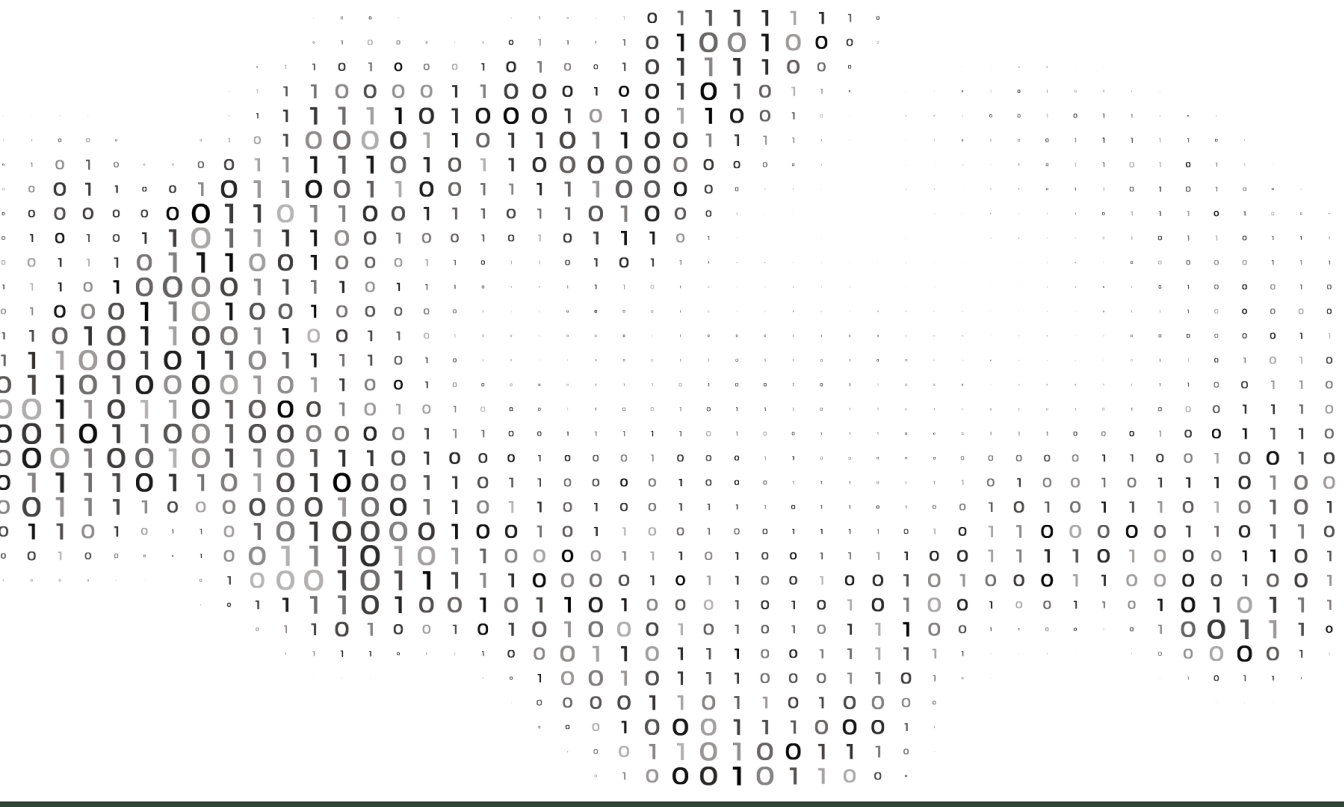


MASTER SOFTWARE ARCHITECTURE

A PRAGMATIC GUIDE

MACIEJ "MJ" JEDRZEJEWSKI

Master Software Architecture

A Pragmatic Guide

Maciej ”MJ” Jedrzejewski

This book is available at http://leanpub.com/master-software-architecture

This version was published on 2024-09-02

To everyone who has helped me become who I am

Contents

Preface

I have good and bad news for you.

The good news: Finding information about software architecture has never been easier.

The bad news: Finding valuable content has probably never been more challenging.

The amount of knowledge to absorb in the current IT world is incredible. New frameworks, libraries, and languages are introduced almost every day. Thousands of people publish new blog posts, create GitHub repositories, and share content on X or LinkedIn.

We live in a wonderful technological era! But how can one person possess all this knowledge? I think I will not surprise you—it is impossible. That is why it is so important to filter the content that surrounds you.

Let me tell you a story. I started as a software developer in 2012. Several years later, in 2016, I shifted my focus heavily towards software architecture. I attended countless conferences and meetups and was involved in mastermind groups and discussions.

Here is what I learned: Microservices, caching, data streaming, NoSQL, Kubernetes, Docker, cloud technologies, and more. These concepts have helped other companies and will benefit the one I work for too (for sure!). I still remember the excitement of discovering these technologies and knowing I had to apply everything I learned.

In the last eight years, I have made almost every mistake related to software architecture that you can think of. To name a few, I:

- applied Onion, Clean or Hexagonal architecture everywhere I could;
- assumed that people not doing test-driven development, continuous deployments or trunk-based development are crazy;
- used microservices in almost every application;
- ran all my tiny applications in Kubernetes;
- used cache everywhere.

Do you see a pattern here? I mindlessly followed each concept without deeper thinking.

Whatever I heard, I applied. I easily justified it to myself because these were always described as silver bullet solutions, and of course, I wanted to follow best practices.

This contradicts pragmatism.

Instead, it would be better to look for solutions that would help solve my problems and support the team in achieving outstanding results. Mix concepts, try various things, have fun with software architecture, and remember that every architectural decision has trade-offs.

My problem was that I lacked patterns and support from experienced mentors. I was wandering in the dark, grasping at one thing or another. Over the years, I have had to learn everything myself, and I cannot count the time I have lost.

That is why I decided to write this book. It is a practical guide where you will learn about business analysis, security vulnerabilities, architectural patterns, deployment strategies, testing, trade-offs, and more. It also shows my thought process and how I build my systems.

I want to show you the map of key areas and fundamentals of software architecture in an easy-to-understand way. However, you are responsible for setting your own path on this map. I encourage you to do so because one of the beauties of software architecture is that we all have our unique ways of dealing with it.

I hope you get a lot out of this book and that it will be worth every moment you spend reading it.

Enjoy the read!

About Me

My name is Maciej Jedrzejewski, but everyone calls me “MJ”—you can imagine how hard it is to pronounce my surname :). I am originally from Poland, but I moved to beautiful Switzerland some time ago (a sheep outside confirms!) and enjoy nature.

Computers and programming have been a part of my life for as long as I can remember. I have fixed tons of issues related to hardware and software, overclocked processors, and burned them. Staying up late to solve another mystery wasn’t unusual.

Apart from the processors, not much has changed over the years. With over 12 years of experience as a software engineer, architect, tech lead, and consultant, I still spend countless hours poring over books, tutorials, and articles. I regularly publish content on software architecture and am active in the open-source community. In 2023, together with Kamil Bączek, we launched a repository on Evolutionary Architecture[1] where we help developers avoid over-engineering. I don’t particularly appreciate over-engineering; my first rule in programming is prioritizing simplicity and clarity over complexity.

Software engineering is more than just my job. It is my life, my passion. Throughout my IT career, I have worked with and spoken to various companies. Some were large-scale and medium-sized, and others were startups. Their problems were different, but they all involved software architecture, engineering processes, or team topologies.

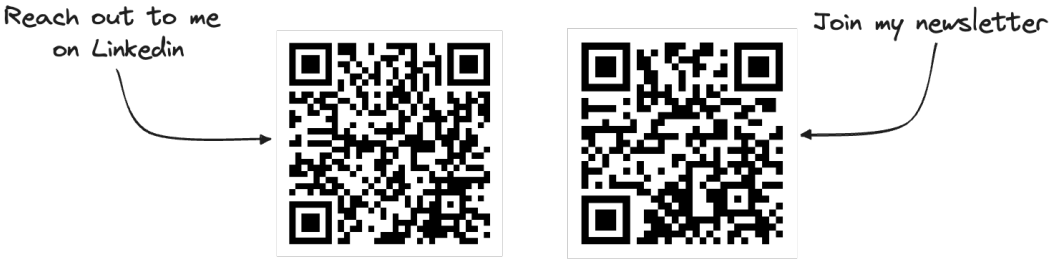
I often encountered poor application performance, too extensive testing before production, high cloud costs, and too expensive feature enhancements. However, these issues can be addressed when building a system around evolutionary and robust architecture.

[1] https://github.com/evolutionary-architecture/evolutionary-architecture-by-example

Some time ago, I decided to leave the 9-5 world and became a fractional architect. As the name suggests, this is a resource-as-a-service solution—I can help you build a successful software product, audit existing applications, or join as a sparring partner in software architecture. I also run extensive workshops[2].

In general, I help to avoid the unnecessary costs of over-engineering while maintaining quality standards.

If you have any feedback or suggestions, please reach out to me on Linkedin[3]. I also invite you to join my newsletter[4] where I write about software architecture topics every week.



[2] https://www.fractionalarchitect.io/workshops
[3] https://www.linkedin.com/in/jedrzejewski-maciej/
[4] https://newsletter.fractionalarchitect.io/

What Will You Find In This Book?

This book covers a comprehensive range of topics related to software architecture, but it cannot cover them all. Over the decades, the scope has grown to an unimaginable size. I have gathered all the key elements that have accompanied me over the past few years and described them here, ensuring you have a solid foundation to build on.

This book takes a holistic approach to software architecture, so expect discussions on topics like business analysis or documentation of architectural decisions. We will also dive into infrastructure, security, deployments, and application structure. Additionally, I will introduce you to various engineering practices that, while not directly related to architecture, influence it. My goal is to explain each topic clearly for easy understanding.

The world of software architecture is vast and ever-expanding. I am certain I will leave out many exciting topics, so I ask for your understanding, dear reader. You can read the book cover to cover, following the logical progression of the chapters. Alternatively, feel free to jump into a specific part or chapter that interests you. The choice is yours! :)

Step 1: Understand Software Architecture

In this step, you will learn about the basics of software architecture, including its key areas, such as business analysis, solution architecture, and infrastructure.

I also describe the role of software architects in modern environments. Who is a software architect? Is this a role, function, or just a way of thinking?

We will also examine architectural drivers such as functional requirements, business and technical constraints, and quality attributes. You will learn how they impact your architectural decisions.

Next, I describe how pragmatism and holism can influence your architectural decisions.

At the end of this chapter, you will have a look at other experts' views on software architecture success.

Step 2: Discover Your Business Domain

Next, we will focus on the most critical part of any system design—the business domain.

What are the key business capabilities? What are the constraints? How do processes interact? These are just a few of the questions you will find answers to.

We will focus on high-level and deep-level workshops for a practical case study that illustrates the real-world application that will accompany us throughout the book. You will learn to discover and analyze crucial processes using Event Storming and Domain Storytelling.

Then, you will use the workshop outcomes to experience strategic domain-driven design. This will enable you to discover subdomains and define bounded contexts using different heuristics. Finally, you will learn how to create a context map based on distilled bounded contexts.

Step 3: Recognize the Environment Around You

Step three focuses on understanding your work environment. How are decisions made? What do you want to build? What are the budget limitations? How do the expectations differ from the perspective of various involved groups, such as stakeholders or development teams?

You will also learn how to assess your team skills using a competency matrix and plan the infrastructure based on various team setups. What should we do if there is a separate and experienced infrastructure team? What if there is a team that needs more expertise? What if we have to handle infrastructure alone?

Another critical point of this step is to discover crucial numbers like total and daily active users, number of requests, storage capacity, and how to calculate the availability defined in the Service Level Agreement (SLA).

Finally, we will determine how to apply the new knowledge to the previously defined case study.

Step 4: Choose Deployment Strategy

In this step, we slowly navigate towards the technical aspects of software architecture. Here, you will find out the strategies of single and multiple deployment units and their key characteristics.

We will focus on modular monolith and microservices, which represent the two most famous strategies. You will learn about communication using commands, queries, and events and how to communicate using abstraction layers, HTTP, in-memory queues, and external components like message brokers. Additionally, you will discover outbox and inbox patterns and find out why you should consider using a dead letter queue.

How do you design your database, deploy changes, or scale the application? This is where you will find the answers to these questions.

By the end, you will be equipped with the knowledge to explore these topics further.

Step 5: Define Release Strategy

Step five focuses on various ways to deliver new versions of your application to customers. In addition to strategies like basic, blue-green, canary, and rolling deployments, you will learn how to leverage the power of continuous delivery and continuous deployment.

All of this is supported by the adoption of engineering practices like swarming, pair and mob programming, feature flags, short-living branches, and trunk-based development. Additionally, you will learn how to organize post-mortem meetings to further analyze and prevent problems that occurred in production.

Finally, you will discover a pragmatic approach to deciding on the release strategy based on the requirements and the environment around you.

Step 6: Focus On Testing

This step explains different ways to test your software to ensure it works well. I focus on the most relevant topics from a software architect's perspective.

First, I describe three testing concepts: the pyramid of tests, the inverted pyramid, and the diamond. You will learn when to use each and how they can support you.

Next, I cover penetration testing, performance testing, and load testing.

By the end of this chapter, you will know different ways of testing and how to use them to make your software reliable.

Step 7: Evolve Your Architecture

This step highlights that architecture is not set in stone—it evolves. You will focus on four key steps of the evolution:

1. Simplicity - Learn how to approach software architecture in the simplest possible way.
2. Maintainability - Learn how to maintain your software architecture over time.
3. Growth - Learn what you can do if your application gains much traction and generates heavy traffic.
4. Complexity - Learn how to tackle code that becomes too complex to maintain and discover patterns that can help you.

I describe helpful concepts at each step, such as CQRS, database replicas and sharding, tactical Domain-Driven Design, and other relevant topics.

My goal is to ensure that you gain the knowledge necessary to maintain the robustness of your application and make informed decisions as it evolves alongside the business.

Step 8: Don’t Forget About Security

This is the final step. Here, you will learn about security vulnerabilities often encountered while working with greenfield and legacy applications.

These include Insecure Direct Object References (IDOR), Supply Chain Attacks, SQL Injection, Cross-Site Scripting (XSS), DDoS, and more. These attacks occur for various reasons. How do we defend against them, and what common misconceptions lead to vulnerabilities?

Furthermore, you will learn why encoding, encryption, and hashing are often confused. This knowledge will help you select the appropriate mechanism for your case.

Extra 1: Other Engineering Practices

This section describes important engineering practices that can help you along the way. As I could not find a suitable place for them in the main body of the book, I have decided to include them at the end.

You will learn about metrics, and developer carousels. You will also learn how to effectively use vertical slices in product design and how to manage technical debt.

Extra 2: Architecture Exercises

To master software architecture, you need to put theory into practice. While reading about concepts is important, hands-on experience solidifies your understanding and sharpens your skills.

I have prepared several exercises to challenge you to apply what you have learned. Each case is presented in a format similar to the one used in this book. All exercises are created to simulate real-world scenarios.

Working through these cases will validate your understanding of the concepts covered throughout this book. It will also highlight areas where you might need further study, directing your continued learning journey.

Who Should Read This Book?

This book is perfect for all software engineers, regardless of your experience—software developers, architects, tech leads, and even project managers who want to understand the nooks and crannies of architecture. The only thing you need is an interest in software architecture. If you enjoy thinking about how to design efficient software systems, this book is for you.

Beginners will find a solid foundation here. If you are new to software engineering, this book will help you understand the architectural concepts and give you a good

start. For experienced engineers, there are advanced tips and fresh perspectives that can further enhance your skills.

Finally, this book is excellent for anyone who loves to learn and improve. The world of software architecture is constantly changing, and we must adapt. Understanding the fundamentals increases your chances of making better decisions and creating software that stands the test of time.

Whether you are a beginner or an experienced architect, this book will equip you with the fundamental knowledge to build resilient and scalable applications.

Feedback

I would love to know what you think about this book. Did you enjoy it? Was it helpful? Or do you have any ideas to make it better?

Your feedback is very important to me. It doesn't matter if your review is good or bad - I want to hear it all!

Please share your honest opinion on Goodreads[5]. This will help other readers decide if the book is right for them.

Feel free to share your thoughts on social media as well - tag me, and let's have a conversation about software architecture!

Thank you for reading and for taking the time to provide feedback.

You are awesome!

[5] https://www.goodreads.com/book/show/216954084-master-software-architecture

Acknowledgments

We are who we are because of the people we meet along the way.

First and foremost, my deepest gratitude goes to my family. Special thanks to my wife for her unwavering support. She understands my passion for software architecture and programming, even when it takes up a lot of my time. To my parents, thank you for shaping the person I am today. You opened my eyes to the wonders of the world and taught me the values that guide my life. And to my grandmother, who taught me how to read; every book I read is a tribute to the bond we shared and the knowledge she passed on.

Next, I want to thank my teachers, especially Dorota Nieznanowska-Gawlik, who taught me to respect others, be resilient, and believe in myself. I also want to thank my direct mentors who accompanied me at the beginning of my career and throughout the years: Piotr Piechura, Stefan Sieber, and Othmar Oeler. Your guidance helped me developed my programming, architecture, and product development skills.

I haven't had the pleasure of working directly with these programmers, but they have influenced me greatly: Scott Hanselman, Jon Skeet, and Michał Franc. I remember reading your blogs and thinking I would be like you one day.

I would like to thank my colleagues from the teams I have worked with, especially Michał Dziurowski, Krzysztof Cichopek, and Adrian Rusznica, for all the discussions, arguments, and great times we had building applications together.

Finally, I would like to thank my beta readers and reviewers—Kamil Kiełbasa, José Roberto Araújo, Jose Luis Latorre, Urs Enzler and Martin Dilger—for their support while I was writing this book. It would not have been possible without you and your time.

Thank you!

STEP 1: Understand Software Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Key areas of software architecture

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Architectural drivers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Importance of software architecture

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

How to stay pragmatic?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

How to stay holistic?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Who is a software architect?

Over time, the architect's role has evolved from being someone who sat in an ivory tower, distant from programmers, to one who works closely with them. They get involved in coding and architectural decisions alongside other developers. There is no more hierarchy, allowing everyone to share responsibility for the application architecture.

Another observation is that architects are drifting away from their traditional roles toward mentoring, coaching, and being partners in discussions and actions. They help teams achieve success, get their hands dirty with code, and experience the consequences of decisions made together with other developers.

Their experience enables them to identify mistakes to avoid and explain the potential consequences. I like the comparison to squeezing a lemon: an architect's experience is the lemon, and the team uses it to its fullest extent until all the juices are squeezed out.

I must admit that I love the way the role has evolved.

Software architects are the glue between different parts of the business—developers, testers, managers, and stakeholders. They can explain complex concepts simply to both technical and non-technical audiences. Regardless of whom they are addressing, they can say "stop." They serve as advisors and conversation partners.

One of the most important aspects of being an architect is understanding the business domain in which you work. This involves knowing its operations, processes, actors, and their interactions. The more diverse sectors you familiarize yourself with, the easier it becomes to learn new ones, as problems like to repeat.

We live in a world that is changing incredibly fast. The plethora of concepts, technologies, languages, frameworks, and libraries makes it impossible to be an expert in everything. Therefore, architects often function as generalists. They have an awareness of:

- Business capabilities
- Infrastructure
- Solution architecture
- Security
- Soft skills

However, they can only be experts in specific areas. For instance, they may understand infrastructure and how to set up pipelines, but an infrastructure specialist could optimize this further. They can differentiate between testing strategies and know which ones to apply, but experts in testing may execute them more effectively.

The rule of thumb: the broader the knowledge, the easier it is to understand the environment and make wise decisions.

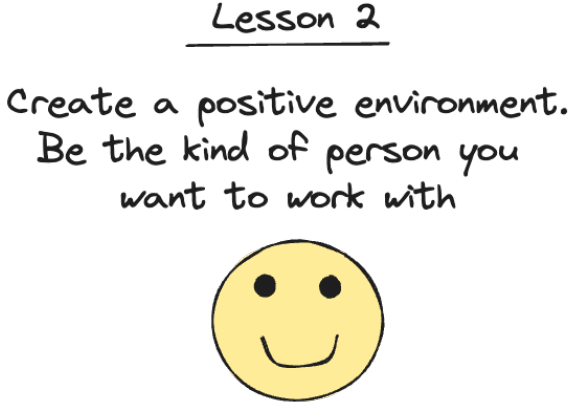
If you take a closer look, you will see that an architect is more of a mindset or a function within a team than a specific role.

In one of my favorite books, The Software Architect Elevator[1], Gregor Hohpe brilliantly illustrates the software architect's role using a building metaphor. He describes architects as riding an elevator between the "engine room" (developers) and the "board room" (stakeholders), passing, depending on the organization, more or less floors. This way architects connect technical aspects with business strategy.

[1] https://www.goodreads.com/book/show/49828197-the-software-architect-elevator

Remember, no one was born a great architect. It requires a lot of experience, ideally in different organizations and systems. The more you are exposed to, the better it is for you.

Don't hurry too much; try to stop and look at the problem from different angles. Help others succeed and be someone they enjoy working with. Focus on enabling your team and creating a positive work environment without a blame culture.



What are the key drivers of a successful software architecture?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Recap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

STEP 2: Discover Your Business Domain

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

The case

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

How do I start?

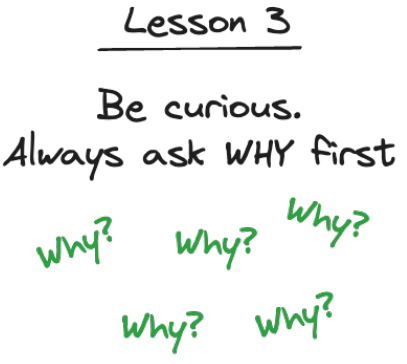
Before starting any official meeting, I always advise having coffee with everyone you will work with. Discuss the company, processes, and the applications they develop. You can ask if they know anything about the application you will be responsible for. It is amazing how much you can learn over coffee and chit-chat.

The next step is to organize a meeting with domain experts and stakeholders, or ensure you are invited to attend one in the first two weeks. By now, you have already gained some knowledge from unofficial talks. Now, it is time to deepen that understanding.

When joining such a meeting, let the others speak first. Be an active listener. Note as much as possible; it will help you in the upcoming weeks. Don't hesitate to ask questions. One of my favorite approaches is to start by asking *Why?* followed by *What?* and concluding with *How?* These questions require deeper thinking before answering, unlike closed questions, where the answer is either yes or no. Here are some examples:

- **Why** do we want to build this product?
- Why did we decide on the SaaS model?
- Why did we decide to build the mobile app for it?
- **What** problem are we trying to solve?
- What is our goal?
- What is the customer's vision?
- What are the alternatives?
- **How** do we measure product success?
- How big is the market demand?
- How can the problem be solved without our software?

You can use the *5 Whys* method, which involves asking "why" five times to find the root cause of a problem. Each question dives one level deeper. If you are interested, you can read more about it here[1].



Such meetings usually take around two hours, and the knowledge you gather there can be used for various purposes. Most often, my notes take around 2-3 A4 pages.

The outcome of the meeting can look like this:

[1] https://www.lean.org/lexicon-terms/5-whys/

> We want to build a SaaS application to sell subscriptions to private medical clinics. It will cover everything you can think of, from appointment scheduling and treatment management to drug prescriptions. Clinics want to keep and manage the medical records of each patient. The central vision of the clinics we spoke with is to expand their reach, attract new customers, and automate processes. Three clinics are already interested in this solution, and the other two are considering it. They plan to onboard around 1,000-1,500 patients in the first months. We have yet to determine how much it will cost and how exactly we will sell it. There are some plans, but a lot must be agreed upon.

When you hear all this, you may feel overwhelmed (the above is an abbreviated version for book purposes). But I have to tell you that you are lucky. I have experienced situations where the development team was simply instructed to replicate another application. Yes, this was the entire description of what we had to build. So it is all right; having more information than less is way better.

Why do I feel so happy when I get so much information that scratches the surface but not the details? Well, the time for details will come during the workshops that you will organize later. For now, we focus on analyzing the information you have already collected. You can do this by highlighting key facts with a highlighter or simply making the font bold.

> We want to build a **SaaS application to sell subscriptions to private medical clinics**. It will cover everything you can think of, from **appointment scheduling and treatment management to drug prescriptions**. Clinics want to keep and **manage the medical records of each patient**. The central vision of the clinics we spoke with is to expand their reach, attract new customers, and automate processes. **Three clinics** are already interested in this solution, and the **other two** are considering it. They plan to onboard around **1,000-1,500 patients** in the first months. We have yet to determine how much it will cost and how exactly we will sell it. There are some plans, but a lot must be agreed upon.

Look how much valuable information can be found in such a short note:

- **SaaS application to sell subscriptions to private medical clinics**. Our company wants to monetize the access with a subscription, as in most SaaS software. It is going to be built for private medical clinics. The next step is to check how they work and the regulations in the countries where they operate.
- **(...) appointment scheduling and treatment management to drug prescriptions**. **(...) manage the medical records of each patient**. Various areas to cover. That is fine. The next step will be to find the key process.
- **Three clinics**, **(...) other two**. That is excellent information. You don't build an application based on your feelings but on real interest. It is important. It very often turns out that you develop an application for a long time only to find that it does not arouse any interest.
- **1,000-1,500 patients**. This information is helpful to know what scale the application has to support. In this case, the scale is small.

This is a common approach to start discovering any business domain. The next step is to organize high-level workshops. How should you handle this?

Organize high-level workshops

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Event Storming

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Domain Storytelling

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Organize in-depth workshops

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Event Storming

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Domain Storytelling

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Combine both techniques

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Use knowledge from workshops

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Subdomains

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Bounded contexts

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Evolution

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Definition

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Context map

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Upstream-downstream

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Description

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Recap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

STEP 3: Recognize the Environment Around You

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

What do you want to build?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

How are product decisions made?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

What are the strengths and weaknesses of your development team?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Are there any other development teams?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Are there any infrastructure teams?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Separate infrastructure team(s) with deep expertise

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Separate infrastructure team(s) with limited expertise

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

No dedicated infrastructure team

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

What does existing infrastructure look like?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

What scale and numbers will the application serve?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

SLA

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Users & requests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Storage

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

What are the budget limitations?

An integral element that we engineers need to remember is the budget within which we operate. It might be enormous or minimal. Fixed or flexible.

But there is always a budget.

As software architects, we need to observe it, but we don't have to think about it all the time. If you exceed the monthly budget by a dozen dollars, it is unlikely that something terrible will happen. But it would look different if your logs generated costs of $500,000 while it was expected to be around $10,000.

That is why making informed decisions with your team is crucial. Compare at least two solutions and analyze them carefully. If the budget is tight, you will plan the architecture differently than if it is flexible, where you can maneuver a bit.

- **Tight budget**. There is no time to learn anything new; together with your team, you must look at your current skills and select the approach based on them. It is similar to a sports team: when a coach joins a mid-tier club, they adapt tactics to suit the players they have.

- **Flexible budget**. There is a space to learn something new, spend time on spikes, compare tools, libraries, etc. However, this does not mean you should choose all the fancy technologies you have ever heard of. I know cases where the team gold-plated every detail of the application with cutting-edge components and continuous deployments, only to run out of budget along the way. The app was never released. In the worst-case scenario, one company went bankrupt.

Budgets will usually look different, depending on the company type:

- **Your own business without external financing**. You will try to cut all costs to a minimum. Every extra few dozen dollars will weigh you down. I recommend that every programmer try it because it changes the point of view. When we work for someone, it usually doesn't matter so much to us.
- **Startup**. Depending on the financing type, there will be more or less flexibility on the budget but much pressure on product delivery. This means that you cannot invest too much time in learning new things, especially before releasing the minimum viable product (MVP).
- **Small and mid-size companies**. Budgets might be higher than in previous cases, but business looks at every cent spent. The rule is simple: you need to earn first, then you can spend part of it.
- **Large-scale organizations**. They have the highest budgets and expenses. You usually plug your application into existing infrastructure so that the initial costs might be smaller. These costs may go unnoticed up to a certain point. However, this does not mean that you should not watch out for them.

The rule of thumb: Always treat it like your own business, minimize costs where possible, and continue expanding your knowledge.

When looking at costs, they are primarily dependent on:

- **Running costs**. It does not matter where you run your application, it always generates monthly costs. The more components you add, the higher costs will be. Start with the minimum infrastructure necessary.
- **Cost as a consequence of the choice**. If you decide to use a language, framework, or library that you do not know yet, you have to learn it first. The more new elements you add to your application, the more there is to learn, which can slow down development. As a result, costs will be increased.

Cloud providers might offer a significant infrastructure budget for free in the first year. However, adding too many components and using too many services can lead to enormous costs in the second year. As a result, you might spend 10 times more than you earn.

What always works great for me is to look at running costs in percentages rather than focusing on specific amounts. As you have probably noticed, these costs can vary depending on the scale of the business. So, instead of looking at something that costs $100, I am checking what percentage of the current costs this will absorb. With current costs of:

- **$100.** 100% more than I pay today, so I will probably be contemplating this for a few days.
- **$1,000.** 10% of the current costs. Not as much as in the previous case, but still a lot.
- **$100,000.** 0.1% of the current costs. Minimal impact on the budget, let's add it.

Another way to consider costs is to calculate how many software licenses you would need to sell to cover the expense of additional components.

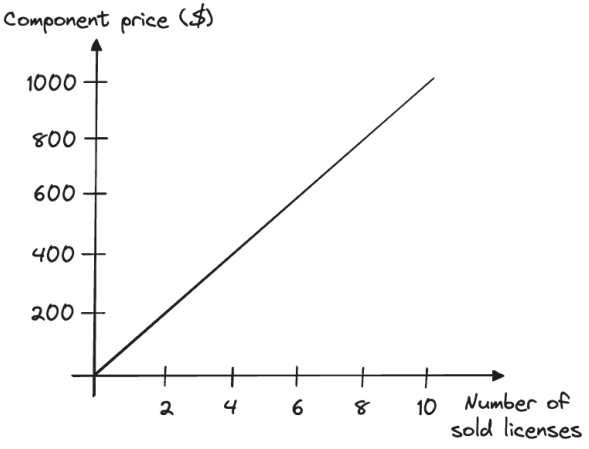


Figure 1. Example of the number of licenses that have to be sold to cover additional component

So, if a component costs $1,000, and each software license is sold for $100, your company would need to sell 10 additional licenses to cover the cost.

You must always consider all of this. In some companies, exceeding the budget by $10,000 might not be a big deal. However, there are also companies with minimal budgets, and you will need to carefully evaluate whether using specific components makes sense.

Answer this question based on our case: What are the budget limitations?

When is the deadline?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

What are the expectations?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Our case

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Recap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

STEP 4: Choose Deployment Strategy

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Single deployment unit

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Multiple deployment units

Multiple deployment units divide your application into smaller and independent units, each running in a separate process. Each unit should encapsulate a closely related set of functionalities (high cohesion):

- **Unit 1**. Responsible for all invoicing operations, including preparation and sending to customers.
- **Unit 2**. Responsible for appointment scheduling and sending reminders for appointments.
- **Unit 3**. Responsible for selecting drugs, preparing prescriptions, and sending copies to an external, public system.

Together, they build the application. Each unit can be deployed separately, as each runs in its own process.

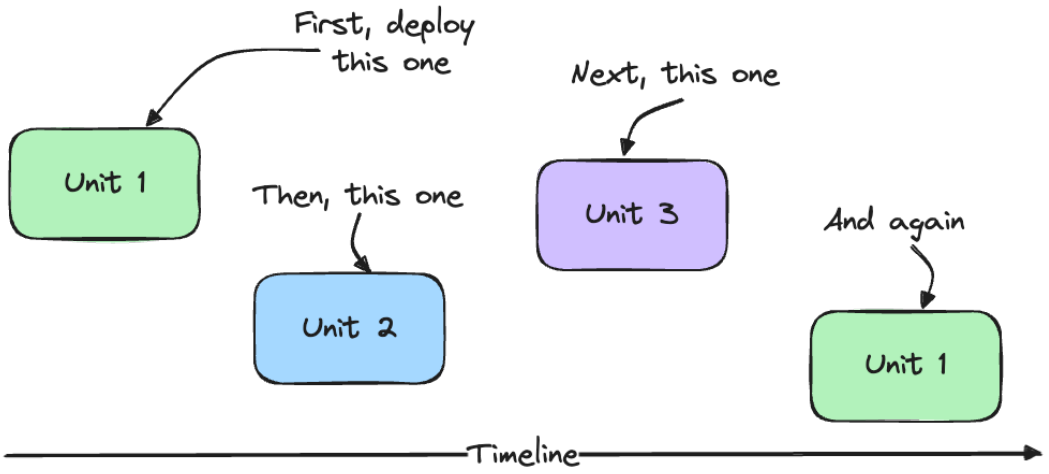


Figure 2. Deploying multiple units to production

If the application needs to be scaled, this strategy allows you to scale units independently. This is especially useful when one unit is used more often than others. You deploy more instances of it while the rest of the units stay untouched.

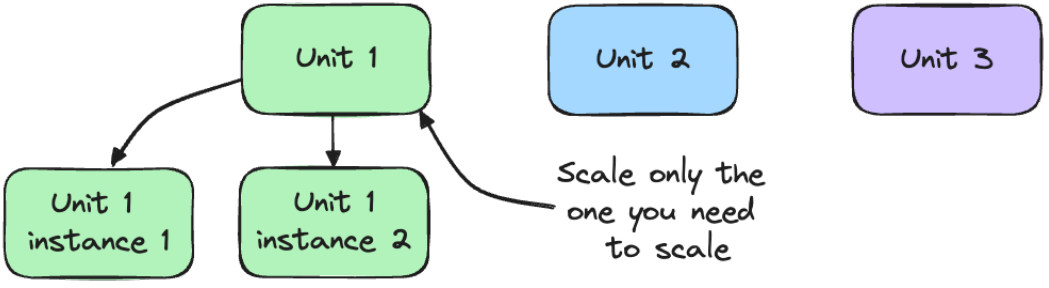


Figure 3. Flexibility to scale a single unit

When one of the units fails, let's say *Unit 1* (responsible for invoicing in the application), the rest continue to work. Thanks to that, customers can continue to schedule their appointments (*Unit 2*), and doctors can prescribe drugs for patients (*Unit 3*).

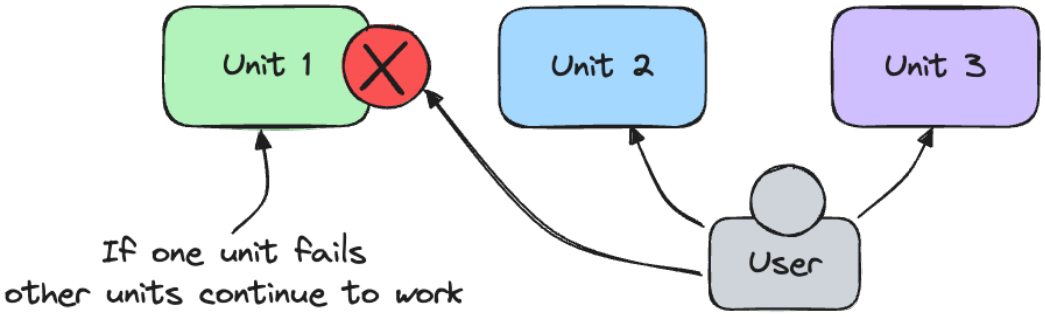


Figure 4. If one unit fails, the others keep working

One of the benefits of using this strategy is the freedom to select any technology or platform you want per unit. For instance, you can opt for Node.js for two units, .NET for three, and JRE for others.

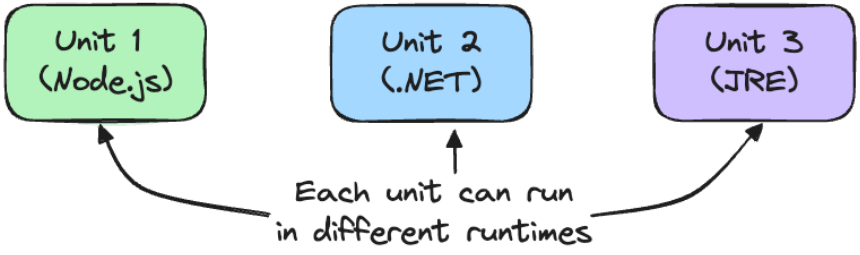


Figure 5. Technology flexibility per unit

The multiple deployment units strategy is a considerable choice. As units are isolated, it gives you flexibility in scaling, deployments, and implementation.

However, there are also disadvantages. As units run in separate processes, you need to tackle problems related to network communication (high potential for errors) and latency, which will be higher than in applications running in a single process.

Also, having multiple entry points into your system increases the risk of security vulnerabilities, but you can reduce this risk by using an API gateway.

Finally, testing might become highly complex, with numerous deployment units running different versions. Imagine having 100 deployment units, each with at least two versions; the number of possible combinations for the testing environment would be enormous!

Pros	Cons
Independent units	Difficult to design properly
Independent scaling	Vulnerability to network errors
Independent deployments	High operational costs
Fast deployment time	Latency
No single point of failure	More entry points = higher security risk
Flexibility in technology selection	Complex testing

Communication

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Commands & queries

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Events

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Strategies for message delivery and processing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

At most once delivery

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

At least once delivery - Outbox pattern

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Exactly once processing - Inbox pattern

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Combination of outbox & inbox patterns

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Dead letter queue

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Modular monolith

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Module boundaries

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Deploying changes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Scaling

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Database

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Communication from the outside

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Communication between modules

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

When to decide on it?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Further education

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Microservices

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Microservice boundaries

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Deploying changes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Scaling

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Database

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Communication from the outside

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Communication between microservices

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

When to decide on it?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Further education

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Our case

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Recap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

STEP 5: Define Release Strategy

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Release versus deployment

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Deployment strategies

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Basic

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Blue-Green

This strategy solves the main problem of the basic one—there is no downtime during the version update.

In short, a current version of the application runs in production. At the same time, we prepare the new version on another instance and execute verification checks. If everything works fine, we switch the entire traffic to it. The old version stays there in idle as a backup.

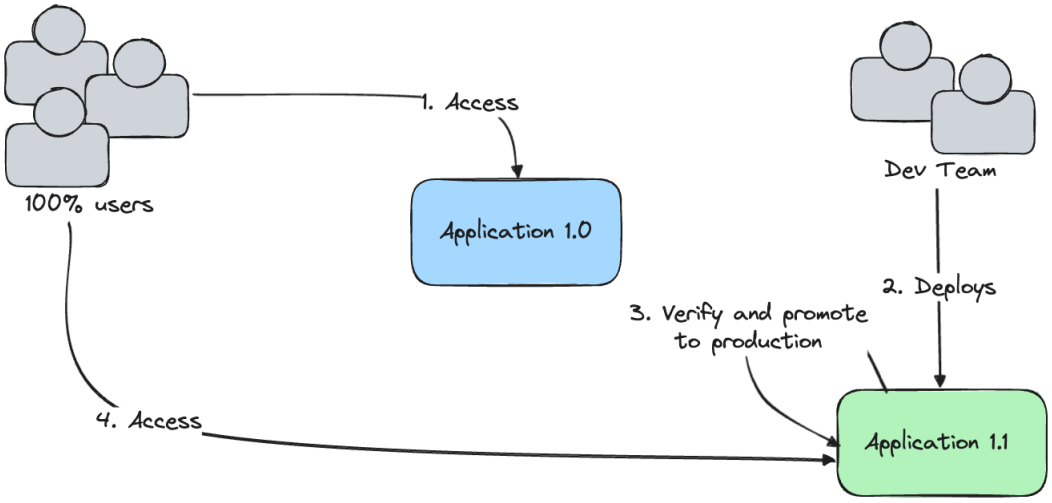


Figure 6. Overview of blue-green deployment

All users access the current version of the application. In the above diagram, the blue environment represents the production version. Next, the development team triggers the deployment of the new version. It can be triggered automatically on commit push or manually.

The new version is deployed to the green environment. Here, a series of verification checks are conducted to ensure the latest version is functioning correctly and meets the required standards. Once these checks pass successfully, all traffic is redirected to the green environment. It means that the green environment replaces the blue one in production. Next time, the blue will replace the green, and so on.

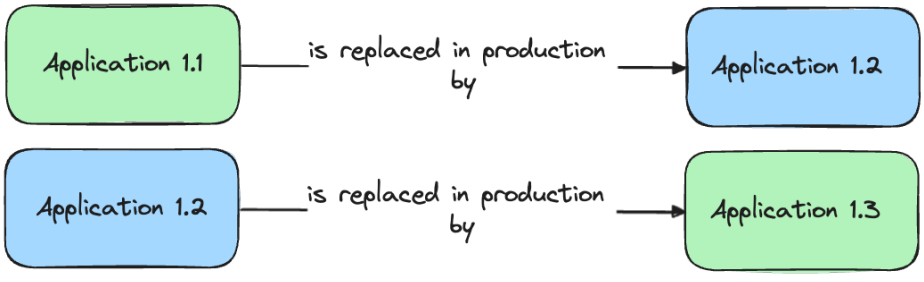


Figure 7. Blue and green environments replace each other in production

Verification checks in the new environment can be done manually, such as testing specific features or functionalities by a tester, or automatically. I recommend automating as much as possible, but it is not required in this strategy.

The cool thing is that popular cloud providers support the blue-green strategy in their platform-as-a-service offering, and it is simple to set up. You can configure it in Azure App Services or AWS Elastic Beanstalk. However, despite reducing (or eliminating) the downtime risk compared to the previous strategy, we still run into the other problem. After a successful update, all traffic is redirected to the new version. All new features and changes will be immediately available to everyone. Fortunately, there is a concept that helps with this.

Feature flags (also known as feature toggles) allow you to activate or deactivate features using configuration without redeploying the entire application so that you can separate the frequency of code deployments from the release of features. Features can be rolled out gradually to subsets of users, enabling testing in production-like conditions and gathering feedback before a full-scale release. The risk is reduced because each feature can be turned off instantly, which can help with severe problems like out-of-memory or stack-overflow exceptions—you do not need to roll back the entire application. However, managing feature flags requires regular maintenance, including cleaning them up when they are no longer needed.

The blue-green strategy can significantly improve the application's deployment process. Its key advantage lies in its ability to minimize the risk of downtime, as there are always two environments running in parallel. Rollback is, in most cases, a smooth and straightforward process, especially when there are no database changes involved.

However, it is important to note that the blue-green strategy can pose some challenges when database changes are involved. Ensuring that these changes are compatible with both versions requires a significant amount of effort and careful planning. More detailed information on this can be found in the compatibility chapter.

Finally, this strategy requires a more complex infrastructure setup than the basic one. It includes setting up and managing two separate environments, as well as

handling the traffic switching process. However, the benefits of reduced downtime and smoother deployments often outweigh the additional complexity.

Canary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Rolling

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Compatibility with previous versions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Scenario 1

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Scenario 2

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Scenario 3

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Fully automated releases - Continuous deployment

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

The change

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Development experience

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Swarming

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Pair & Mob programming

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Trunk-based development

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Example flow

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Semi-automated releases - Continuous delivery

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Pragmatic approach

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Short-living branches

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

All hands on deck

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Post mortem

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Our case

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Recap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

STEP 6: Focus On Testing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Traditional testing

Traditional testing, also known as shift-right testing, is a practice where extensive testing is done at the end of the development process. It is called "shift-right" because it is done on the right side (end) of the process. Testing is treated as a separate step after the developers have finished their work, and is usually handled by a separate team of testers.

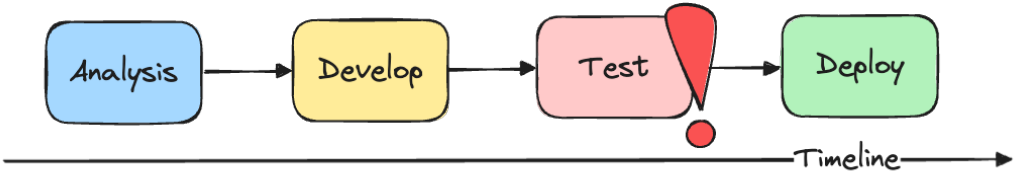


Figure 8. Testing at the end of the development process

Testing software at the end of development has major drawbacks:

1. **Last-minute bug surprises**. If you discover significant bugs late in the development process, you must perform extensive code rewrites or completely redesign the part of the application. First, it takes time. Second, it costs a lot of money. Finally, it can compromise the integrity of the entire application. It is like building a house and realizing at the very end that the foundation is cracked.

2. **High costs**. When changes are needed in later stages, the costs pile up quickly. Consider this scenario: the development team spends two weeks building a feature. After the testing phase, they realize it needs to be completely redone. You have now used up two weeks of development time plus the time spent testing, and you still need to rebuild the feature from scratch.
3. **Time pressure**. Because testing takes place close to project deadlines, there is much pressure to resolve problems quickly. This rush will likely lead to hasty fixes that may introduce new bugs or miss less obvious but critical issues.
4. **Missed opportunities**. Many issues or architectural weaknesses could be identified and addressed much earlier if testing were integrated throughout the entire development process.
5. **Limited scope for improvement**. By the time testing begins, major design decisions have been made, limiting the scope for improvements or alternative solutions that could have been implemented if problems had been identified earlier.

Cost of fixing bugs

Fixing software bugs gets more expensive the later you find them. If you catch a problem early, such as during planning or design, it is usually cheap to fix. But if you find it during acceptance testing or after release, the cost skyrockets. It is not just a matter of paying developers to fix it—late stage bugs can delay projects, cause you to miss market opportunities, and damage your product's reputation. That is why more and more teams are taking a shift-left approach by testing earlier and more often. This helps catch bugs earlier, saving time and money in the long run.

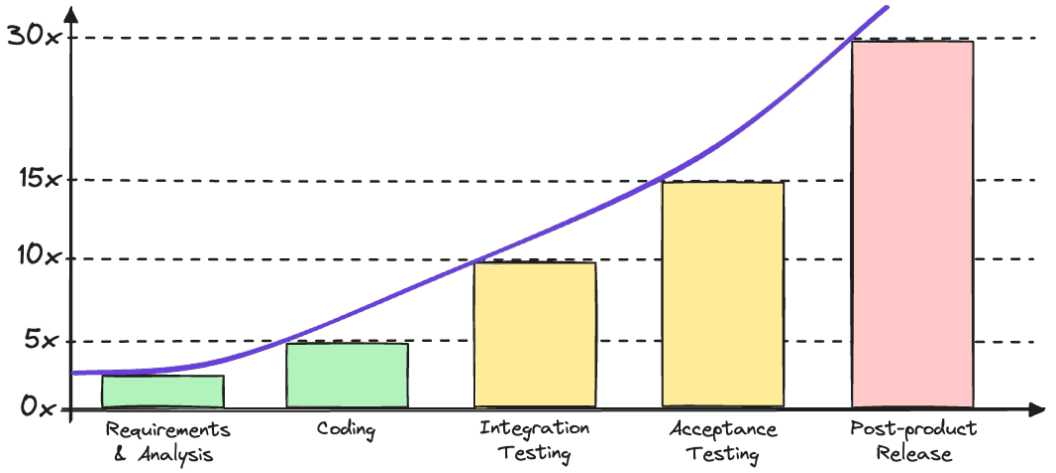


Figure 9. Relative cost of fixing defects at different stages of software development

The figure above which is based on this NIST report (table 5-1)[1], shows the relationship between the cost of fixing a bug and the stage at which it is discovered. As you can see, the later you find it, the higher the cost. That's why it is so important to catch bugs as early as possible.

Shift-left testing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Shift-left vs. traditional testing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

[1] https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf

Testing pyramid and its variations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Pyramid

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Inverted pyramid

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Diamond

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

From my diary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Performance

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Load testing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Stress testing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Penetration testing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Our case

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Recap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

STEP 7: Evolve Your Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

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.

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

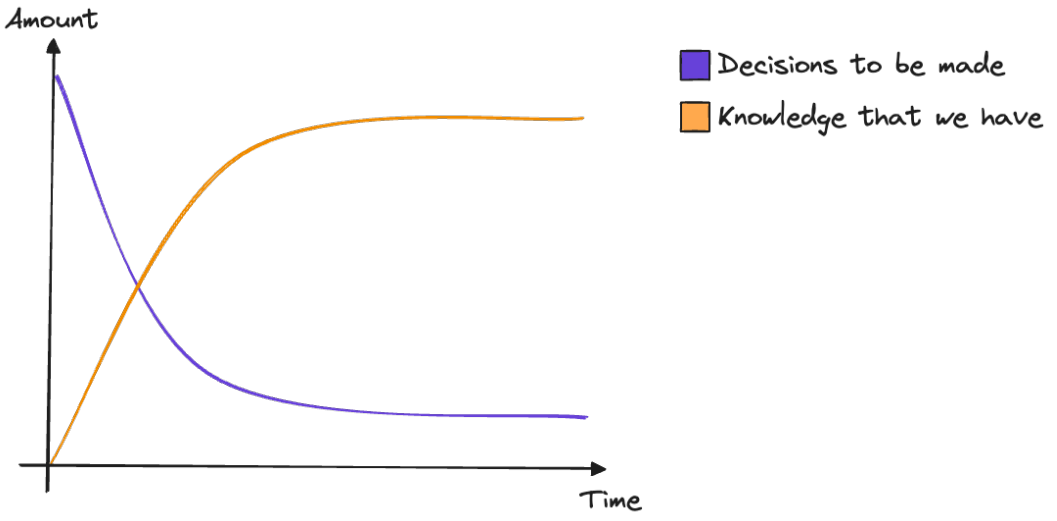


Figure 10. Relation between the amount of knowledge and the number of decisions to be made

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.

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.

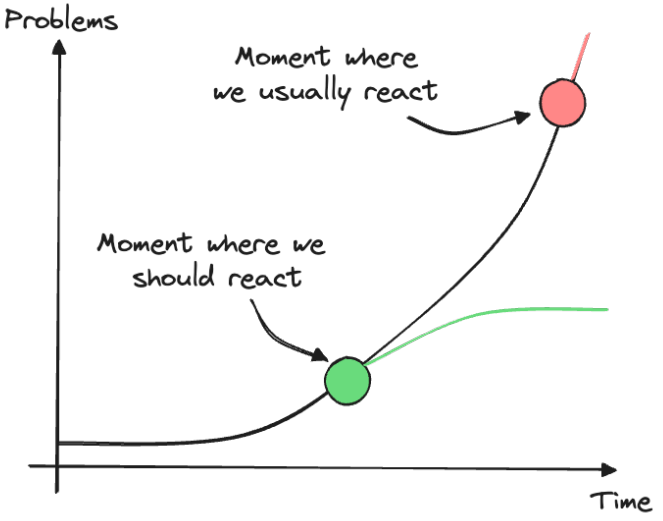


Figure 11. Increasing problems of maintaining trivial architecture for too long

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.

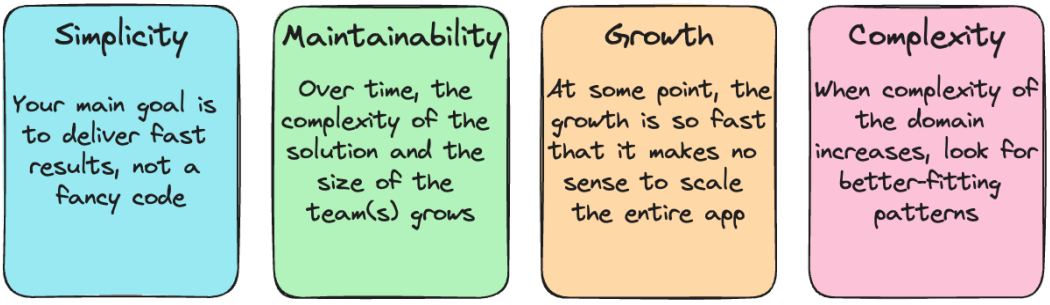


Figure 12. Four steps of evolution

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 content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Requirements

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Main considerations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Decisions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Code structure

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Behavioral entities

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

API endpoints

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

API versioning

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Workflow

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Architecture tests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Database

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Communication

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Architecture Decision Log

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Result

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Second: Focus on maintainability

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

What changed?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Maintainability: Problems to address

Code structure

Multiple projects

Transaction Script

CQRS

Team structure

Communication

Result

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Third: Focus on growth

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

What changed?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Growth: Problems to address

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Scaling

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Vertical versus horizontal

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

API

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Database

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Review indexes and queries

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Read replicas

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Partitioning

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Sharding

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Cache

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Microservice extraction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Communication

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Result

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Fourth: Focus on complexity

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

What changed?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Complexity: Problems to address

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Redesign the module

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Domain Model

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Value objects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Entities

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Aggregates

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Domain events

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Result

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Recap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

STEP 8: Don't Forget About Security

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Insecure Direct Object References (IDOR)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Exposure to DoS and DDoS attacks

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Unnecessary public endpoints

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Full path as endpoint parameter to download files

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Lack of encryption on sensitive information

Imagine you have built an application for medical clinics that contains sensitive information such as treatment details, patient medical records, and addresses.

During implementation, data masking was not considered. Since the endpoints are appropriately secured, you assumed no one could access the database.

Unfortunately, an attacker exploited a security hole in the hosting server and stole the database. Now, they can read all the patient data, as it is exposed in plain text.

Patient	Disease	Diagnosis Date
John Doe	Spinal injury	2023-11-09T13:17:01
Anna Smith	Lung cancer	2024-06-18T11:15:04
Mark Reacher	Arm fracture	2024-09-12T09:06:47

Figure 13. Sensitive data stored as plain text in a database

The attacker could sell this data or attempt to blackmail you by threatening to reveal it. If this data were to leak, it would have severe consequences for your company.

However, it would be impossible to read this data if it had been masked.

Patient	Disease	Diagnosis Date
92B1D802FA780D9019A4E9CB312699FECA9ED17B12B21CC8C33969DC750A90F5	B76F596DBB4AA9845A444A83CB8E9AD54CCCFD9B13F20BF5161C9BFA2F82C4FE0	2023-11-09T13:17:01
3633D12CA8294AD27F197536F871786758 15CC92451A6CF89B35F757289396E63	085FCDAD8DA91086F43E75C5E3294BB0C5229F286822C3AE47F7ABD82FBE6FD0	2024-06-18T11:15:04
72DB93E175ED3925AAEC3ED42F80DA107CF6623EF3C523E6495610F2F3CD6219	BFF85BFA02A690613945 0E744E3405332015600B65923B758F88EF2097AE5AA8	2024-09-12T09:06:47

Figure 14. Sensitive data stored as masked text in a database

Why does this happen? We often overlook the importance of protecting sensitive data. When we do think about it, our focus is usually on protecting passwords. We tend to pay less attention to protecting personal information such as names (which can reveal an individual's identity), as well as sensitive data like medical history, credit card numbers, and financial transactions.

How can you protect sensitive data? One effective method is encryption. Encryption converts plain text into a coded format known as cipher text using an encryption key(s) and a selected algorithm. This process renders the original text unreadable, so anyone who wants to read it must obtain the correct key.

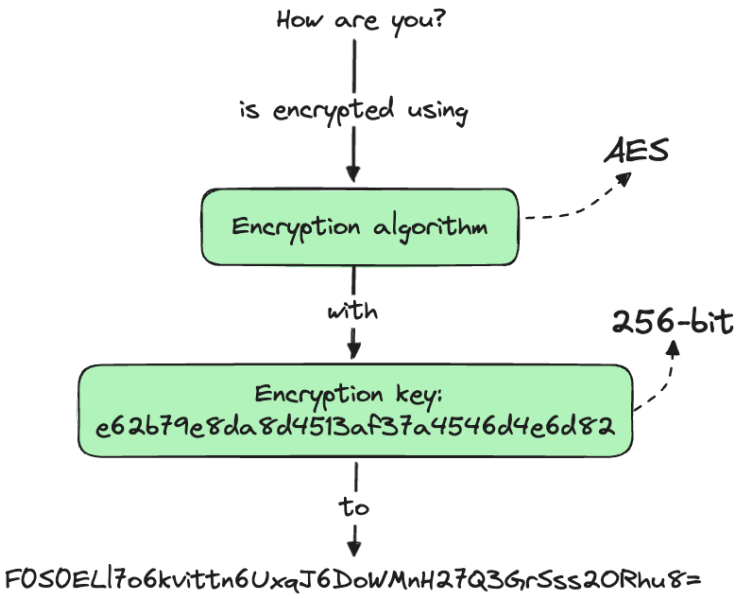


Figure 15. Encryption of a plain text to a cipher text

Encryption can be done using one of the following methods:

- **Symmetric**. This method is based on a single (private) key. The sender uses this key to encrypt the plain text into cipher text, and the recipient uses the same key to decrypt the cipher text back into plain text. An example of a symmetric encryption algorithm is AES (Advanced Encryption Standard).
- **Asymmetric**. This method is based on two keys: public and private. The public key is available to anyone who wants to encrypt plain text, but the private key is only accessible to authorized recipients. This way, there is no need to share the same key for encryption and decryption. However, asymmetric encryption is slower than the symmetric one. An example of an asymmetric encryption algorithm is RSA (Rivest–Shamir–Adleman).

How does the entire process of encrypting and decrypting the text work? Let's look at an example using the AES algorithm.

1. The sender uses the encryption key (in my example, 256-bit, but you can also use 128 or 192) and AES to transform the plain text into cipher text.
2. The sender sends the cipher text to the receiver.
3. The receiver uses the same key and AES to decrypt the cipher text back into plain text.
4. The receiver can now read the decrypted text.

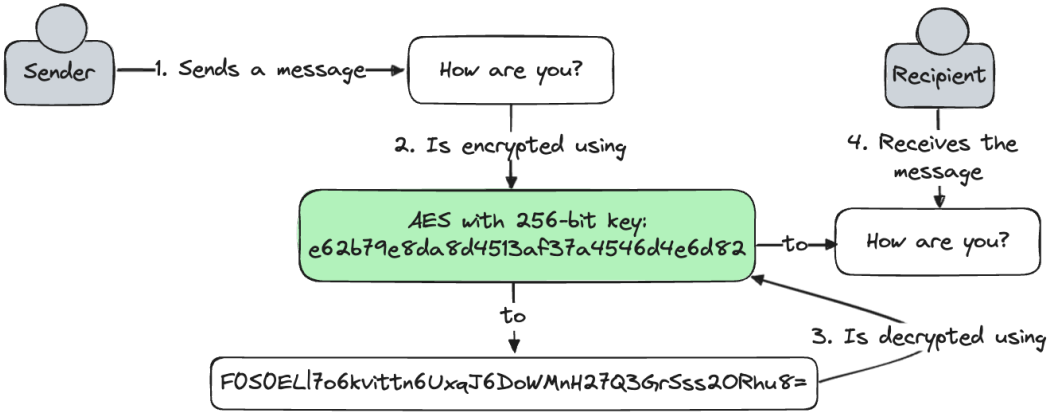


Figure 16. Example of the flow of encryption and decryption using AES and 256-bit key

Both encryption and decryption can be seamlessly integrated into your code by automating the process during data storage and retrieval. This eliminates the need for repetitive implementation.

```
// Decrypt and convert to original value on read
when reading:
    decryptedValue = Decrypt(dataFromDatabase)
    set entity.Property to decryptedValue

// Encrypt the value before writing
when writing:
    encryptedValue = Encrypt(dataToStoreInDatabase)
    write encryptedValue to database
```

Note that most modern ORMs (Object Relational Mappers) have built-in support for these operations. You can either configure it on the entity level with an attribute directly set on the property or in the place where you configure entities (e.g., entity builder in Entity Framework).

A common misconception: Our API is secure, so there is no need to encrypt the data in the database. While cloud providers and hosting services offer many security options, proper configuration still lies at our side. If we misconfigure things, like setting the wrong access controls or exposing the database online, it could lead to data leaks.

How do you determine which data is sensitive and which is not? If, in the event of a data leak, the data could negatively affect users, reveal information that should remain confidential, or impact you financially, then it is a strong candidate for encryption.

Sensitive information in logs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Supply chain attack

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Cross-Site Scripting (XSS)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

SQL Injection

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Misconfiguration of infrastructure

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Bonus: Exposure of passwords

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Recap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Epilogue

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Extra 1: Other Engineering Practices

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Working with metrics

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Vertical slices

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Developer carousels

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Addressing technical debt

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Extra 2: Architecture Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Case 1

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Case 2

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.

Case 3

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/master-software-architecture.