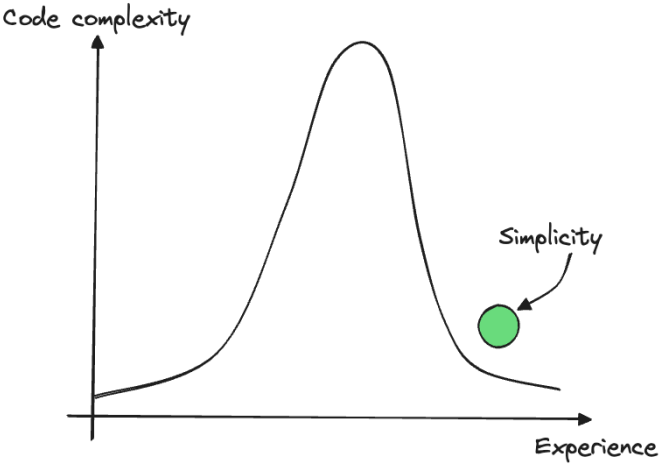


**Figure 155. Code complexity depending on experience**

The application architecture should be as simple as possible while fulfilling the requirements. Keeping this in mind will increase our chances of delivering a successful product.

Together with everyone involved in the product, we reviewed and summarized everything we had analyzed to make informed decisions about the software's