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 2024-10-29

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

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

Maciej "MJ" Jedrzejewski
Author

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!

Maciej "MJ" Jedrzejewski

STEP 1: Understand Software Architecture

Before taking you on a software architecture journey, let's take a moment to define it.

Here is the beauty: there is no official definition. Each architect has their own, and each may understand it quite differently. Sometimes, there are even heated discussions on this topic, which is good as it brings about further development in our vibrant IT world.

The following definition is my interpretation of software architecture. Feel free to use or modify it.

Software architecture
: is a holistic process that allows us to produce the software. It is a set of patterns, principles, processes, and relationships that make up the overall design.

What is important is that software architecture is not static. You can't say:

> *Okay, we have applied the architecture, and from now on, we no longer need to care about it.*

It evolves alongside the application it supports. As businesses change, so must their software architecture. Moreover, our understanding of the business deepens over time, even if the business itself remains stable. What seemed adequate in the early stages may prove insufficient.

Imagine that you have just started working as a software architect for a fitness studio. Your goal is to create a system that will automate the following tasks:

- Preparation of contracts

- Signing of contracts
- Registration of passes

You prepare a plan and start implementation. Soon, your product is in production. Customers are delighted because they no longer need to visit the studio to sign contracts. Stakeholders are satisfied because it has improved the customer service process and reduced costs. You are proud of what you did as a team.

Everything seems great. However, dark clouds are gathering on the horizon. More features are being requested, and there are also more customers. As you add these features, you realize that the current architecture is reaching its limits.

Your boss comes to you and tells you that another fitness studio has been acquired, and now you need to staff both. Within a few months, you take over several other studios until you finally become a chain and start offering franchises.

At some point, you also start offering supplements and consultations with nutritionists.

⚠ In such a situation, the initial assumptions about the software architecture are unlikely to remain the same. In most cases, business change is equal to software architecture change.

The sooner you accept this fact, the faster your systems will become robust and maintainable. Your customers will be satisfied, and your team will be happy.

Key areas of software architecture

The multitude of topics around software architecture can be overwhelming. That is why, to make my day-to-day work easier, I often use the divide-and-conquer method.

Divide-and-conquer
is a way to tackle a complex problem by breaking it into smaller parts and solving each part individually.

Using this method, I can split my software architecture problem into the following key areas:

- **Business analysis**. What does my business domain look like, and what are the key processes and boundaries?
- **Infrastructure**. Do we need to operate in a cloud-agnostic way, and if so, at what scale and budget? Do we need a database? A cache?
- **Deployment strategy**. Do we want to go with a single or multiple deployment units?
- **Solution architecture**. Where do we want to apply which patterns? How do we want to communicate between modules?
- **Test strategy**. How do we want to approach testing in our application?
- **Release strategy**. How often do we want to release new application versions to customers?
- **Teams**. How should we split and assign our teams? How experienced are they?

Each of these areas is divided further. In business analysis, I focus on defining boundaries and their relationships. In infrastructure, I address issues related to monitoring and scaling. In release strategy, I consider continuous delivery and canary releases.

This approach helps me stay well-organized at every stage of software development.

Architectural drivers

Every project is unique, differing in economic sectors, size, purpose, business capabilities, and more. Each time you plan software architecture, it differs from your previous experiences.

Looking for a one-size-fits-all architecture is like asking an alchemist to find a way to produce gold. It is impossible.

Microservices might be the best fit for one project, while a monolith could be more suitable for another. Event streaming can be beneficial in some scenarios but may be excessive in others. A document database could be ideal in one context, yet a relational database might be preferable elsewhere.

I could go on and on.

In one of his articles[1], Kamil Grzybek says:

> *Each of our decisions is made in a given context. Each project is different (it results from the project definition), so each context is different. It implies that the same decision made in one context can bring great results, while in another, it can cause devastating failure. For this reason, using other people's/companies' approaches without critical thinking can cause much pain, wasted money and finally - the end of the project.*
>
> —Kamil Grzybek, *Modular Monolith: Architectural Drivers*

Deciding on the architecture based on our feelings would make little sense. Instead, we can leverage the power of architectural drivers. These are categorized into four groups:

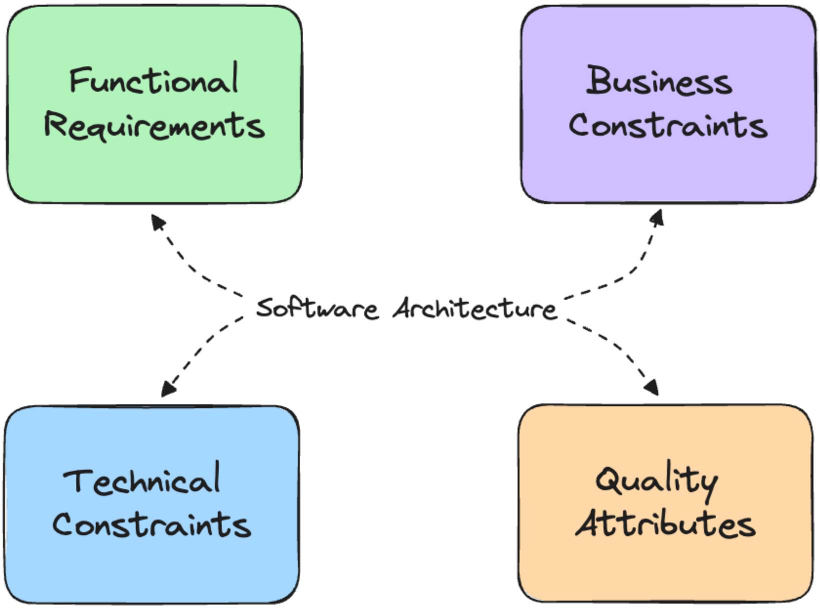


Figure 1. Four groups of architectural drivers

[1] https://www.kamilgrzybek.com/blog/posts/modular-monolith-architectural-drivers

Functional Requirements specify what actions or tasks a software system should perform to fulfill its purpose and meet users' needs. These requirements should be stated clearly and simply (unfortunately, in the real world, they almost never are), focusing on what the system must do, not how it will do it.

Example: *Users shall be able to search for content 'A' using specific keywords* or *Risk should be assessed for every new customer before signing a contract.*

Business Constraints refer to the limitations, law regulations, rules, or guidelines set by the company or external organizations on a software project due to budget, timeline, resources, or market conditions. These constraints ensure the solution considers factors that may affect its success.

Example: *We must implement it before the 'A' event in two weeks* (time constraint); *Only three developers can be used for this project* (resource constraint); *Due to GDPR, we must obtain explicit consent from users before collecting or processing their personal data* (regulation constraint).

Technical Constraints refer to limitations caused by technology, infrastructure, or other technical considerations when developing a software solution. They help to ensure that the resulting product is compatible with existing systems or industry standards.

Example: *We have to use a platform 'A' built internally*, or *Solution has to be implemented using C# because other systems are using it.*

Quality Attributes define the characteristics or qualities of a software system to ensure that it meets user expectations and business needs. These can be defined as:

- **Maintainability**. The ease with which the system can be modified to fix bugs, improve performance, or adapt to changing requirements.
- **Complexity**. The degree to which the system's structure and functionality are understandable and manageable.
- **Scalability**. The ability of the system to handle increased load without compromising performance.
- **Performance**. The speed and efficiency with which the system processes requests and performs tasks.

Example: *System has to be available at least 99.5% of the year* (availability), or *Generation of the 'A' report must not take more than 7 seconds at the peak* (performance).

Of course, there are other quality attributes like flexibility, portability, and reliability. It is your choice what to focus on.

Enough theory—let's focus on a real-world example. Your task is to build an application. Here is the plan:

- There will be 10 features (*Functional Requirements*)
- Deadline is set to 3 months from now (*Business Constraint*)
- Application must run on existing infrastructure (*Technical Constraint*)
- Application should be available during 99% of the year (*Quality Attribute: Availability*)

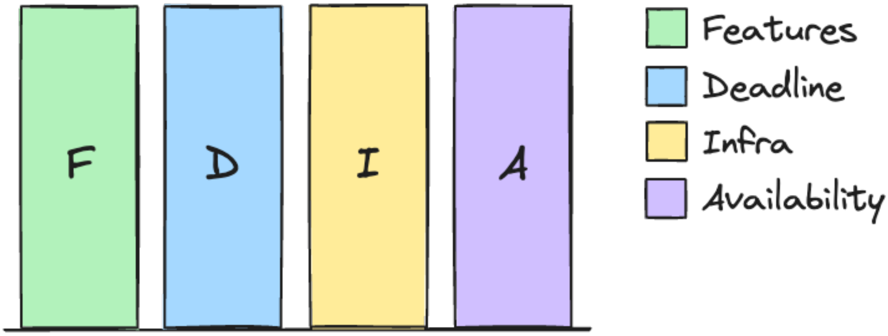


Figure 2. Initial harmony between architectural drivers

Now let's look at how things might change. In the real world, requirements often change over time. We will look at three different cases where these initial plans are changed. This will show you how flexible you need to be when building software.

Case 1: The company wants to release the minimum viable product (MVP) one month earlier. However, it is not possible to deliver it faster with the same resources. You need to reduce the number of features to meet the new deadline.

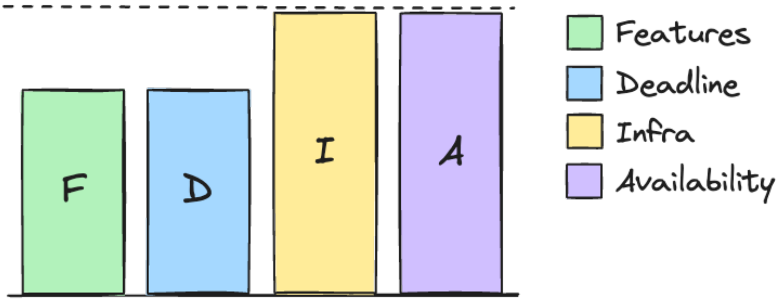


Figure 3. Earlier deadline requires feature reduction

Case 2: The initial plan was to guarantee 99% availability. However, there is a new requirement: it has to increase to 99.9%. There are two problems now. The current infrastructure cannot meet this level, so you need to add new components. As this takes time, the deadline must be extended.

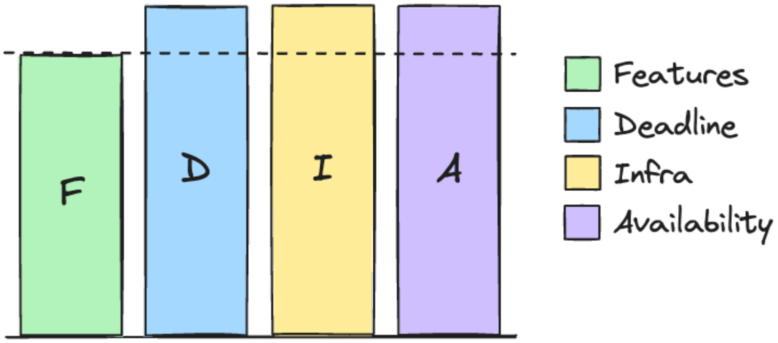


Figure 4. Increased availability requires additional infrastructure and extended deadline

Another option is to reduce the number of features to meet the previous deadline.

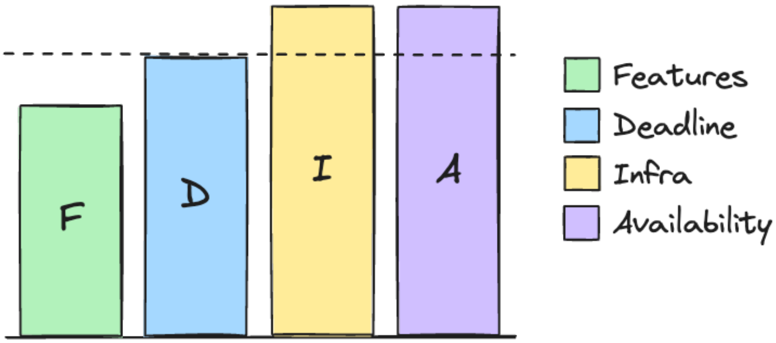


Figure 5. Increased availability requires additional infrastructure and feature reduction

Case 3: There is another feature that must be included in the MVP release. Adding more functionality will impact the deadline as it must be extended now.

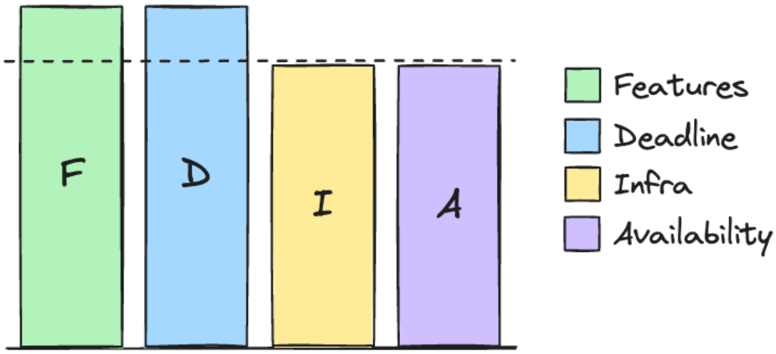


Figure 6. Additional feature requires deadline extension

As you can see, architectural drivers can help us, but they are also dependent on each other—a change in one causes a change in another. If we increase complexity, we reduce maintainability. If we optimize for flexibility, we might sacrifice performance. Prioritizing performance might increase costs.

You have learned the basics of architectural drivers, which will guide you throughout the rest of the book, supporting your decisions regarding selecting infrastructure blocks, deployment, release strategies, and more.

Importance of software architecture

Sometimes, we forget how important software architecture is. Instead of addressing business problems first, we start with implementation. We add features over features and new components. Our application grows, and customers are happy.

Suddenly, problems start to appear.

The application page takes forever to load, but it used to be lightning-fast. Adding one small change takes two weeks. Testing alone lasts one month. Fixing bugs in one place causes several others to appear. Components are tightly coupled. When you look at the reference map, you see a picture similar to what you drew when you were two—a giant scribble.

As a result, the software's reputation suffers, and customers lose confidence in its reliability. Frustrated by the constant firefighting, top developers seek opportunities elsewhere while the rest of the team stays demotivated. Your business cannot quickly adapt to market needs and slowly becomes uncompetitive.

In contrast, when you focus enough on your software architecture, you can evolve the system continuously. Moreover, investing in architecture pays dividends beyond the implementation phase. Customers are satisfied because the application solves their needs, and they spread the good word to others. The development team is happy because they can focus on improvements and features instead of firefighting. New features can be implemented quickly, and thanks to that, your business stays competitive.

So why is architecture often overlooked? One of the most common reasons is that, from a business perspective, we see it as a cost. We want to release the product as soon as possible, and focusing on architecture will extend that time, often by several weeks. We can easily see that focusing on architecture equals higher upfront costs.

One of the challenges with a well-designed architecture is that its benefits are not immediately visible. It might take 2-3 years for someone to realize that maintenance costs remain low, but they may not quickly connect this to proper architecture.

- **Immediate savings, shifted costs**. We saved $50,000 initially but incurred $1 million in maintenance costs over two years.

- **Immediate costs, shifted savings.** We spent $50,000 more initially and saved $1 million in maintenance costs over two years.

As software architects, it is our duty to make others aware that these long-term benefits are due to careful attention to architectural quality.

Lesson 1
Enough time spent on architecture
=
Better adaptability & lower costs
$$$

By planning our architecture thoughtfully and accepting its evolution, we ensure that applications remain maintainable and continue to serve us for many years.

How to stay pragmatic?

First, let's take a look at the definition of pragmatism. According to the Cambridge Dictionary:

Pragmatism[2]

> the quality of dealing with a problem in a sensible way that suits the conditions that really exist, rather than following fixed theories, ideas, or rules

So, it urges us to address immediate needs based on current conditions, not predefined theories or rules. In short: observe the application and react to problems that you have (or will have soon).

In the IT world, this means going against the grain. Why? One of the well-known traits in our environment is the desire for continuous development. We attend

[2] https://dictionary.cambridge.org/dictionary/english/pragmatism

conferences and meetups, read books, watch tutorials, and get certifications. During social outings, we talk about our jobs and share experiences.

This, in turn, makes us not want to be left behind. When we hear someone applying microservices at their job, we want to do the same. It is the same with orchestration, data streaming, and a mass of other things.

Should we be ashamed of this? Absolutely not. Should we be wary of it? Definitely yes.

If we get caught up in the whirlwind of technology, we will encounter the common problem of too complex architecture—even if we don't need it.

We add more and more components. Once we realize the solution is too complex, it is often too late to make significant changes. Replacing several components incurs enormous costs. In such cases, the only solution is to replace them continuously (step-by-step decomposition), but you will probably need help from external consultants (an expensive option!).

A better approach is to start lean. Build the application's MVP in weeks and release it to the first customers. As you get feedback and more traffic, tackle problems that occur. Add components to improve performance and monitoring, and scale up gradually.

> Build your architecture based on what you need now, not what you might need in the future. Use the power of real data and observation to support your decisions. But don't box yourself in – always leave room for growth and change in your design. In short, always keep some options open.

This way, you are following the pragmatic path. When you see the problem—or that it will come in the next few days or weeks—you react to it. You don't think about what will happen next year. Who knows, your app might not be around anymore.

We should not blindly follow theories and concepts because there are no silver bullets.

Silver bullet
: a solution that is believed to solve all problems or be universally applicable.

The same applies to adopting solutions from others; what works for them may not necessarily be suitable for us due to differing environments and challenges.

Each decision we make involves trade-offs. As software architects, our role is to choose the optimal one that fits our needs, often by combining various concepts.

How to stay holistic?

Alongside pragmatism, being holistic is another important thing. Let's again look at the definition from the Cambridge Dictionary:

Holism[3]
: the belief that each thing is a whole that is more important than the parts that make it up.

Software architecture should be treated as a whole. It comprises various elements, including infrastructure, code, testing strategies, and security. Each element should be treated with a similar focus. If we leave out any part, it can cause problems.

Over the years, I have observed that architects who come from infrastructure backgrounds prioritize infrastructure, while those from development backgrounds prioritize code. This is natural and expected, as long as they also focus on other aspects.

You don't have to be an expert in every area, but you should be like a construction manager. He does not need to be an expert in plastering, fitting pipes, or building a roof. But he has to know how to put them together to build something habitable. The same goes for architecture - you don't want a leaky pipe in your wall, do you? :)

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

[3] https://dictionary.cambridge.org/dictionary/english/holism

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.

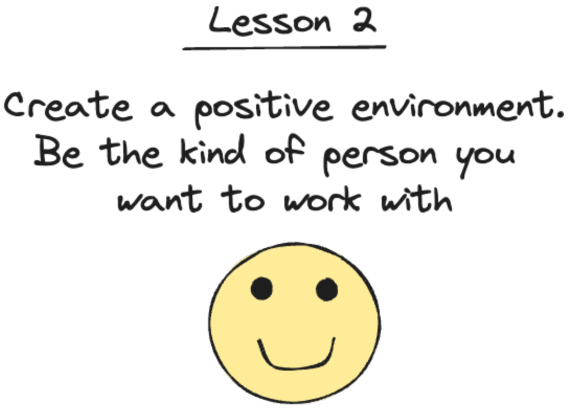
The rule of thumb: the broader the knowledge, the easier it is to understand the environment and make wise decisions.

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[4], 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.

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.



Lesson 2

Create a positive environment. Be the kind of person you want to work with

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

What are the key drivers of a successful software architecture?

You already know my perspective on software architecture. To make this book complete, I asked top experts to share their opinions on the subject as well. Vlad Khononov, Milan Jovanović, Oskar Dudycz, Milan Milanović, and Denis Čahuk have agreed to share their views.

Each of them answered the title question: *What are the key drivers of a successful software architecture?*

Vlad Khononov
Author of Balancing Coupling in Software Design (Pearson)
and Learning Domain-Driven Design (O'Reilly)

The term "software architecture" itself, I dare say, is somewhat oxymoronic. Architecture is a set of design decisions that are hard to make and even harder to change. This holds true for any type of architecture, be it software or urban planning. Yet, software architecture is quite unique. Let me explain.

Software has to be *soft*. Healthy systems continuously evolve. The evolution might be driven by new functionalities requested by users or needed to stay ahead of the competition. The system's success may lead to infrastructural changes: initial choices might become technically or financially ineffective as the system scales. Furthermore, organizations developing software systems evolve as well. Organizational growth impacts software architecture; design decisions that made sense for a startup often don't work for an enterprise.

All those evolution drivers make architectural decision-making even harder. There is no such thing as perfect design. Even if your design achieves all of its functional and non-functional goals today, nothing guarantees it will be the case tomorrow. New requirements might invalidate the assumptions on which your design is based. Therefore, I believe the key driver of a successful software architecture is its ability to withstand changes.

We don't have crystal balls that predict the future of software systems. Yet, we have other tools to address uncertainty in our designs. The hardest one to acquire is experience. After working on multiple systems and enduring painful architectural changes, you will notice common change patterns among most systems. That awareness will help you make design decisions that minimize the impact of such changes in the future.

Arguably, the most important tool for managing uncertainty is modularity. To address uncertainty, you must make it explicit. If you are making assumptions about the behavior of users or the system's infrastructural needs, encapsulate them in explicit modules. Revisiting such a decision in the future should impact only a single module, and you should know exactly which module it is.

In summary, to be successful, software architecture has to withstand uncertainty. Experience definitely helps, but nothing beats good old modular design.

Milan Jovanović
Software Architect, Educator

A successful software architecture stands upon two fundamental pillars: functionality and quality.

Functionality is the system's ability to satisfy the specified requirements and deliver the desired features. This involves a well-defined structure, clear modules, and efficient algorithms that work together seamlessly. In essence, a functional software architecture fulfills the software's core purpose. However, a system that merely functions is not necessarily successful.

On the other hand, quality consists of non-functional requirements such as performance, reliability, maintainability, security, and scalability (collectively known as "-ilities"). Performance determines how efficiently the software executes its tasks. Reliability ensures consistent operation even in unexpected conditions. Maintainability allows for easy modifications and updates. Security is about shielding the system and its sensitive data from malicious users. Scalability allows the system to adapt to increasing workloads.

A good software architecture ensures that the software not only performs its intended functions but also performs it efficiently and securely, even under varying loads and conditions.

However, software architecture is more than just a technical effort. It is a strategic tool that must align with our business goals. A successful architecture helps the business achieve its key objectives. A key objective could be rapid time-to-market to capitalize on emerging opportunities. Another key objective could be reducing costs to enhance profitability or fostering innovation to stay ahead of the competition. For example, an e-commerce platform might prioritize scalability and performance to handle peak traffic during sales. A healthcare application might focus on security and data privacy. We must understand these business drivers and incorporate them into the architecture.

Change is the only constant (an old saying). This is equally true in software architecture. Software architecture is a continuous journey of improvement. A successful architecture constantly evolves. We must incorporate customer feedback, adapt to emerging technologies, and address new challenges. A successful software architecture embraces the principles of modularity, loose coupling, and well-defined interfaces. By adopting a culture of continuous evolution, we can ensure that our software systems remain efficient, effective, and competitive in the long run.

Oskar Dudycz
Event-Driven Architecture enthusiast
and Open Source builder

I never feel comfortable being asked about software architecture drivers. How do you give someone a checklist for good software?

It is funny that we called our industry **SOFT**ware, but it is all about making **HARD** decisions. Usually, we make them when we are the dumbest: we don't know the business domain, we don't know the user needs, and we are unsure of technology choices. Plus, even if we do, our changing environment is open to proving us wrong.

For me, architecture decisions are more a process than a set of specific rules. It is a process of answering the following questions:

WHY? Understanding the product vision and business model. Consider where the money flows: who the customer and the user are. That's an important fact, as we should care about all users but optimize for customers, especially those who bring money. In the end, our product should generate revenue.

WHAT? Understand what we actually need to build. Set a mental model of the business workflow. This is an excellent moment for collaborative tooling, brainstorming, and modeling practices like Event Storming, Domain Storytelling, etc.

HOW? Think about the requirements and guarantees you need to have. Find architecture patterns and the class of solutions that will fulfill your requirements. So, the type of databases, deployment type, and integration patterns, not the specific technologies. Consider tools like C4 and other ones to structure your findings.

WITH. Select the tooling based on the outcome of the previous point. It has to fulfill not only functional but also non-functional requirements, like costs, match team experience, ease of use, etc.

Then rinse and repeat.

Architecture is not created in a vacuum. Talk and collaborate with business, users, and your technical fellows.

Consider the team you (can) have. Most of the time, the best technology is the one that your team knows. We are building new tools, but to be true, rarely sophisticated ones. Most of them are regular lines of business applications. And hey, let me share the secret with you: your decisions will be wrong. Mine also. And that's fine. We don't need to be flawless; our system also doesn't need to be. Expect the change; it will come.

So don't be afraid to make decisions, but don't rush yourself. Always consider alternative solutions. Record your decisions together with discarded ideas. Provide the context and explain WHY, WHAT, HOW, and WITH. Provide the assumed limits. Suggest how to evolve if, for example, your system is a huge success and becomes overwhelmed by traffic. Some problems are good to have, but they don't need to be solved immediately.

We should optimize not for maintainability but for removability. If our system is built so that we can relatively easily remove pieces from it, then we can drop bad ideas and move on to new ones. Also, by accident, we are getting a system that's easier to maintain.

Dr Milan Milanović
Chief Technology Officer

When we talk about architectural drivers, we mean the early on in the project from which our architecture results. Architectural drivers are guidelines that a software architect should consider while weighing different alternatives.

If we take into account key drivers, those would be:

Business drivers

When we talk about business drivers, the first thing we need to understand is business requirements. The architecture must meet the project's requirements to ensure the developed system aligns with stakeholder needs and helps the company reach its goal. We need to engage with stakeholders early to gather those requirements, which ensures the architecture meets business needs and user expectations.

As per Conway's Law, team composition significantly influences a software system's architecture. The system's structure often mirrors the communication patterns and organizational structure of the team developing it. Architects must understand this and design the architecture to enable effective communication and collaboration within the team.

We are also usually constrained by these two essential components: how much time we have to complete something and our budget. This limits the time we have to research and weigh trade-offs.

Technical drivers

Choosing the right technology stack that aligns with project requirements and team expertise is critical. The stack should support the desired performance, scalability, and security needs while being sustainable in the long term. Sometimes, the client may have this requirement regarding operating systems, specific libraries, etc.

The architecture should also seamlessly integrate with existing systems, third-party APIs, and future technologies, ensuring interoperability and extensibility.

Functional and non-functional requirements

Functional requirements establish the software's core operations to meet user and stakeholder needs, including user stories, use cases, and specific features. They are easier to understand as their purpose is usually straightforward, as every piece of software needs some job. Yet, focusing only on functional requirements could result in more secure and faster software, which is why we need non-functional requirements (quality attributes).

Non-functional requirements are the properties of a system that influence its operation and user experience but are not directly tied to specific functionality. They define the system's overall qualities, such as performance, security, maintainability, and usability. Understanding and addressing these attributes is crucial for building robust and reliable software.

Denis Čahuk
Engineering Expert, Coach, XPer,
Author, Speaker

Architecture is code's companion. Simple code has simple architecture. Complex code most often comes with complex architecture. Great architecture is when the architecture of complex code makes it appear as if it were simple.

Simplification is the most challenging problem in software design. It is the marriage of code craftsmanship, software design, systems design, and organizational communication.

Architecture is a lexicon and a map. It gives meaning, metaphors, and names to various areas in code that did not have a natural name as they grew, similar to districts in cities, regions in countries, and organs to cells in your body. It is no coincidence that most depictions of software architecture are modeled geographically: next to something, close to something else, similar to, or neighbor of—two-dimensional space with borders and crossings.

Just like a city, software products will attract users to live inside them—thousands, even millions. As customers become used to the product's style and cityscape, the architecture also becomes more difficult and slower to change. This scale and usage are challenges for *successful* products as they are actively used while being shaped, re-shaped, and scaled outwards with new features.

Unlike typical construction, within software, *everyone* is an architect to some degree. Engineers, testers, managers. Architecture is influenced by usage patterns, language, tools and frameworks, and especially organizational composition. Such is the nature of *soft*ware.

Teams create architecture whose shape matches that of their communicational barriers and flow.

A great architect will capture the natural seams and boundaries of this growth and, to capture them, modularise them into named wholes of cohesive units. These cohesive units define the span of awareness and cognitive load necessary for future engineers and architects to make changes.

To be successful as an architect means to be adaptive and resourceful when faced with a rigid, frequent system. A software architect's duty is to courageously balance the conversation of what the code *does*, what it *should do*, and *what changes it enables* in the immediate future.

Recap

In this step, I introduced you to the basics of software architecture. We discussed what it is and why it is important in today's fast-paced tech world.

I described my understanding of software architecture. You learned that architectural drivers can support us in building systems. I explained some key ideas that shape how we design software, like making sure it can grow (scalability), is easy to fix (maintainability), and runs effectively (performance).

I also talked about being pragmatic when designing software, having a holistic view of the entire environment, and finding a balance between doing things well technically and meeting business needs.

You learned about the role of a modern software architect. This person is part of the development team, uses their expertise to help others, and connects all technical and non-technical people.

Finally, guest experts have expressed their opinions about what makes software architecture successful. I am confident this knowledge will stay with you for a long time.

It is time to discover the business domain.

STEP 2: Discover Your Business Domain

Whenever you start a new project or move to another company, the key is understanding what problem you are trying to solve, why, and for whom. These questions will help you understand the business domain in which you work, and you can design systems tailored to customers' needs.

> It is only sometimes possible to get a clear answer to a question. Often, the information you receive from different people is mutually exclusive (one person says this, another says something else). One of your duties as a software architect is to make a common understanding of the problem you are trying to solve.

Let's look at some real-world examples of business domains:

- **Education**. Learning platforms, assignment tracking, student enrolment, grading
- **Healthcare**. Medical research, pharmaceutical development, patient management, appointment scheduling, medical treatment
- **Finance**. Managing money, providing financial advice, facilitating transactions
- **Retail**. Online shopping, product catalog, order processing, payments
- **Logistics**. Warehousing, transportation, inventory management, route optimization, delivery schedule

You can see that each domain covers many areas. Sometimes, your task will be to design software that covers the entire domain (all parts of *Logistics*), and sometimes just a part of it (*Transportation* and *Route optimization*). There is also a chance that the company you joined will operate in various domains (*Retail* and *Logistics*).

Although these domains differ, they often face the same problems. For such cases, you can use similar patterns and solutions. It is like having a cheat sheet in front of you. For example, payment processing or appointment scheduling might be needed in both the *Education* and *Healthcare* domains, which can likely be addressed using similar solutions.

Understanding the business domain at a high level requires spending many hours talking to domain experts, stakeholders, and customers, actively participating in workshops, and implementing or maintaining the application. It is expected to feel like you are on a rollercoaster filled with architecture diagrams, tons of requirements, new teammates, organizational things, etc. Even if you are an experienced architect, you may need help with all the information coming your way.

Due to a flurry of duties, you may not have the time to explore the business domain thoroughly, or someone might tell you that all the requirements are already there, precise, and you no longer need to focus on them. This is where most problems and legacy applications start.

You should clearly communicate the need to go through the business domain and all processes yourself—you are new and cannot understand them just by reading documents (e.g., Excel or Word files) someone else created. It would be best to organize at least a few workshops with domain experts. Usually, you will need at least several days to understand small businesses and several weeks for larger ones.

Convincing people to participate can be challenging when trust has not yet been established. Usually, you only get one chance. The workshop has to be a success—if it is high-quality and participants are heavily involved, it sets the stage for future workshops.

How can you navigate through all of this without getting lost? Let’s begin our journey.

The case

We will use a real-world case that will guide us through all the steps in this book. Our objective is to build an application for a *Healthcare* domain to support private medical clinics and automate their business processes.

Category	Definition
Business Domain	Healthcare
Processes	Appointment scheduling, medical records management, patient treatment, drug prescription, invoicing
Current procedures	Clinic workers & doctors use their computers to note everything in a spreadsheet. Doctors who have home visits record everything on their phone
Involvement	Executives want to be involved in the decision-making process and the design of the product
End customers	Private medical clinics in Europe
Business model	SaaS
SLA	99.5% (availability)
Expected number of customers	3-5 (first three months)
Expected number of patients	1,000-1,500
Expected number of employees	60-100
Existing dev teams	No, current is inexperienced
Existing infrastructure	No
Budget limitations	Up to $500,000
Deadline	2 months from now

This case will always be mentioned and linked whenever it is necessary to refer to it in the following chapters.

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.

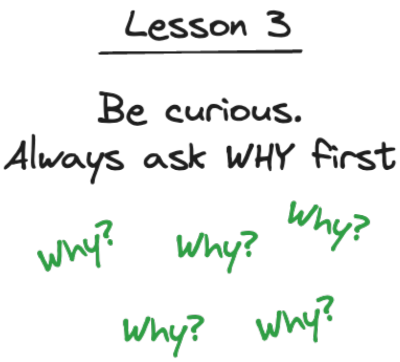
The next step is to organize a meeting with domain experts and stakeholders, or ensure you are invited to attend one in the first two weeks. By now, you have already gained some knowledge from unofficial talks. Now, it is time to deepen that understanding.

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].

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



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:

> 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

Armed with the basic knowledge, you are ready to organize the first interactive workshop with people who represent groups of decision-makers and domain experts. Sometimes, domain experts are represented by non-technical people, and sometimes by developers. Yes, developers!

When someone works on a legacy system, they often have comprehensive knowledge of business processes, as they need to maintain it, fix bugs, and extend functionality.

I prefer sessions with less than ten people, including me as the facilitator. This way, I can focus on everyone, and I can easily keep track of what is happening. If there are more people, I start feeling lost. However, this is an individual issue. I know people who are comfortable running workshops with twenty participants.

Often, more than one high-level workshop will be needed. Large-scale systems with complex processes might take dozens of sessions. In such cases, start with the most critical process. It might be:

- Appointment scheduling in the *Healthcare* domain
- Claim assessment in *Insurance*
- Order placement in *Retail*
- Risk assessment in *Banking*
- Supplier selection in *Supply Chain*

When you finish the first process, continue with the next ones.

⚠ Be careful, as there might be a situation where several processes are already in the in-depth phase while others are still in the high-level phase. In such a case, I recommend having at most 3-5 processes that you deal with simultaneously. Otherwise, it will be challenging to track them all.

Try to split the workshop into several parts. Long sessions are a productivity killer. For the first session, I would recommend planning 3.5 hours that are divided into:

- 1 hour, 5 minutes break

- 1 hour, 5 minutes break
- 1 hour, 5 minutes break
- Plan an additional 15 minutes at the end. This extra time is useful for resolving any heated discussions or wrapping up loose ends.

I often split the workshop into six parts, each lasting 30 minutes with a 5-minute break. It allows for complete focus and high engagement. Participation without full attention makes no sense. When I see that certain people are not engaging in discussion and modeling, I usually don't invite them to the subsequent workshops.

There are two states of the business process that you can model:

- **As-is**. Represents its current state.
- **To-be**. Represents its future (desired) state.

When discovering processes, you typically begin by understanding their current state, known as the "as-is" state. The second technique is used when you want to find out how the process can be improved using the application you will build. For example, today, all appointments are scheduled manually in a notebook (as-is). In the future, they will be done using our application (to-be). Please keep in mind, however, that this is a wishful state and much can change along the way.

How do you discover a business domain? Are there any techniques that can help you? Of course, there are many of them. In this book, we will focus on the two that are quite popular:

- Event Storming
- Domain Storytelling

These are entirely different techniques, but they are often combined. We will examine them more closely when modeling the patient treatment process, which starts with scheduling the appointment and finishes when the invoice is sent to the patient.

Event Storming

Event Storming is one of the simplest ways to handle workshops where you expect much participant engagement. It can be used for almost everything—my colleague even used it with his wife to plan their wedding, and I frequently use it to organize my activities.

How can it help you in business analysis? The same way! There is no difference between wedding planning and invoice preparation—both are processes. Furthermore, if you plan your application to be based on events, there is no easier way to model it. That is why this is a brilliant way to discover processes.

As the name suggests, the central focus area is events, representing business facts that already happened, like *Appointment scheduled* or *Treatment completed.*

Event Storming is divided into three levels:

- **Big Picture**. At this level, focus on creating the draft of the process. Identify the business events, order them chronologically, group them, find the first hot spots, note any external systems, and add comments.
- **Process**. Dive into the details of the process. Identify triggering commands, relevant policies, and participating actors. Establish read models.
- **Design**. Explore modeled processes, focus on design, find aggregates, and refine business rules. At this level, pseudo-code may accompany sticky notes.

To handle the on-site workshop, you will need the following:

- **A long wall**. The longer, the better.
- **Sticky notes of different colors**. My recommendation is to use static notes instead, as it is easier to move them around.
- **Markers**. They will be used to fill the notes or write something on the wall.

> It is a good idea to prepare the room where the workshop will take place so there is nothing to sit on (unless someone has to sit for health reasons). During my workshops, people were much more engaged while standing. But you have to remember that people can also sit on the windowsill, and this cannot be so easily removed from the room :)

Another option is to conduct the workshop remotely. In addition to some communicator that allows you to talk to people, you will need an application that allows you to manage the board and the notes efficiently. In my opinion, the easiest-to-use software at the time of publishing this book was Miro[2], but you can use whatever tool you like.

First, we will handle the *Big Picture* workshop. We will focus on the current state of the patient treatment process. During this session, you are going to use four types of notes:

Event

- **Color**: Usually orange
- **Purpose**: Represents every business event in our domain. It does not represent a technical event, so *Button clicked* is not something you should care about. The rule of thumb is to name it in the past tense (something happened) because it is an event that has already occurred and cannot be changed.

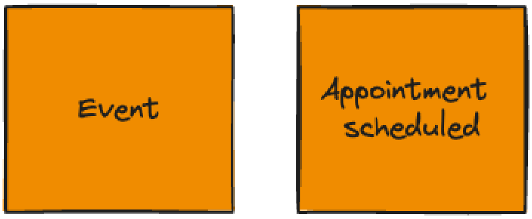


Figure 7. An example of a business event

Hot Spot

- **Color**: Usually red
- **Purpose**: Represents every question or input that is unclear or cannot be answered during the workshop. This ensures that these concerns are not forgotten and can be addressed in future workshops.

[2] https://miro.com

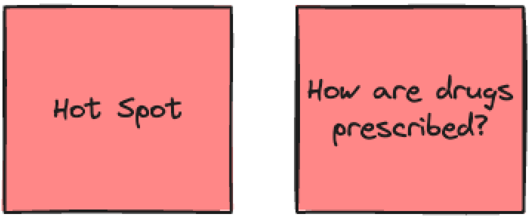


Figure 8. An example of a hot spot

External System

- **Color**: Usually pink
- **Purpose**: Represents an external system that interfaces with the domain. External systems are not directly controlled by the modeled system but play a significant role in its operation. They may send or receive messages, data, or events to and from the system, influencing its behavior or being influenced by it.

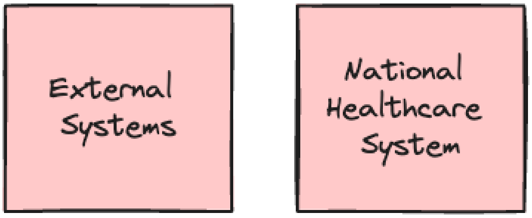


Figure 9. An example of an external system

Comment

- **Color**: Usually yellow
- **Purpose**: Represents additional information, clarifications, or observations that participants want to add to the conversation. They ensure that important information is not overlooked.

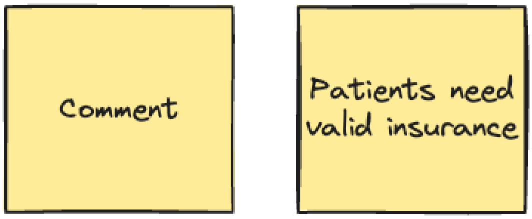


Figure 10. An example of a comment

Before starting the workshop, ensure that participants get as many notes in the above colors as possible and give them markers. Next, you can conduct a short training session. Model a familiar process, such as pizza delivery or bike rental, which most participants will know. This should take no more than 30 minutes and will help participants understand the key mechanisms.

When ready, explain which process from the business domain you will focus on. Ensure that all participants are aware of the workshop's objectives and goals. Here are a few examples:

- **Objective 1**: Understand the current patient treatment process from end to end.
- **Objective 2**: Identify and document key events and find hot spots.
- **Goal 1**: Creation of a detailed visual map that represents the entire patient treatment process.
- **Goal 2**: A common understanding of the patient treatment process.

You can start the workshop by sticking the first orange note, representing a random event, in the middle of the wall. This initiates the process, allowing participants to place their notes on the right side for events that occur after yours or on the left side for events that occur before.

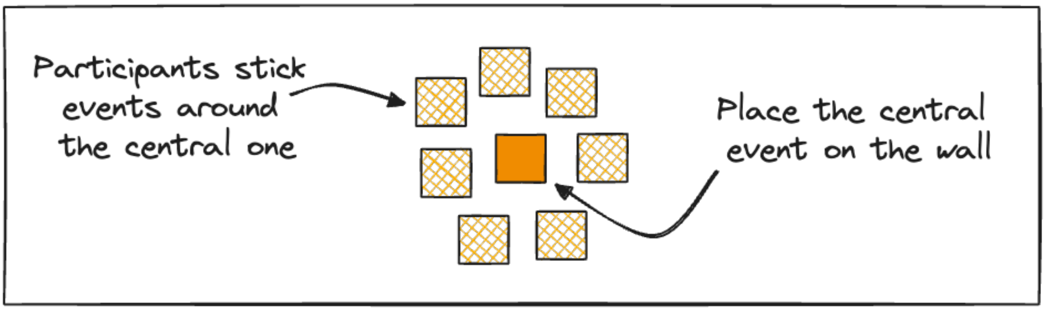

Participants stick events around the central one
Place the central event on the wall

Figure 11. The starting point of the Big Picture workshop

Let everyone know there are no wrong ideas. Each participant should write what comes to their mind—of course, related to the modeled process!

> When you notice events repeating during the session, please clarify with the participants who added them to the wall. In many cases, these duplicates represent the same event and can be removed. However, an event with the same name may actually be a part of another business operation and should remain on the wall.

During the workshop, you should facilitate discussions and ask questions like:

- Is there any other event that can impact this one?
- Are there any negative consequences when the patient has no valid insurance?
- What happens if the drug is unavailable?

When you encounter a heated discussion with no resolution in sight, leave the red hot spot note on the wall and proceed with the workshop. Your role is to assist others in addressing any issues or misunderstandings that arise.

In the first hour—the most turbulent—during which participants will have the most ideas and energy, the wall should be filling up with notes at a high rate. You should feel that you are in a session where thoughts clash.

If this is not the case, there are two possibilities: either you have chosen the wrong people, or certain aspects require further clarification. You may not be able to do much about the first scenario, but try to provide a few clarifying examples in the second.

After the first part, the wall should look similar to this:

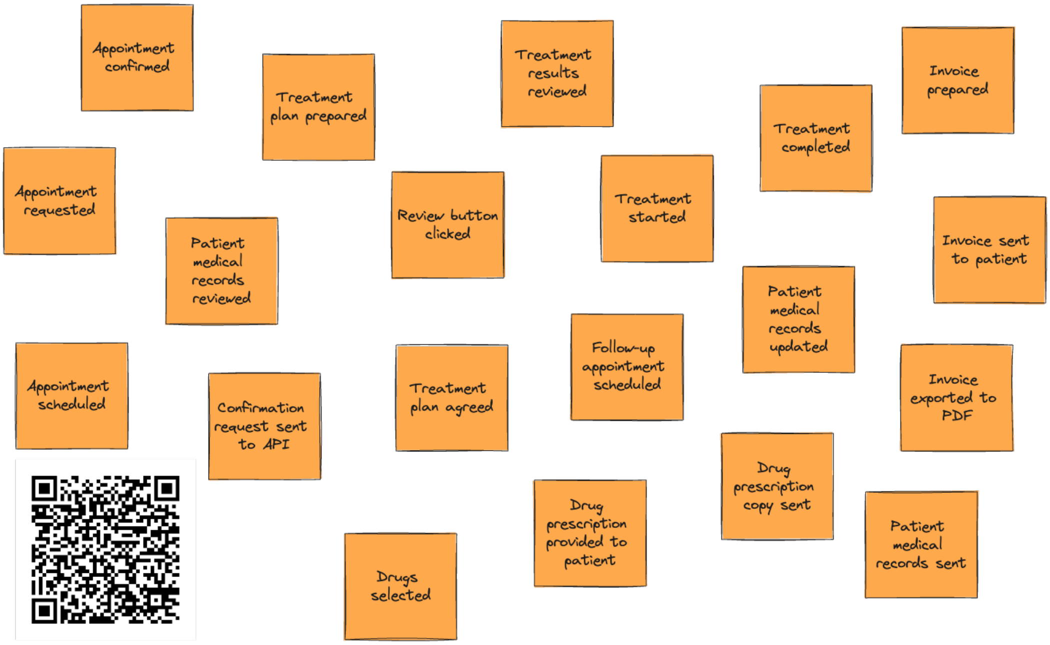

Appointment confirmed
Appointment requested
Appointment scheduled
Treatment plan prepared
Patient medical records reviewed
Confirmation request sent to API
Review button clicked
Treatment plan agreed
Drugs selected
Treatment results reviewed
Treatment started
Follow-up appointment scheduled
Drug prescription provided to patient
Treatment completed
Patient medical records updated
Drug prescription copy sent
Invoice prepared
Invoice sent to patient
Invoice exported to PDF
Patient medical records sent

Figure 12. Results of brainstorming during the Big Picture workshop

There are a lot of events. If you look more closely, you will see both technical and business-related ones:

- Confirmation request sent to API
- Appointment scheduled
- Review button clicked
- Invoice prepared
- Invoice exported to PDF

The first thing you want to do is to get rid of the technical events, as they are of no interest:

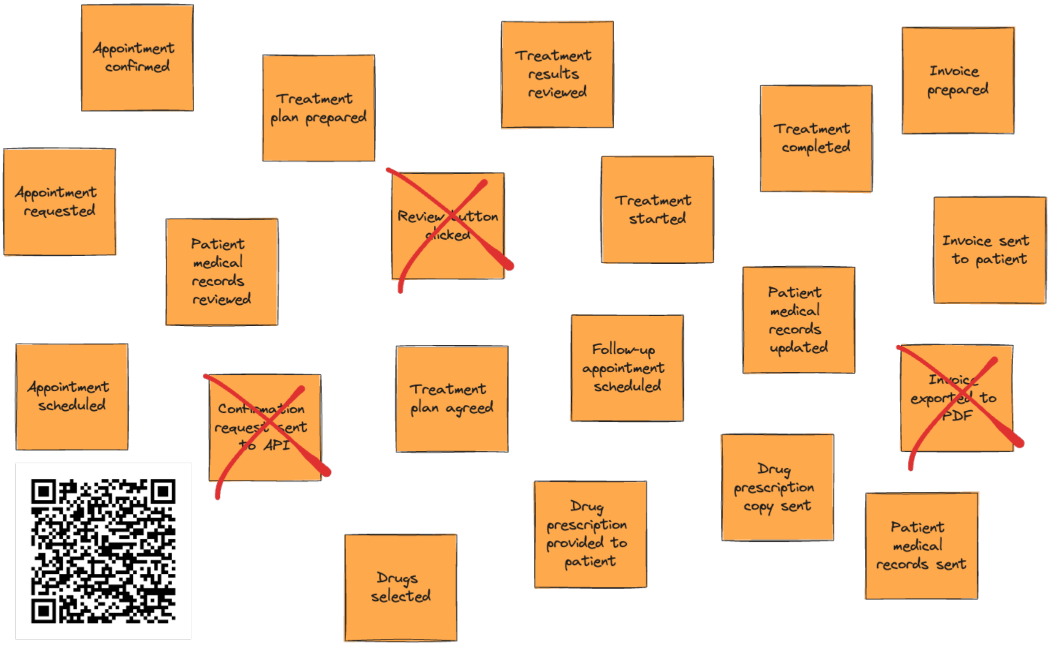

Appointment confirmed
Treatment plan prepared
Treatment results reviewed
Invoice prepared
Treatment completed
Appointment requested
~~Review button clicked~~
Treatment started
Patient medical records reviewed
Invoice sent to patient
Patient medical records updated
Follow-up appointment scheduled
Appointment scheduled
~~Confirmation request sent to API~~
Treatment plan agreed
~~Invoice exported to PDF~~
Drug prescription copy sent
Drug prescription provided to patient
Patient medical records sent
Drugs selected

Figure 13. Removal of technical events

Next, you group these events chronologically. It can be approached:

- **From left to right**. You start with the first event and then move forward with the next event until you reach the end of the process.
- **From right to left**. You start with the last event and then work backwards with the previous event until you get to the beginning of the process.

When dealing with various outcomes, such as success or error scenarios, you can still organize events chronologically. However, you will need to consider multiple possible paths and model them separately.

Both techniques are useful. When we use the former, we follow the nature of our brain; thus, we can model detailed processes and there is less chance of forgetting an event. Using the latter allows us to think outside the box, which can help us find something we wouldn't see by going from left to right. However, the disadvantage of this technique is that it is easier to miss an event.

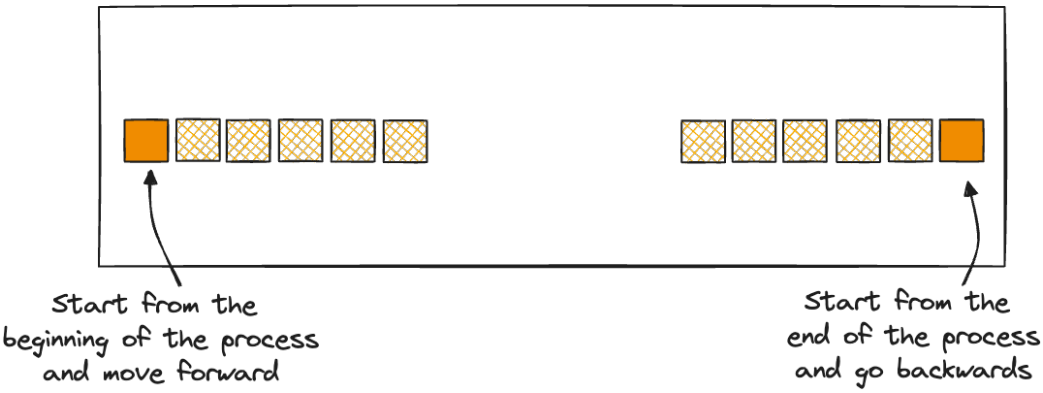

Start from the beginning of the process and move forward
Start from the end of the process and go backwards

Figure 14. Various techniques for grouping events

After grouping, you should see an image similar to the one below.

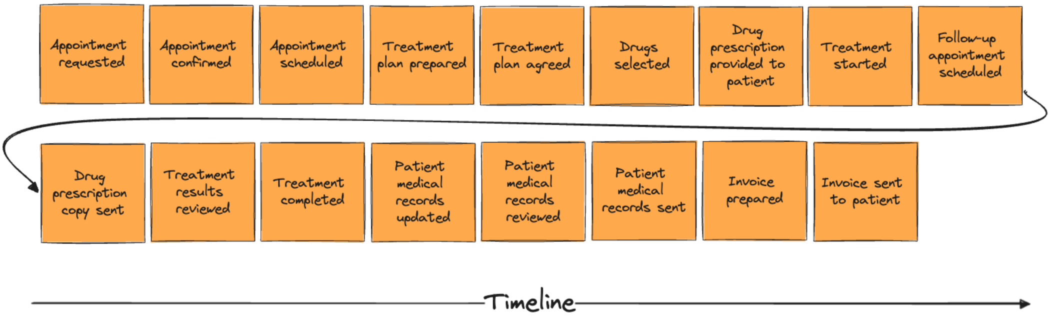

Appointment requested
Appointment confirmed
Appointment scheduled
Treatment plan prepared
Treatment plan agreed
Drugs selected
Drug prescription provided to patient
Treatment started
Follow-up appointment scheduled
Drug prescription copy sent
Treatment results reviewed
Treatment completed
Patient medical records updated
Patient medical records reviewed
Patient medical records sent
Invoice prepared
Invoice sent to patient
Timeline

Figure 15. Business events in chronological order

You can see a long "snake" formed by the events—a single line on the wall. However, you don't want to keep it in this state; it tells nothing more than the order of events. This is an ideal time to discuss pivotal events. You can think of them as:

- Key events (marking milestones or checkpoints) in the business process.
- A new subprocess is triggered after that event, or it begins with it.
- Actions taken during these events typically cannot be easily reversed without significant effort or consequence.

- Often, there is a switch of actors. For example, the doctor reviews the patient's medical records that were updated by the front desk.
- They often mark boundaries of other (sub)processes.
- **They are difficult to reverse**. This is the most important and helpful characteristic of pivotal events. For example, when an invoice is booked, or a payment is made, they are not easy to undo.

There will be multiple pivotal events in each business process you will model.

Let's look at the diagram. Candidates are marked with the letter 'p', and dotted lines represent potential boundaries.

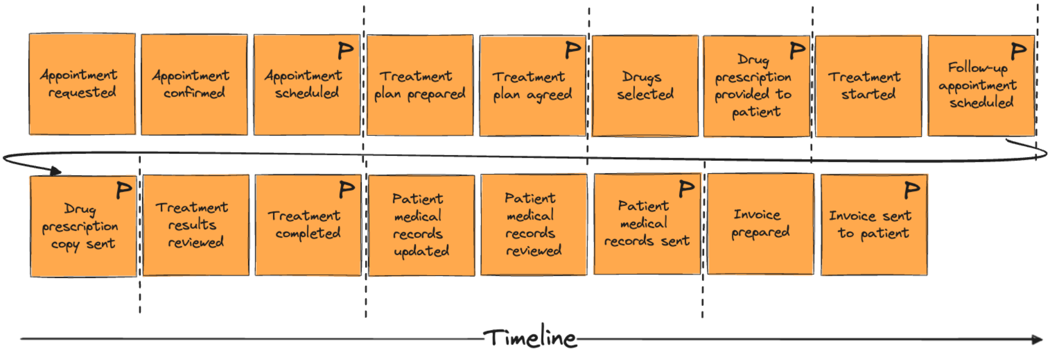

Appointment requested
Appointment confirmed
P
Appointment scheduled
Treatment plan prepared
P
Treatment plan agreed
Drugs selected
P
Drug prescription provided to patient
Treatment started
P
Follow-up appointment scheduled
P
Drug prescription copy sent
Treatment results reviewed
P
Treatment completed
Patient medical records updated
Patient medical records reviewed
P
Patient medical records sent
Invoice prepared
P
Invoice sent to patient
Timeline

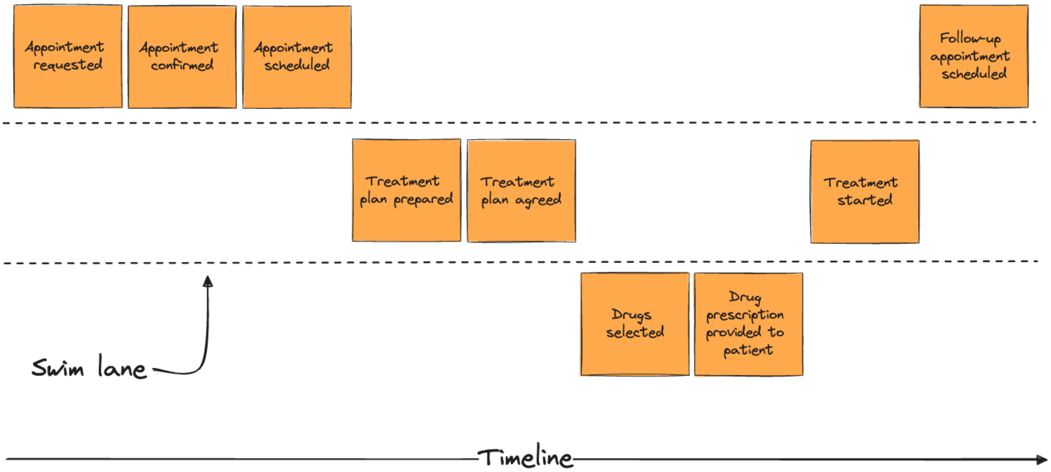
Figure 16. Candidates for pivotal events in the modeled process

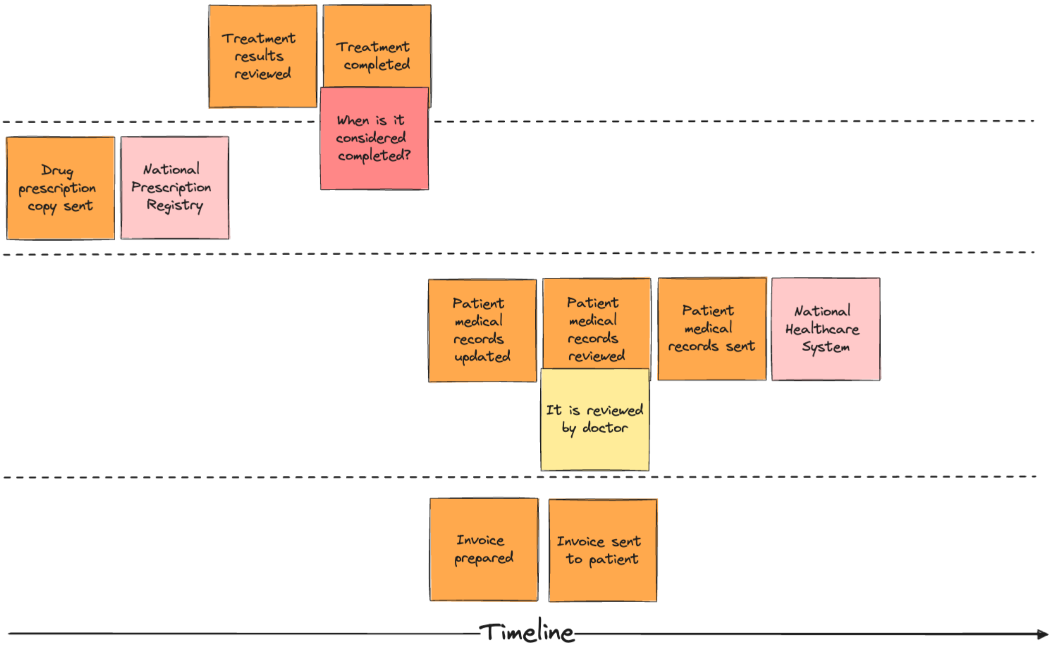
Each candidate that is marked on the above diagram is not easy to reverse:

- **Appointment scheduled**. To unschedule it, it requires you to contact the doctor, contact the patient, and unbook the examination room.
- **Treatment plan agreed**. When the plan is prepared and agreed upon, it cannot be undone without re-evaluation and mutual agreement between the patient and the doctor.
- **Drug prescription provided to patient.** To revoke or change the prescription, the patient must contact the prescribing doctor.
- **Follow-up appointment scheduled.** Same as in the case of the appointment.
- **Drug prescription copy sent.** No changes are possible once the prescription copy has been sent.

- **Treatment completed.** It is impossible to undo this event. When the treatment is completed, then it is only possible to start another treatment.
- **Patient medical records sent.** After sending patient medical records to the national healthcare system, you can't undo it. You will probably need to contact the support desk for the correction or send another update.
- **Invoice sent to patient.** In case there is a need to correct the existing invoice, you usually have to create another invoice that will be a correction of the previous one.

One of my favorite ways of grouping subprocesses is to separate them using swim lanes. This way, I can see that all my actions which are related to appointment scheduling are in one group, all actions related to drug prescription in another, and so on:




Treatment results reviewed
Treatment completed
When is it considered completed?
Drug prescription copy sent
National Prescription Registry
Patient medical records updated
Patient medical records reviewed
It is reviewed by doctor
Patient medical records sent
National Healthcare System
Invoice prepared
Invoice sent to patient
Timeline

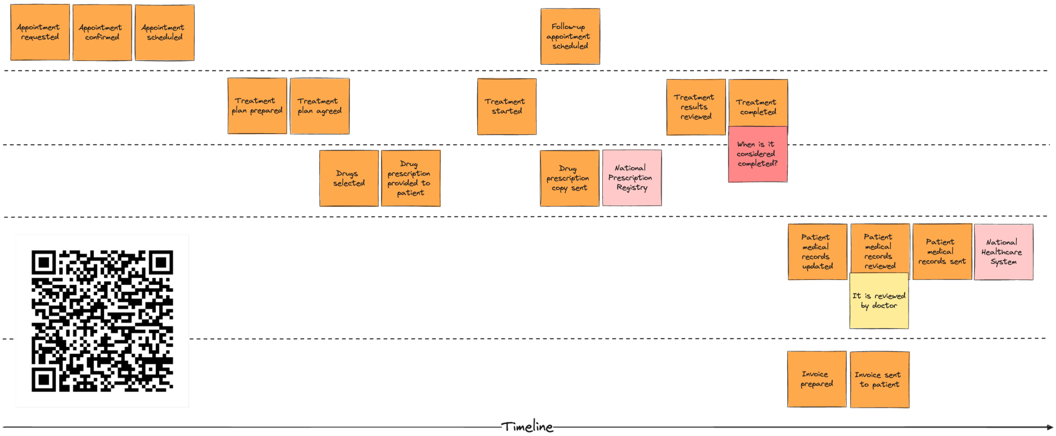

Appointment requested
Appointment confirmed
Appointment scheduled
Follow-up appointment scheduled
Treatment plan prepared
Treatment plan agreed
Treatment started
Treatment results reviewed
Treatment completed
When is it considered completed?
Drugs selected
Drug prescription provided to patient
Drug prescription copy sent
National Prescription Registry
Patient medical records updated
Patient medical records reviewed
It is reviewed by doctor
Patient medical records sent
National Healthcare System
Invoice prepared
Invoice sent to patient
Timeline

Figure 17. Grouping events using swim lanes

It is important to note that each group is highly cohesive.

High cohesion
elements within a group are closely related and focused on a specific purpose

or business capability.

For example, *Appointment Scheduling* is responsible for managing appointments but knows nothing about preparing or providing prescriptions, which is a responsibility of *Drug Prescription*.

During the grouping process, it is noted that clarification is needed regarding when treatment can be considered complete. Since no one can answer this question, you mark it with a red hot spot note to address later.

Additionally, it is unclear who should review patient medical records. One of the participants marks it with a comment note stating that doctors always do the review.

Other sticky notes include information about external systems that the modeled system needs to communicate with. These are marked with a pink external system note.

The last step in the workshop is to name the groups that you defined. It is up to the participants to decide what they want to name it. In the end, the diagram should look similar to the one below.

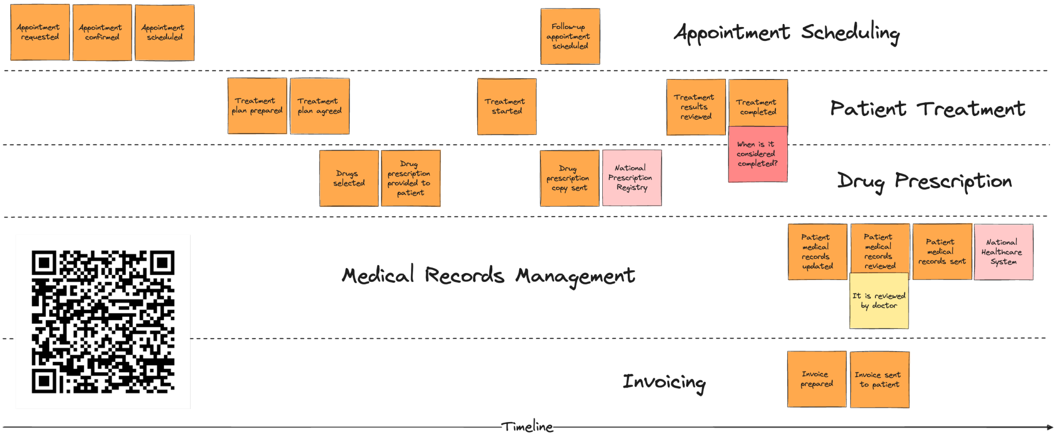

Appointment requested
Appointment confirmed
Appointment scheduled
Follow-up appointment scheduled
Appointment Scheduling
Treatment plan prepared
Treatment plan agreed
Treatment started
Treatment results reviewed
Treatment completed
When is it considered completed?
Patient Treatment
Drugs selected
Drug prescription provided to patient
Drug prescription copy sent
National Prescription Registry
Drug Prescription
Medical Records Management
Patient medical records updated
Patient medical records reviewed
It is reviewed by doctor
Patient medical records sent
National Healthcare System
Invoicing
Invoice prepared
Invoice sent to patient
Timeline

Figure 18. Results of the Big Picture workshop

That is it. We are done with the high-level workshop. The result will be used during an in-depth *Process Level* workshop.

Domain Storytelling

Domain Storytelling is another technique that can be used for high-level modeling. Unlike Event Storming, it allows you to capture and analyze domain knowledge through Storytelling. In simple terms, someone describes the process to you like a story, and you note it down as a diagram, ask questions, and share the current outcome with others. Each diagram represents a domain story.

In Event Storming, the central focus is on business events. Domain Storytelling, on the other hand, utilizes the interaction between people or systems using objects like:

- Phone
- Prescription
- Invoice

In essence, the difference lies in "what happened" with Event Storming versus "what happens" with Domain Storytelling. Teams often choose one or the other technique based on the desired outcome they aim to achieve. Sometimes it is a matter of preference. In one environment, people may prefer one, and in another, they may prefer the other.

> I recommend using a combination of both techniques. It is much easier for me to conduct Event Storming workshops when I provide a domain story to all participants before the workshop. As a result, I know that all participants understand the process and are adequately prepared for event-based modeling.

To handle the workshop on-site, you will need the following:

- **A calm place**. You must stay focused while storytellers describe the process to you.
- **A beamer**. During modeling, it is a great idea to continuously share the outcomes of the conversation. This way, you avoid misunderstandings because the storyteller can see what they describe.

- **Someone who will draw a diagram**. I noticed this during my workshops, especially when there are multiple storytellers. Focusing on what they say, creating diagrams, and asking questions all at once can be quite demanding. Therefore, it is a good idea to have a partner who can take over the diagram creation.

Another option is to conduct this workshop remotely. In addition to a communicator that allows you to talk to people, you will need an application that allows you to create diagrams. I like to use Egon[3], but again, you can select whatever tool you want.

If given the option to choose between on-site or remote, I prefer handling my workshops remotely. This way, I make sure that I stay in a calm place where no one can disturb me and entirely focus on my storyteller(s).

This technique works with domain stories. Each can be represented first and foremost by the degree of detail. It can be:

- **Coarse-grained**. At this level, you want to have just an overview of the domain or process and explore it without going into details.
- **Fine-grained**. Here, you want to do a deep dive into the process that was modeled in the first phase.
- **Something between**. Neither coarse-grained nor fine-grained. It shows the process with enough information. What does enough mean? It is up to you; there is no official definition of it.

For a high-level workshop to discover what the process of patient treatment looks like, it is recommended to decide for:

- Coarse-grained level
- As-is state

Sometimes, I choose to combine coarse and fine-grained levels for high-level workshops. I have found that I often miss a lot of helpful information when I focus only on the overview, when more detail might actually help me.

[3] https://egon.io

Before we start the modeling session, let's look at the fundamentals:

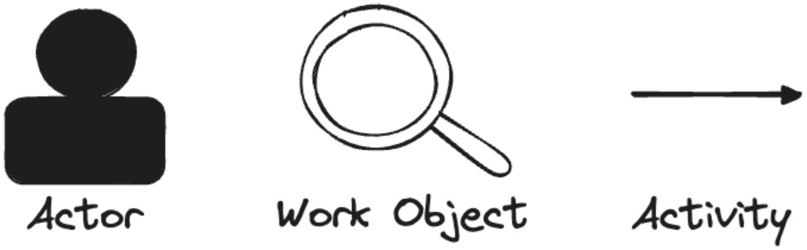


Figure 19. Basic elements for representing interactions

Actor represents one person, many people, or a system. They trigger actions and interact with each other using work objects. Their function or role defines them, so they should be represented by doctor and patient rather than James and Sophie.

Work Object is represented by a noun. It can be anything to interact with—a phone, prescription, drug, appointment, invoice, etc.

Activity is represented by a verb, pictured with an arrow. It describes the interaction between actors and work objects.

Each domain story is built on top of several sentences, which will be ordered on the diagram using a number representation. If this sounds complicated, don't worry too much; you will soon see how easy it is!

Imagine that you are at the workshop. There are domain experts who are going to describe the flow of our process:

> You: There is a patient who does not feel well. What is the first step?
>
> Storyteller: First, the patient has to call the front desk and ask to book an appointment. The front desk schedules the appointment.

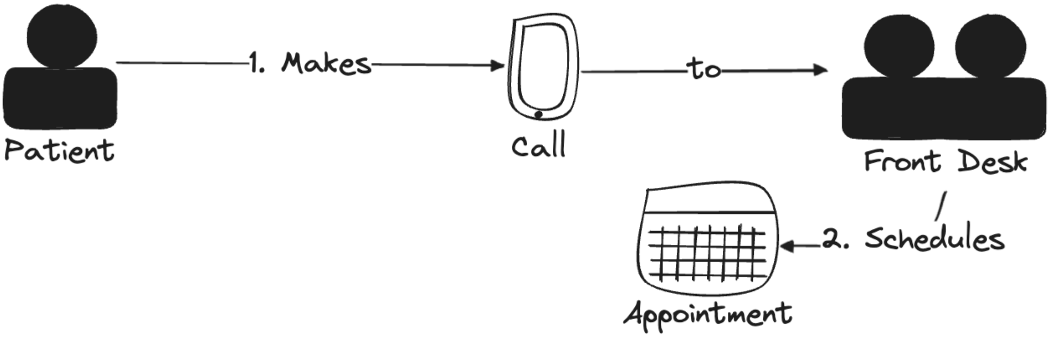


Figure 20. Patient calls front desk to schedule an appointment

Storyteller: Then, the appointment takes place, and once the examination is complete, the treatment starts. Additionally, a follow-up appointment is scheduled.

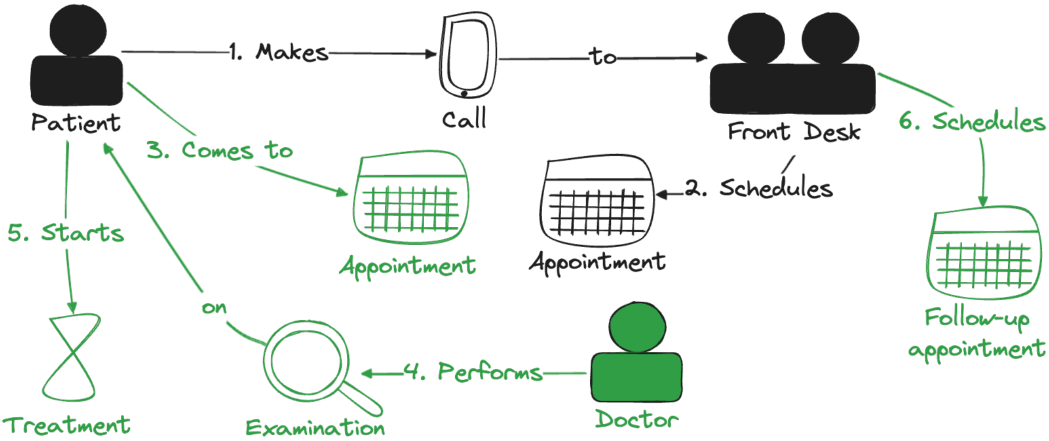


Figure 21. Appointment takes place and the treatment starts

You: What happens if the treatment does not work?

Storyteller: Then, it is reviewed and adjusted, and the patient receives new drugs.

We will not put this one on the coarse-grained diagram. This negative path can be modeled on a separate diagram during this or another session. Instead, you can mark it with a new element, the annotation. The rule of thumb is not to model negative paths on process overview diagrams.

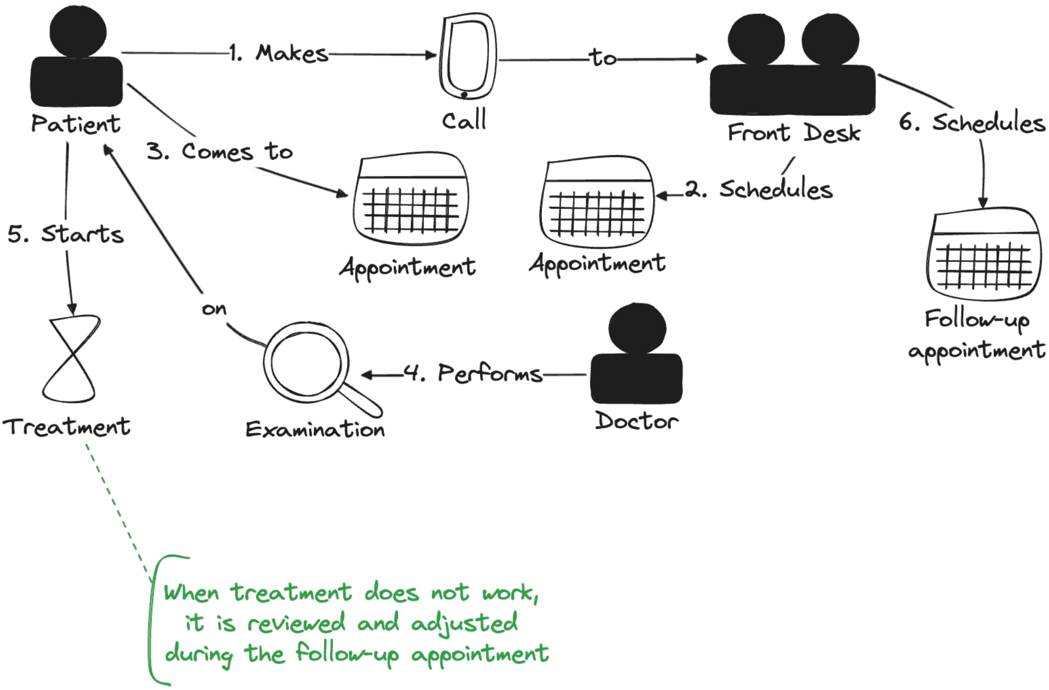


Figure 22. Document negative case with annotation

You: What is the next step?

Storyteller: The patient comes to a follow-up appointment, and the treatment results are reviewed. Next, the treatment is considered complete.

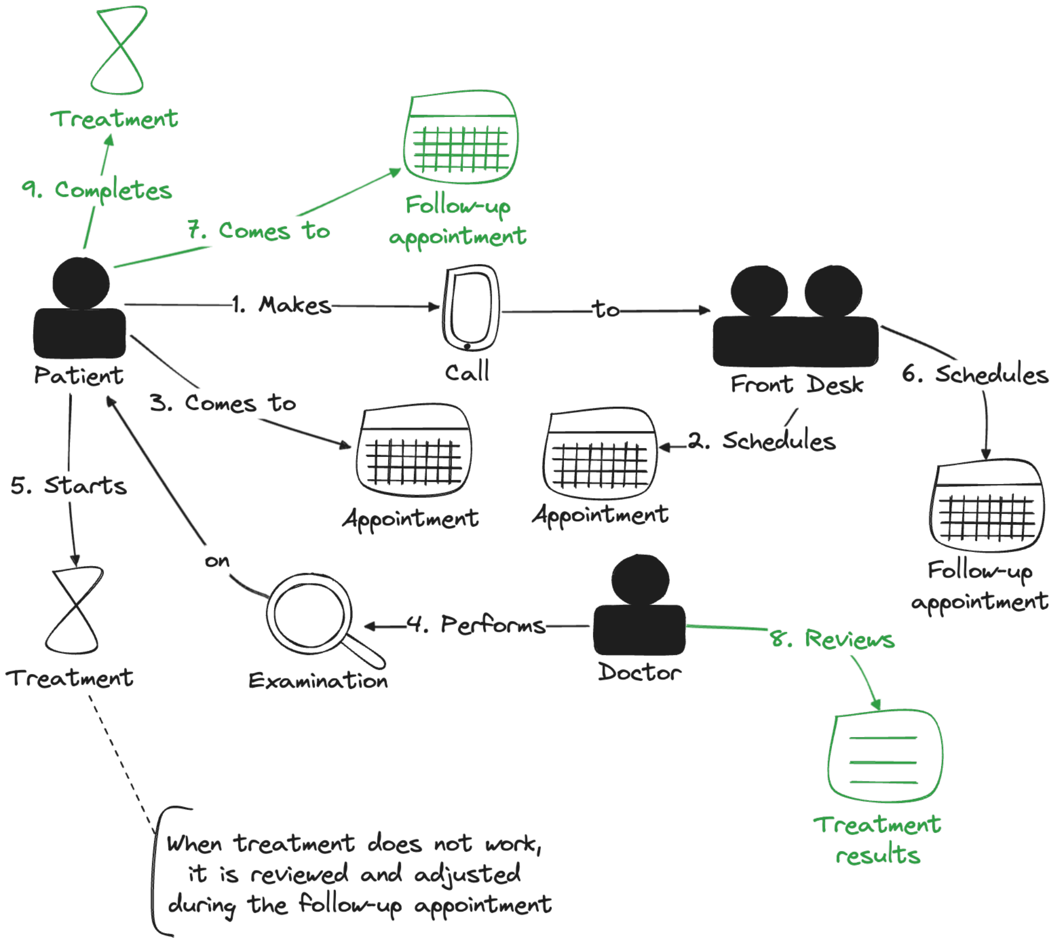


Figure 23. Patient completes the treatment

You: Is there anything else that should be considered?

Storyteller: Well, yes. When the treatment is completed, the front desk prepares the invoice and sends it to the patient.

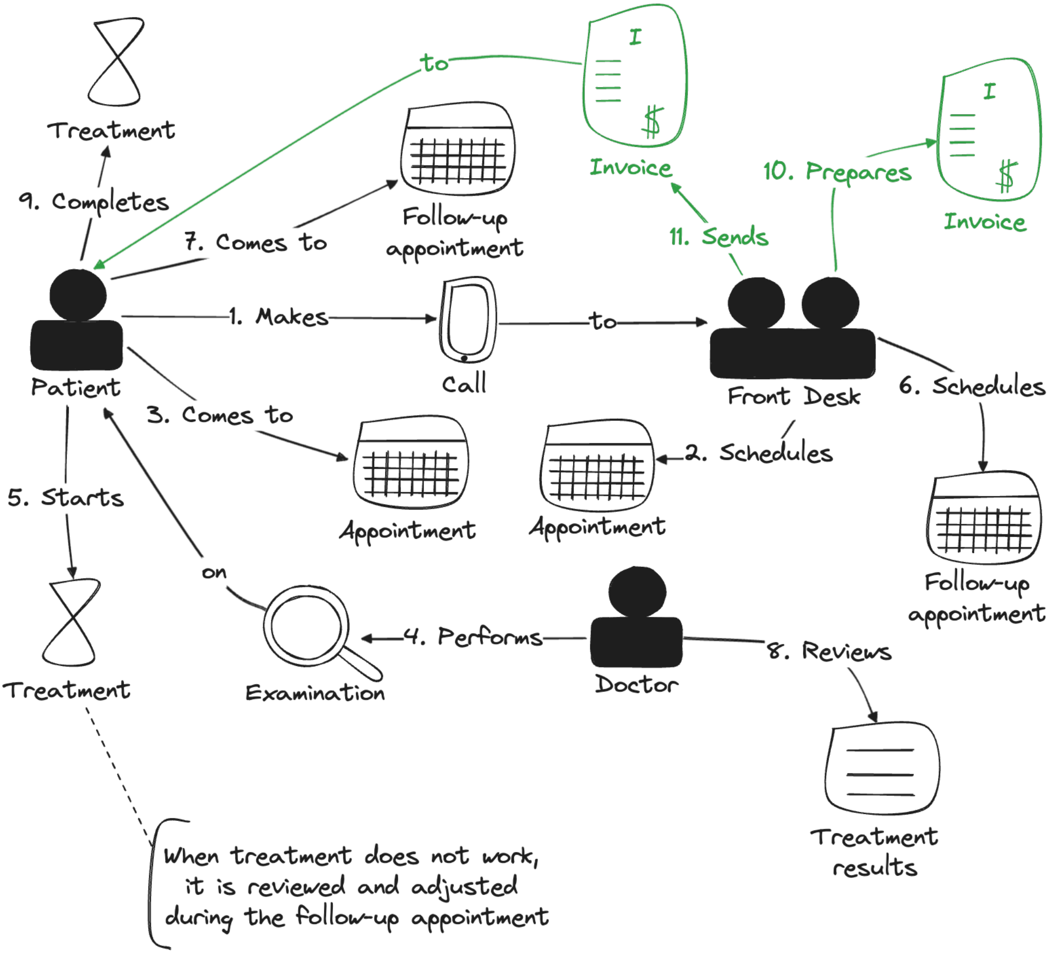


Figure 24. Front desk prepares and sends the invoice to the patient

With an overview of the entire patient treatment process, it is possible to find cohesive groups similar to what was done during Event Storming. The following indicators can help you:

- **Change of the activity**. As one type of activity finishes, another one starts. For example, an appointment is scheduled first, and then the patient is examined.
- **Change of the actor**. The patient completes the treatment, and the front desk starts to prepare an invoice.

- **Change of the object meaning**. Product means something different to a factory that produces it compared to a product that is an application developed by the IT startup.

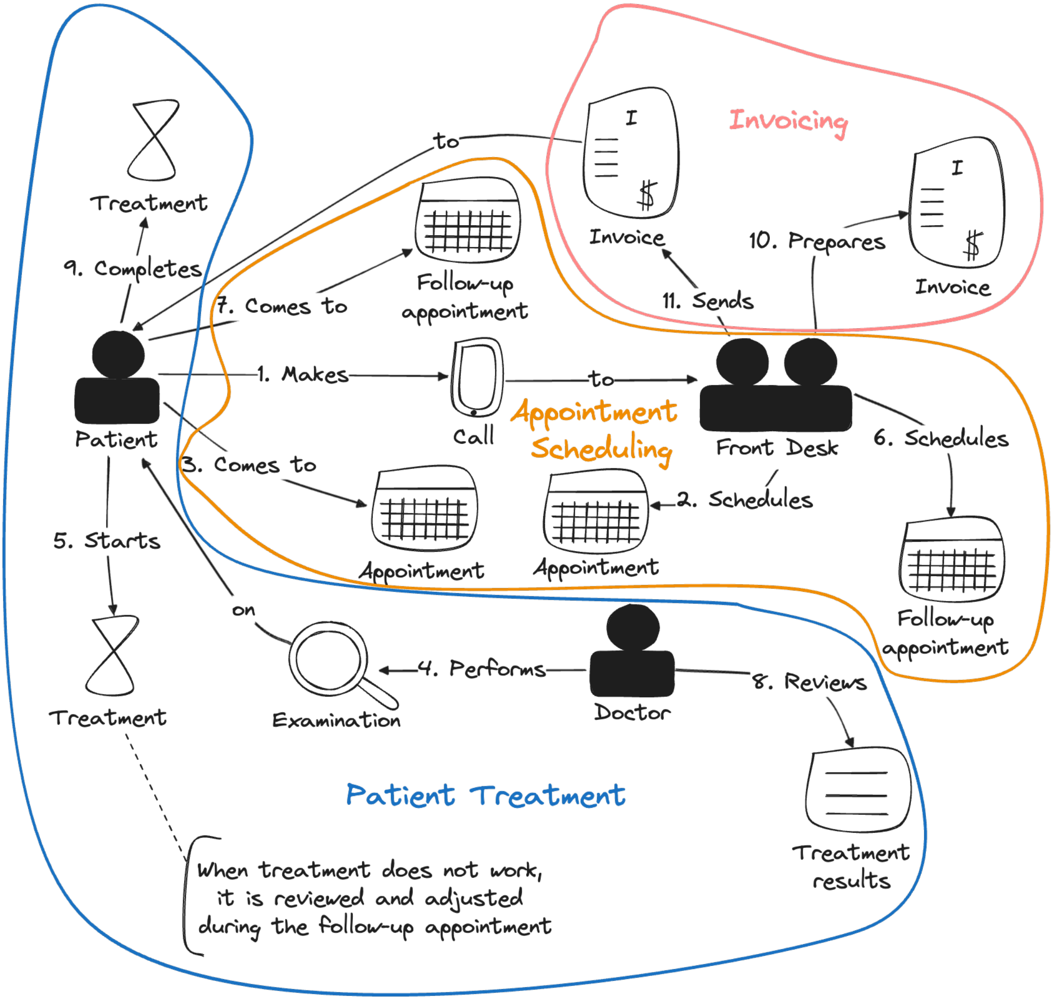


Figure 25. Grouping of interactions based on high cohesion

There is a problem—we completely missed two areas:

- Drug Prescription

- Medical Records Management

At the beginning of this section, I mentioned that focusing on a coarse-grained level can sometimes lead to such situations. The storyteller might not mention it because it is obvious to them, and the problem is that you did not ask enough questions. No worries; you can still fix it during the in-depth workshop.

Organize in-depth workshops

Regardless of the technique, the moment you realize that the high-level discovery of the process is complete, it is time to plan follow-up workshop(s) to analyze it further. Modeling will be much more challenging; here, you will focus on details.

The location, duration, and other organizational factors remain the same. Based on the previous outcome and your observations, you can adjust the number of people attending. You may also find that you invite people who did not participate earlier—that is fine.

Event Storming

It is time to shift our focus to *Process Level* workshops. These sessions are also highly collaborative, building upon the models created earlier. At this level, the goal is to examine the process in-depth, mostly focusing on its subprocesses to identify:

- Actors
- Commands
- Policies
- Read Models

This approach allows for a thorough and comprehensive understanding of the process by the end of the day. Before we begin modeling, I would like to introduce some new types of notes:

Actor

- **Color**: Usually yellow
- **Purpose**: Represents a person or role interacting with the domain. Actors can be the initiators of commands or recipients of information. Actors help identify who or what is interacting with the system, providing insights into user roles or other entities that play a part in the domain.

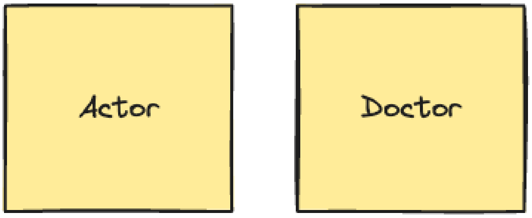


Figure 26. An example of an actor

Read Model

- **Color**: Usually green
- **Purpose**: Represents the result of a translation of a business event. It is the data a user or system needs to execute the action. Read models could still be a little bit vague, so treat them as a draft.

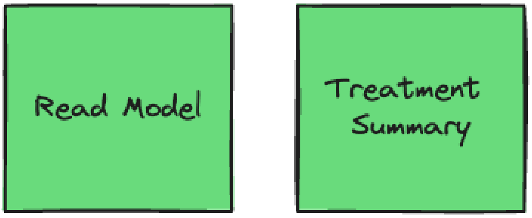


Figure 27. An example of a read model

Command

- **Color**: Usually blue
- **Purpose**: Represents an action taken by a user or system that changes the state of the domain. It is usually a verb or a verb phrase, like *Schedule appointment* or *Start treatment*. Commands are crucial for understanding what actions can be performed in the system and what triggers these actions.

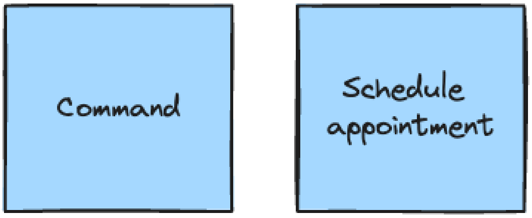


Figure 28. An example of a command

As you add commands, be sure to minimize situations where a command handles identical information as an event. The *Prepare treatment plan* command and the *Treatment plan prepared* event are excellent examples. Having both the command and the event repeat the same information creates unnecessary redundancy. Instead, you could have a *Analyze exam results and existing medical records* command that ends in the *Treatment plan prepared* event.

Policy

- **Color**: Usually purple
- **Purpose**: Represents a business policy that dictates what action should be taken under certain circumstances. Policies are typically derived from business rules or legal requirements. They guide the decision-making process within the domain and often lead to the generation of commands based on certain conditions.

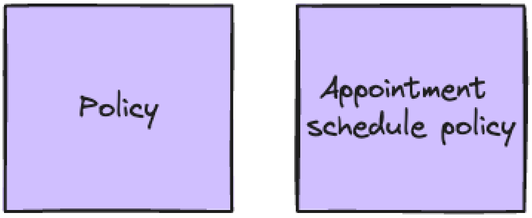


Figure 29. An example of a policy

With this knowledge, we can begin our workshop. The key participants are the domain experts who know the process inside out. Before we start, let's look at what we modeled before, as this is the starting point of the next step.

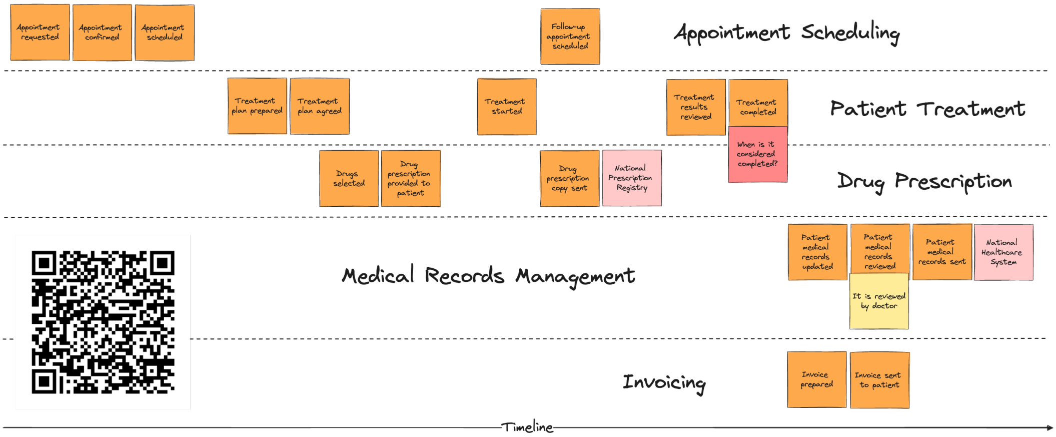

Appointment requested
Appointment confirmed
Appointment scheduled
Follow-up appointment scheduled
Appointment Scheduling
Treatment plan prepared
Treatment plan agreed
Treatment started
Treatment results reviewed
Treatment completed
When is it considered completed?
Patient Treatment
Drugs selected
Drug prescription provided to patient
Drug prescription copy sent
National Prescription Registry
Drug Prescription
Medical Records Management
Patient medical records updated
Patient medical records reviewed
It is reviewed by doctor
Patient medical records sent
National Healthcare System
Invoicing
Invoice prepared
Invoice sent to patient
Timeline

Figure 30. The summary of the Big Picture workshop

The initial operation is to schedule the appointment for the patient. There are three events: *Appointment requested*, *Appointment confirmed*, and *Appointment scheduled*. During the discussion, we can see that:

- The patient calls the front desk and asks for an appointment.
- The front desk confirms the appointment date with one of the doctors and then confirms it with the patient.
- The front desk schedules an appointment in one of the examination rooms.

- An appointment cannot be scheduled when a patient has no insurance.

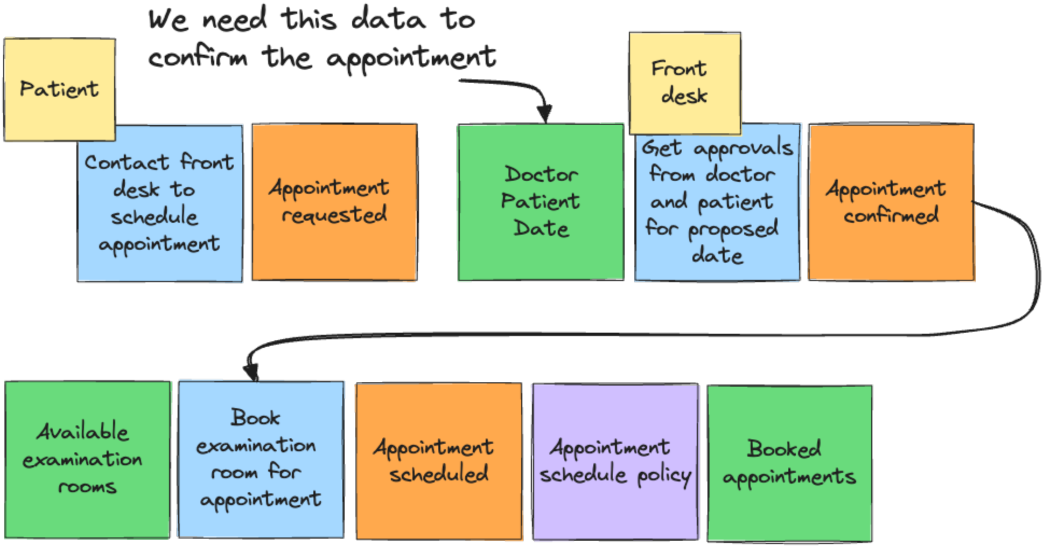

We need this data to confirm the appointment
Patient
Contact front desk to schedule appointment
Appointment requested
Doctor Patient Date
Front desk
Get approvals from doctor and patient for proposed date
Appointment confirmed
Available examination rooms
Book examination room for appointment
Appointment scheduled
Appointment schedule policy
Booked appointments

Figure 31. Front desk schedules an appointment for the patient

- **Actor** notes represent the patient and the front desk.
- **Read Model** notes represent the doctor, patient, available date, the list of available examination rooms, and the list of booked appointments.
- **Command** notes represent actions taken by the patient and the front desk.
- **Policy** note represents the rule that an appointment cannot be scheduled without the patient's insurance.

When the appointment takes place, the doctor prepares the treatment plan with the patient and prescribes needed drugs before the treatment starts:

- The doctor prepares the treatment plan based on the patient's examination result and existing medical records.
- The patient has to agree to the proposed plan before treatment can begin.
- The doctor looks at the list of available drugs and selects the right ones.
- If selected drugs have no contraindications for the patient, the doctor provides the drug prescription to the patient. Treatment starts.

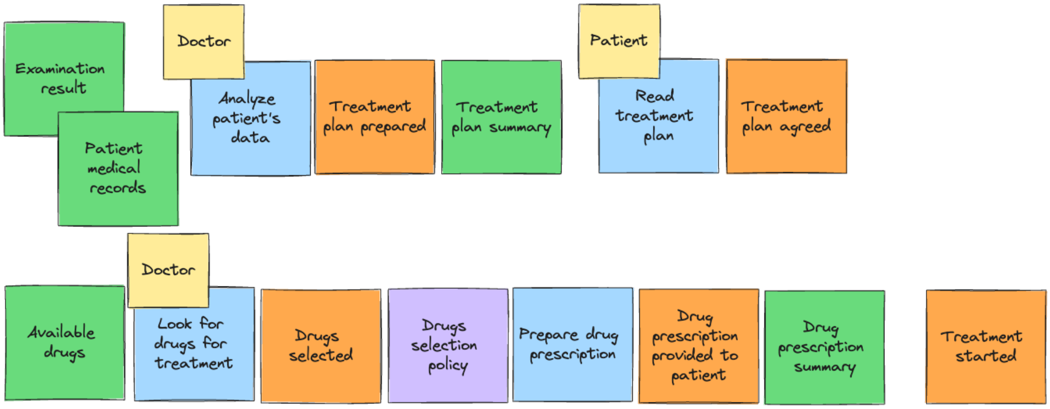


Figure 32. Start of the patient treatment

- **Actor** notes represent the doctor and the patient.
- **Read Model** notes represent the examination result, the patient's medical records, available drugs, drug prescription summary, and the treatment plan summary.
- **Command** notes represent actions taken by the doctor and the patient.
- **Policy** note represents the rule that selected drugs have no contraindications for the patient.

When the appointment is finished, the front desk schedules the follow-up appointment based on the doctor's and patient's availability.

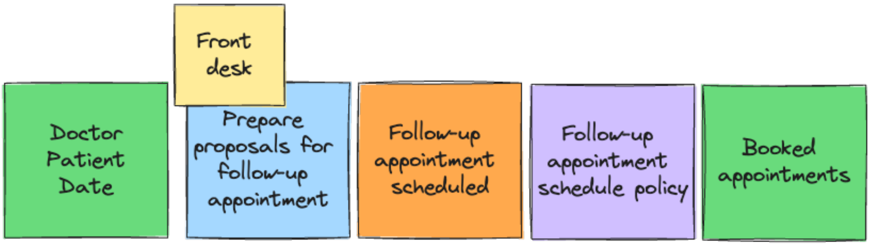

Doctor
Patient
Date
Front desk
Prepare proposals for follow-up appointment
Follow-up appointment scheduled
Follow-up appointment schedule policy
Booked appointments

Figure 33. Front desk schedules a follow-up appointment for the patient

- **Actor** note represents the front desk.

- **Read Model** notes represent the doctor, patient, available date, and the list of booked appointments.
- **Command** note represents action taken by the front desk to schedule the follow-up appointment.
- **Policy** note represents the rule that a follow-up appointment cannot be scheduled without a prior appointment.

As you may have noticed, there is a chance that the follow-up appointment scheduling process is the same as the appointment scheduling process. If you have such a feeling, clarify it during the workshop - it can simplify the work that must be done, as you might reuse the same process.

In the meantime, the front desk prepares a copy of the prescription and sends it to the national prescription registry. It is done once at the end of the day.

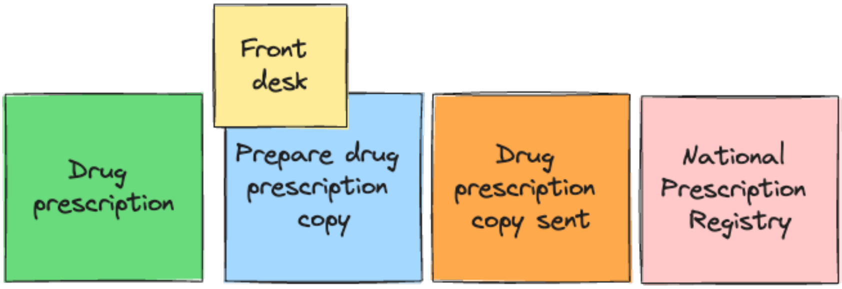

Front desk
Drug prescription
Prepare drug prescription copy
Drug prescription copy sent
National Prescription Registry

Figure 34. Front desk sends copy of drug prescription to national registry

- **Actor** note represents the front desk.
- **Read Model** note represents the existing drug prescription.
- **Command** note represents action taken by the front desk to prepare and send a copy of the drug prescription.

Next, when the follow-up appointment takes place:

- The doctor examines the patient and analyzes the treatment and examination results.
- If everything is fine, the doctor completes the treatment and prepares a summary.

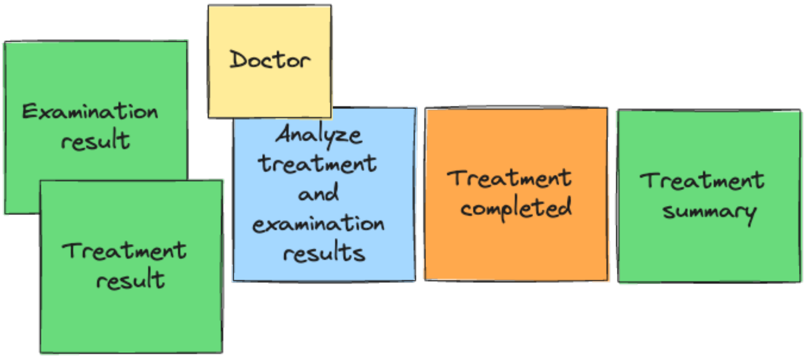


Figure 35. Doctor analyzes the results and prepares a summary

- **Actor** note represents the doctor.
- **Read Model** notes represent the patient examination result, the treatment result, and the treatment summary.
- **Command** notes represent actions taken by the doctor to analyze treatment and examination results and complete the treatment.

One of the events - *Treatment results reviewed* - from previous workshop was dropped. It makes no sense to keep it because the *Analyze treatment and examination results* command already describes the action taken by the doctor.

When the treatment is completed, two subprocesses are carried out simultaneously. First, patient medical records are updated and sent to the national healthcare system:

- Based on the treatment summary, the front desk updates patient medical records.

- The doctor reviews and approves them.
- The updated records are automatically sent to the national healthcare system.

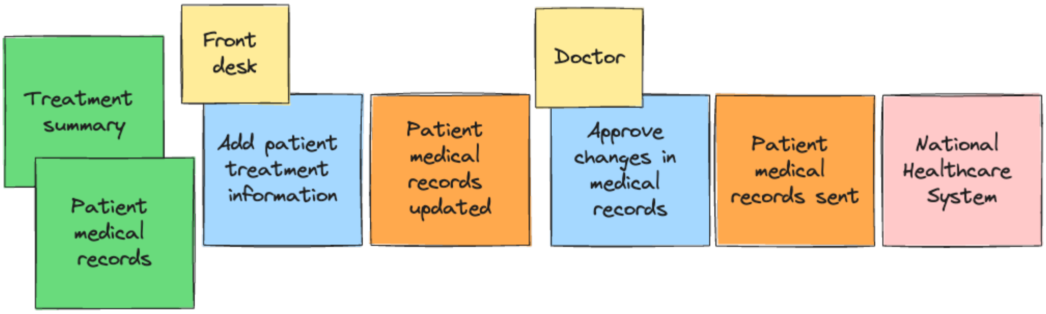

Treatment summary
Patient medical records
Front desk
Add patient treatment information
Patient medical records updated
Doctor
Approve changes in medical records
Patient medical records sent
National Healthcare System

Figure 36. Update and send the actual medical records to the national healthcare system

- **Actor** notes represent the front desk and the doctor.
- **Read Model** notes represent the treatment summary and the patient medical records.
- **Command** notes represent actions taken by the front desk and the doctor to update, review, and send medical records.

The second subprocess, which is triggered when the treatment is completed, is about preparing and sending the invoice to the patient:

- The front desk prepares the invoice based on the treatment summary.
- The front desk sends the invoice to the patient.

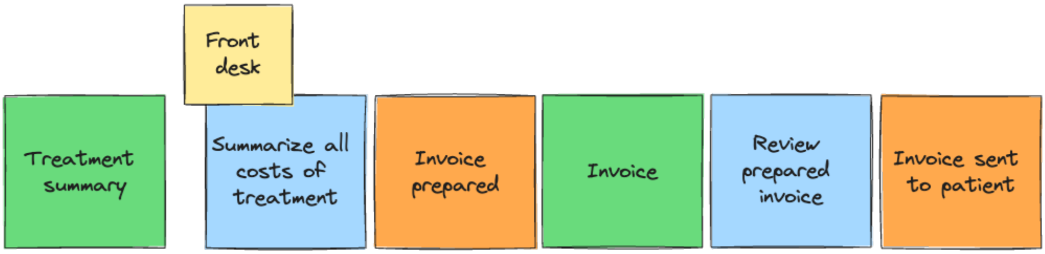


Figure 37. Front desk prepares and sends the invoice to the patient

- **Actor** note represents the front desk.
- **Read Model** notes represent the treatment and invoice summaries.
- **Command** notes represent actions taken by the front desk to summarize all treatment costs and review the invoice.

The entire patient treatment process is now completed, but there is one more thing to mention. As you may recall from the previous workshops, a question was raised: *What does it mean that treatment is completed?* This was addressed in the current workshop, and that is why there is no hot spot anymore.

If there is still no clear answer, leave it there. Of course, other hot spots might also occur during the current workshop. In such cases, add them to the current diagram.

When you look at the wall, it should look similar to this:

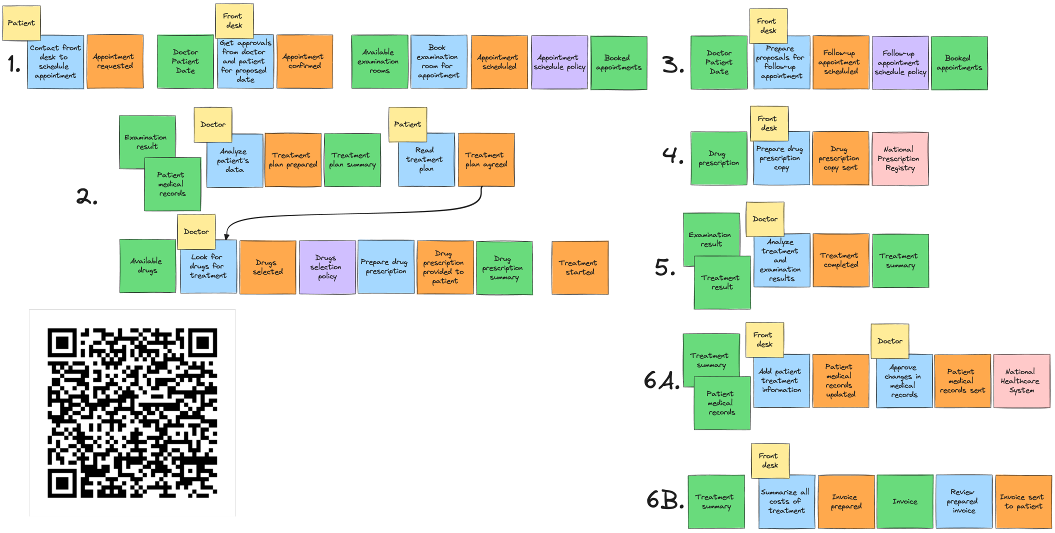

1.
Patient
Contact front desk to schedule appointment
Appointment requested
Doctor Patient Date
Front desk
Get approvals from doctor and patient for proposed date
Appointment confirmed
Available examination rooms
Book examination room for appointment
Appointment scheduled
Appointment schedule policy
Booked appointments
2.
Examination result
Patient medical records
Doctor
Analyze patient's data
Treatment plan prepared
Treatment plan summary
Patient
Read treatment plan
Treatment plan agreed
Available drugs
Doctor
Look for drugs for treatment
Drugs selected
Drugs selection policy
Prepare drug prescription
Drug prescription provided to patient
Drug prescription summary
Treatment started
3.
Doctor Patient Date
Front desk
Prepare proposals for follow-up appointment
Follow-up appointment scheduled
Follow-up appointment schedule policy
Booked appointments
4.
Drug prescription
Front desk
Prepare drug prescription copy
Drug prescription copy sent
National Prescription Registry
5.
Examination result
Treatment result
Doctor
Analyze treatment and examination results
Treatment completed
Treatment summary
6A.
Treatment summary
Patient medical records
Front desk
Add patient treatment information
Patient medical records updated
Doctor
Approve changes in medical records
Patient medical records sent
National Healthcare System
6B.
Treatment summary
Front desk
Summarize all costs of treatment
Invoice prepared
Invoice
Review prepared invoice
Invoice sent to patient

Figure 38. Results of the Process Level workshop

As you can see, Event Storming is relatively easy, but there are some nuances around it. To deepen your knowledge about it, I encourage you to read this book[4].

[4] https://leanpub.com/introducing_eventstorming

Domain Storytelling

In Domain Storytelling, you can also focus on diving deep into the process using fine-grained diagrams. There are no new objects; everything will be based on the previous workshop. However, my advice is not to edit the coarse-grained diagram. It will take much more time than creating it from scratch. Also, looking at the high-level process on the side is helpful.

The approach I recommend for handling in-depth workshops is to focus on something between coarse-grained and fine-grained, and on the current state (as-is state).

As before, the workshop will include domain experts and your assistant, who will create diagrams while you interact with the storyteller(s). Unlike high-level sessions, fewer participants are expected, as many may lack detailed knowledge of the process.

Everyone has been invited, and the session starts:

> You: There is a patient who does not feel well. What is the first step?
>
> Storyteller: First, the patient has to call the front desk and ask to book an appointment. The front desk looks for available appointment dates and confirms the doctor's and patient's availability. Next, the appointment is scheduled.

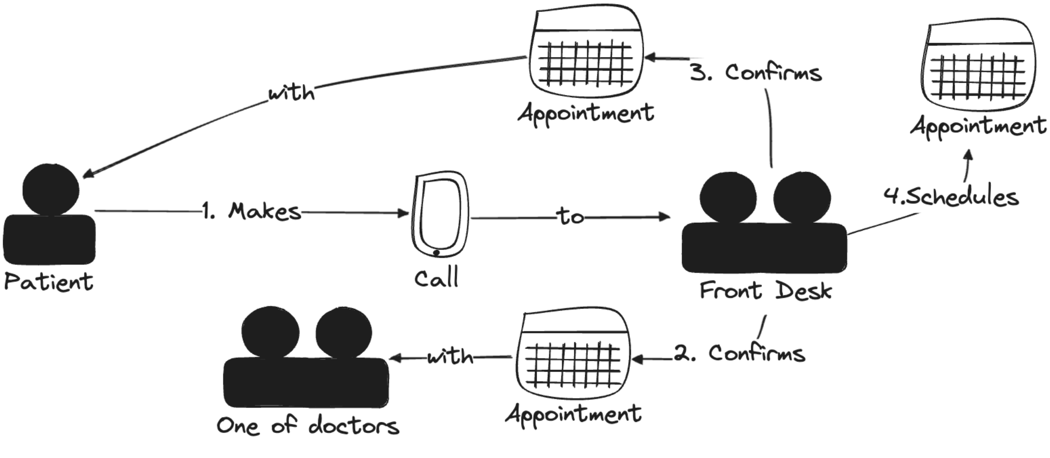


Figure 39. Patient calls front desk to schedule an appointment

Storyteller: The appointment takes place. The patient is examined, and the doctor prepares the treatment plan based on the patient's medical records. Once the plan is accepted by the patient, the doctor prescribes drugs and provides the drug prescription to the patient. Treatment starts, and additionally, the front desk schedules a follow-up appointment.

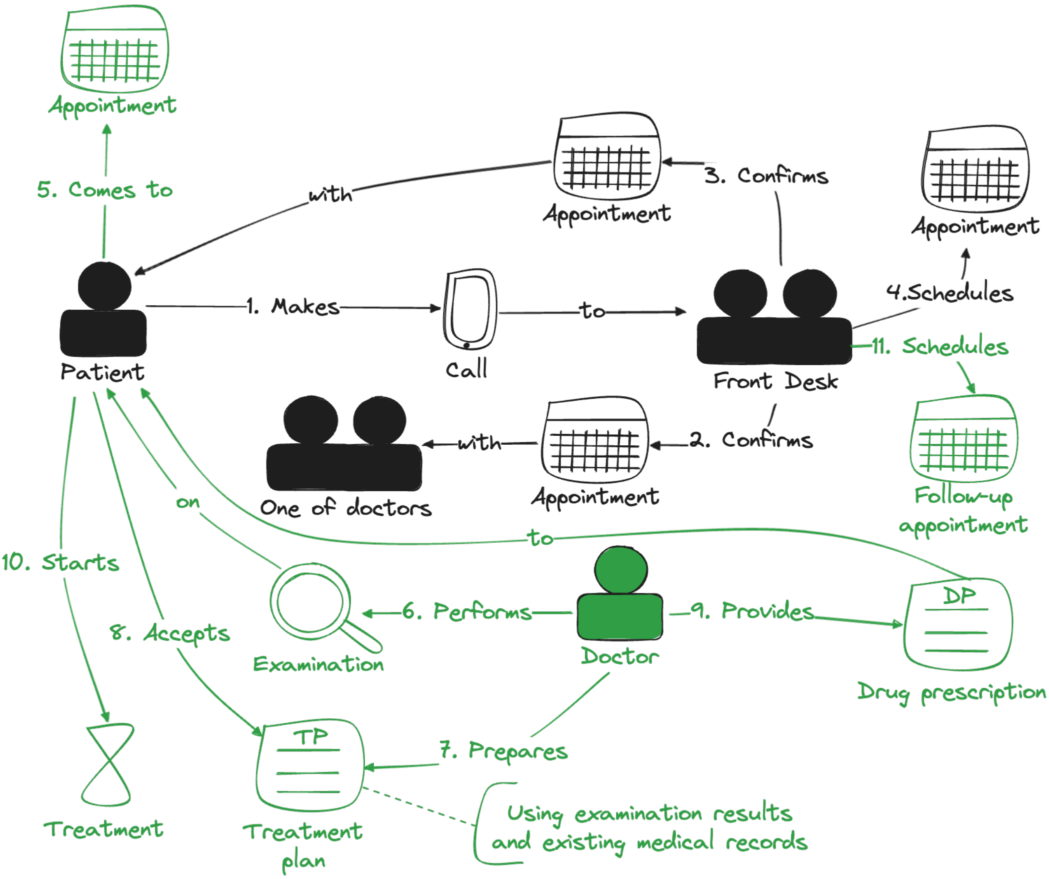


Figure 40. Appointment takes place and the treatment starts

Storyteller: Meanwhile, the front desk sends a copy of the drug prescription to the national prescription registry. It is required due to regulations.

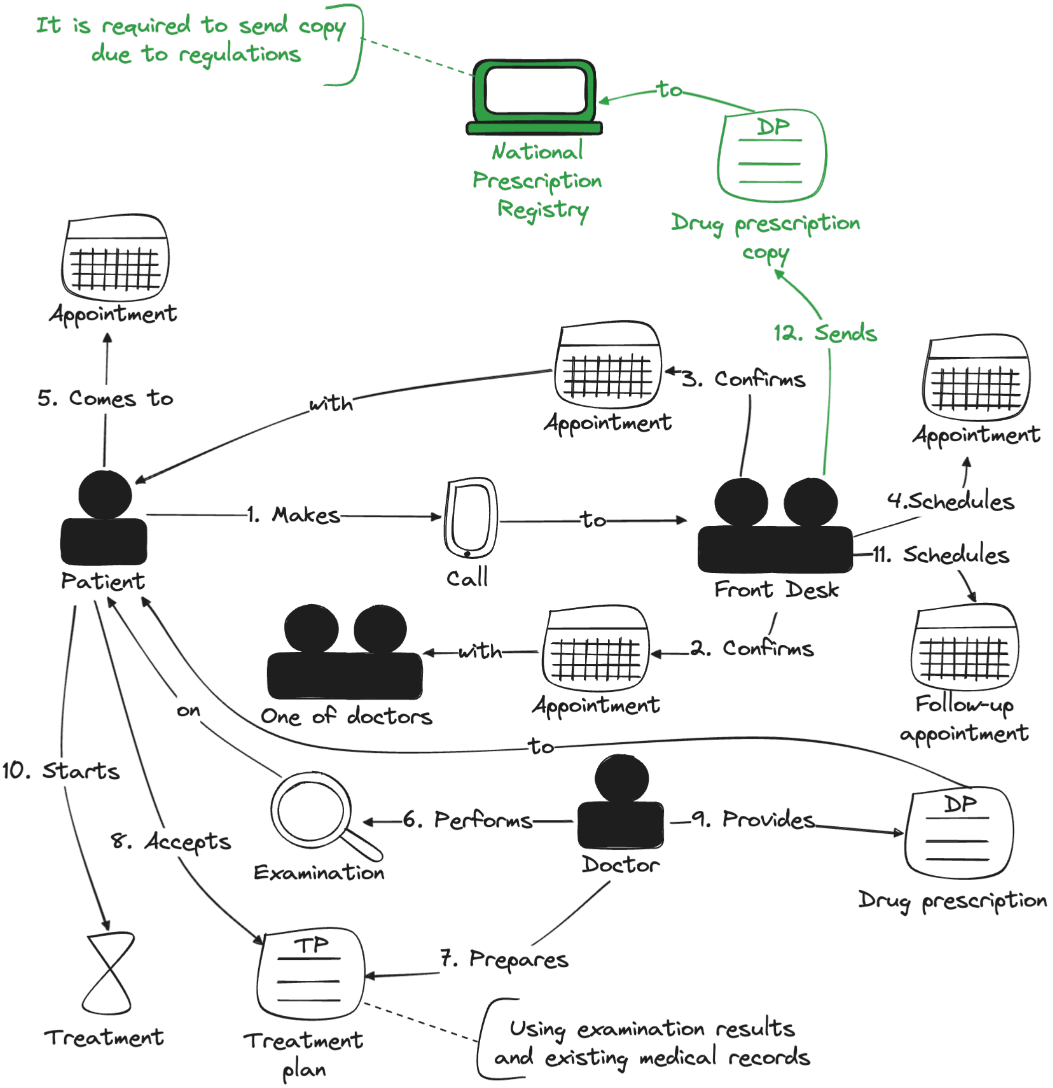


Figure 41. Front desk sends the drug prescription copy to the national prescription registry

You: What happens next?

Storyteller: The patient attends the follow-up appointment. The doctor examines the patient and reviews the treatment results. If everything is okay, the patient completes the treatment.

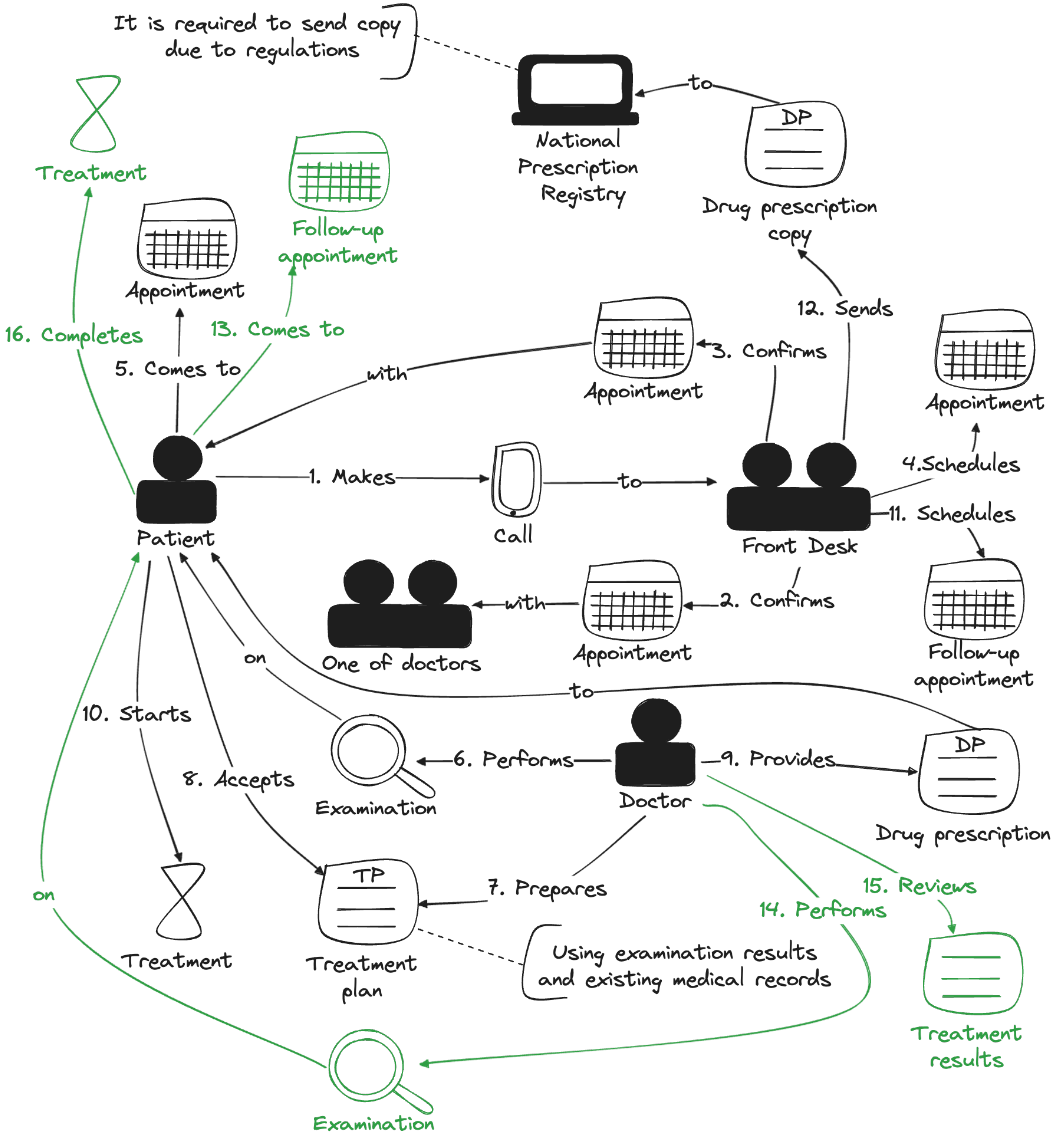


Figure 42. Patient completes the treatment

Storyteller: When the treatment is completed, two things happen. First, the front desk updates the medical records, which the doctor then reviews and approves. When done, it is automatically sent to the national healthcare system. Second, the front desk prepares and sends the invoice to the patient.

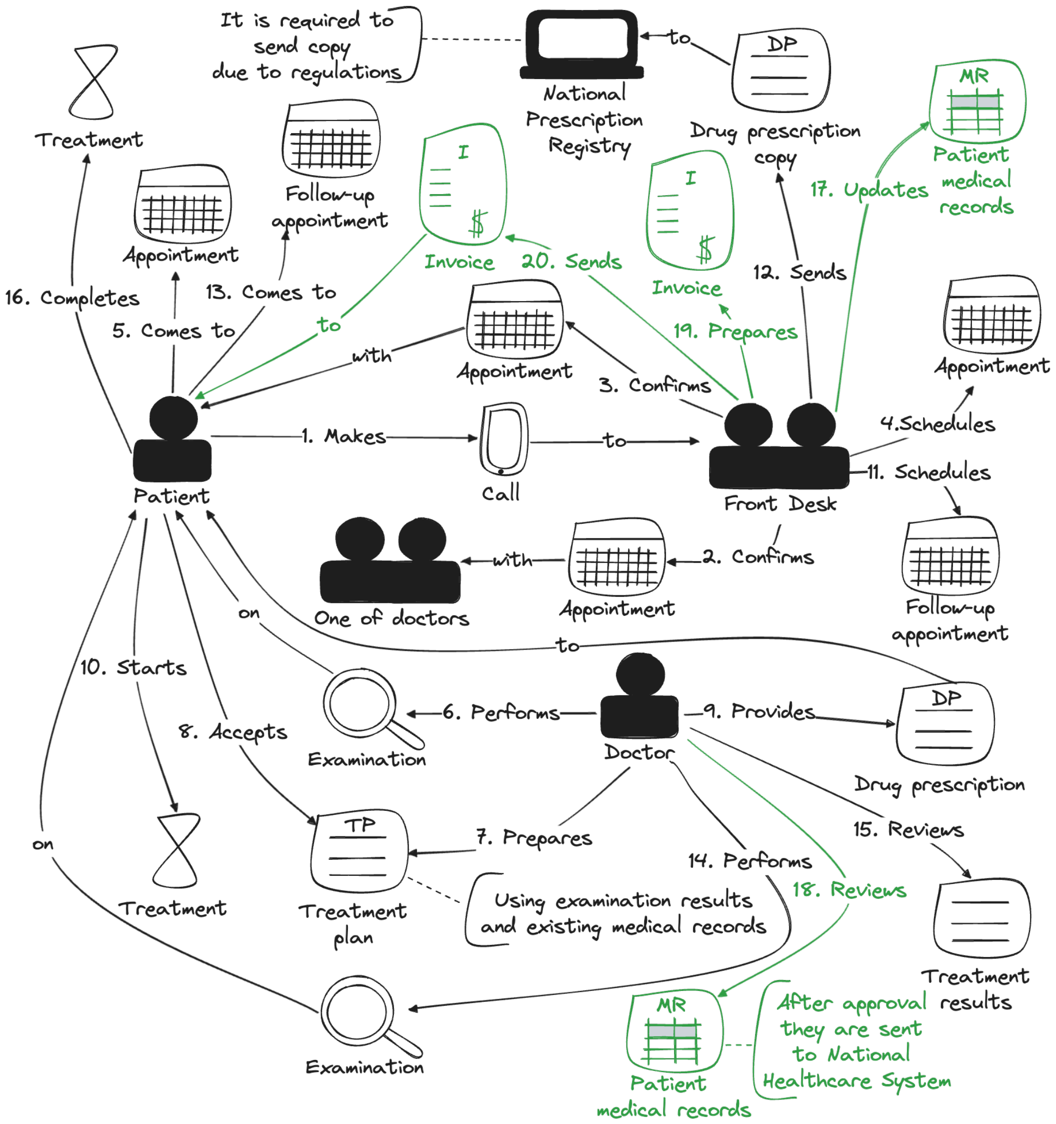


Figure 43. Closure of the patient treatment process

Anyone who receives the diagram from us should be able to read it without problems just by following the numbers. You can also mark and name all the cohesive areas (like in high-level workshops).

This time, you discovered all the information related to:

- Appointment Scheduling
- Patient Treatment
- Medical Records Management
- Drug Prescription
- Invoicing

At the end of the workshop, everyone should have a clear understanding of the modeled process. All concerns should be marked with annotations, and if further modeling of negative cases is needed, you should plan another session.

Domain Storytelling has much to offer, and we have only touched on some of its areas. The ones that I showed you should be enough in most cases, but if you would like to learn more about it, there is a must-read book[5] from the authors of this technique.

Combine both techniques

Event Storming and Domain Storytelling can be used independently for business domain discovery. Both techniques are collaborative and easily understandable for workshop participants. They excel in discovering business capabilities, although other approaches may be more suitable in highly technical domains.

Event Storming is particularly effective for visualizing the flow of events and understanding the sequence of actions within a domain.

Pros	Cons
Very easy to understand by both technical and business people.	May lead to oversimplified diagrams.

[5] https://www.goodreads.com/book/show/58385794-domain-storytelling

Pros	Cons
Extreme collaboration level between stakeholders, developers, and domain experts, which can uncover misunderstandings early.	Time-consuming, especially when multiple stakeholders are involved.
It is easy to note what happens in a single process.	In large and complex systems that are built on top of many processes, it might be tricky to have a clear overview of what happens.

Domain Storytelling provides insights into systems, users, their roles, and interactions with objects in the domain's context.

Pros	Cons
Valuable for understanding user interactions and behaviors.	May not provide sufficient technical detail.
Relies on natural language descriptions.	Not everyone is a good storyteller.
As humans, we are used to listening to stories.	Stories are subjective, which can lead to different interpretations.

I used to run workshops using either Domain Storytelling or Event Storming, depending on the context and preferences of participants. If participants preferred describing the domain through storytelling, I chose Domain Storytelling; otherwise, I opted for Event Storming workshops.

However, at some point, I noticed I was missing some information that could have been discovered using the other technique. I reviewed my approach and decided to mix both (when possible). This brought fantastic results!

How do I do it?

My first step in domain discovery is to invite domain experts for high-level sessions where we focus on a few critical processes. For each process, I use Domain Storytelling. I meet with them in a room, listen to what they have to say, model it so they can see it (e.g., using a beamer), ask questions, and in the end, I have a whole domain story in the form of a diagram. I use something between coarse and fine-grained— like in in-depth chapter.

Next, I organize the *Big Picture* (Event Storming) workshop. At the start, I always ask participants to look at the outcome of the Domain Storytelling session and try to understand the process. In the past, I would send it via email before the workshop session, but it did not work well; some participants forgot about it, missed it entirely, or lacked the time to review it. Therefore, dedicating 10-15 minutes of the workshop session to this review could be more effective.

If you want to have an overview of the entire process, I think there is no better way than Domain Storytelling diagrams. But if you want to focus on details, model negative paths, etc., I find Event Storming a better fit.

Other ways can also be used for domain discovery, such as UML or Storystorming. UML's complexity makes it less suitable for conducting collaborative workshops, especially with non-technical people. However, when it comes to documenting the domain afterward, it works perfectly well.

Storystorming combines several techniques, so it gives you all the tools you need. Unfortunately, I haven't yet used it in practice, so it is not covered in this book. If you want to explore it, you can find information about it here[6].

[6] https://storystorming.com/

⚠ Remember, no technique guarantees complete coverage of all the business logic in your application. Some areas will inevitably remain unexplored. Even thorough techniques may leave gaps in addressing all aspects of your business needs.

Use knowledge from workshops

A common question I receive after workshops is:

> *Well, OK, we have discovered the domain, but how can we use the outcome?*

First, it would be good to retain this knowledge, so I recommend taking pictures after each workshop or downloading and uploading diagrams to a repository or some other commonplace. My suggestion is to keep it as close to the application as possible, so a repository is a good choice.

The next step is to prepare a map of the business domain based on the discovered areas. This is where strategic Domain-Driven Design comes in. It answers the following questions:

1. How is the business domain divided (subdomains)?
2. What are the boundaries (bounded contexts)?
3. How are all the pieces interconnected (context map)?

The knowledge you will gain from this section will allow you to properly modularize your system, whether you decide to go for microservices, modular monolith, or anything else.

Subdomains

During workshops, you distilled groups of events and separated them by swim lanes. Then, the next step was to name these groups.

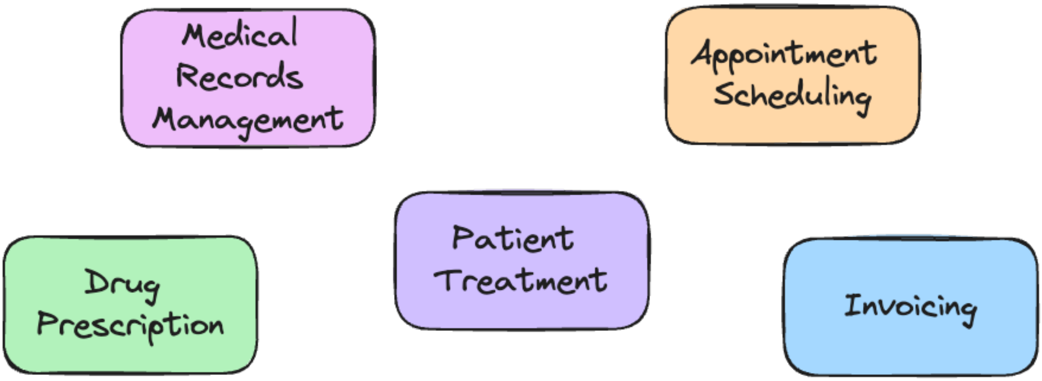


Figure 44. Groups we have found during our workshops

They might cover the entire process from the beginning till the end:

- **Appointment Scheduling**. It covers requesting the appointment, confirmation, and scheduling.
- **Invoicing**. It covers the preparation and sending of the invoice to the patient.
- **Medical Records Management**. It covers updating patient's medical records and informing the national healthcare system about changes.

There are also cases where multiple groups can be involved in a single process. For example, starting patient treatment requires examining the patient and preparing the treatment plan (*Patient Treatment*), selecting drugs, and providing a drug prescription (*Drug Prescription*).

Each group represents unique business capabilities—*Appointment Scheduling* cannot handle sending invoices, and *Drug Prescription* cannot schedule an appointment. Furthermore, they operate on various objects—the first knows nothing about drugs, and the latter knows nothing about appointments.

In Domain-Driven Design nomenclature, such a group is called a *subdomain.*

Each business domain is built on top of multiple subdomains. Each subdomain represents its cohesive part.

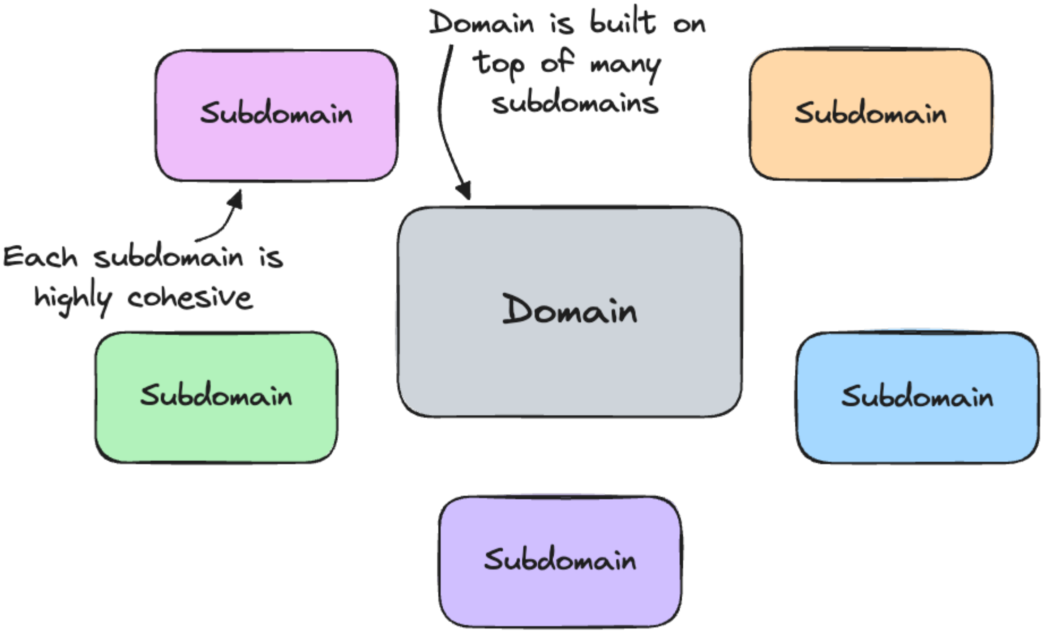

Domain is built on top of many subdomains
Subdomain
Subdomain
Each subdomain is highly cohesive
Domain
Subdomain
Subdomain
Subdomain

Figure 45. Business domain and its subdomains

These subdomains allow you to discover and understand your domain step-by-step rather than focusing on the entire domain at once.

⚠ I caution against taking shortcuts and defining subdomains based solely on departments within the company. More often than not, such assumptions are incorrect, as processes usually cross several departments. If there is organizational chaos in your company, you will transfer it to your software.

When analyzing and splitting complex domains, you need to be very careful. You may discover that one of the distilled subdomains covers dozens of processes, and it makes sense to divide it further into other subdomains. In such a case, you are probably dealing with a completely separate domain. Remove it from the workshop and plan a follow-up. It is not uncommon to develop a separate application for such domains.

To illustrate it, let’s take a look at this diagram:

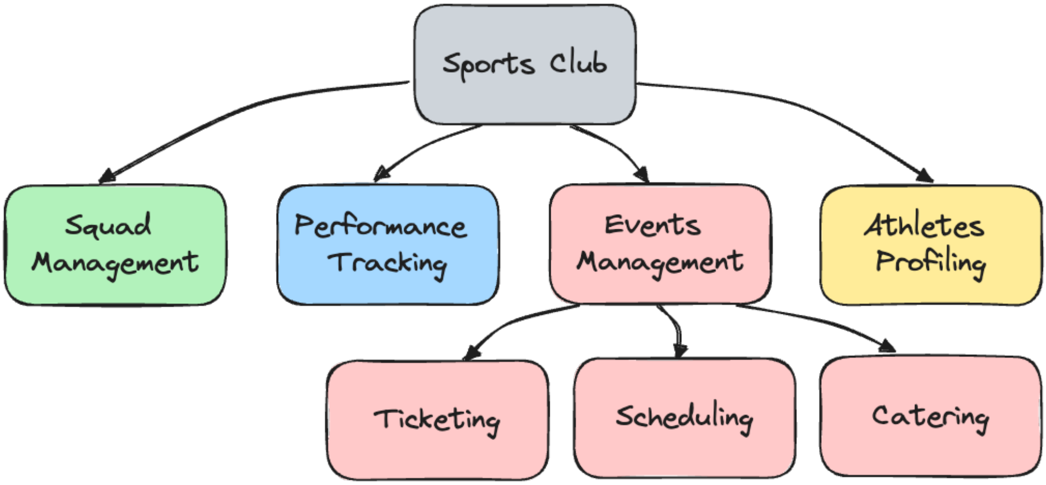


Figure 46. A complex domain where one of its subdomains is another domain

The diagram represents a sports club domain. During workshops, you discovered subdomains related to profiling athletes, tracking their performance, managing the squad, and managing events.

You start digging deeper into *Events Management* and see that it covers dozens of processes that can be further divided into *Ticketing*, *Scheduling*, and *Catering*. This is the moment to pause, remove it from the current workshop, and plan to handle it in another session, most probably with different domain experts.

Over time, as you gain more experience in distilling subdomains, you will spot recurring patterns. There are popular subdomains like invoicing or payments, which will appear in various domains, although they will not always be resolved in the same way. Sometimes, you can use an off-the-shelf solution; other times, you must implement it in-house due to the process's complexity and uniqueness.

Bounded contexts

When subdomains are already distilled, it is time to think about how to build a software around them.

What you need is a model to represent them. One of the first thoughts might be:

> *Let's create one unified model for the entire domain!*

You begin building it, then realize that each subdomain has its own characteristic terms, for example, *Treatment*, *Prescription*, or *Appointment*. Additionally, there is a chance that the same term might be used in another subdomain, with a completely different meaning.

Imagine having two subdomains of *Drug Prescription* and *Addiction Treatment*. The term *Drug* won't have the same meaning in each of them. The model you created contains only one definition for it (a medication used to cure patients) but will be used interchangeably in other contexts (stimulants).

This will lead to problems with application maintenance, as you have to distinguish between different meanings inside one model, creating a complex and tangled system, often referred to as a *big ball of mud*[7] in software development. In a nutshell:

Big ball of mud

> A state of the application where everything is tightly coupled, duplicated and messy, causing changes in one part to affect multiple others. This makes the application extremely difficult to maintain and evolve.

Another approach is to divide the system into multiple models. Each model will represent its ubiquitous language, business capabilities, and rules. This approach allows models to be extended independently without affecting the others.

Let's take a more practical example. You need to create an application that focuses on the following team sports:

- Football (soccer)
- Basketball
- Volleyball

[7] http://www.laputan.org/mud/mud.html#BigBallOfMud

You decide to create a single, unified model for them. After all, all of these sports have coaches, players, squads, and tactics.

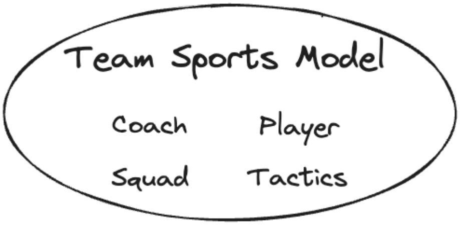


Figure 47. Team sports represented as a single unified model

Everything is fine so far, but you need to add a game duration concept.

- **Football.** 90 minutes, divided into two halves, possible overtime for each half of the game.
- **Basketball.** 48 minutes, divided into four quarters, the time can be paused.
- **Volleyball.** No time limit; the game is divided into sets. The first team to win three sets wins the game.

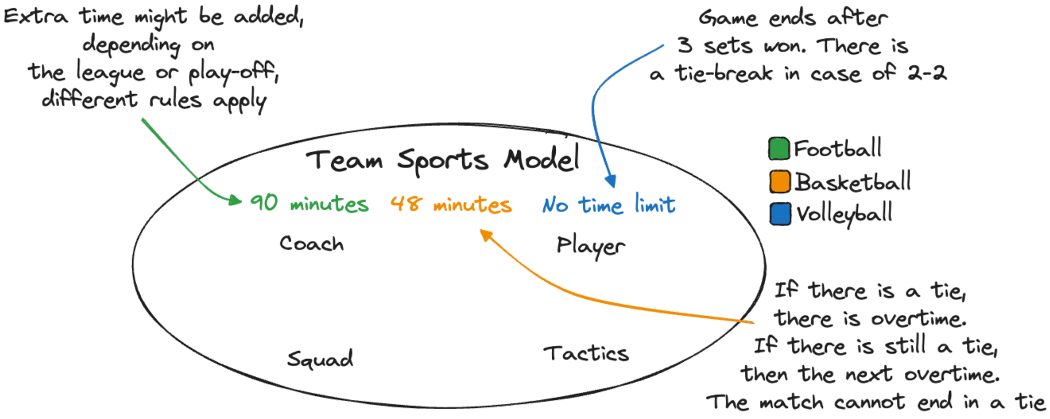

Extra time might be added, depending on the league or play-off, different rules apply
Game ends after 3 sets won. There is a tie-break in case of 2-2
Team Sports Model
90 minutes
48 minutes
No time limit
Coach
Player
Squad
Tactics
🟩 Football
🟧 Basketball
🟦 Volleyball
If there is a tie, there is overtime. If there is still a tie, then the next overtime. The match cannot end in a tie

Figure 48. Game duration concept in a unified model

You start by adding different validation checks - if it is football, consider 90 minutes; if it is basketball, 48 minutes; otherwise there is no time limit. Then you extend it to check for halves, quarters, sets, overtime in basketball and overtime in soccer. Then you have to distinguish between league games and playoff games (different rules). You start to cook spaghetti slowly.

Next, you dive into the number of players that can be on the field at the same time.

- **Football**. There are 11 players.
- **Basketball**. There are 5 players.
- **Volleyball**. There are 6 players.

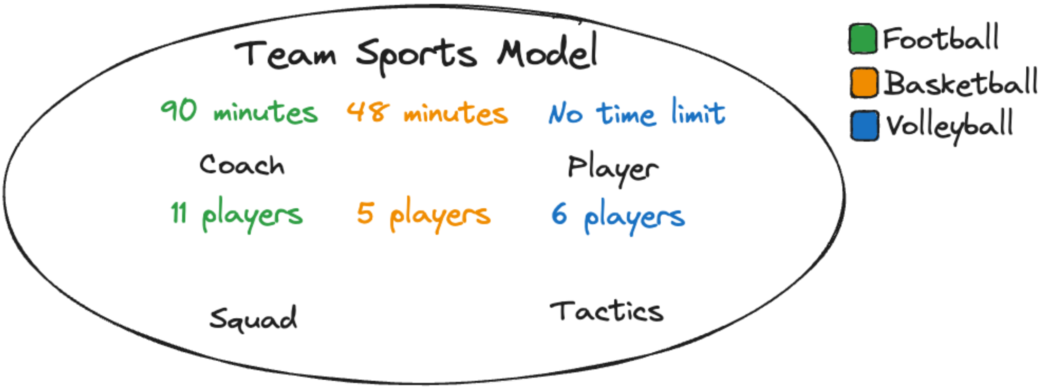


Figure 49. Player concept in a unified model

Again, validation checks must be performed for each sport. If soccer, then 11 players; if basketball, 5, otherwise 6. If we add player substitutions, it turns out that in basketball and volleyball players can return to the game, but in soccer they can't, and the coach can only do five.

Finally, it makes sense to add the concept of sets.

- **Football**. There is no such concept as a set.
- **Basketball**. A set play refers to a pre-designed play that is intended to create a scoring opportunity.
- **Volleyball**. A set is a subdivision of the game. At best, there are three sets (3:0); at worst, there are five sets (3:2).

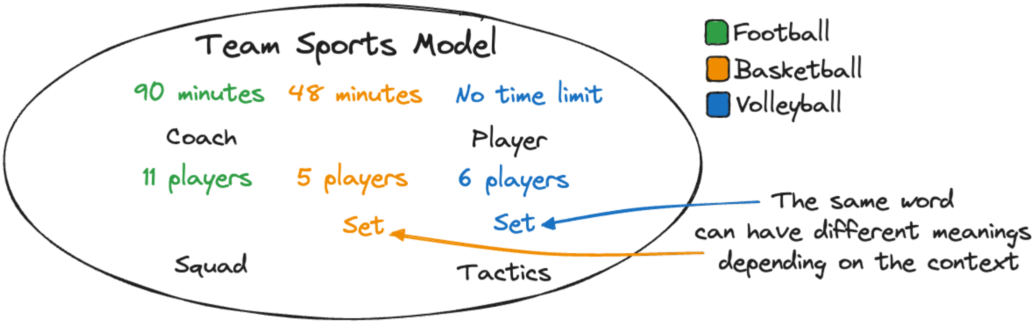


Figure 50. Same terminology, different meaning

Oops, there is a problem. The same terminology (set) means different things in different sports. In this case, in addition to checking whether it is basketball or volleyball, we come to differentiate the word depending on the context.

Sooner or later, the model will grow to such a size that it will be impossible to maintain. It will mix business rules from different concepts (football, basketball, volleyball) and the same terminology with different meanings (set).

Instead, you could split it into multiple models represented by separate bounded contexts.

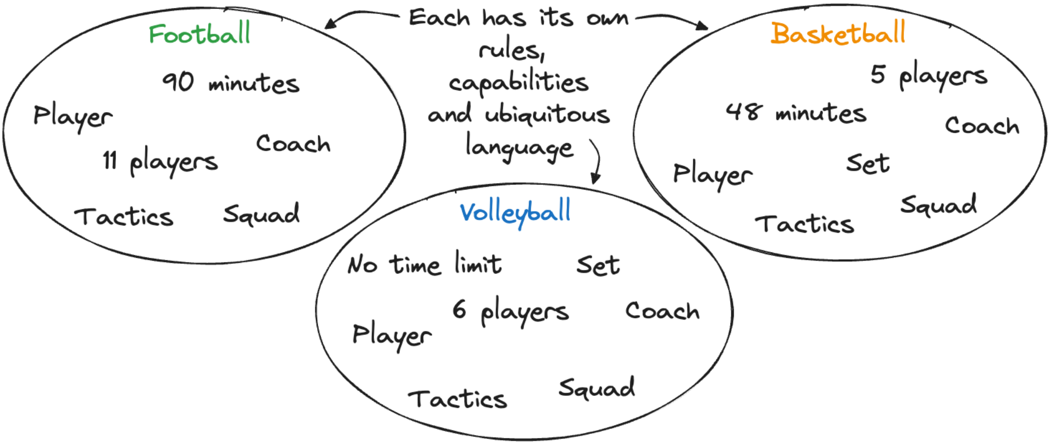


Figure 51. Separate models represented by bounded contexts

Where each bounded context has its own business rules, ubiquitous language, and capabilities.

When starting a greenfield project, you will most often face one of the following situations:

- **Each subdomain is represented by a single bounded context**. Initially, when your knowledge about the domain is limited, treating each subdomain as a separate bounded context is a really good idea. It helps with a smooth start. However you need to remember that bounded contexts will evolve over time.
- **Multiple subdomains are represented by a single bounded context**. When you observe that two or more subdomains use identical terms, and there is a clear indication that these subdomains will require frequent communication with each other, it makes sense to consolidate them into a single bounded context.

Imagine you distilled three subdomains during workshops: *Assessments*, *Progress Tracking*, and *Virtual Coaching*. You have discovered that:

- Terms and concepts such as trainee, coach, assessment, and progress have the same meaning in all subdomains.
- The rules, workflows, and processes are consistent across the subdomains.
- It is common for these subdomains to communicate with each other.

Putting them in a single bounded context called *Personalized Training* makes sense.

Evolution

Bounded contexts will evolve together with your business. As your company introduces new services, changes the structure of existing business, or acquires other businesses, you will likely need to adjust bounded contexts or create new ones.

Let's look deeper into this using our case.

Private medical clinics that use our software also want to offer telemedicine. We extend the *Patient Treatment* bounded context to accommodate this change. It looks like it fits there – after all, it is just another treatment method.

However, with telemedicine, the patient does not need to come for an appointment, and there is no physical examination. Additionally, no follow-up appointment will be planned. We start wondering if this is a good idea, but the decision is to leave it as it is.

Three months later, clinics decide to offer specialized treatment. When a patient needs a consultation with an external specialist, they have to call the specialist and book an appointment.

Once a month, all specialists collaborating with the clinic send their invoices. Then, the clinic sends another invoice to the patient. As this is another treatment method, you decide to extend the *Patient Treatment* again, despite the process being entirely different from the initial state.

Problems arise. Maintenance requires increasing effort. More than one development team is needed. Too many people are working in the same area, and communication between them begins to fail. Terms start to be used in different contexts— *Doctor* refers to either an internal doctor or an external specialist. The former is on a contract of employment with the clinic, while the latter sends invoices to the clinic.

Ultimately, *Patient Treatment* becomes enormous, and you spot that the scope, rules, terms, and behaviors are completely different from what was initially defined.

> There is also a chance that, at some point, another team will join the development of a selected bounded context due to the need to extend it to a new area (like telemedicine). The first team pulls it in their direction, and the latter in theirs. As a result, bounded context ends up as a big ball of mud. Be careful!

The wise decision is to split it now into several new bounded contexts:

- **Telemedicine**. This covers remote patient consultations and treatment. It includes virtual visits, digital communication, and remote monitoring of patients.

- **On-site Treatment.** This represents the initial process of patient treatment where the patient visits the clinic, is examined, agrees to the treatment plan, and starts the treatment.
- **Specialized Treatment.** This handles interactions with external specialists and ensures that the collaboration with them is streamlined and distinct from the internal operations.

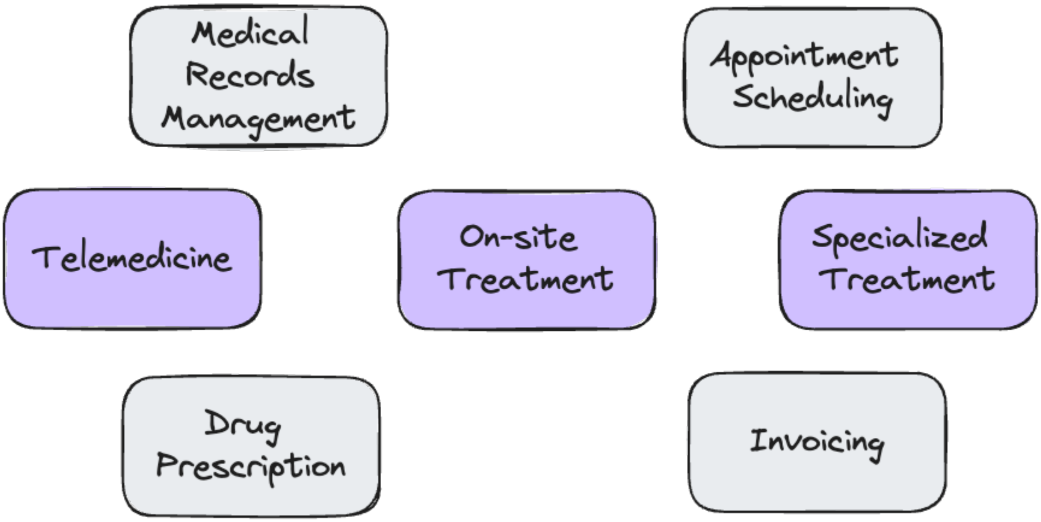


Figure 52. Evolution of a single bounded context to several others

Definition

It is challenging to decide whether you need a given bounded context and what should be included in it. It would help if you had the right tools. Fortunately, this has improved in recent years, and there are various of them at our disposal, including my two favorites:

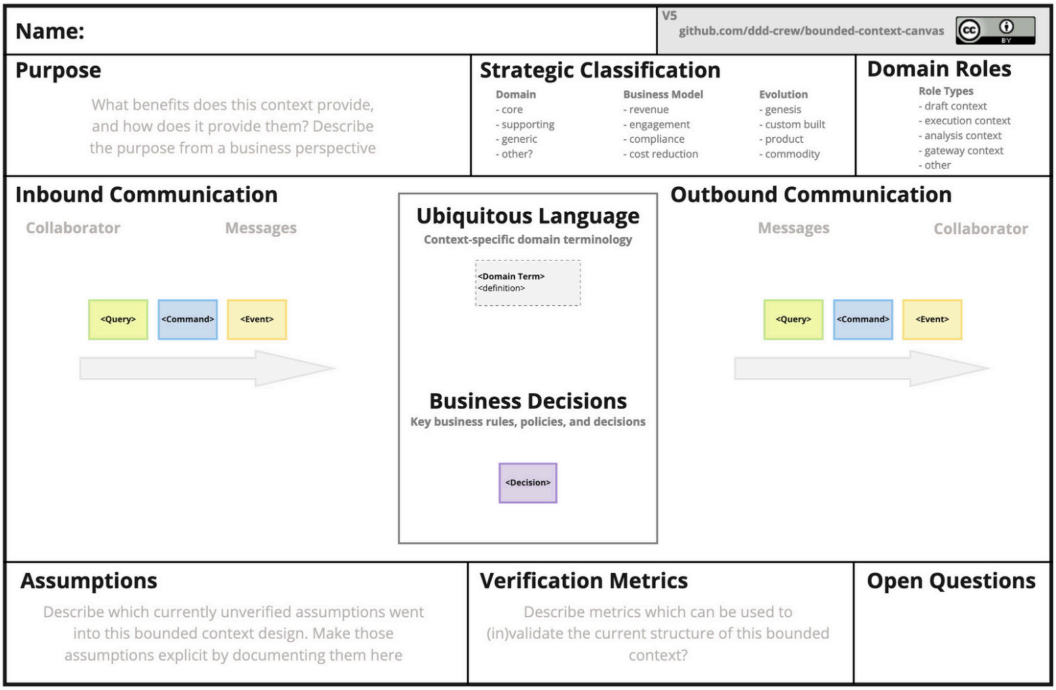
- **Bounded Context Canvas.** Proposed by DDD Crew, it represents a canvas[8], where you can add information about the strategic classification of a bounded context, inbound and outbound communication, business rules, ubiquitous language, and what kind of domain role the context would have.

[8] https://github.com/ddd-crew/bounded-context-canvas

- **Bounded Context Event Storming.** An additional level[9] of Event Storming proposed by Radek Maziarka as a step between *Process Level* and *Design Level* workshops. The main problem it addresses is that the first focuses on the representation of business processes (too abstract), their actors, and policies, while the second already shifts the focus to technical aspects (too deep) of modeling.

When I first saw the *Bounded Context Canvas* template, I immediately tried it in the next workshop I planned. The outcome surpassed my expectations. It helped us understand the different elements clearly and showed that we did not need all the bounded contexts we had originally planned.

The template is structured as follows:



Name:

V5
github.com/ddd-crew/bounded-context-canvas

Purpose

What benefits does this context provide, and how does it provide them? Describe the purpose from a business perspective

Strategic Classification

Domain
- core
- supporting
- generic
- other?

Business Model
- revenue
- engagement
- compliance
- cost reduction

Evolution
- genesis
- custom built
- product
- commodity

Domain Roles

Role Types
- draft context
- execution context
- analysis context
- gateway context
- other

Inbound Communication

Collaborator Messages

<Query> <Command> <Event>

Ubiquitous Language

Context-specific domain terminology

<Domain Term>
<definition>

Business Decisions

Key business rules, policies, and decisions

<Decision>

Outbound Communication

Messages Collaborator

<Query> <Command> <Event>

Assumptions

Describe which currently unverified assumptions went into this bounded context design. Make those assumptions explicit by documenting them here

Verification Metrics

Describe metrics which can be used to (in)validate the current structure of this bounded context?

Open Questions

Figure 53. Bounded Context Canvas template

[9] https://radekmaziarka.pl/2022/03/12/event-storming-bounded-context-level-en/

The *Name* represents just the name of the bounded context. Next is *Purpose*, one of my favorite fields because it forces people to justify why this context is needed.

Before we look at the field of *Strategic Classification*, I would like to let you know that each bounded context can be classified as:

Core

- **Description**: The central part of your business that differentiates it from competitors. It is where the primary business value (ROI) is created and is often the reason why customers choose your product or service over others.
- **Characteristics**: It is unique to your business and requires internal development to meet specific business needs. It is complex, frequently evolving, and needs continuous refinement and innovation.
- **Focus**: Highest priority in terms of resources and attention.

Supporting

- **Description**: It is necessary for the business but does not constitute its competitive advantage. It supports the core ones but isn't the primary focus of the business.
- **Characteristics**: Less complex than the core, these areas still require specific business logic but are not as critical for the unique selling proposition of the product or service.
- **Focus**: Often developed in-house but can be as well outsourced.

Generic

- **Description**: It represents common areas across many businesses and industries. They don't provide a competitive advantage and are not specific to the business, like payment gateway or invoice creation.
- **Characteristics**: These are often standardized processes or features with well-established solutions in the market.
- **Focus**: Businesses opt for off-the-shelf solutions.

There are two schools of thought regarding the definition of types (Core, Supporting, and Generic): one focuses on the subdomain level, and the other on the bounded context level. I prefer a pragmatic approach that combines both, depending on the context. In greenfield applications, where there is a high chance that a subdomain will directly map to a bounded context, I prefer to define the type at the bounded context level to avoid redundancy. In legacy applications, I often use both approaches because it helps me better identify and address issues.

Now, you need to decide where to classify the selected bounded context. You can use the *Core Domain Charts*, another excellent material[10] provided by DDD Crew (CC BY 4.0).

It is a chart representing the current and future state of your bounded contexts. Based on its model complexity and business differentiation, you can place each bounded context on it and predict where it might evolve in the next months or years.

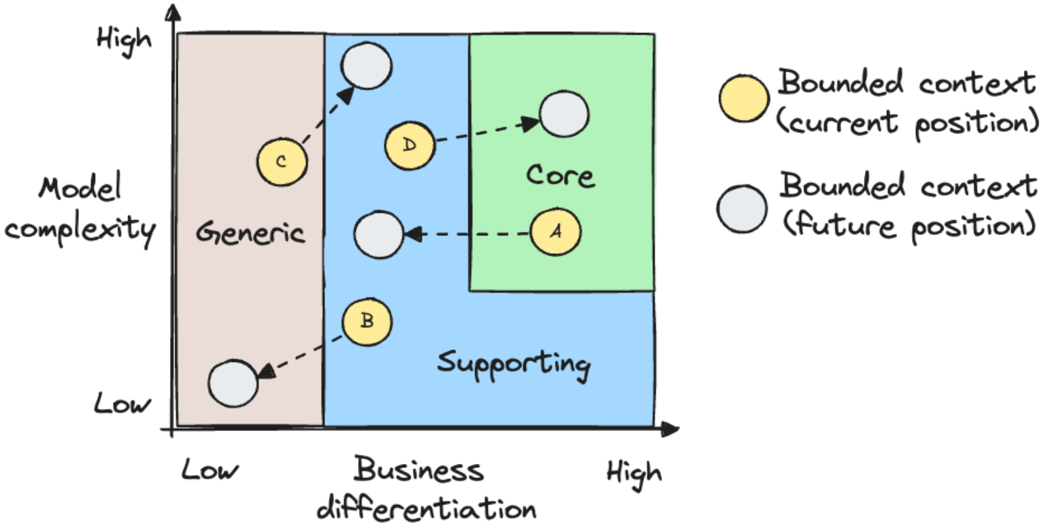


Figure 54. Possible movements of bounded contexts

Today, one of the bounded contexts (*BC A*) is quite complex and highly differentiates business from the market. You mark it as *Core*. However, you heard that several

[10] https://github.com/ddd-crew/core-domain-charts

other companies will offer the same functionality one year from now. This bounded context will no longer differentiate the business as it does today. You predict that there is a high chance that it will become *Supporting*.

Another bounded context (*BC B*) has low complexity, but there are no off-the-shelf solutions. You still have to develop it on your own. However, there is a high chance that in several months, there will be a product that you can buy and use. This means that it might become *Generic*.

The next bounded context (*BC C*) is highly complex, but the problem it solves seems generic. After several days of analysis, you found an off-the-shelf solution. However, it only partially solves your problem. Since the deadline for the release of the application is relatively short, you decide to adapt your process to the tool (trade-off), but you know that you will have to make a custom solution at some point. You mark the future position as *Supporting*.

The last bounded context (*BC D*) is complex and differentiates the business from most companies. You see a high potential to make it unique over the upcoming months. You predict that sooner or later, it will become *Core*.

> ℹ These charts can be used for various purposes, not only for marking bounded contexts. You can do similar for subdomains. It also works well when you want to mark which team should be assigned to which bounded context.

With this new knowledge, we are ready to return to the canvas and define *Strategic Classification*:

- **Its type (Domain)**. Should it be classified as *Core*, *Supporting*, *Generic*, or other.
- **Business model**. Does it directly generate revenue? Is it the primary reason customers use your application? Does it enhance your business reputation? Or does it primarily reduce costs?
- **Evolution**. Is it already explored, and are there off-the-shelf solutions you can buy? Or is it in an emerging market, and you need to build it yourself?

Next, we have to define the *Domain Roles* of the bounded context:

- **Analysis**. Receive data, analyze, and provide insights about it.
- **Enforcer**. Ensure that other contexts comply with procedures like anti-money laundering.
- **Funnel**. Receive data from other contexts, rework, and pass it further.
- **Execution**. Trigger workflows and other bounded contexts.
- **Engagement**. Provide features that attract customers to use your product.

There are more roles, and you can find them on DDD Crew GitHub[11].

Next, we have to look at the *Inbound Communication.* Here, you define who communicates with the bounded context (*Collaborators*):

- Another bounded context
- External system
- Frontend application
- User (direct interaction)

And how it communicates (*Messages*):

- **Query**. Ask the bounded context for some information. For example, *Get treatment plan details* or *Get available drugs.*
- **Command**. Order the bounded context to do something (change the state). For example, *Review treatment plan* or *Confirm appointment.*
- **Event**. Inform the bounded context about some business event. For example, *Appointment scheduled* or *Treatment completed.*

The central part of the canvas is *Ubiquitous Language* and *Business Decisions.* For the first one, we collect terms that are specific within this bounded context and their definitions:

- **Appointment**. Scheduled meeting between the patient and the doctor where the patient is examined.
- **Drug**. Any substance or compound that is used to cure the patient. The doctor prescribes them.

[11] https://github.com/ddd-crew

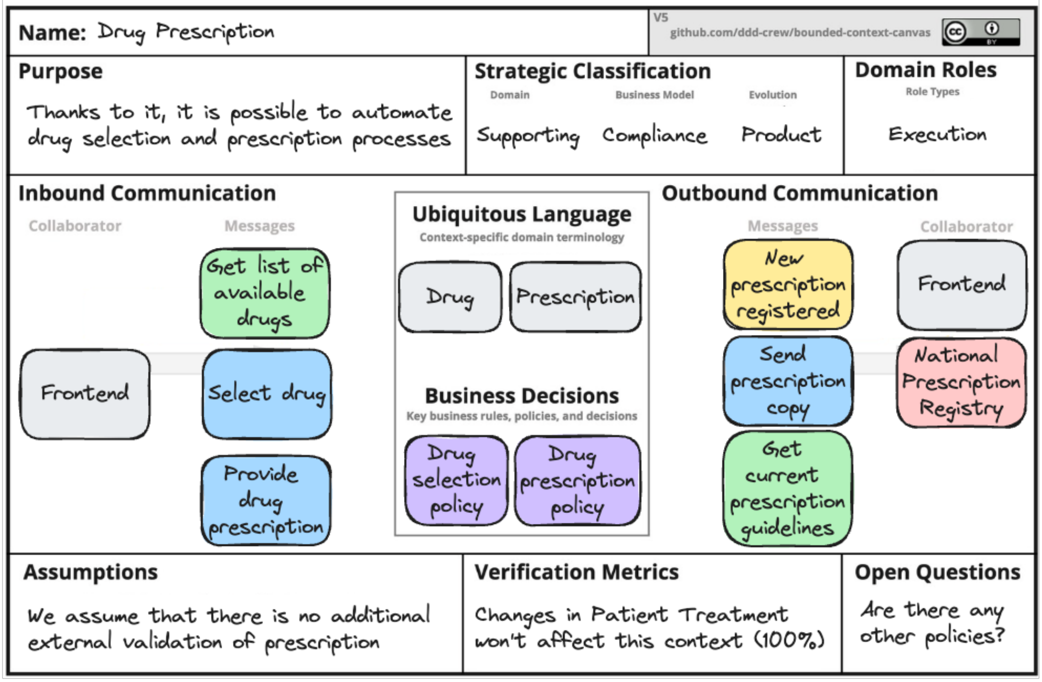
Business Decisions field contains all business rules and policies that are important inside the bounded context:

- **Appointment schedule policy**. Appointment cannot be scheduled without the patient's insurance.
- **Drug prescription policy**. The prescribed drug must not have any contraindications for the patient.

Next, you define *Outbound Communication* in the same way as *Inbound*. The last part of the canvas is used to write:

- **Assumptions**. You will always make some assumptions as it is impossible to have clear answers for everything. This is the place to write them.
- **Verification Metrics**. How would you verify that defined boundaries are correct? These metrics can indicate if the bounded context is triggered too often or if changes in it cause issues in other bounded contexts.
- **Open Questions**. Here, you write all the questions you could not clarify while defining bounded context.

OK, enough theory. Let's look at how we can use all this to define one of our bounded contexts, *Drug Prescription*.



Name: Drug Prescription

V5 github.com/ddd-crew/bounded-context-canvas

Purpose

Thanks to it, it is possible to automate drug selection and prescription processes

Strategic Classification

Domain	Business Model	Evolution
Supporting	Compliance	Product

Domain Roles

Role Types

Execution

Inbound Communication

Collaborator	Messages
	Get list of available drugs
Frontend	Select drug
	Provide drug prescription

Ubiquitous Language

Context-specific domain terminology

Drug

Prescription

Business Decisions

Key business rules, policies, and decisions

Drug selection policy

Drug prescription policy

Outbound Communication

Messages	Collaborator
New prescription registered	Frontend
Send prescription copy	National Prescription Registry
Get current prescription guidelines	

Assumptions

We assume that there is no additional external validation of prescription

Verification Metrics

Changes in Patient Treatment won't affect this context (100%)

Open Questions

Are there any other policies?

Figure 55. Example description of the Drug Prescription bounded context

Drug Prescription is not the core of our business (it would be *Patient Treatment*) and is used for compliance. Few products exist on the market, but we are still determining if it is possible to meet all requirements using them. Its role is to execute processes related to the prescription of drugs.

Whenever a drug prescription is provided, the information about it is later sent to the national prescription registry. Additionally, we inform our frontend about the registered prescription.

When there is a need to get information about current guidelines for drug prescriptions, we query the national prescription registry.

We assume that the prescription is not externally validated. If any change in the *Patient Treatment* bounded context triggers changes in the *Drug Prescription*, we need to review it. Also, there was a question that we could not answer—we have to clarify it later with our domain experts.

As a result, you have a clear description of the selected bounded context. Repeat all the steps for others.

Now, it is time to define relationships between all of them.

Context map

In the previous section, I showed you how to define bounded contexts using the canvas method. However, when many bounded contexts exist, your canvas pile will grow, making it difficult to find relevant information.

One of the solutions for the above problem is to create a context map.

You can think of it as a map of countries (bounded contexts and external systems). Each country has its unique characteristics: borders (boundaries), language (ubiquitous), regulations (policies), and key economic sectors (business capabilities).

Additionally, it includes information about the import and export relationship (inbound and outbound communication).

Together, they provide a complete picture of the ecosystem.

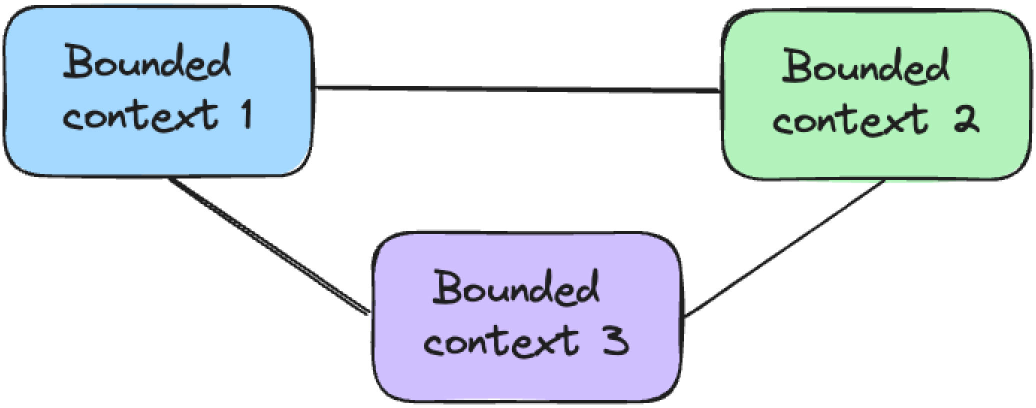


Figure 56. Basic representation of the context map

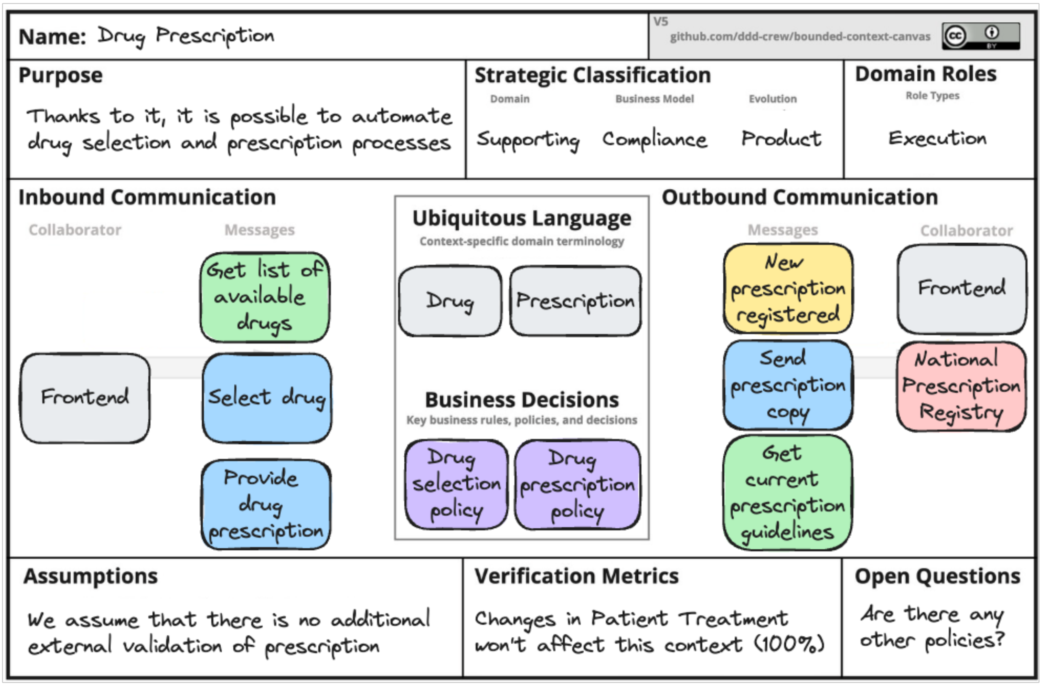
Upstream-downstream

To create a context map of your system, start by writing down all bounded contexts and external systems you discovered in previous steps:

- Appointment Scheduling
- Medical Records Management
- Patient Treatment
- Drug Prescription
- Invoicing
- National Healthcare System
- National Prescription Registry

Next, focus on one selected bounded context and review the canvas that was earlier created for it. Here, the most interesting parts are inbound and outbound communication.

As an example, let's take another look at *Drug Prescription*.



Name: Drug Prescription

V5 github.com/ddd-crew/bounded-context-canvas

Purpose

Thanks to it, it is possible to automate drug selection and prescription processes

Strategic Classification

Domain: Supporting

Business Model: Compliance

Evolution: Product

Domain Roles

Role Types: Execution

Inbound Communication

Collaborator: Frontend

Messages: Get list of available drugs; Select drug; Provide drug prescription

Ubiquitous Language

Context-specific domain terminology

Drug; Prescription

Business Decisions

Key business rules, policies, and decisions

Drug selection policy; Drug prescription policy

Outbound Communication

Messages: New prescription registered; Send prescription copy; Get current prescription guidelines

Collaborator: Frontend; National Prescription Registry

Assumptions

We assume that there is no additional external validation of prescription

Verification Metrics

Changes in Patient Treatment won't affect this context (100%)

Open Questions

Are there any other policies?

Figure 57. Drug Prescription bounded context

The entire inbound communication comes directly from the frontend, so no other bounded contexts or external systems are involved. However, there is an outbound communication to the national prescription registry system.

What you have to define now is which of them acts as upstream and downstream in their relationship:

- **Upstream**. The provider or source of data, services, or functionality. It often acts as the sender in a relationship.
- **Downstream**. The receiver or consumer of the upstream system's data, services, or functionality. It relies on the upstream system.

Before doing that, let's check the interactions between both systems:

- Retrieval of the current guidelines for prescriptions
- Sending of the prescription copy

Since the national prescription registry system is the source of truth for current prescription guidelines and requires the registration of prescription copies, it can be considered upstream.

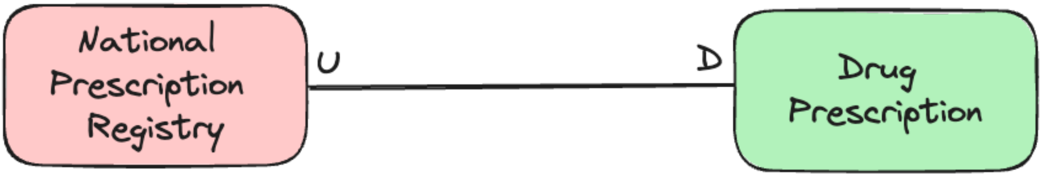


Figure 58. Upstream-downstream relationship between the National Prescription Registry and Drug Prescription

Now, repeat all the above steps for all bounded contexts.

Description

Instead of having only a bounded context name and its relationship with others, you can also include a description based on the information from the canvas. This way, you don't need to look into canvas unless you need detailed information:

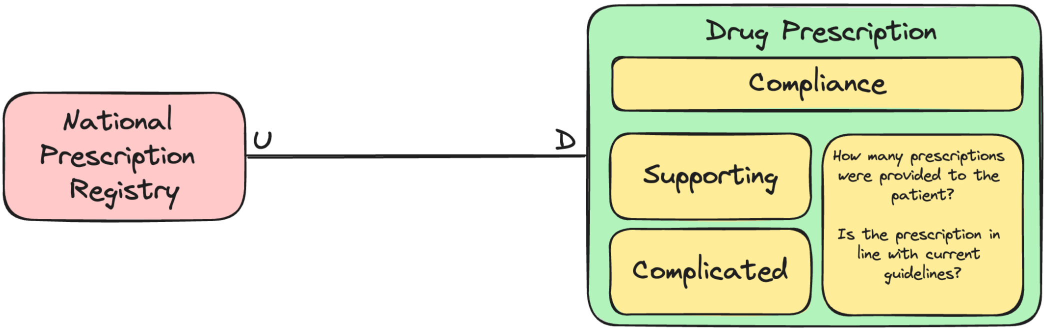

National Prescription Registry
U
D
Drug Prescription
Compliance
Supporting
Complicated
How many prescriptions were provided to the patient?
Is the prescription in line with current guidelines?

Figure 59. Detailed description of a bounded context (based on canvas)

My recommendation is to describe the following:

- **Strategic Classification**. Core, Supporting, or Generic. Revenue, compliance, engagement, and more.
- **Complexity**. Simple, Complicated, Complex.
- **Knowledge**. Questions that this bounded context can answer. This way, you have a full overview of its capabilities.

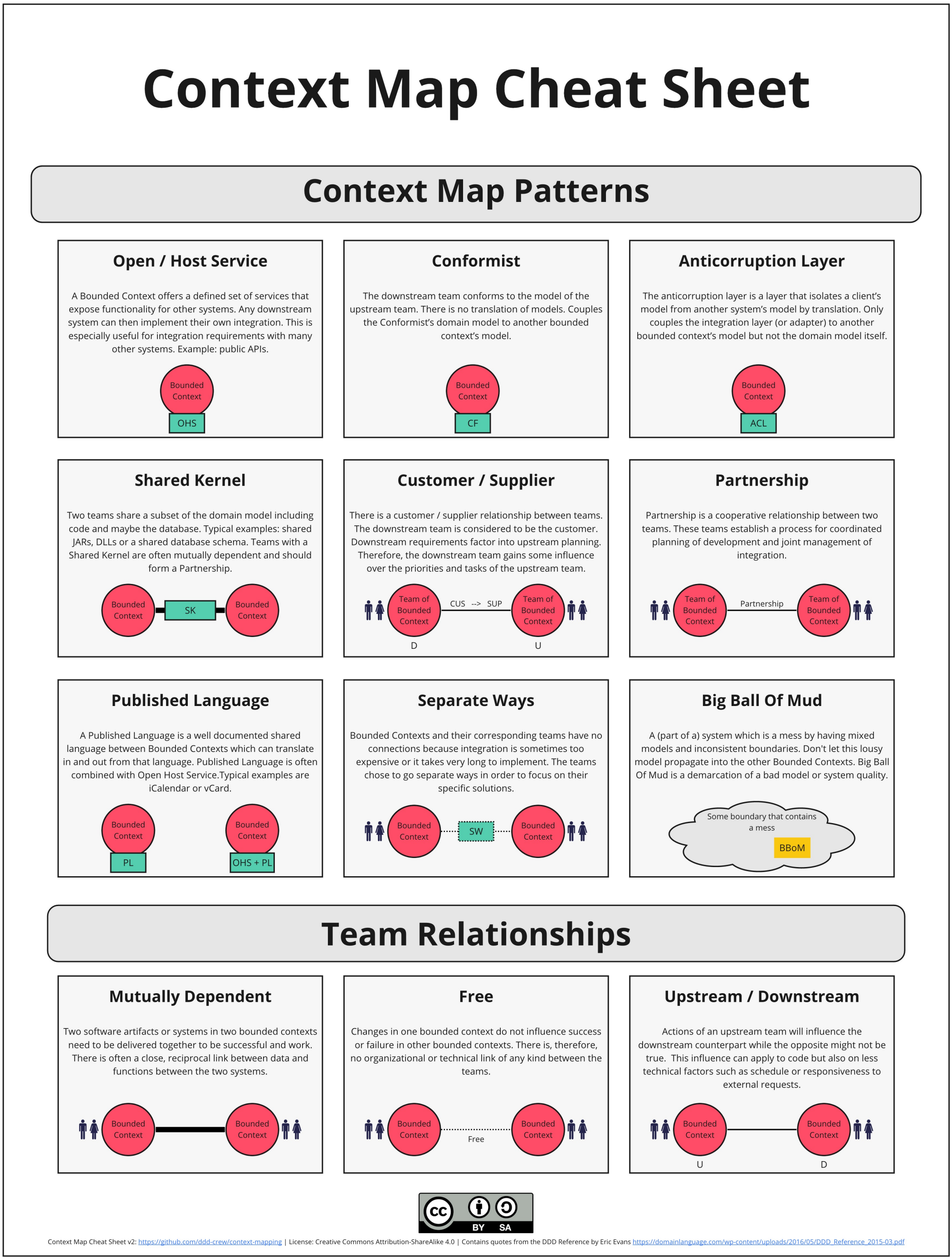
Please note that each bounded context can answer way more questions than on the diagram. I included only a few of them due to space limitations.

Patterns

Each relationship between two or more bounded contexts differs, and various patterns can be used to describe it. These relationships can range from tight integration to separate ways, depending on the needs of the system and the teams involved. Common patterns include open-host service, customer-supplier, conformist, and anti-corruption layer.

The DDD Crew has compiled these patterns into a comprehensive cheat sheet[12], which I recommend using as a reference.

[12] https://github.com/ddd-crew/context-mapping



Context Map Cheat Sheet

Context Map Patterns

Open / Host Service

A Bounded Context offers a defined set of services that expose functionality for other systems. Any downstream system can then implement their own integration. This is especially useful for integration requirements with many other systems. Example: public APIs.

Conformist

The downstream team conforms to the model of the upstream team. There is no translation of models. Couples the Conformist's domain model to another bounded context's model.

Anticorruption Layer

The anticorruption layer is a layer that isolates a client's model from another system's model by translation. Only couples the integration layer (or adapter) to another bounded context's model but not the domain model itself.

Shared Kernel

Two teams share a subset of the domain model including code and maybe the database. Typical examples: shared JARs, DLLs or a shared database schema. Teams with a Shared Kernel are often mutually dependent and should form a Partnership.

Customer / Supplier

There is a customer / supplier relationship between teams. The downstream team is considered to be the customer. Downstream requirements factor into upstream planning. Therefore, the downstream team gains some influence over the priorities and tasks of the upstream team.

Partnership

Partnership is a cooperative relationship between two teams. These teams establish a process for coordinated planning of development and joint management of integration.

Published Language

A Published Language is a well documented shared language between Bounded Contexts which can translate in and out from that language. Published Language is often combined with Open Host Service.Typical examples are iCalendar or vCard.

Separate Ways

Bounded Contexts and their corresponding teams have no connections because integration is sometimes too expensive or it takes very long to implement. The teams chose to go separate ways in order to focus on their specific solutions.

Big Ball Of Mud

A (part of a) system which is a mess by having mixed models and inconsistent boundaries. Don't let this lousy model propagate into the other Bounded Contexts. Big Ball Of Mud is a demarcation of a bad model or system quality.

Team Relationships

Mutually Dependent

Two software artifacts or systems in two bounded contexts need to be delivered together to be successful and work. There is often a close, reciprocal link between data and functions between the two systems.

Free

Changes in one bounded context do not influence success or failure in other bounded contexts. There is, therefore, no organizational or technical link of any kind between the teams.

Upstream / Downstream

Actions of an upstream team will influence the downstream counterpart while the opposite might not be true. This influence can apply to code but also on less technical factors such as schedule or responsiveness to external requests.

Figure 60. Context map patterns cheat sheet

Which one should you choose and on what basis?

The national prescription registry is used not only by our system but also by thousands of others. There is already a public API that can be called to retrieve the relevant data, so we can leverage it and mark it as the open-host service (OHS) on our map.

The *Drug Prescription* bounded context will call it and get responses. However, the model provided by the upstream is messy, and we don't want to consume it as defined. To solve this issue, the anti-corruption layer (ACL) can be used that allows us to translate one model to another that fits the system better.

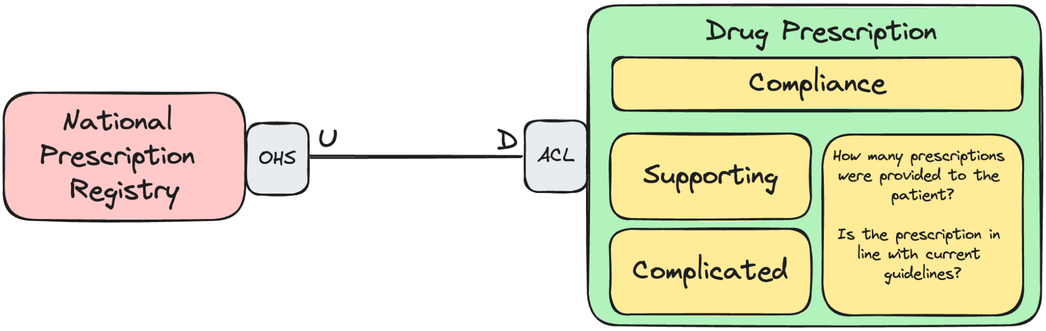

National Prescription Registry
OHS
U
D
ACL
Drug Prescription
Compliance
Supporting
Complicated
How many prescriptions were provided to the patient?
Is the prescription in line with current guidelines?

Figure 61. Application of various patterns

The anti-corruption layer is especially helpful when working with legacy or third-party systems. Models from such systems are often too complex, or the language defined there does not fit our model.

Such a translation layer can be implemented per bounded context. Still, in larger systems, it can also be solved by having one "fake" bounded context that communicates with the legacy systems. Thanks to this, all other bounded contexts can communicate with the "fake" one, knowing nothing about the legacy ones.

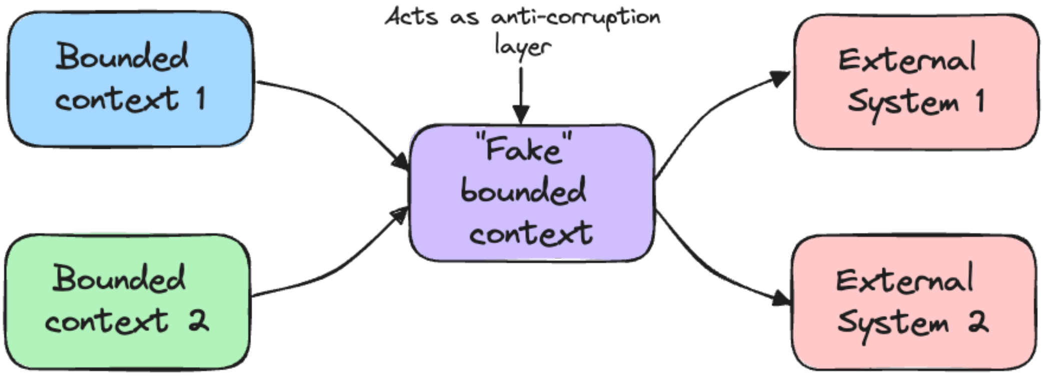


Figure 62. Additional bounded context acting as anti-corruption layer

Sometimes, you must accept that the model coming from the external system cannot be translated. This may be due to regulations requiring that data from the external system to always be stored in the same format.

Let's have a look at another relationship between *Medical Records Management* and *Patient Treatment* contexts:

- The first stores the information about the patient's medical history, all visits, diseases, and more.
- The second is used when the doctor examines the patient, prepares the treatment plan, and the patient completes the treatment.

Medical Records Management is the source of the information (patient's medical history) needed to prepare the treatment plan, so it acts as the upstream. *Patient Treatment* consumes this information as a downstream.

Changes inside *Medical Records Management* should not directly affect downstream, but the data should be easily accessible. It makes sense to use open-host service on the upstream. Since both contexts are under our control, the downstream can act as a conformist and consume the upstream model without any translation.

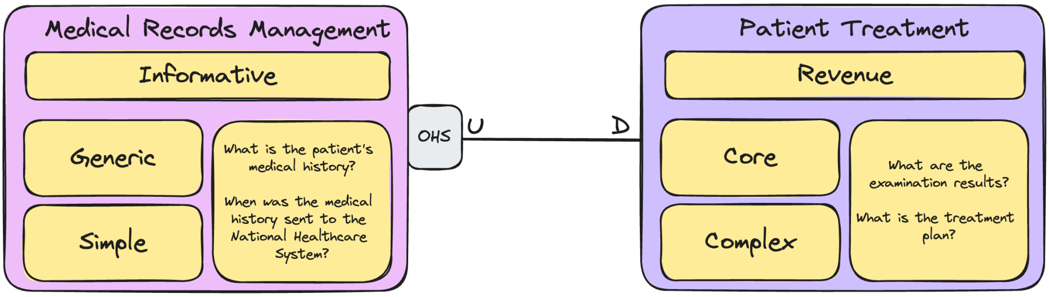

Medical Records Management
Informative
Generic
Simple
What is the patient's medical history?
When was the medical history sent to the National Healthcare System?
OHS
U
D
Patient Treatment
Revenue
Core
Complex
What are the examination results?
What is the treatment plan?

Figure 63. Relationship between Medical Records Management and Patient Treatment

Repeat similar steps for all other relationships. In the end, you should have a clear overview of the entire system.

Recap

In this step, we explored the *Healthcare* domain, and you learned how to analyze a business domain at both high and in-depth levels.

I described two techniques—Event Storming and Domain Storytelling—that can support you during workshops with domain experts, developers, and non-technical stakeholders, and explained how you can combine both.

I showed you how to transition from high-level workshops, where you gain an overall understanding of the domain, to in-depth workshops, where you dig deeper into specific areas.

Next, we used the outcomes of these workshops to focus on strategic Domain-Driven Design. This involved defining subdomains and bounded contexts using elements of a canvas: strategic classification, domain roles, inbound and outbound communication, business rules, and ubiquitous language.

Finally, we described the relationships between bounded contexts and communication patterns using a context map.

It is time to recognize the environment around you.

STEP 3: Recognize the Environment Around You

Discovering a business domain can be incredibly exhausting. Often, we may be tempted to skip further analysis and dive straight into implementation. After all, there are features waiting for us, and we might think we already know everything, right?

Not so fast. Before you can do that, you need to recognize the environment in which you will be working:

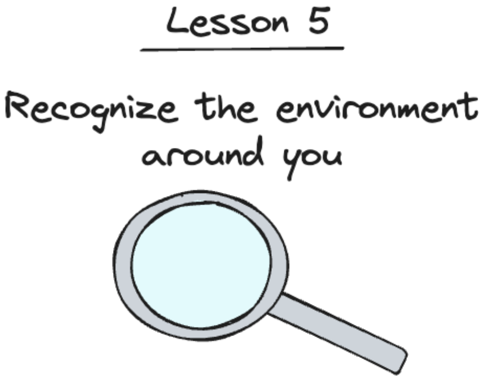
- What do you want to build?
- How are product decisions made?
- What are the strengths and weaknesses of your development team?
- Are there any other development teams?
- Are there any infrastructure teams?
- What does existing infrastructure look like?
- What scale and numbers will the application serve?
- What are the budget limitations?
- When is the deadline?
- What are the expectations?

By answering these questions, you can spot potential challenges and recognize opportunities. This ensures that your work aligns with the business's goals and current environment.

These insights will help you determine whether you're building a web or mobile app (or both), assess if you can leverage existing infrastructure, or understand that a three-month deadline may necessitate prioritizing quality or functionality.

Additionally, early recognition of these factors will influence future decisions and enhance collaboration between development teams, infrastructure teams, product managers, stakeholders, and others.

This way, you build a strong foundation for your project, increasing the chances of delivering an application that meets the requirements and expectations.



After each section, you will find a small exercise. Try to answer the question according to our case. If you do not know it, you can find all answers at the end of this step.

What do you want to build?

During workshops, we discovered the business domain and its processes. The next step is to define what we want to build from the technical point of view.

It could be:

- Web application
- Desktop
- Mobile
- Embedded

Or a combination of them.

It is crucial to understand how the selection was determined. Oftentimes, it is based on gut feeling rather than focusing on customer needs. We tend to focus on what is easier to implement instead of what is essential to the product.

During discussions with customers, you will learn, for example, that they work in the following way:

- Field workers take measurements in the field using custom hardware devices (surface scanners). They then provide the measurements to office workers.
- Office workers use their computers to analyze the measurements and record everything in a spreadsheet.

With further questions, you can clarify that the most comfortable solution is to use the web or desktop application in the office and embedded applications on scanners. A backend solution must also be built to allow synchronization between the scanner and the web or desktop app. Now, you have a high-level design.

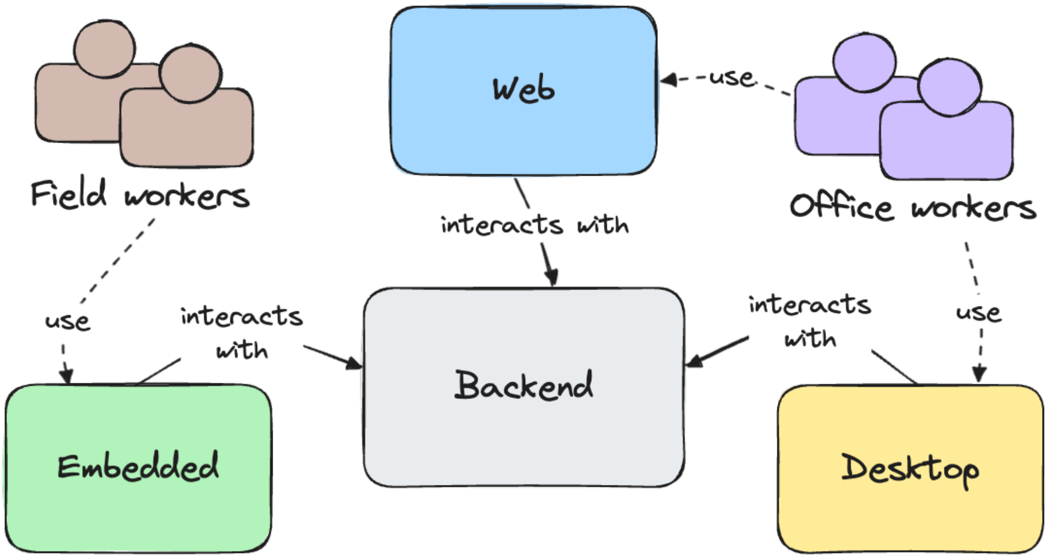


Figure 64. Basic overview of the application

At this point, you don't have to go into further details. Whether it will be container-based, a native app, a progressive web app, or which database you will use doesn't matter at this stage. The same goes for load balancing, fault tolerance, and disaster recovery. The most important thing is that you already have enough understanding of the application you are building.

One of my favorite approaches for implementing applications is to let the product team work for several days with people who will use the application, such as those in logistics or warehouse operations. This way, they can thoroughly understand the challenges faced by the end customers and their workflows. By doing so, the team can develop bespoke software solutions tailored to the customers' specific needs.

✎ Answer this question based on our case: What do you want to build?

How are product decisions made?

We often don't think about it, but it significantly impacts how we plan the implementation of the application. In some companies where the hierarchy is flat, product decisions are made by the product team. In others, decisions require approval from a single person or must pass through several people or multiple departments.

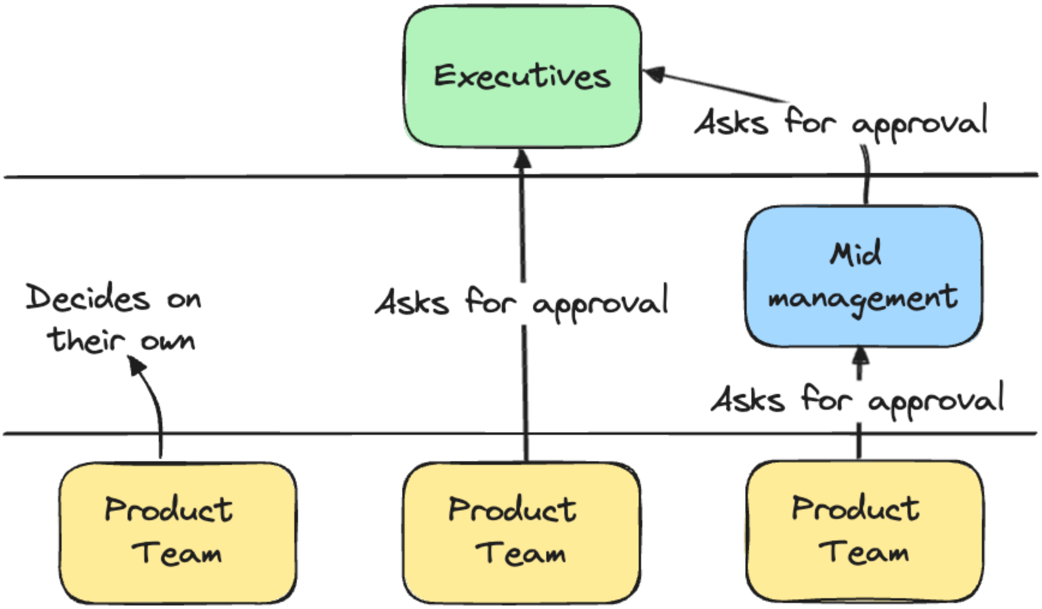


Figure 65. The decision-making process varies between companies

Imagine that each product decision has to go through several departments. If your customers are asking for a file upload feature or you want to switch to a new database provider different from the company's standard, you will find that approval can take about a month on average.

To save time while waiting for approval, it is smart to plan your product strategy carefully. For instance, if you know it takes a long time to get infrastructure changes approved, you might delay those changes until they are approved and use an in-memory database for now.

But there are other issues to consider too, like bottlenecks. What if a key decision-maker is on vacation or unexpectedly hospitalized? Do you have to wait for them to return, or can someone else make the decision in their absence?

There is also the issue of interpersonal dynamics. What if some people involved in the decision-making process don't get along? Could personal conflicts actually block a decision from being made, despite its merits? It might sound unlikely, but internal politics and personal relationships can surprisingly affect product development outcomes more often than you would expect.

Product decisions can also go wrong because people overuse their authority. The product owner may pretend to be in charge of everything and make decisions without asking others - team members, stakeholders, or customers - what they think. In the end, the team wastes time building something that does not meet real needs.

Answer this question based on our case: How are product decisions made?

What are the strengths and weaknesses of your development team?

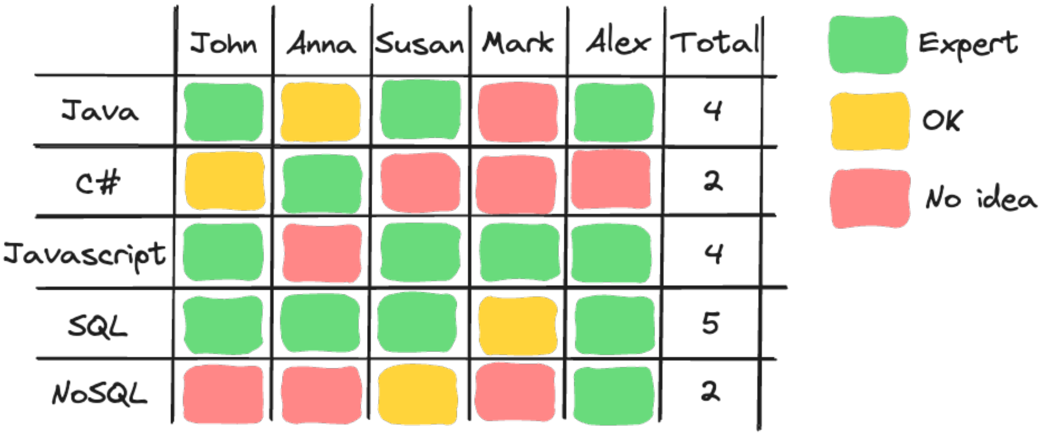
As you work with different teams of developers, you will often find yourself working with people with varying experience and skills. Sometimes, you work with very experienced guys, while other times, you have a less experienced team.

You can meet people who are experts in certain areas, such as building large-scale distributed systems or optimizing infrastructure. Some people might know a lot about NoSQL but have no idea about relational databases or the other way around.

To make sure your project is successful, it is important to understand what each person on the team can do well. Try to use the team's strengths and hide weaknesses while deciding on technologies and concepts that you will use to build the application.

One way to gather and analyze this information is by using the competency matrix[1]. It helps identify what each person knows, what they are good at, and what areas they struggle with.

It is crucial to involve the entire team in workshop sessions when creating the competency matrix. Doing this without the presence of others is wandering in the dark, and besides, you may offend someone with an inadequate assessment of competence. This is what the competency matrix can look like:



	John	Anna	Susan	Mark	Alex	Total
Java	🟩	🟨	🟩	🟥	🟩	4
C#	🟨	🟩	🟥	🟥	🟥	2
Javascript	🟩	🟥	🟩	🟩	🟩	4
SQL	🟩	🟩	🟩	🟨	🟩	5
NoSQL	🟥	🟥	🟨	🟥	🟩	2

🟩	Expert
🟨	OK
🟥	No idea

Figure 66. Team competency matrix

- **On the left side** there are competencies that should be evaluated. They will vary depending on the context, and you can, of course, assess more than five.
- **In the central part**, you can find all team members and their competency assessments. Assessments are represented using the following color codes: a

[1] https://management30.com/practice/competency-matrix/

green box (expert in this area), a yellow box (competent with some skills), or a red box (no knowledge in this area).
- **On the right side** is a summary of the assessment. To assess if the team is competent in an area, add the green and yellow boxes, skipping the red ones. If the majority have sufficient skills, you can consider this when deciding which direction to take.

You can assess the team's strengths and weaknesses any time you need them—it is not a one-time action. Environments and personnel change, expertise grows, and you may need to evaluate competencies in entirely new areas.

Furthermore, you can monitor the team's growth. For instance, while only two people currently feel reasonably confident in C#, a year from now, it could be four people or the entire team.

✎ Answer this question based on our case: What are the strengths and weaknesses of our development team?

Are there any other development teams?

Reach out to other development teams within the organization and engage with them. Schedule a coffee meeting—either on-site or virtually—to discuss technical and business matters.

Other teams are usually a gold mine of knowledge about processes, pain points, and collaborations with different teams. You might discover that there are tech guilds[2] inside the company where people help each other—take advantage of it!

Here are some questions to ask the other team:

- How do they handle their APIs?
- How do they scale their system?
- Which load balancer do they use?

[2] https://www.infoq.com/articles/architecture-guild-800-friends/

- What type of database do they use?
- How do they host the application?
- How do they make product decisions?
- How long does it take for product decisions to get approved?
- How often do they deploy to production, and why?

The list of questions is endless and can show you the company's current state of processes and knowledge. Make sure you ask enough questions to get the desired information.

Keep in mind that some teams may be so used to the status quo that they will not see the potential for growth or change. Therefore, treat this as helpful knowledge rather than as an oracle. There is always room for change.

Answer this question based on our case: Are there any other development teams?

Are there any infrastructure teams?

One of the biggest problems with infrastructure is that it is often isolated from the development teams. What do I mean by this? I have worked with organizations of various sizes, from small startups to large enterprises. Often, companies establish a separate team responsible for the infrastructure at some point. Over time, this team may divide further to manage areas such as identity access management, orchestrators, API gateways, and system version control. This division results in development teams being responsible for fewer aspects of infrastructure. Consequently, developers become less concerned with infrastructure matters over time.

> *DevOps team will handle this!* (irony)

As a result, our infrastructure skills begin to decline, making us more dependent on others. Neglecting infrastructure is an anti-pattern that you should be aware of. That is why it is important to recognize how infrastructure is handled in your company and check what skills you lack in your team.

Each company has its own way of handling infrastructure. Regularly, I face one of the following setups:

- **Separate infrastructure team(s) with deep expertise**. These teams can do everything, including deployments, orchestration, cloud setup, account management, and system administration. It is great to have such expertise in a company. However, it creates bottlenecks and might lead to potential breaches caused by a lack of skills in development teams.
- **Separate infrastructure team(s) with limited expertise**. These teams are usually good in one area, like system administration or account management, but lack skills in others, like deployments, setup of pipelines, orchestration, and more. In this case, a lot of infrastructure blocks are on your team's shoulders.
- **No dedicated infrastructure team**. This setup is a standard in early-stage startups and small companies. It is obvious that your development team is responsible for the entire infrastructure. It has its drawbacks, especially when there is a lack of expertise.

Depending on the case, you will need to approach the infrastructure topic differently.

Separate infrastructure team(s) with deep expertise

First, meet with the team(s) as soon as you join the company. Ask questions and try to understand what and how they operate. Here are examples of valuable insights that you can obtain during the meeting:

- The entire company runs on Kubernetes; all new web applications must run on it as well.
- 80% of applications run in AWS, 15% in Azure, and 5% in GCP.
- Each application has to pass a security audit. This audit is done by one of the infrastructure teams.
- They observe all environments using Prometheus and Grafana.
- The deployments are done through GitHub Actions.

With this information, you know which areas they cover and which can be covered by your team. You might hear that they do not care which libraries you use for logging, but you must use structured logging. You might also use any database you want. However, your application must run in the existing Kubernetes.

If you have any suggestions for change, that would be great. However, keep in mind that the larger the organization, the more people you have to convince and the more time such a change will take. Also, especially when you are starting out as an architect, changing things just for the sake of change might not be wise.

Lesson 6

The proposal for change should be measurable

When you see that your application will require a change to the current infrastructure—for example, the introduction of API gateway—and it makes sense to reuse its mechanism in the organization, then make sure that you plan it as soon as possible with the team's resources. Do not leave this problem for the future because you may run out of time due to slow processing.

There are three things I dislike about this setup:

1. Isolating infrastructure from development teams can increase the likelihood of security weaknesses. Without insight into security protocols and optimal procedures, developers may unknowingly introduce vulnerabilities that could result in cyberattacks or information exposure.
2. Developers must wait for infrastructure modifications or fixes, which results in slowed development cycles and potential delays in product launches. Such bottlenecks may negatively impact time-to-market.
3. When the development team does not participate in decision-making or managing the infrastructure, it may lead to a lack of ownership and accountability.

Separate infrastructure team(s) with limited expertise

This is a special case, and many things need to be addressed. Still, the more information you get, the better. You can find out that:

- Part of the infrastructure is running on Azure, and part is self-hosted.
- They can help you in all user, access, and account management areas. Also, they have some experience in serverless solutions.
- No one has heard about infrastructure-as-a-code (IaaC).
- If there is a security audit, then it is done by an external auditor.
- They can't help in migration to another cloud due to a lack of skills.

You may learn that the team can assist with user identity management, but you must handle all other infrastructure matters yourselves.

This setup leaves much room for maneuvering (flexibility) because you can do anything you like (within budget constraints, of course!). When you take any action, it is a good idea to establish and document some standards for the next teams, such as conventions for naming resources or deployment pipelines. It would also be nice to share the knowledge about anything you do with the infrastructure team.

There are two things I dislike about this setup:

1. Sometimes we get support for certain processes, but for others, we are left to figure things out on our own. It feels like some responsibilities are unclear and split between us and another team, leading to a bit of chaos.
2. Another concern is the risk of security breaches or errors in how things work. This can happen when the team lacks specific knowledge, leading to accidental disabling or overlooking of parts of the infrastructure.

No dedicated infrastructure team

First, check if there are other development teams in your company. They may already be using some cloud provider, infrastructure, or observability tools, so it might be worth going in the same direction.

When your team is alone, it is a greenfield game. You can plan and do whatever you want, and every infrastructure-related topic is on your team's shoulders. Having at least one team member with strong expertise in this area is beneficial. If such a person isn't available, consider hiring one. However, it is crucial that all team members are involved in planning and executing ideas. This way, you won't create bottlenecks.

Another thing that I would recommend is delegating some tasks, such as a security audit or penetration testing, to an external company. This way, you get top-notch skills for the most important tasks, and your team can concentrate on other tasks—not necessarily infrastructure-related.

When starting out and not chasing any deadline, it is worth implementing some rules, standards, conventions, or mechanisms that you can later reuse in other applications. It might slow you down but will speed up the implementation process in the future.

There are two things I dislike about this setup:

1. If we are the only development team in the company, there is no one else to validate our infrastructure plans and actions. This means we might miss mistakes or overlook better solutions.
2. Without a budget to hire someone with infrastructure expertise, many of our decisions are based on gut feeling rather than informed knowledge.

Answer this question based on our case: Are there any infrastructure teams?

What does existing infrastructure look like?

Checking the existing infrastructure is one of my favorite tasks during discovery. The more infrastructure there is, the faster and easier it usually is to deliver results.

Unfortunately, more infrastructure does not always mean an easier life. Sometimes, there is such spaghetti that using it is not a good idea. The question is whether we can integrate new infrastructure alongside the existing one, which isn't always possible.

Probably, no one will care if you use one library or another. The same goes for database choice. However, some infrastructure has a massive impact on the entire organization:

- Cloud providers
- Version control systems
- Central observability components
- Identity and Access Management (IAM)
- CI/CD pipelines

In such cases, you either have to accept the situation or propose a change and wait a long time for it to be implemented.

Even when there are no strict rules, it is often a good idea to follow common practices. Take cloud providers, for example. If most of your company uses AWS, it is usually smart to stick with it. This approach has at least two benefits:

1. **Existing knowledge**. Why go against the grain, especially when talking about the cloud, where the quality of service is comparable between providers? Use the knowledge and experience from other teams and focus on improvements instead of learning it from scratch.
2. **Future migration**. If the company decides to move all applications to AWS, the tight coupling with other vendors' services would make the migration challenging.

Answer this question based on our case: What does existing infrastructure look like?

What scale and numbers will the application serve?

Numbers have been a part of human history for centuries, guiding us in exploring and understanding the world. Thanks to them, we have most inventions—cars, airplanes, telephones, cameras, computers, etc. They represent facts that help us measure and plan our actions. We can make informed decisions by focusing on actual conditions rather than gut feelings.

Designing IT systems should also be based on facts. Before we decide which component or database to use or how our API should be built, among other things, we should be looking at:

- SLA and SLOs
- Number of total users
- Daily active users
- Number of activity hours
- Size of the files we are going to store
- Expected number of requests
- Response time for requests

These factors give us an overview of the current and projected state and can significantly impact our infrastructure. It is vastly different to manage an app where 5,000 users shop, another where 5,000 users sync gigabytes of data from their devices within the same 15 minutes, and yet another where 100,000 users regularly publish and interact with various types of content like articles, photos, and videos throughout the day.

When I started my career, there was a specific role dedicated to calculating infrastructure capacity—determining how much an application could handle in terms of users, data, or requests. Over time, this role has become extinct, much like the dinosaurs. With the rise of cloud computing, we have moved away from this practice and stopped paying as much attention to it.

> *Just add another instance!*

Unfortunately, this approach often leads to excessively high infrastructure costs. It is not uncommon for a company that spends $100,000 a month on infrastructure to reduce costs by as much as 60-80% after reassessing the infrastructure and replacing services.

That is why I recommend looking at the numbers before you add a new component or change an existing process. The following sections will show you that it is not that difficult.

SLA

Service level agreement (SLA) is a term commonly heard in nearly every IT project. It represents a contract between you—the service provider—and your customer and defines the minimum level of quality for a service.

Quality can be related to performance, response time to fix bugs, availability, and other attributes. For example:

> *We guarantee that our service is available 99.99% of the year.* (availability)

> *In the case of a critical bug, we respond within 2 hours.* (response time)

Most often, when people talk about SLA, they are referring to uptime (availability). It is represented in the form of "X nines." I most often encounter the following SLAs:

- **99.5% (Two nines five)**. Your services are available for 99.5% of the year. It translates to a total downtime of 1 day, 19 hours, 28 minutes, and 8.8 seconds in a year.
- **99.9% (Three nines)**. Your services are available for 99.9% of the year. It translates to a total downtime of 8 hours, 41 minutes, and 38 seconds in a year.
- **99.99% (Four nines)**. Your services are available for 99.99% of the year. It translates to a total downtime of 52 minutes and 9.8 seconds in a year.

Adding additional nines to your SLA can be quite expensive and might be unnecessary for most products. Achieving five nines (99.999%) means your service can be unavailable for only 5 minutes and 13 seconds per year, which requires complex infrastructure with failovers and fallbacks.

The number of nines you define should always rely on numbers. It could be the cost of a minute during downtime:

- **Transaction system**. Some transaction systems process thousands of transactions per minute. From each transaction, the system gets a provision. Let's say that each minute it earns $10,000. If it experiences 522 minutes of downtime in a year (three nines), it could cost around $5.2 million. In this case, aiming for at least four nines may be more sensible.

- **Niche e-commerce system.** If your system handles around 100 purchases per day at various times and you have few competitors, customers are likely to return even after downtime. In this scenario, complex infrastructure may not be necessary, and two nines (99%) could be sufficient.

Let's say you decide to go live with your monolithic application. It needs to run in an environment such as a virtual machine, platform-as-a-service (like AWS Beanstalk or Azure App Service), or any other. This environment guarantees 99.9% availability.

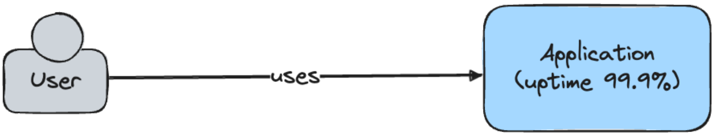


Figure 67. Overview of the basic setup of a monolithic application

You decide to add a database to your application. The database is provided as platform-as-a-service with 99.5% availability.

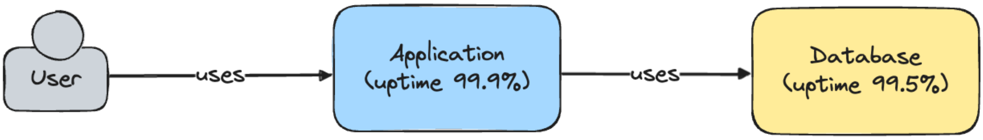


Figure 68. Overview of the setup of a monolithic service with a connection to a database

To calculate current availability, you multiply both components:

$$CombinedAvailability = 99.9\% \times 99.5\% = 0.999 \times 0.995 = 0.994005 \approx 99.4\%$$

Oops, it results in a decrease from 99.9% to 99.4%. When you add another component, you have to multiply it as well. The more elements you have, the greater the impact on overall availability (composite SLA).

How can you raise availability again? You can use a fallback or a failover. A fallback allows you to have a backup plan when your component (here, a database) fails. It can be another database, queue, or anything else. This way, when one fails, another can handle the operation.

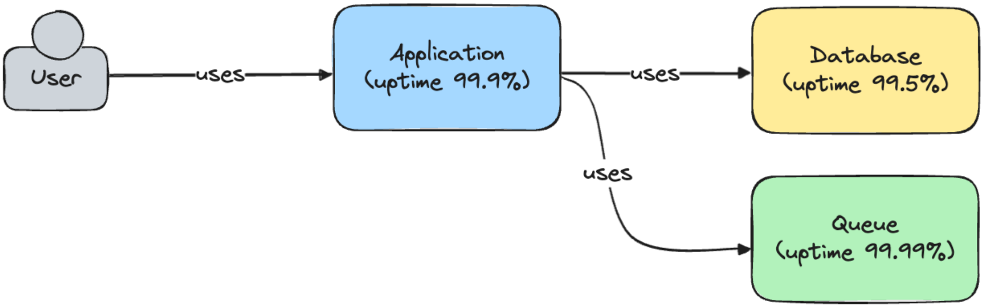


Figure 69. Overview of monolithic service setup with database fallback (queue)

To calculate availability for a component that has a fallback mechanism, the following formula is used:

$$ComponentAvailability = 1-[(1-PrimaryAvailability) \times (1-FallbackAvailability)]$$

$$ComponentAvailability = 1-[(1-0.995) \times (1-0.9999)] = 0.9999995 \approx 99.99999\%$$

Since the result of the calculation for the component (database) with fallback (queue) is ready, let's calculate the combined availability for the entire system:

$$CombinedAvailability = ApplicationAvailability \times ComponentAvailability$$

$$CombinedAvailability = 99.9\% \times 99.99999\% = 0.999 \times 0.999999 \approx 99.9\%$$

Adding a fallback mechanism made it possible to increase composite SLA back to 99.9%. However, there are at least two drawbacks:

1. **Higher infrastructure costs**. Adding new resources raises the total infrastructure expenses.
2. **Increased complexity**. As there are additional resources, the complexity of the

entire solution increases. Therefore, maintenance becomes more difficult, and the entry threshold becomes higher.

Failover is another useful mechanism for improving uptime. Instead of adding a fallback mechanism in the form of a queue or a different database (e.g., non-relational), you can add another instance of the same database. If one instance fails, the other instance will handle the traffic.

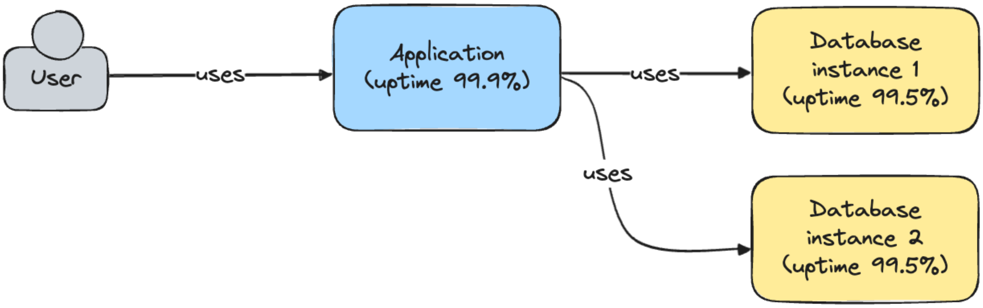

User
uses
Application (uptime 99.9%)
uses
Database instance 1 (uptime 99.5%)
uses
Database instance 2 (uptime 99.5%)

Figure 70. Overview of monolithic service setup with database failover (another instance)

Calculate the component availability the same way as in the fallback case:

$$ComponentAvailability = 1 - [(1 - PrimaryAvailability) \times (1 - FallbackAvailability)]$$

$$ComponentAvailability = 1 - [(1 - 0.995) \times (1 - 0.995)] = 0.999975 \approx 99.99\%$$

Since the result of the calculation for the component (database) with failover (another instance) is ready, let's calculate the combined availability for the entire system:

$$CombinedAvailability = ApplicationAvailability \times ComponentAvailability$$

$$CombinedAvailability = 99.9\% \times 99.99\% = 0.999 \times 0.9999 \approx 99.9\%$$

To calculate a composite SLA, start by reviewing the SLA documents from your hosting and service providers. You will find the guaranteed SLAs for each component. Another point to consider is things outside of the infrastructure, such as offline data migration (when your application may be unavailable for an hour), a bug that crashes the application for a period of time, or a misconfiguration that makes your application unavailable.

Users & requests

Users are usually represented by two numbers:

- **Total number of users**. The current count of users in the application or the projection for the initial weeks and months.
- **Number of daily active users (DAU)**. The count of users who engage with the application daily or the forecast for the first weeks and months.

Once you determine the number of potential users your application will serve, the next step is to figure out how many requests per second (RPS) they will generate on average. Predicting this can be quite challenging.

Each application has different characteristics. For instance, on social media platforms, users may generate an average of several thousand requests each, while in banking or intranet applications, it might be dozens or hundreds. In addition, in real-world applications, traffic is not evenly distributed. Your application needs to be able to handle spikes.

To calculate the number of requests per second, you need to divide the product of the daily active users (DAU) and the daily requests per user (DRPU) by the number of seconds in a day.

Be careful: a single user request can trigger dozens of other requests running behind the scenes (calls to databases, message brokers, other modules, etc.).

A day can be represented by 24 hours (86,400 seconds) when your application works 24/7 in the entire world. It can also be represented by 8, 10, and 12 hours when it serves, for example, only European customers during standard working hours.

$$RPS = \frac{DAU \times DRPU}{day(seconds)}$$

Let's assume the application has 10,000 DAU where each does 100 requests:

$$RPS = \frac{10,000 \times 100}{86,400} = \frac{1,000,000}{86,400} \approx 12$$

Now you know the infrastructure will handle around 12 requests per second. At first glance, this may not seem like much, but it depends on the nature of these requests. If each request takes 50 milliseconds, a basic infrastructure should manage it adequately. However, if each request takes up to 10 seconds, a more complex solution will be necessary, as each second requests will pile up:

- First second - 12 requests
- Second second - 24 requests
- Third second - 36 request
- Tenth second - 120 requests

Once you have calculated the number of requests you will need to handle, it is good practice to validate the infrastructure against 3x traffic (some suggest 2x, others 5x). In cloud environments, you can leverage auto-scaling, while in on-premise environments, you must prepare the infrastructure upfront before testing.

> Planning long-term traffic isn't practical since predicting how much traction the application will get is difficult. Once the application is live, all future calculations should be based on the metrics you collect.

Storage

Another useful calculation to consider is for storage requirements.

Let's assume each user can upload a profile picture up to 5MB in size, and you expect 10,000 users to register in the first month, we need to calculate the total storage space required for these photos.

$$SizeInGB = \frac{NumberOfUsers \times ProfilePictureSize}{1,000}$$

Taking into consideration the above numbers:

$$SizeInGB = \frac{10,000 \times 5MB}{1,000} = \frac{50,000}{1,000} = 50GB$$

The total storage you need to plan is only 50GB, which is not that much.

I recommend always planning for the worst-case scenario, rather than assuming that only 50% will upload their photo. Better to be prepared for the worst.

Now, let's consider a more complex scenario where your app allows users to share their songs. With an expected initial user base of 10,000 and each song having a maximum size of 3MB, you need to calculate the required storage space.

Another factor to consider is how many songs each user will upload in the first few weeks. It can be 1, 10, 100, or more. In such cases, you can reach potential users, or check dozens of small artists on other sites to see how many songs they have authored. This will give you an overview of the situation.

Assuming that each artist authored ten songs on average:

$$SizeInGB = \frac{10,000 \times 10 * 3MB}{1,000} = \frac{300,000}{1,000} = 300GB$$

This gives a total of 300GB of storage, which may not seem like much at first. However, as your user base grows, you will need more space. If you pay $0.01 for 1GB of storage per month, and you store 5PB (5,000,000GB), it will cost $50,000 per month. So be careful :)

Answer this question based on our case: What scale and numbers will the application serve?

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.

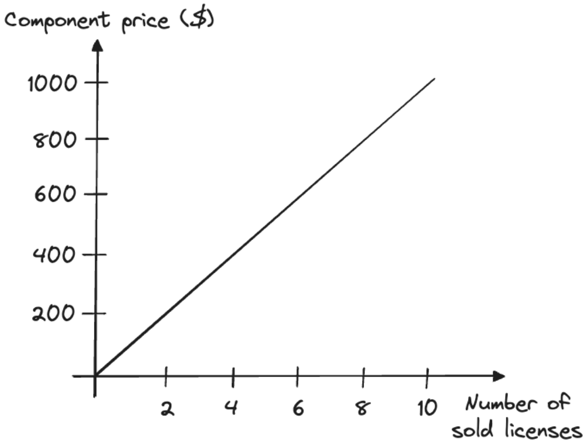


Figure 71. Example of the number of licenses that have to be sold to cover additional component

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?

Deadlines play a significant role when planning the software architecture of the application. They drive your decisions. The approach will be vastly different if you have to deliver the product in 3-4 weeks (MVP) compared to planning for a first release a year from now.

In the first case, you would focus on delivering the solution as fast as possible under extreme time pressure. Your business's future depends on your product's success, so you will do everything you can to make the product work and customers happy.

Ask yourself: Will you consider microservices, data streaming, caching, and orchestration? Probably not. Proper logging and monitoring? I don't think so.

You may look for no-code and low-code solutions to speed up development. But how would that work when features get more complicated and require custom implementation? What about maintainability? You do not need to answer these questions now, as you do not know whether this application will succeed.

In such a situation, plan a temporary solution. Of course, it has to change at some point, but you can plan it conditionally. If it succeeds, adapt to current needs; if it fails, pivot and explore another business idea.

Pivot

in this context, it means a significant change in the direction of the business.

> If a startup discovers that its original business model or product is not meeting market needs, it might pivot to focus on a different feature, market segment, or business strategy. This change can lead to various parts of the application becoming more or less critical.

⚠ Make sure that you clearly communicate to stakeholders that you have decided to implement a temporary solution and ensure that it is approved. It is crucial to avoid surprises—no one wants to unexpectedly face the cost of rewriting the entire application if it proves successful.

In contrast, when working on a product where development has been planned for a year, we usually don't think about the deadline every day. It is also hard to plan our work for the entire year.

As we do not feel time pressure, it is not uncommon for a project to be late. We had plenty of time in January, but suddenly, in October, we realized we would need more time to deliver it. It is natural—less stress about time causes us to overthink.

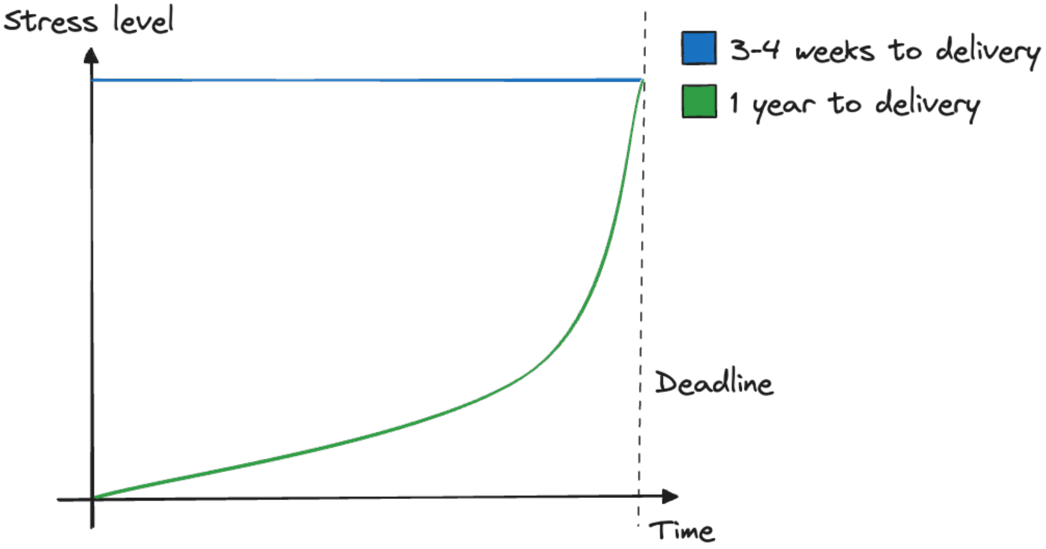


Figure 72. Difference in stress levels over time

Plan the software architecture that fits your current needs. Don't look at what will happen next year or when your application is successful. This way, the architecture evolves together with your business.

What worked well for the teams I worked with was to start extremely simple and deliver critical features as soon as possible. When you are ready with the first delivery, some people can focus on other features, while others might concentrate on improvements. If you see that you won't be able to deliver all that you planned, release the app without it. It is usually better to do it this way rather than shifting the release.

Answer this question based on our case: When is the deadline?

What are the expectations?

You have to be prepared for the fact that there will be different expectations depending on the group:

1. **Customers**. They are the end users of the product. They care about how well the software meets their needs, how easy it is to use, and whether it provides good value for money. They are interested in useful features, reasonable pricing, and helpful customer support. Their satisfaction is crucial for the product's success.
2. **Stakeholders**. These are often investors or board members. They look at the big picture of the project or company. Their main concerns are financial performance, long-term strategy, and managing risks. They want to ensure the company is heading in the right direction and meeting its business goals.
3. **Mid-level Managers**. They focus on day-to-day operations, turning high-level goals into actionable plans. Their job involves managing teams, allocating resources, and keeping projects on schedule. They care about meeting deadlines, using resources efficiently, and maintaining team morale.

4. **Product Managers**. They define what the product should be and why. They research market needs, analyze competitors, and prioritize features. They balance what customers want, what the business needs, and what is generally possible. Their goal is to build a product that people will find valuable and want to use.
5. **Software Engineers**. They build and maintain the product. They care about the architecture of the system, how different parts of the software interact, and ensuring the code is clean and maintainable. Keeping up with new technologies and best practices is important to them, as it helps them improve their skills and the quality of their work. While they are deeply involved in the technical details, they also need to understand the business side of the project.

All of these groups need to work together and agree on where the product is going. When everyone talks openly and understands each other, they make better products faster. But when groups don't talk to each other or work separately, problems arise. The product may not work as expected, teams may miss deadlines, and it becomes difficult to make decisions. People may start blaming each other instead of solving problems together. That's why it is so important to work as one team.

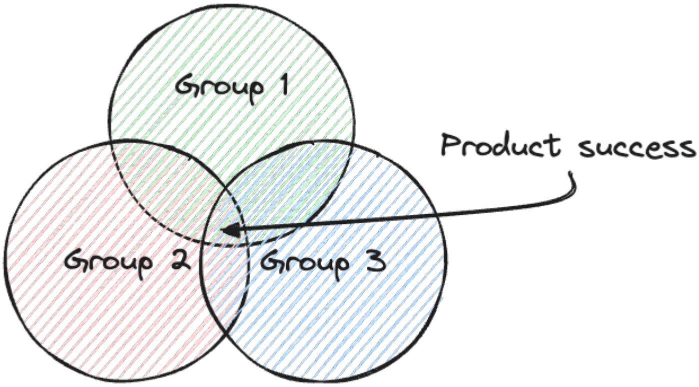


Figure 73. The success of the product depends on cooperation between different groups

I like it when a product team has a very flat structure. This team (together with customers) directly impacts the product's appearance, eliminating bottlenecks and potential mistakes made by people far away from the product.

How can the expectation be defined?

Expectation	Definition
Target	Europe, USA, China
Sold licenses	500 (first year)
Customer retention rate	>70%

Some business expectations might translate directly into technical solutions. Let's take the first row from the table. We want to sell access to the app to customers in Europe, USA, and China. At first glance, everything seems pretty simple. We host an application somewhere, collect data, store it, and process it.

Unfortunately, the following problems arise:

- **Europe**. We must follow the rules of the GDPR (General Data Protection Regulation) when handling personal data. To comply with the GDPR, we have to take steps to protect people's data. This means putting safeguards in place to keep information secure. We also need to get clear permission from people before we use their personal data, give them the right to know what information we hold about them, and give them right to ask us to delete it.
- **USA**. We may be subject to sector-specific regulations (e.g., healthcare or financial services) and state-level privacy laws.
- **China**. In the past, we decided to host our application on a global cloud provider. However, due to data policies, our Chinese customers require the app to run on one of the local clouds. That is fine. We have to start supporting both clouds. Traffic inside China will go through the local cloud, while the rest will go through the global one. To do this, your application should be cloud-agnostic. If it is not, you have to invest much time rewriting it to agnostic components.

That is why it is so important to recognize the environment and analyze business requirements carefully before the application is developed.

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

Our case

Let's look at our case and possible answers:

Question	Answer
What do you want to build?	Web app for people working in a clinic, mobile app for doctors who have home visits.
How are product decisions made?	There is no previous experience, but many decisions need to be approved by executives (CEO, CTO).
Are there any other development teams?	No, we are the first dev team.
Are there any infrastructure teams?	No, our dev team is responsible for it.
What does existing infrastructure look like?	N/A
What scale and numbers will the application serve?	3-5 clinics, 1,000-1,500 patients - first three months forecast.
What are the budget limitations?	Up to $500,000.
When is the deadline?	Two months from now.
What are the expectations?	Europe (EEA), SLA 99.5% (availability), SaaS, five licenses to be sold in the first three months.

When dealing with your case, please do not limit yourself to the questions above. There are many more you can ask, such as:

- Does the system handle sensitive information?

- What are our interactions with external systems?
- Who are our domain experts, and how can we best work with them?
- How crucial is data consistency for this project?
- What are our projections for data growth?

Feel free to ask about any aspect that could improve your understanding or benefit the project.

Recap

In this step, I showed you how to recognize and analyze the environment around you effectively.

You learned how to identify the type of application you aim to build and understand how product decisions are made within your company, which helps in aligning development goals with business objectives.

Next, we focused on evaluating the development team's strengths and weaknesses using a competency matrix. This technique helps you map out the skills and expertise within your team, ensuring you can leverage individual strengths and address any gaps.

You also learned the importance of communicating with other development teams to gain insights into the current environment. Understanding the existing infrastructure, identifying existing infrastructure teams, and assessing how the infrastructure supports your application is crucial.

We focused on key metrics, such as service-level agreements (SLAs), storage requirements, the number of requests per second, and the total number of users. With these metrics, we can plan and scale the application effectively.

Additionally, we considered business constraints such as budget limitations and deadlines.

Finally, we discussed the importance of understanding and meeting the expectations of various groups, including customers, stakeholders, managers, and the development team. Balancing these expectations is one of the keys to your project's success.

It is time to decide on the deployment strategy.

STEP 4: Choose Deployment Strategy

You might have heard or even participated in discussions that microservices are one of the best things that have happened to us as developers, or that we should only use modular monoliths in our applications.

However, the decision of which one to choose is not that simple. It is a complicated process that requires careful consideration of various factors. There is no single answer to this except the most obvious one—it depends.

Modular monoliths and microservices represent different deployment strategies. We determine whether an application should rely on a single (e.g., modular monolith) or multiple (e.g., microservices) deployment units.

Choosing the wrong strategy can lead to significant consequences. It is not just about the technical challenges like scaling, performance issues, and network problems. It is also about the potential impact on the development team and the project's budget. Furthermore, it can cause the development team to face issues that could have been avoided by choosing a different strategy.

The appropriate choice should heavily depend on architectural drivers, budget and numbers that we have discovered. Thanks to them, we can base our decisions on objective factors rather than subjective thinking.

In this section, we will explore both deployment strategies, understand their typical implementations, and learn how to make decisions that best suit the project's environment.

Single deployment unit

The easiest way to understand a single deployment unit is to look at the monolith, which contains all the application's functions, libraries, dependencies, and configurations within one large codebase.

Each time you decide to adjust something, you must build and redeploy it as a whole.

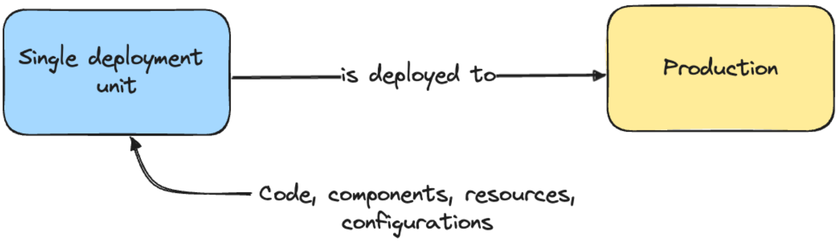


Figure 74. Deploying a single unit into production

Once deployed, the application runs in a single process. This means that the entire communication between various parts of code occurs within that process. They can reference each other directly or communicate via an in-memory queue. It is important to note, however, that while the application code runs in a single process, external dependencies such as databases, caches, or third-party services still run as separate processes.

If the application gains a lot of traction and needs to scale, you can deploy additional instances, but only if it is designed to be stateless. This allows you to create multiple identical copies of your application, each capable of handling requests independently. However, scaling becomes much more difficult if your application stores session data or other temporary information internally. In this case, the application architecture may need to be reviewed.

App instance 1
App instance 2
App instance 3

Figure 75. Scaling a single unit to multiple instances

If only one component requires scaling while the others are only used occasionally, you still must deploy a copy of the entire application. There is no option to scale it partially.

Imagine that you want to buy a wine subscription. Every month, you get a case of wine: six bottles of different brands and types.

One of these wines was excellent; you want to buy another bottle. However, they are not in normal distribution, and to get one, you have to order the whole case again, even when other wines are awful. Each time you want this concrete wine, you must buy an entire case.

This approach will work, but it is not too efficient.

In case of a critical failure, the entire application goes down. A slight change in one of the components will affect all others. For example, if an adjustment in the *Appointment Scheduling* component causes an out-of-memory exception, all other components will immediately stop working.

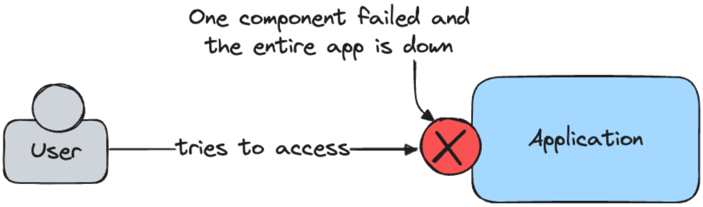


Figure 76. When one component fails, the entire application goes down

A single deployment unit is a reasonable choice. It simplifies the deployment process and ensures low latency, as there is no need to communicate over a network between components. You do not need to orchestrate deployment and worry that part of your application won't be deployed. Testing is more straightforward than in distributed systems because the entire application runs in a single environment.

One of the frequently raised issues is technology lock-in. When you choose one platform, all of your code must be written in the language supported by its runtime environment.

Please note that there are cases where one platform supports multiple languages, like .NET (support for C#, F#, or Visual Basic). In such cases, you can use the combination of supported languages.

Looking further at disadvantages, replicating the entire application may be challenging at some scale. Another problem is the single point of failure—as it runs in one process, the whole application will stop functioning when one of its components fails. As complexity grows with the application's size, you can expect longer build times and problems related to too many teams working on the same codebase.

Pros	Cons
Managing and deploying a single deployment unit is straightforward and requires less operational overhead.	Scaling a single deployment unit requires replicating the entire application.
Low latency.	Long build times.
All code is deployed together.	Single point of failure.
Easy versioning.	Technology lock-in.
Testing is easy because the application runs in a single environment.	Small changes require redeployment of the entire application.

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.

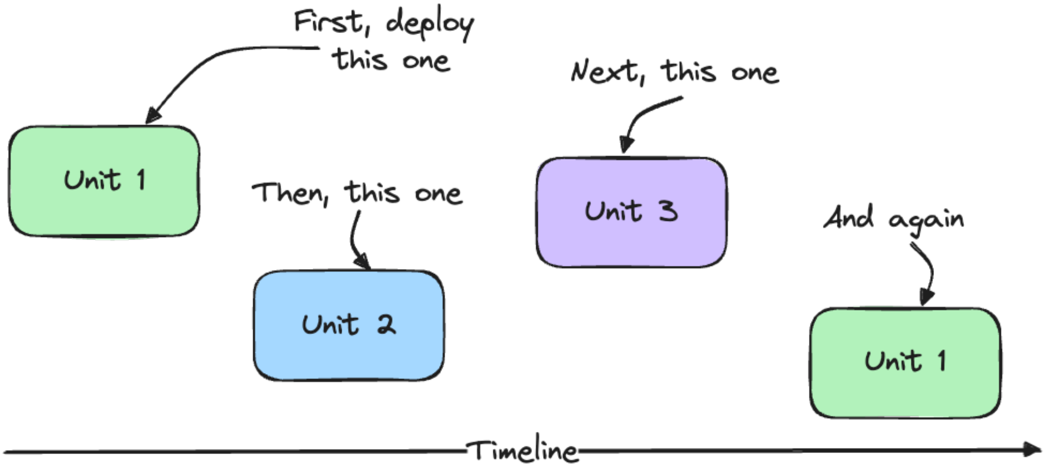


Figure 77. Deploying multiple units to production

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.

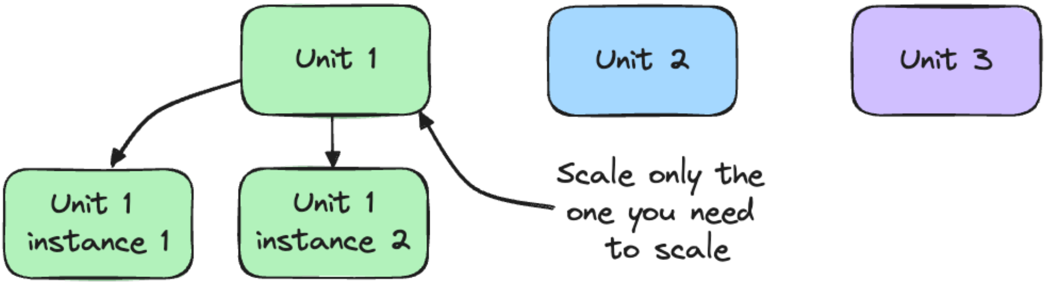


Figure 78. Flexibility to scale a single unit

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*).

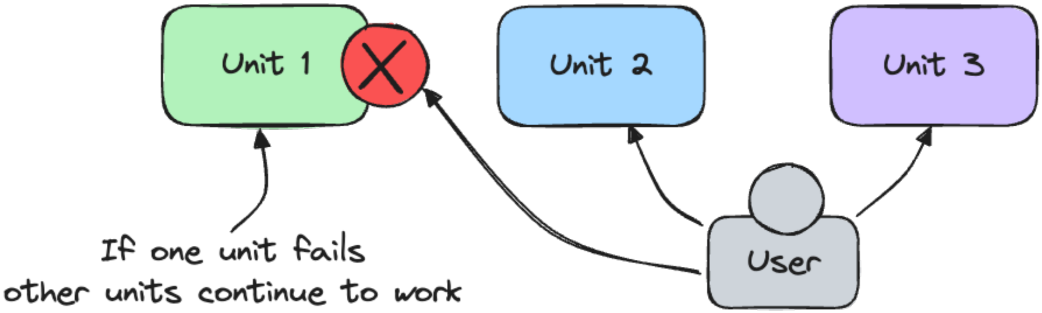


Figure 79. If one unit fails, the others keep working

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.

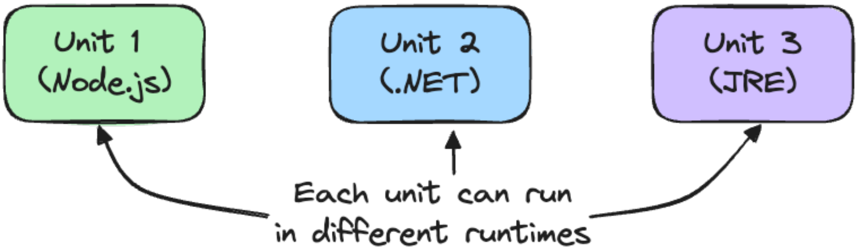


Figure 80. Technology flexibility per unit

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

Whether you decide on a single or multiple deployment units, inbound and outbound communication is one of the most important points of any application. Inbound, because we need to trigger the system somehow, and outbound, because it has to deliver the results of the system's processing.

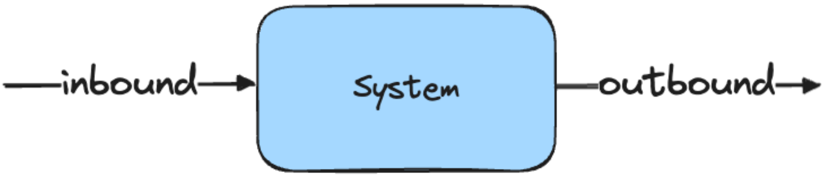


Figure 81. Inbound and outbound communication

In most systems, communication occurs through the following methods:

- **Command**. Used to order the system to do something. As a result of this order, you change the system's state. Examples: *Book an appointment*, *Prescribe drug*, or *Send an invoice*.
- **Query**. Used to retrieve information from the system without changing its state. Examples: *Get last 10 prescriptions*, *Retrieve available dates*, or *Show how much we earned in 2023*.
- **Event**. Used to record business facts and inform other components that an action has happened. It might trigger further logic. Examples: *Appointment scheduled*, *Drug selected*, or *Treatment completed*.

Commands & queries

When we talk about commands and queries, we enter the request-response world. These interactions are typically handled through HTTP calls to REST (`POST, PUT, PATCH, DELETE, GET`) or GraphQL (mutations and queries) APIs. You send a request like *Get available dates* and receive an object containing the list of available dates. Theoretically, these operations are synchronous.

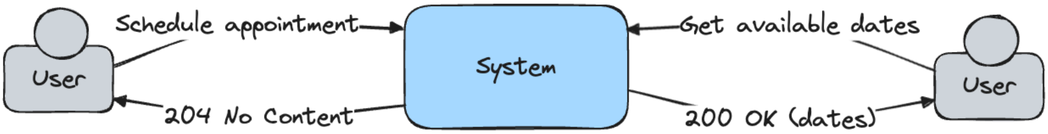


Figure 82. Example of communicating with a system using command and query

However, this is not always the case. Consider a scenario where we send a request that triggers a command. We immediately receive the `200 OK` response, indicating that our request has been realized. Yet, unknown to us, this command sets off a series of subsequent actions that execute asynchronously under the hood. An example of this could be the *Schedule appointment* command. After we send it to our API, it gets processed. This process can yield two results:

1. Almost immediately you will get the `204 NoContent` response indicating that the appointment was successfully scheduled.

2. Under the hood, an event *Appointment scheduled* is sent and consumed by another module responsible for appointment reminders. Thanks to that, you will receive an appointment one day before the visit.

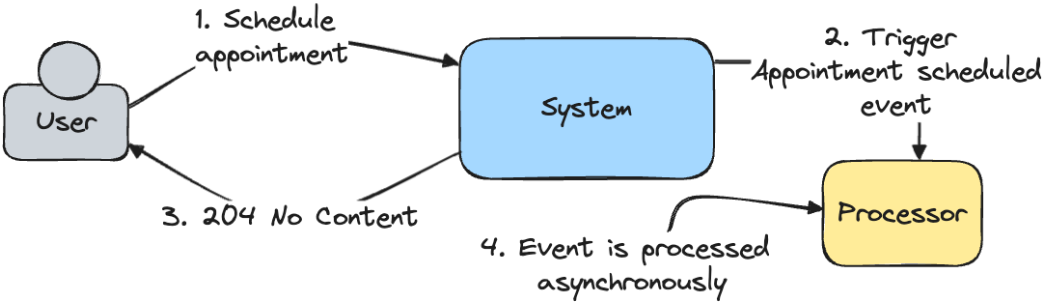

1. Schedule appointment
User
System
2. Trigger Appointment scheduled event
Processor
3. 204 No Content
4. Event is processed asynchronously

Figure 83. Using a command to trigger an asynchronous process

It is well described by Oskar Dudycz in his article[1] about the differences between events and commands:

> *Commands are usually assumed to be synchronous, and events to be asynchronous. We typically send commands via, for example, Rest API, and events via queues (In-memory, RabbitMQ, Kafka, etc.). This distinction comes from custom. When sending a command, we'd like to know immediately whether it has been done. Usually, we want to do an alternative scenario or handle errors when the operation fails. Likewise, we typically assume it is better to immediately stop the process, e.g., buying a cinema ticket, than wait and refresh to see if it has worked.*
>
> *It makes sense, but it is not always so obvious. Take a bank transfer, for instance—when initiated, it doesn't happen immediately; there's usually a waiting period. The same applies to making online purchases: placing an order and processing a payment doesn't instantly complete the entire transaction. It involves subsequent steps like shipping and issuing an invoice. These processes are asynchronous, meaning the outcomes of our commands can also be asynchronous.*
>
> —Oskar Dudycz, *What's the difference between a command and an event?*

[1] https://event-driven.io/en/whats_the_difference_between_event_and_command/

Events

On the contrary, when dealing with events, we are most likely in the asynchronous world. Events happen in stages: first, they are triggered, then they are processed and consumed. Each represents a business fact like *Appointment scheduled* or *Treatment completed*. They can be private or public—you might also find other nomenclature: internal and external, domain and integration.

- **Private (Internal, Domain)**. This type of event is only visible within the component where it occurred. It is never shared with other components and is used mainly for internal processing and coordination. For example, in the *Drug Prescription* component, you want to record the *Drug prescribed* event and store it for audit purposes or perhaps for further processing within that component.
- **Public (External, Integration)**. These events are meant to inform other parts of the application or external systems that something has happened within a component. They are mainly intended for communication and interaction. For instance, the *Patient Treatment* component notifies *Invoicing* when a treatment is completed. This allows *Invoicing* to generate and send the invoice to the patient promptly.

Typically, when using events, the application will do one of the following:

- Send them to the in-memory queue.
- Send them to an external message broker.
- Store them in the database or as files. Next, process them using a recurring task or similar mechanism.

Your task as a development team is to decide which approach best suits your specific case. It might be one or another or a combination of them.

Strategies for message delivery and processing

While designing the system around asynchronous communication, you must decide how information will be delivered from one component to another and processed:

1. You can send the message once; the receiver will receive and process it (no error case).
2. You can send the message once, which the receiver will never receive (error in the receiver).
3. You can send the message once, which will be received but never processed by the receiver (error in the receiver before or during processing).
4. You can send the same message multiple times and hope the receiver will process it only once.
5. You can send the same message multiple times, and the receiver will process all of them each time it receives it.
6. You might attempt to send the message, but it fails (error in the sender).

When discussing strategies, they are known as *at most once*, *at least once* and *exactly once*. There is no silver bullet—each time, you must decide which solution fits your problem best.

> The issue you face is also known as the *Two Generals' Problem*.
>
> The enemy army occupies a city.
>
> Two armies led by two generals must attack the enemy simultaneously; otherwise, they won't succeed.
>
> They need to communicate to agree on when to attack. However, they can only do so by sending messengers through enemy territory. One general might send a messenger to the other general with the attack plan. But there is a risk: the messenger could be captured or killed. The other general then needs to send a messenger back to confirm that he got the message, but this messenger might also be captured or killed.
>
> To improve their chances, the generals could send multiple messengers. However, this increases the risk of more lives being lost. And even if they send many messengers, there is still no guarantee that any of them will get through.

If you want to read more about this problem, it is well described in Wikipedia[2].

[2] https://en.wikipedia.org/wiki/Two_Generals%27_Problem

At most once delivery

At most once delivery means that a system will try to deliver a message only once. If the message doesn't get delivered, that's okay and the system won't try again. This approach is acceptable in situations where it is not critical for every message to be received.

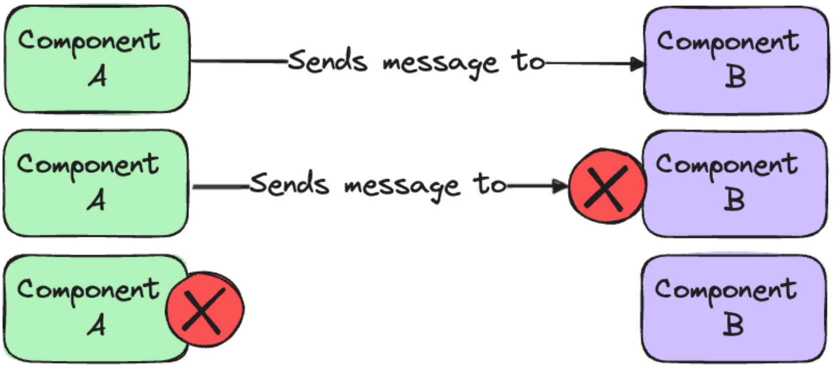


Figure 84. Possible scenarios while using at-most-once delivery

1. It might be sent from the sender and received by the receiver.
2. It might be sent from the sender but never received by the receiver.
3. It might fail while sending and thus never reach the receiver.

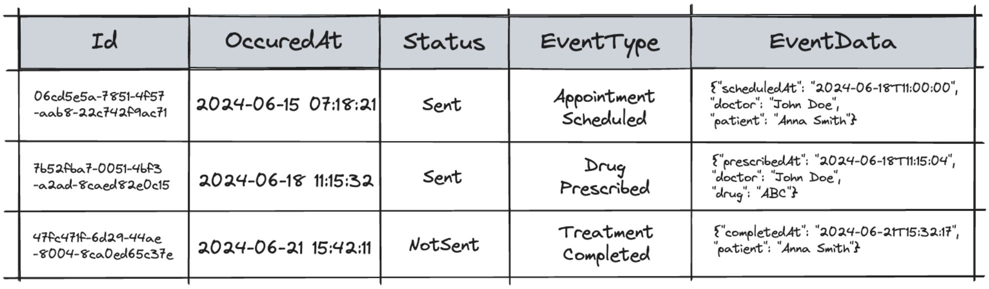
For instance, imagine that your system sends SMS notifications to your customers about daily discounts in your shop. If the message is delivered to the SMS service successfully, that's great. The service will send the SMS with the discount information, and your customer will receive it. If the message fails to deliver to the SMS service, it won't send the SMS, and the customer won't be aware of discounts.

Of course, it would be perfect if customers always received the messages. However, in this case, it is acceptable if some messages are not delivered. This approach works for scenarios where missing a message is not critical.

This solution doesn't work when your system requires guarantees that messages are delivered and processed.

At least once delivery - Outbox pattern

The outbox pattern can be a solution for ensuring messages are delivered at least once (but can lead to duplicates). In short, besides storing the result of your action in a database, you additionally store the message that contains an event in a separate database table.



Id	OccuredAt	Status	EventType	EventData
06cd5e5a-7851-4f57-aab8-22c742f9ac71	2024-06-15 07:18:21	Sent	Appointment Scheduled	{"scheduledAt": "2024-06-18T11:00:00", "doctor": "John Doe", "patient": "Anna Smith"}
7b52fba7-0051-4bf3-a2ad-8caed82e0c15	2024-06-18 11:15:32	Sent	Drug Prescribed	{"prescribedAt": "2024-06-18T11:15:04", "doctor": "John Doe", "drug": "ABC"}
47fc471f-6d29-44ae-8004-8ca0ed65c37e	2024-06-21 15:42:11	NotSent	Treatment Completed	{"completedAt": "2024-06-21T15:32:17", "patient": "Anna Smith"}

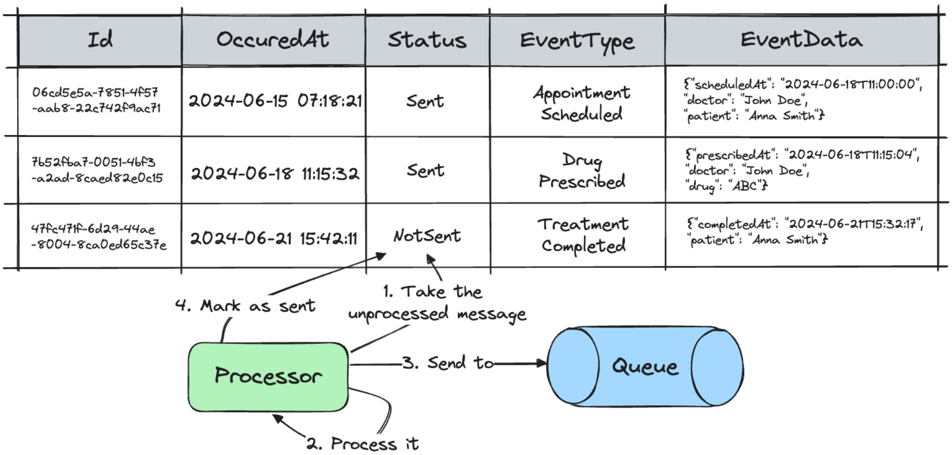
Figure 85. Event stored as a message in the outbox table

Both the action and event storing are handled inside one transaction. So if there is any error, the entire transaction is rolled back.

```
BEGIN TRANSACTION
 var actionResult = performAction();
 database.actions.add(actionResult);

 database.outbox.add(occuredAt, actionEventType, actionEventData);
 database.saveChanges();
END TRANSACTION
```

If all goes well, the event (stored as a message) will be processed later, for example by a background task. The task will check the outbox table for any unsent messages. If there are any, it will send the message to the queue and mark it as sent in the outbox table.



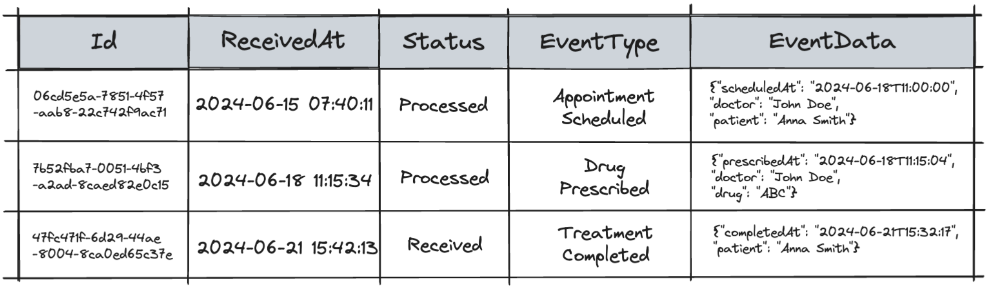
Id	OccuredAt	Status	EventType	EventData
06cd5e5a-7851-4f57-aab8-22c742f9ac71	2024-06-15 07:18:21	Sent	Appointment Scheduled	{"scheduledAt": "2024-06-18T11:00:00", "doctor": "John Doe", "patient": "Anna Smith"}
7b52fba7-0051-4bf3-a2ad-8caed82e0c15	2024-06-18 11:15:32	Sent	Drug Prescribed	{"prescribedAt": "2024-06-18T11:15:04", "doctor": "John Doe", "drug": "ABC"}
47fc471f-6d29-44ae-8004-8ca0ed65c37e	2024-06-21 15:42:11	NotSent	Treatment Completed	{"completedAt": "2024-06-21T15:32:17", "patient": "Anna Smith"}

Figure 86. Processing of the message stored in the outbox table and sending the message to a queue

In case of a random failure while sending the message, the system will retry the delivery to ensure that the message is delivered at least once. The main trade-off of this approach is that you need to store the information about outgoing events in a separate database table, and it won't be sent immediately—you have to wait until the message is processed.

Exactly once processing - Inbox pattern

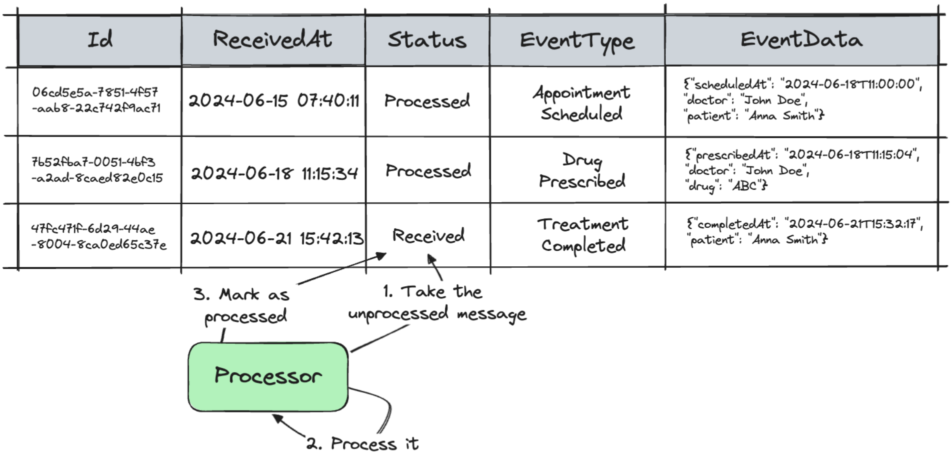
The inbox pattern is similar to the outbox pattern but is used on the receiver side. The difference is that you store messages about the events that were received.



Id	ReceivedAt	Status	EventType	EventData
06cd5e5a-7851-4f57-aab8-22c742f9ac71	2024-06-15 07:40:11	Processed	Appointment Scheduled	{"scheduledAt": "2024-06-18T11:00:00", "doctor": "John Doe", "patient": "Anna Smith"}
7b52fba7-0051-4bf3-a2ad-8caed82e0c15	2024-06-18 11:15:34	Processed	Drug Prescribed	{"prescribedAt": "2024-06-18T11:15:04", "doctor": "John Doe", "drug": "ABC"}
47fc471f-6d29-44ae-8004-8ca0ed65c37e	2024-06-21 15:42:13	Received	Treatment Completed	{"completedAt": "2024-06-21T15:32:17", "patient": "Anna Smith"}

Figure 87. Event stored as a message in the inbox table

After receiving the message, it is stored in the database, e.g., with a *Received* status. Next, it is processed by, for example, a background task. If everything goes well, the message is marked as *Processed.* In case of failure, it will retry processing.



Id	ReceivedAt	Status	EventType	EventData
06cd5e5a-7851-4f57-aab8-22c742f9ac71	2024-06-15 07:40:11	Processed	Appointment Scheduled	{"scheduledAt": "2024-06-18T11:00:00", "doctor": "John Doe", "patient": "Anna Smith"}
7b52fba7-0051-4bf3-a2ad-8caed82e0c15	2024-06-18 11:15:34	Processed	Drug Prescribed	{"prescribedAt": "2024-06-18T11:15:04", "doctor": "John Doe", "drug": "ABC"}
47fc471f-6d29-44ae-8004-8ca0ed65c37e	2024-06-21 15:42:13	Received	Treatment Completed	{"completedAt": "2024-06-21T15:32:17", "patient": "Anna Smith"}

Figure 88. Processing the message stored in the inbox table

This pattern helps ensure idempotency, meaning if the same message is delivered again, it won't disrupt your system. However, it doesn't guarantee idempotency by itself.

The main trade-off of this solution is that, similar to an outbox, you have to store the messages in a separate table (additional overhead), and they won't be processed immediately (increased latency). You also have to design the message processing logic to be idempotent so that reprocessing a message does not cause adverse effects.

Combination of outbox & inbox patterns

It is a good idea to use a combination of both of these patterns:

- Outbox to ensure that the message will be delivered at least once.
- Inbox to ensure that the message will be processed exactly once.

Using both patterns together minimizes the risk of missed events, though it does add complexity to the overall solution. This complexity may not be justified in small applications where missing or re-triggering the event has minimal impact on the system.

Both outbox and inbox tables will grow over time. If you decide on these patterns, I highly recommend archiving the old processed data in a separate database table or clearing it regularly—you can do it using a recurring task that runs once per day. This approach helps mitigate potential performance issues.

Dead letter queue

There is a chance that some of your messages may have incorrect schema, fail in processing logic, or cannot be processed for other reasons. They won't go through even if you process them a million times.

It makes no sense to keep retrying the processing of such messages. In such cases, you can use the dead letter queue, a holding area for messages that a messaging system cannot deliver or process successfully.

You can configure your system so that messages are pushed automatically to it after 5 or 10 retries. Then, developers can inspect the messages to determine why they failed. When the reason for failure is found, it can be corrected, if necessary, and reprocessed. This means important data is not lost because of a failure in the initial processing attempts.

Without the dead letter queue, we would need to manually delete the message from the queue or let it remain there, potentially clogging the system and causing further issues. The dead letter queue acts as a safety net, preventing system overload and ensuring that problematic messages do not disrupt the operational flow.

Modular monolith

The modular monolith is one of the most popular representations of a single deployment unit. It is a variation of a monolith, where your application still runs in a single process, but its code is logically divided into modules, allowing for better organization and maintainability.

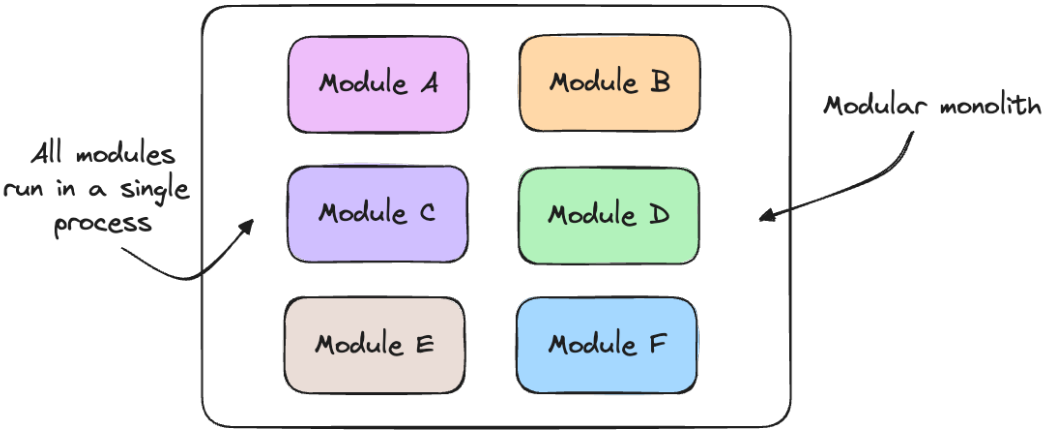


Figure 89. Modular monolith runs in a single process and is built on top of modules

Depending on the platform and language, modules can be represented by:

- Packages
- Namespaces
- Projects
- Folders

One of the fundamental principles of a modular monolith is high cohesion. For example, in a financial application, modules could be organized around different aspects of financial operations, such as transaction processing, account management, and fraud detection.

Each module would contain the necessary code, data, and business logic related to its specific context, promoting clarity and maintainability in the codebase. It is also recommended to logically split the data from the beginning—each module could have its own schema. Do not split it into separate databases until you really need it.

While a modular monolith offers many advantages, it is important to be aware of its limitations. Every change in the modular monolith requires redeployment of the whole. The same applies while scaling—you cannot scale just a single module.

Overall, a modular monolith strikes a balance between the simplicity of a monolith and the modularity of microservices, making it a pragmatic choice for most applications.

Module boundaries

During the modeling phase, we used the canvas to define bounded contexts, including their inbound and outbound communication. We can now reuse these contexts and directly translate them into modules. Moreover, we already have events, commands, and queries—it would be a mistake not to take advantage of this.

This is why I usually treat each bounded context as a module when I start a greenfield application using a modular monolith approach. It might change, but it is a perfect starting point.

> There are also other approaches for module definition. The most popular, next to the one mentioned, is either one subdomain or a single process per module.

Before we continue, I want to point out one more thing. You will not always work with the domain related to business; you may be building a technical solution, such as an algorithm library. What should you do in such a case? Can you also use a modular monolith for this?

Yes, you can, but your modules will look different. Suppose that you build this library with various algorithms. You could structure it based on the algorithm task. This way, you would have modules for:

- **Sorting.** It will include all sorting algorithms like bubble, insertion, selection, merge, or quick sort.
- **Searching.** It will include all searching algorithms like linear, binary, depth-first, breadth-first, or interpolation search.

You create a new module and include related algorithms each time you need to add a new algorithm task.

Let’s have a look what a modular monolith structure would look like for bounded contexts that we found during the business analysis of our healthcare domain.

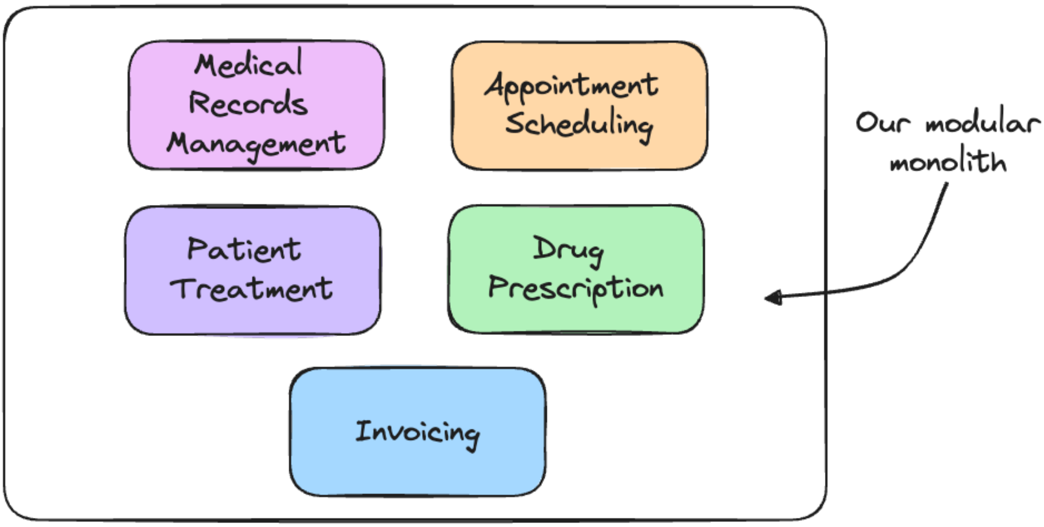


Figure 90. Modules based 1:1 on bounded contexts found during business domain analysis

There are five modules, each representing a selected bounded context:

- Medical Records Management
- Appointment Scheduling
- Patient Treatment
- Drug Prescription
- Invoicing

While this one-to-one mapping between bounded contexts and modules is often effective, it is not a universal rule. In some cases, you may need to structure your modules differently due to technical constraints, team organization, or specific system requirements. For example, you might combine two closely related bounded contexts into a single module for better cohesion, or split a large, complex bounded context into multiple modules for easier management. Always consider the unique needs of your project when deciding on the final modular structure.

Deploying changes

Modular monolith runs in a single process. This means that changing anything in one of its modules requires redeployment of the entire application.

Let's say you adjust the code in the *Appointment Scheduling* module. Next, you want to push it to production. The problem is that you cannot deploy just one module—all other modules, like *Patient Treatment* or *Drug Prescription*, will also be redeployed.

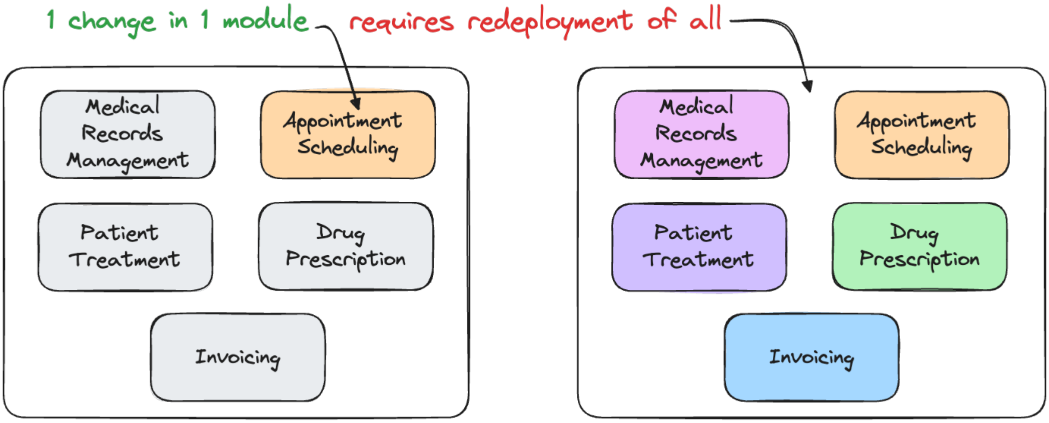


Figure 91. Change in one module requires redeployment of the entire modular monolith

Is this a problem? Well, it was not a big deal in most environments where I worked. But it does not mean this will also be true in your case. Issues usually appear over time when the application grows. The following circumstances are the ones I observed the most:

- **Increasing deployment time.** Running through all pipelines starts taking significant time. You need to constantly run all validation checks for all modules, not only the one you changed.
- **Several development teams.** Working on a modular monolith might be tricky when several development teams are involved. Each team might work relatively independently on their specific modules, but since all modules are part of a single application, it might be tricky to handle that many changes and common parts like building blocks (reusable middleware and other mechanisms).

- **Risk of failure**. An issue in one module could lead to cascading failures throughout the application, affecting areas not directly changed in the recent deployment. At some point, the risk and consequences of failure might be too high for your business.

Scaling

At some point, your application may need to scale. Scaling means adding new instances of the same API, database, or other component. In the case of the modular monolith, each time you add a new instance, the entire application is replicated.

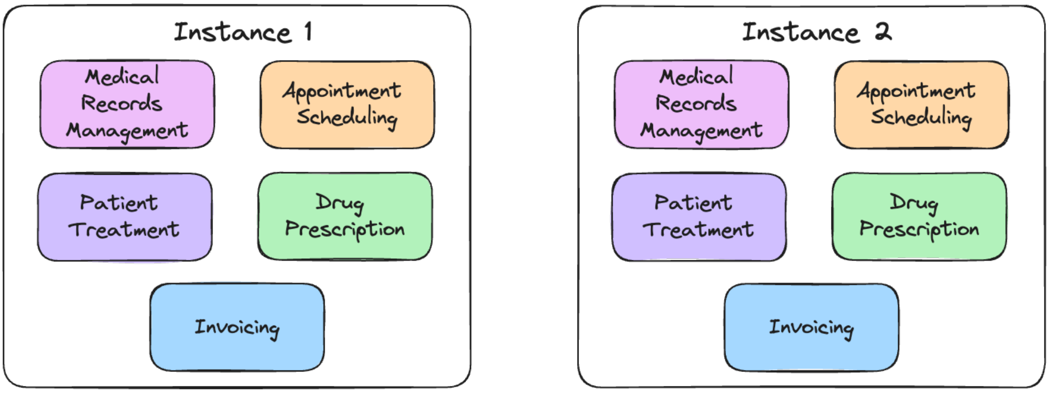


Figure 92. Multiple instances of our modular monolith

This works well with several instances but might lead to too high costs and overly complicated handling if too many of them exist.

Consider an environment where you must scale to 200 instances because of one heavily used functionality. You cannot scale it alone; you must scale the entire application.

Each instance costs an additional $50 (example value) per month. In comparison, scaling only one module would cost only $5. The difference is $9,000 per month - quite a bit. If the extraction is a one-time cost of $10,000, you would save almost $100,000 per year.

Adding a new instance is almost immediate when your application is relatively small. The problem starts when it takes dozens of seconds because your application is too large, and each time, it is replicated as a whole.

> Imagine that you are responsible for the application that sells tickets to some event—concert, football game, or e-sports.
>
> It takes dozens of seconds to add another instance. Tickets start selling at 10 a.m. I guarantee that there will be a huge increase in requests starting at 9:55 a.m. and peaking at 10 a.m.
>
> If you are not ready to scale quickly (cloud) or did not prepare enough instances earlier (on-premise), customers won't be able to access the site and buy tickets.
>
> Eventually, this will lead to dissatisfied customers and a potential loss of money.

Additionally, when running at least two instances, you must remember to distribute the traffic reasonably between them. Luckily, off-the-shelf solutions can handle this problem, but you will have another component in your setup.

Usually, in the modular monolith, you will handle traffic with a load balancer that will be either a cloud service or an external component.

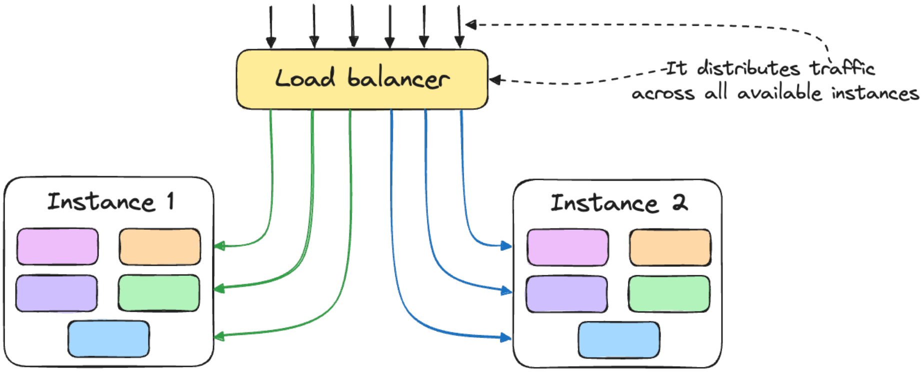


Figure 93. Load balancer distributes the traffic between 2 instances of the same application

After adding it to your setup, all incoming requests will pass through it to determine which instance they should be directed to. This approach helps evenly distribute traffic across all your instances.

The larger the application, the less sense it makes to scale it as a whole. But remember—every decision we make in software should be based on a profit and loss calculation. If extracting a module into a separate deployment unit will cost more than scaling the whole application for a year, it will likely make little sense. Make wise decisions!

Database

In most cases, your application needs to store the data somewhere. A database is one of the most common ways to do this. We have already divided business logic into cohesive modules. It would be great to do the same with our data.

Before isolating data, let's look at a typical big ball of mud application. When you add a new feature, you prepare a data structure for it. In the case of a relational database, this would be a table. This table has its columns. Then, you add new columns when this feature extends. Next, you add another feature, and the process repeats. Features begin to cross each other. The table from one feature references the one from another. At some point, you would have a massive database with dozens or hundreds of tables with too many references.

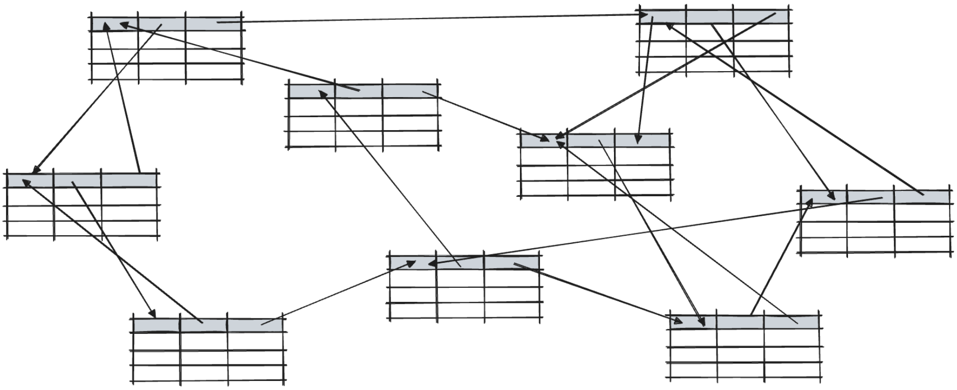


Figure 94. An overview of a database in a big ball of mud application

Imagine doing the same in a modular monolith. You have done a fantastic modularization at the code level, but already, at the database level, you start cooking spaghetti.

Sooner or later, someone will add another column, and it will turn out that the table used by *Appointment Scheduling* will also partially be used by the *Drug Prescription* module. After some time, someone will find it worth adding another column because it will be easier for the *Medical Records Management* module to use the data from it.

This way, our table can now be called *Medical RecordsOfDrugPrescriptionsForScheduledAppointments*. I can tell you one thing: you are already on a highway to hell.

What options do you have to prevent this?

Separate schemas, single database. Make the most straightforward move by leveraging the schemas. Schemas offer you the logical division of your data within one database. Each module will have its own schema. When changes occur in one module, you only need to adjust its corresponding schema, simplifying maintenance and reducing the risk of unintended side effects. To access data from another schema, you must explicitly specify the schema name in your queries, which promotes clear data boundaries and helps prevent accidental cross-module data access.

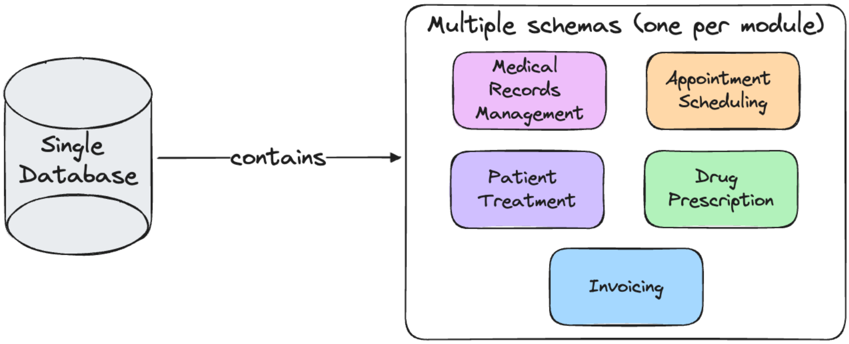


Figure 95. Schema per module approach running in a single database

Separate databases, same instance. As your application grows and your data management needs become more complex, you may consider evolving your architecture beyond schema-based isolation. The next logical step in this evolution is to physically separate the data—moving beyond schemas to separate databases while still running them on the same instance. This approach provides better isolation and the flexibility to move the database to another instance in almost no time.

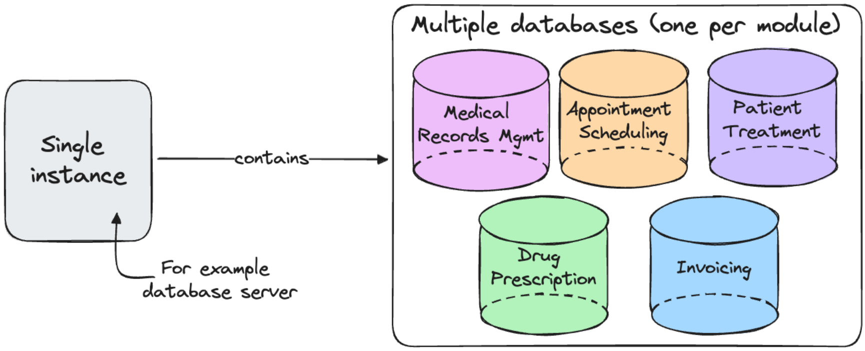


Figure 96. Database per module approach running on the same instance

You can migrate gradually from one approach to another. Imagine that two modules grow fast and are used more frequently than others. Extract them to a separate database that you can later optimize for reads or writes. The other modules will continue using schemas. Then, you observe that another module starts growing, so you also extract it to a separate database. Next, you can repeat these steps if needed, until all modules are successfully migrated.

Communication from the outside

Like other applications, modular monoliths can be called from the outside. Such calls are usually made by humans or other systems using the HTTP protocol.

Each module within a modular monolith can have its own public HTTP API, allowing clients to call module endpoints directly.

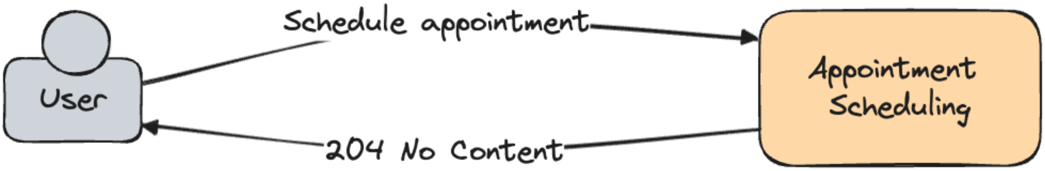


Figure 97. User calling the module's endpoint directly through its public API

An alternative approach is to place an API gateway in front of the modular monolith. This gateway serves as a single point of entry for all external communication. It provides several advantages in managing interactions with the system. The gateway can handle traffic management, ensuring that requests are properly routed to the appropriate modules. It also provides an authorization layer that enhances security by controlling access to different parts of the system.

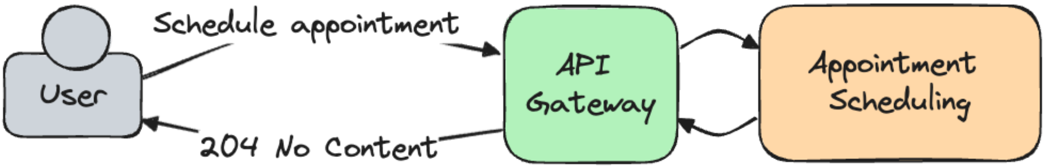


Figure 98. User calling the module's endpoint through the API gateway

The API gateway can also implement rate limiting. This feature throttles calls when there are too many requests, preventing system overload and ensuring fair usage.

Note that this approach requires adding another external component to your application architecture, which you must deploy, host, and pay for.

Communication between modules

There are three main methods of handling communication between modules:

- Using synchronous communication (tight coupling, request-response)
- Using asynchronous communication (loose coupling, event-based)
- Combination of both

Synchronous communication can be implemented in various ways, for example:

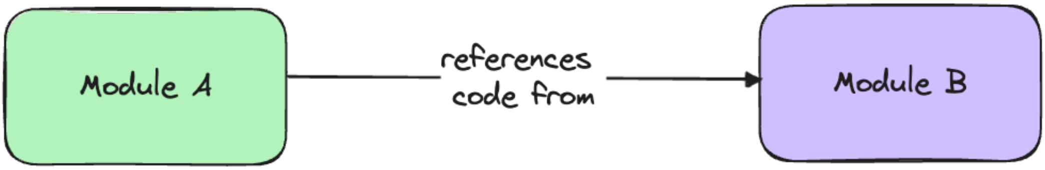


Figure 99. One module references and calls another directly

It looks easy. When one module needs something, it references another one and uses its logic. However, it creates tight coupling between them, which can lead to spaghetti code, as one module has full access to other modules' logic.

This problem can be partially solved by another approach—a public API. It is similar to directly referencing one module from another (you still need to do that) but offers an important advantage.

Instead of allowing full access to all the code in a module, you create a public API for it. This API only exposes specific methods that other modules are allowed to use (acts as a facade). All other internal logic and details of the module remain hidden.

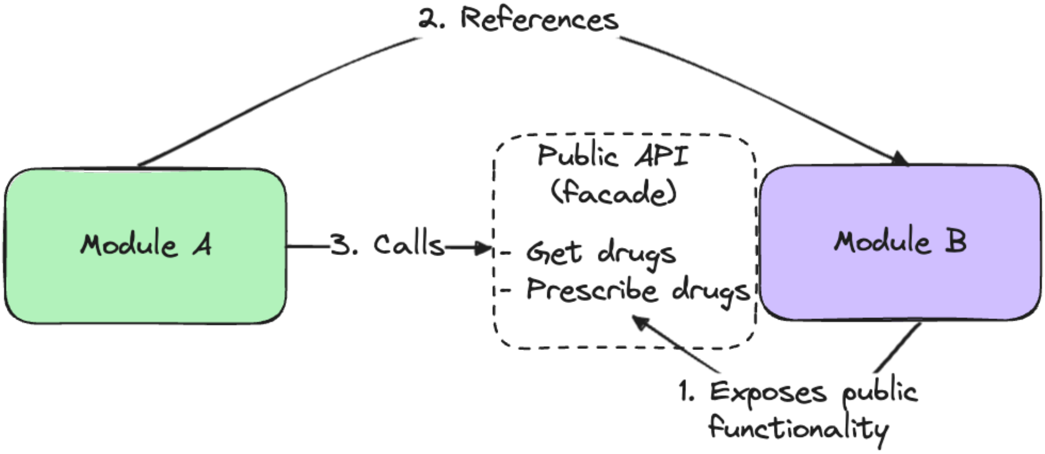


Figure 100. One module references and uses another via its public API

When I say "public API" here, I am not talking about REST APIs or network communication. I mean a set of public methods or functions that other modules can call by reference.

Advantages? If done correctly, there is no chance that another module will impact the logic as it is hidden. It works similarly to REST API—you know what actions you can call, but you do not see how they are implemented.

The disadvantage is that it still requires coupling—one module must know about the other module's existence.

The next option is to manage module communication by using a gateway. Instead of referencing module B directly, you create a simple gateway to handle communication with it. From this point on, each module will only reference and use this gateway.

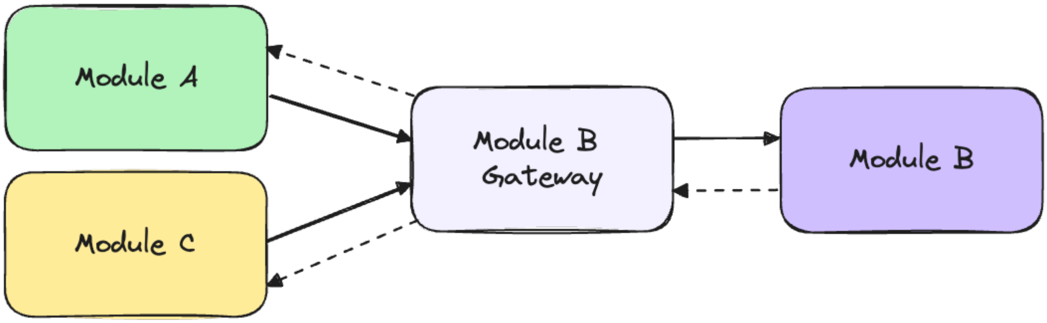


Figure 101. Modules call another via gateway

The advantage? No tight coupling to a specific module. This can make the system more flexible and easier to modify in the future.

Disadvantage? Increased complexity. Adding a gateway adds another layer to the system. This can make the overall architecture more complex, and potentially harder to understand and maintain.

It is important to remember that all communication between modules takes place within a single process. There are no HTTP calls. Using HTTP calls to communicate between modules makes no sense, because you have to do it over the network. A lot can go wrong (network errors), and it will be slower (latency).

Another approach is to use events for inter-module communication. As mentioned earlier in the general communication section, this can be implemented using methods like in-memory messaging, external broker, database, or files.

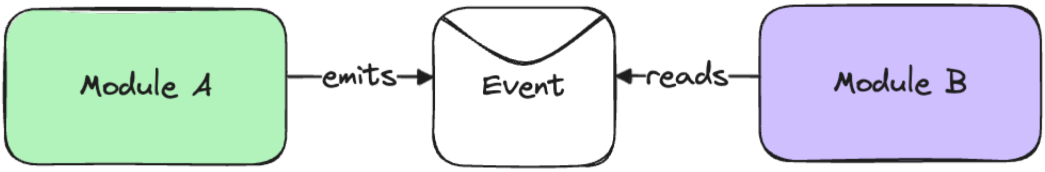


Figure 102. One module emits an event and another reads it

If you start building your application based on modular monolith and want to use event-based communication, I recommend using an in-memory queue first. This is the simplest setup for asynchronous communication in applications that run in a single process.

Some of my friends rightly point out that using an external broker in a modular monolith loses the advantages of a system that runs in the same process. We have an additional component and are exposed to the same problems that arise in distributed systems, such as network errors and higher latency. Such argumentation makes much sense.

Consider migrating to an external broker when you see a growing user base and high traffic in some modules. Migrating to an external broker while still working with a modular monolith is an excellent preparation for possible future distribution (multiple deployment units). This way, you can extract any module into a separate unit (e.g., a microservice) at any time because inter-process communication is already guaranteed. However, you must remember that by introducing an external broker, you lose the advantage of running the application in the same process—fast and reliable calls without network communication.

You don’t always have to go one way or the other. It is also possible to combine both approaches. Sometimes it makes sense to call another module using direct calls, while other times it makes sense to communicate using events.

Stay pragmatic. If you think something should be done a certain way, do it. There is no one-size-fits-all approach to architecture—it is about finding what works best for the given context.

When to decide on it?

The question of when to use a modular monolith often arises in discussions of system design. While it is often recommended for new applications, it is important to approach this advice with caution. There is a tendency to view the modular monolith as a universal solution, leading to thoughts such as *I am starting a new project, and everyone says modular monoliths are great for greenfield applications, so that's what I am going to use.* However, this oversimplified approach can be problematic.

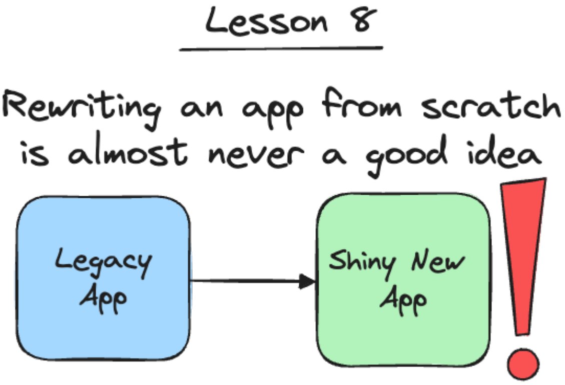
Each environment and the application we work with differ significantly. Over the years, I have prepared some heuristics that I use when deciding on modular monolith:

- **Lots of unknowns**. When project requirements are ambiguous or frequently changing due to factors like evolving market trends, unclear stakeholder expectations, or experimental product development, adopting this approach can provide greater adaptability. This design allows for a more straightforward incorporation of future changes without the added complexity and overhead associated with distributed systems.
- **Expecting average numbers**. When you forecast that your application will be used by several or a dozen thousand users, where write and read operations won't be too long, it makes sense to go with modular monolith and scale it if needed. Sometimes, however, despite the low number of users, operations will take much time. Imagine that 1,000 hardware devices will synchronize their data several times per day. The synchronization process takes up to 15 minutes, and the data amount is between 5 and 50GB per device. In such cases, modular monolith might not be the best choice.
- **Single development team**. There is no need to complicate things by deciding on a distributed system while having just one development team. Taking care of a single codebase and deployment unit will be much easier. When the team grows, and you have to split it into another one, you can revisit your decision.
- **Lack of experience with distributed systems**. If we haven't yet had a chance to work with distributed systems, we often tend to glorify them, elevating them to almost mythical status. What can go wrong? Unfortunately, a lot. It requires knowledge and expertise in fault tolerance, network communication, scalability, orchestration, and other topics.

- **Full responsibility for infrastructure.** When you cannot count on any other team regarding infrastructure, you must deal with it alone. You have to worry about all the infrastructure components, so it is easier if there are fewer of them.

One might ask about using a modular monolith approach for legacy applications. Can I start refactoring it towards modular monolith? Of course, you can. If you are working in a big ball of mud, try to find the simplest process. Next, decompose it into a cohesive module(s). Finally, repeat it for other parts.

Too often, when dealing with legacy applications, one of the first ideas is to rewrite everything from scratch. You will miss many grey zones (that happen outside of your application), parts of existing processes, and valuable knowledge built up over the years. I have tried it several times, and this approach has consistently failed. In 99% of cases, it is better to follow the step-by-step solution (behavioral decomposition, tactical forking, strangler fig pattern).



Further education

I tried my best to present the most important aspects, but it was only possible to cover some of them in such a limited space.

I do not know any good book directly related to modular monoliths. However, I can recommend a few resources:

1. **Software Architecture: The Hard Parts**. The book[3] describes the process of architecture modularization and shows how to decompose a monolithic application. It is written as a story where engineers face a big ball of mud, then extract modules from it and distribute them.

There are two repositories worth checking out. Kamil Grzybek created one directly focused on the modular monolith. The second one is the result of our work—mine and Kamil Bączek—where we show an evolutionary approach to architecture.

2. **Modular Monolith with DDD by Kamil Grzybek**. Repository[4] shows how to build your application using modular monolith for meetings organization. It deeply describes its concepts and shows how you can approach it using both strategic and tactical Domain-Driven Design.
3. **Evolutionary Architecture by Example by Maciej Jedrzejewski and Kamil Bączek**. Repository[5] that shows how you can build your application using an evolutionary approach. It is divided into four chapters: Simplicity, Maintainability, Growth, and Complexity. The first two focus on the modular monolith, and the second two on a combination of the modular monolith and microservices.

Happy learning!

Microservices

Microservices are one of the most common (but not the only) representations of multiple deployment units. Each microservice acts as a separate and independent unit.

[3] https://www.goodreads.com/book/show/58153482-software-architecture
[4] https://github.com/kgrzybek/modular-monolith-with-ddd
[5] https://github.com/evolutionary-architecture/evolutionary-architecture-by-example

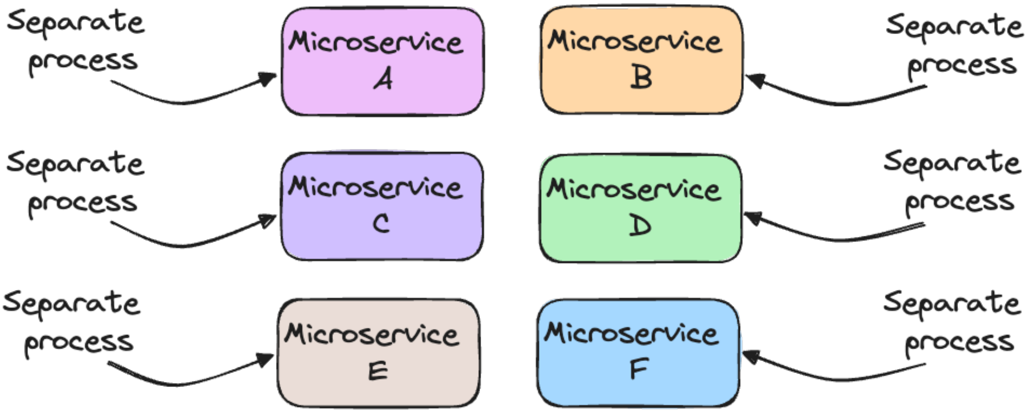


Figure 103. Application built on top of independent microservices

To be considered a distributed system, the application must be built on at least two microservices. Otherwise, it would be monolithic.

During my career, I have seen applications with 5, 100, and even over 500 microservices. The largest systems can be based on thousands of them. Therefore, defining a precise number and saying, *OK, X will be enough*, is impossible.

Microservices should have clear boundaries, set around selected business capabilities. Otherwise, you will end up with microservices that are too chatty. If you want to perform one operation, and the microservice has to contact five other microservices, then it indicates a possible mistake while setting boundaries.

One of the main advantages is that when you adjust something in one microservice, you can redeploy it without impacting others. When you need to scale, you scale it independently from others.

The trade-off is that it increases the system's complexity. It requires decent observability. Additionally, you expose the system to errors related to network communication and more vulnerabilities. There are other issues, and you will learn about them in the upcoming sections.

That is why it would be great to base your decision on numbers, such as the number of users, requests, development teams, data amount, and profit and loss calculation, in case you decide to go with microservices, as it is a highly complex topic.

Microservice boundaries

At first glance, it can be assumed that a microservice is simply an extracted module. As you remember from the modular monolith section, my favorite way of defining it is to take the bounded context and map it one to one. A similar approach can be applied to microservices, especially when you start with limited knowledge about the numbers in the greenfield environment.

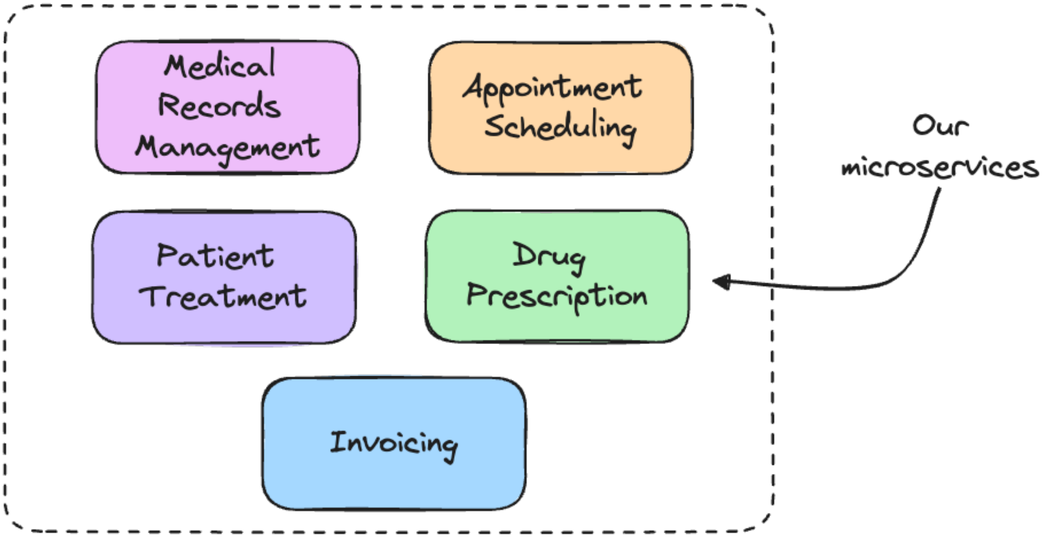


Figure 104. Microservices based 1:1 on bounded contexts discovered during business domain analysis

But there is a catch here. A modular monolith runs in the same process. This means that modules can communicate with each other without using the network. If one module calls another one multiple times, nothing special happens. In contrast, microservices communicate over the network (separate processes). The more calls between them, the worse the performance and the higher the chance of a network error.

That is why when you find that two modules are close to each other and often communicate, consider having one microservice cover both of them. It is an extra point when one of the modules is relatively small.

In our application, this might be a combination of *Drug Prescription* (used to prescribe drugs) and *Drug Availability* (used to check if the selected drug is available). Each time a doctor wants to provide a prescription, he has to check the availability of the drug.

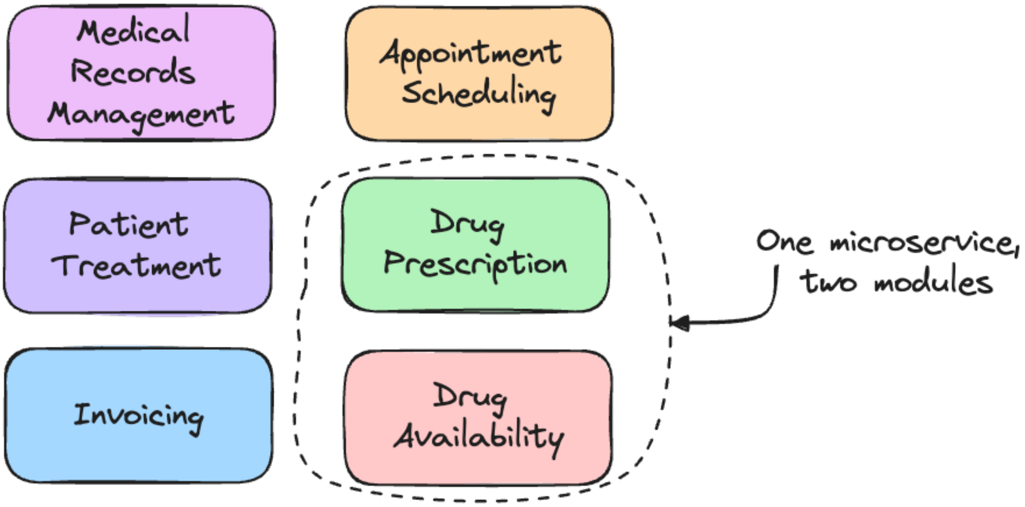


Figure 105. Two modules wrapped into one microservice

Another thing to remember is that the boundaries that we initially set will grow over time. It might make sense to spread them to other boundaries at some point.

> Imagine that we have implemented reminder functionality in *Appointment Scheduling*. The patient wants to be reminded that the scheduled appointment is coming up. There is a default reminder (one day before) and the possibility to add more reminders. Compared to the MVP version, we have also added functionality to manage reminders (list, remove, change).

It used to work quite well, but now, as we serve 100,000 patients, it has become a bottleneck. The team decides to extract it to *Appointment Reminding* microservice.

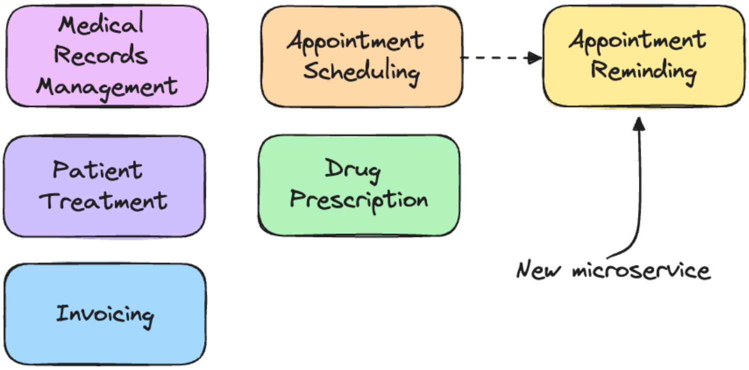


Figure 106. Extraction of the reminder functionality into a separate microservice

Instead of scaling the entire microservice responsible for *Appointment Scheduling*, we scale only the one responsible for reminders. This way, we gain more flexibility.

Deploying changes

In monolithic applications, you must deploy everything together, even if you change one line of code in one area. In the microservices world, this is entirely different. As each microservice is an independent unit that runs in its own process, you only need to redeploy the one you changed. All others stay untouched.

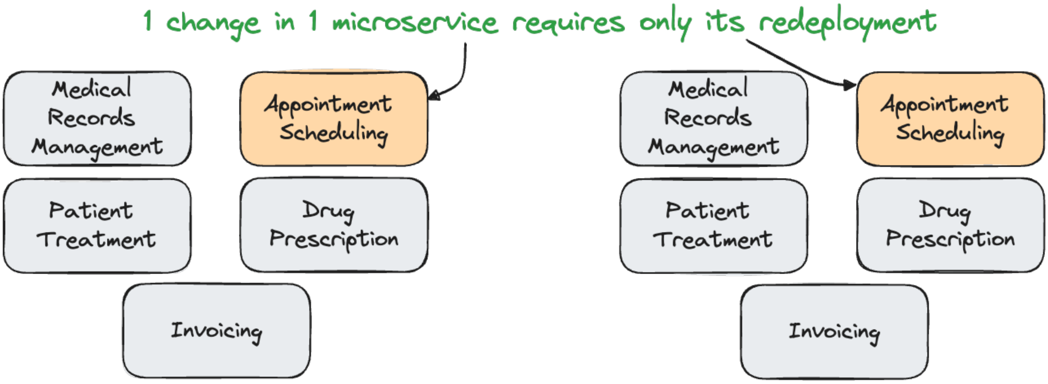


Figure 107. Change in one of the microservices requires redeployment of only that microservice

This is what I like most about distributed systems—the change isolation. What do we get as a result?

- **Decreased deployment time**. When you need to deploy only part of your application, it will naturally be faster compared to deploying the entire system. This capability allows for quick validation of changes and enables fast rollback if any issues arise.
- **Nice solution for several development teams**. You can assign teams per boundaries represented by microservice(s). This way, they own part of the software and become less dependent on other teams. Additionally, as the code is probably split, they do not affect other teams with their changes.
- **Reduced risk**. Even if one microservice encounters a problem and stops working, the rest of the application can continue to work smoothly because each runs in its own isolated process. Keep in mind, however, that this is only possible if other microservices can act without calling the failing one.

Scaling

One of the noteworthy advantages of microservices is that they can scale independently. You may find that four out of the 16 microservices need to scale to at least three instances, while another two need to scale up to ten. No problem. Your setup will look like this:

- 10 microservices running at 1 instance each
- 4 microservices running on 3 instances each
- 2 microservices running on 10 instances each

In total, you run $10 \times 1 + 4 \times 3 + 2 \times 10 = 42$ instances. You scale where necessary. Also, scaling is fast because there is less code and fewer verification checks. Here is an example of scaling in our healthcare domain:

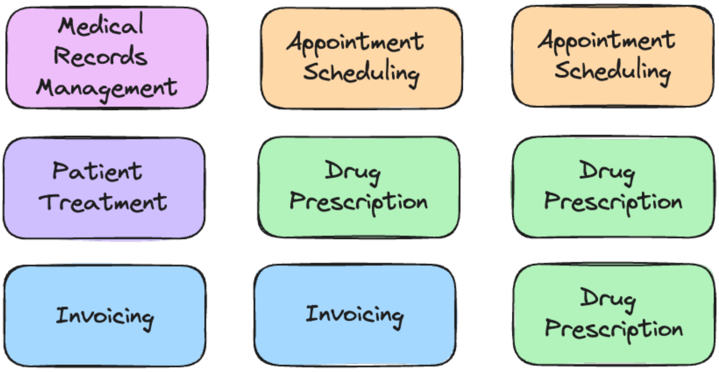


Figure 108. Each microservice can scale independently

Of course, whenever there are at least two instances, you need to balance traffic between them. What can you do? Use the load balancer, which is similar to dealing with multiple instances in the modular monolith.

The load balancer takes all requests and distributes them evenly among all the available instances of each microservice.

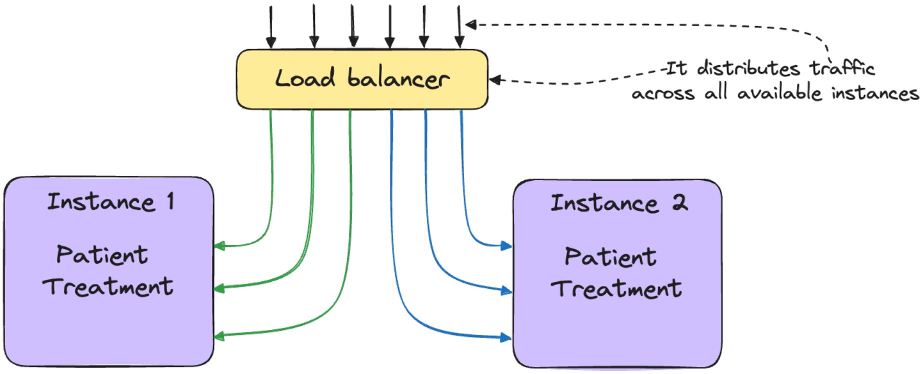


Figure 109. Load balancer distributes the traffic between 2 instances of the same microservice

When one feature inside the microservice is heavily used, consider extracting it to either a nanoservice or a serverless function. This way, you can optimize the costs that you generate with a large number of microservice instances.

Database

Both patterns presented in a modular monolith can also be used for microservices. As a reminder, these are:

- Separate schemas, same instance
- Separate databases, same instance

Following the above approaches means that even if your microservices run in separate processes, they still connect to the same instance to access their data. When the instance goes down, none of the microservices can access any data. This is where the third solution comes in handy.

Separate databases, multiple instances. First, you can assume that each microservice will access a separate instance on which the desired database is located. Also, there is a chance that several microservices will have their databases on one instance and others on another. There are cases where a considerable approach is to have all databases for all microservices on one instance and then replicate all of them in other instances.

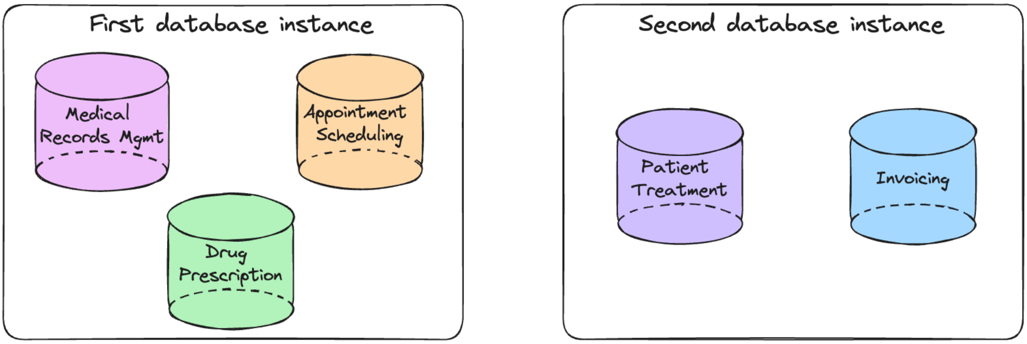


Figure 110. Microservices databases running on multiple instances

Selecting the appropriate approach depends on the size and the usage of the application. When it is relatively small and a single instance is enough, consider adding another instance (replica) as a backup.

When your application grows and is too large for one instance, spread the databases into multiple instances. Each instance could have its replica for backup purposes as well.

If you encounter a situation where a microservice spans two modules, you can either use two databases (one per module) or one database with two separate schemas.

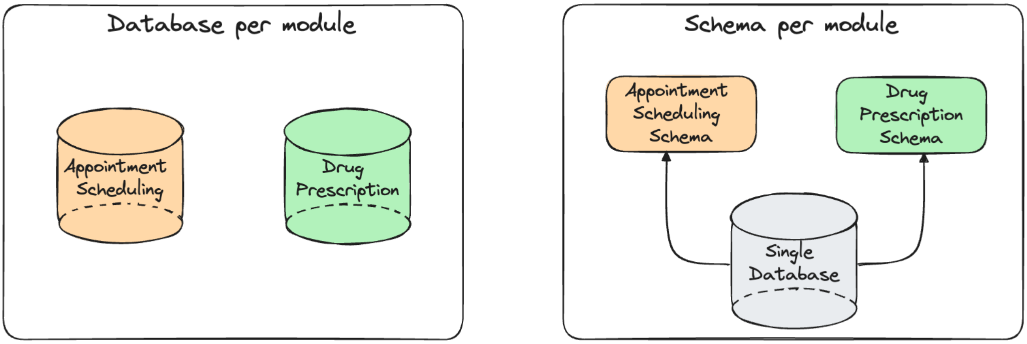


Figure 111. Possible database solutions when a single microservice covers two modules

Communication from the outside

Microservices can be invoked directly or through an entry point, such as an API gateway. Calls are typically made using the HTTP protocol. Every microservice has its own public HTTP API.

Direct communication with microservices, while possible, presents several challenges. You need to authorize access to each of them, handle rate limiting separately, and expose them to the public, which can increase security risks as there are more entry points. Additionally, building microservices using various technologies will complicate things even more, requiring separate solutions for all of the above. These complexities underscore the need for a centralized solution.

In practice, such communication is often centralized using an API gateway. This approach allows you to handle all of the above with a single component and can reduce the risk of vulnerabilities. While it introduces a single point of failure, you can mitigate this risk by running more than one instance. Moreover, it significantly simplifies the maintenance of the application, saving you from the tedious task of repeating all the steps for dozens of microservices or every time you add a new one.

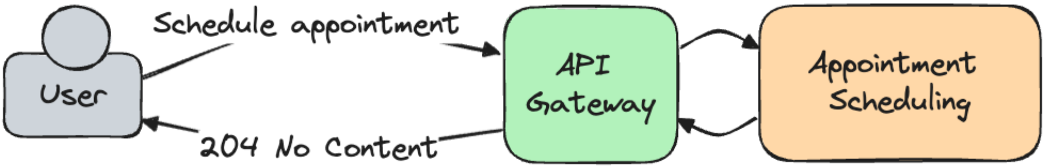


Figure 112. User calls microservice endpoint via API gateway

Communication between microservices

In distributed systems, inter-service communication is handled over the network. Each microservice runs in a separate process, so there is no physical way to reference code directly from one microservice to another.

While handling network communication, you are exposed to problems like partitions or latency. You should never assume that the network is 100% reliable. There will be network partitions that will lead to the division into two subnetworks being unable to communicate with each other.

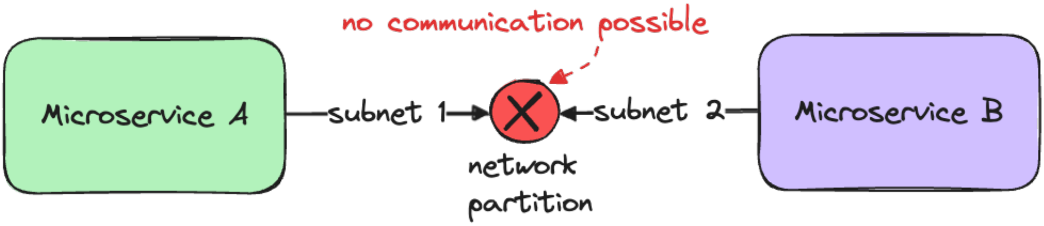


Figure 113. Network communication between two microservices

It would help if you planned what to do in such a case and how to ensure the information is recovered.

When you decide on the communication type between microservices, you usually select one of below approaches:

- Synchronous communication (temporary tight coupling, response-request)
- Asynchronous communication (loose coupling, event-based)
- Combination of both

Ideally, each microservice operates as a separate unit, unaware of others, promoting loose coupling. In practice, there are cases where temporary tight coupling is an acceptable solution, for example, when you need an immediate response.

In synchronous communication, one microservice calls another using the HTTP protocol, and most likely REST or gRPC. It creates a temporary tight coupling—there is a request, and now it must wait for a response. The call can be made through direct communication or through an intermediator.

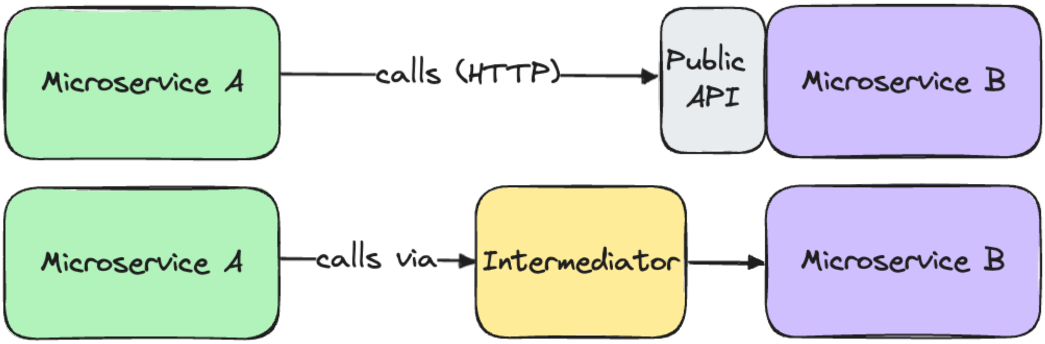


Figure 114. Calling one microservice from another one using a direct HTTP call or an intermediator

One advantage of this type of communication is the immediate response. Call the endpoint and receive its processing result—fast and easy. On the other hand, tight coupling is created between microservices.

What are the possible issues?

When microservice B is unavailable, microservice A must wait and retry the call. This may take a long time, and if a complete failure occurs after several calls, it must be handled somehow. Furthermore, if microservice B has to call an external system or experiences high latency, the response time will also be affected.

Another issue would appear when one microservice needs to use data from another one. Either A will call B each time (inefficient), or you will need to introduce another component—a cache. Now, each time B changes the data, it must update the cache and inform A that it has changed. This way, A can react to the changes. As your system grows, one microservice might need to communicate with several others, complicating the situation even more.

> Imagine that there is an event organizing application.
>
> Someone creates an event (*Events Registration Microservice*). Now, this microservice has to inform another one (*Ticketing Microservice*) that there is a new event for which it has to create a pool of tickets.
>
> Additionally, it must book a location for the event (*Location Booking Microservice*).
>
> Next, catering has to be planned (*Catering Microservice*). To plan catering, food must be ordered (*Food Ordering Microservice*)

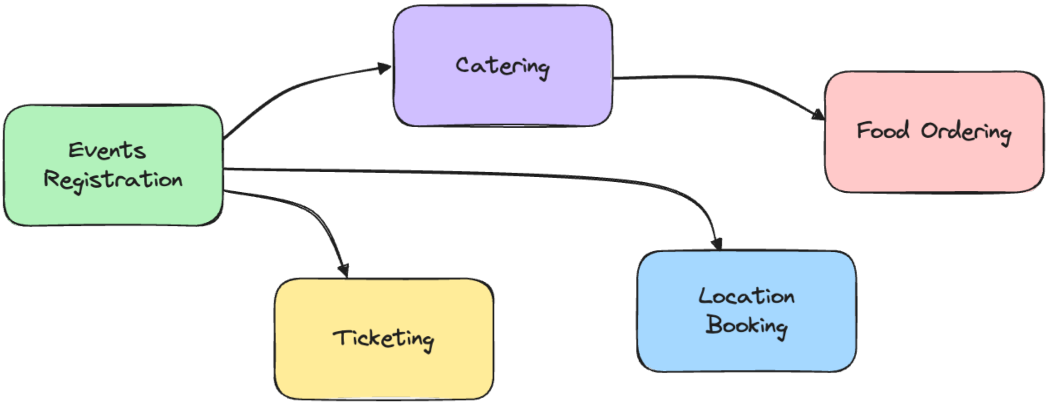


Figure 115. Microservices that use direct calls to inform other microservices of the results of their actions

Synchronous communication has its place in distributed systems, especially when immediate responses are necessary, but it is important to be aware of the mentioned trade-offs.

In asynchronous communication, one microservice does not call another directly. Instead, it uses event-based communication to inform others about the result of its operation. This allows microservices to be loosely coupled.

One of the most popular ways to exchange events in distributed systems is to use external components like message brokers that allow inter-service communication through messaging protocols.

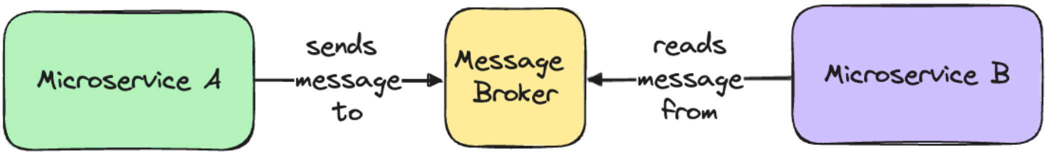


Figure 116. One microservice sends a message to a message broker while another reads that message

> The most popular message protocols are *AQMP (Advanced Message Queuing Protocol)* and *MQTT (Message Queuing Telemetry Transport).* There is a high chance that you will use one of them.

One microservice sends a message (that represents an event) to the message broker, e.g., *Treatment completed*; the other reads it and can send the invoice to the patient. Both microservices do not know about each other as they use an intermediator (message broker) in between.

Well, that is fine, but now we have a focal point that can fail - a message broker. The good news is that you can have multiple instances (replicas) of it. That way, if one instance goes down, you have another one that will work. I prefer to have three instances. Two instances provide basic redundancy, but there is still a risk. If one goes down, you are back to a single point of failure. Also, with three instances, you can do maintenance or updates on one instance while maintaining full redundancy with the other two.

One of the key advantages of asynchronous communication in distributed systems is its non-blocking nature. You don't have to wait for a response to your request. Instead, you inform others about an event, and they react to it at their own pace.

However, this approach introduces eventual consistency. The state of the data might differ between two or more microservices. Customers may operate on outdated data as the message has not yet been processed. If this is a problem, consider another approach.

> It is important to remember that the choice of communication type is not a binary decision. Your system may have varying requirements across different areas. Some processes may demand strong consistency, while others may be fine with eventual consistency. Feel free to mix both of them.

In contrast to the monolithic approach, I recommend using event-based communication as the default approach for microservices. Direct calls (temporary coupling) should be reserved for specific scenarios where it is truly necessary.

Of course, implementing mechanisms like outbox, inbox, and dead letter queue is also advised to handle unexpected issues.

When to decide on it?

My usual recommendation is to start smoothly with a single deployment unit and then, based on the needs, slowly evolve part by part to microservices.

When evaluating the need for microservices, I consider several key factors. In my experience, the presence of at least two of these factors often signals that a transition to microservices may be beneficial:

- **You know the numbers**. Sometimes, some applications replace legacy ones. If you know that as a company, you have one million customers, and these customers use your old application (or parts of it) heavily, this is an indicator that microservices might be worth it.
- **Going off the scale**. Up to a certain point, scaling the entire application is not a big deal and can be achieved in a monolith. However, when it becomes trendy, it might be unprofitable to scale more in the selected model sooner or later. It is time for you to choose those modules that are used most often and extract them to separate microservices. At stake—scaling vs. extraction costs.
- **Frequency of changes**. The code in some modules may be modified much more frequently than in others. The problem is that each time a change is made, you have to deploy an entire unit that includes, for example, six unchanged modules along with the two that were modified. If this situation occurs repeatedly, it is a sign that a change is needed—extract these two frequently updated modules to separate units.
- **High usage of a module**. If you find that the module responsible for calculating potential tax is extensively used by your customers, and it is the only one with such high usage, you face inefficiencies. Scaling the entire application to handle this traffic is costly and time-consuming. This is a strong indicator that you should extract and scale only this heavily used module.

- **Different requirements for security**. In some applications, some areas require increased attention regarding security. If your change affects the entire application, it may be an ill-advised choice. Analyze the situation and see if you also need such a high level of security elsewhere. If not, maybe this is the right time to extract this module.
- **Number of development teams**. It is relatively easy to drive the implementation with a single development team of 5 people. They can manage the code changes and rarely run into conflicts when merging their work. However, as your application grows, one team might not be enough. In larger organizations, you might even need 2-3 teams already from the start. When multiple teams are involved, it becomes harder to coordinate work on a single, large codebase.

Further education

Microservices are complex. No single book, let alone a section, can cover everything. Here are some books I recommend:

1. **Building Microservices: Designing Fine-Grained Systems by Sam Newman**. My favorite book[6] on microservices. You will find knowledge about deployment, testing, observability, security, etc. An absolute must-read if you work daily with microservices.
2. **Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith by Sam Newman**. This book[7] is an excellent addition to the above. Here, you will learn about different patterns that will support you while migrating from monolith to microservices.
3. **Strategic Monoliths and Microservices: Driving Innovation Using Purposeful Architecture by Vaughn Vernon and Tomasz Jaskula**. A completely different book[8] from the others. It focuses more on strategic thinking, how to set goals for your organization, how to reach domain-driven results, and also how to move both modular monolith and big ball of mud to microservices.
4. **Designing Data-Intensive Applications by Martin Kleppmann**. If you are looking to deepen your knowledge of data handling, this book[9] is an absolute must-read.

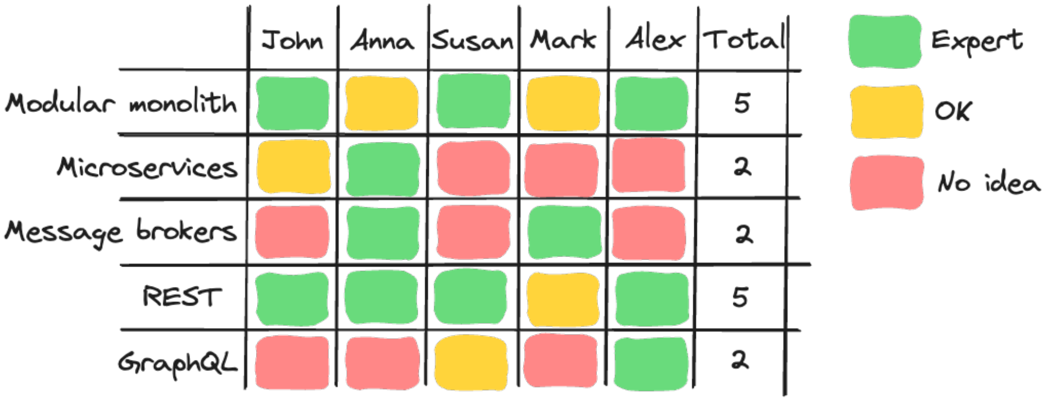
[6] https://www.goodreads.com/book/show/22512931-building-microservices
[7] https://www.goodreads.com/book/show/44144499-monolith-to-microservices
[8] https://www.goodreads.com/book/show/55782292-strategic-monoliths-and-microservices
[9] https://www.goodreads.com/pl/book/show/23463279

I am sure you will enjoy reading these books. Over the past 12 years, I have encountered both high-quality and subpar materials, and these are definitely worth every minute of reading.

Our case

As in previous steps, we need to decide on the best options for our application that will support private medical clinics. Before you continue to read, look at the case.

First, we gathered together with all team members and conducted a quick session to assess our skills related to this area.



	John	Anna	Susan	Mark	Alex	Total
Modular monolith	🟩	🟨	🟩	🟨	🟩	5
Microservices	🟨	🟩	🟥	🟥	🟥	2
Message brokers	🟥	🟩	🟥	🟩	🟥	2
REST	🟩	🟩	🟩	🟨	🟩	5
GraphQL	🟥	🟥	🟨	🟥	🟩	2

🟩	Expert
🟨	OK
🟥	No idea

Figure 117. Team skill assessment

Next, we looked at key points:

1. There are 5 bounded contexts. Additionally, we need to handle account registration and login processes.
2. The expected number of private medical clinics that will use our application in the first months: 3-5.
3. SLA is 99.5%. The expected number of patients is relatively low: 1,000-1,500 (< 1s for response).
4. The current dev team is inexperienced in distributed systems.
5. There is no existing infrastructure.

6. We have a budget limitation of a maximum of $500,000.
7. The deadline is set for two months from now.

Finally, we focused on architectural drivers:

Consideration	Architectural driver
5 bounded contexts	Functional Requirement
Account registration functionality	Functional Requirement
Login functionality	Functional Requirement
Budget limitation	Business Constraint
Deadline	Business Constraint
Low experience in distributed systems	Technical Constraint
No existing infrastructure	Technical Constraint
SLA	Quality Attribute (Availability)
Response < 1s	Quality Attribute (Performance)

And selected key ones:

- **Simplicity**. We need to cover only several functional areas. We have an inexperienced dev team, and there needs to be an existing infrastructure. There are max. 1,500 users planned (not a high number). Having in mind that there is a deadline in two months, there is no other option but to focus on the simplest possible solution.
- **Performance**. Our application users (1,000-1,500) should receive a response under 1 second. Depending on the number of requests, it might affect scalability, but not in our case.
- **Cost-Efficiency**. Due to our budget limitations, we should focus first on delivering functionality. This means that it would be best to keep basic infrastructure and most likely delegate as much as possible to the cloud provider.
- **Availability**. The agreed SLA is at 99.5% level. This allows for up to 1 day, 19 hours, 28 minutes, and 8.8 seconds of downtime per year. That is good because we can still use some basic infrastructure to meet this requirement, so we keep our costs low.

Considering the above, we decide to go with a single deployment unit (modular monolith). In the future, depending on our needs, we might extract it part by part to multiple deployment units.

Recap

In this step, you learned about single and multiple deployment units and the nuances of synchronous and asynchronous communication using commands, queries, and events.

Furthermore, we explored the inbox and outbox patterns needed for reliable message delivery and processing. You now understand various ways of handling messages, including at-most-once delivery, at-least-once delivery, and exactly-once processing.

I explained the concept of a modular monolith, covering how to handle communication within it, planning the database, and discussing strategies for scaling and deploying changes. This will help you understand how to balance modularity and monolithic simplicity.

Additionally, we explored microservices, focusing on their communication methods, database planning, scaling, and deployment. You learned about problems that occur there, like network errors and high latency.

Now, you should have a solid knowledge of when to apply each of these concepts based on your project requirements and environment constraints.

It is time to focus on the release strategy.

STEP 5: Define Release Strategy

Once you have a deployment strategy in place, you need to decide how often you want to release new versions of your application to customers and the approach you will use. Over the years, I have seen various approaches. Here are the most popular ones:

1. **Release application several (usually 4-6) times yearly**. Each release preparation takes around one month, sometimes more. It involves manual testing by the QA team, then project (or product) managers, fixing bugs, and only then release to the public.
2. **Release application based on release trains**. A set of features or changes are bundled together and released at the same time. There is a deadline, e.g., in 2 months. Until then, all teams will work on features and release them together. When you cannot deliver on time, the feature will have to wait for another release train.
3. **Release application multiple times per month (2-4)**. Each release is based on the outcome of a week or two of work. If the team works in Scrum, it is often tightly coupled to sprints—release at the end of each sprint.
4. **Release the application multiple times per week**. There is a release in the morning or afternoon every day. It contains all commits pushed that day.
5. **Release multiple times per day**. This approach allows you to release your application hundreds or even thousands of times per year, with multiple releases per day. While manual validation may be sufficient for less frequent releases, as deployment rates increase, the process becomes increasingly burdensome without automation.

Choosing the right approach that aligns with your unique business needs is a complex task. It requires careful consideration and close collaboration between technical and non-technical people.

You might work for a company that delivers social media or e-commerce application, where nothing stops you from continuous releases that can happen multiple times a day.

However, you may find that you can't show new features to your customers because a marketing campaign must be run first or a feature has to go through an external audit (e.g., in public systems). In such cases, nothing stops you from continuous application releases, but you might need to hide new features until the official release.

You might also deliver software to hardware devices. Assuming that such a device might damage someone's health or cause someone's death, each version has to first go through a complex validation process similar to Waterfall, and it might take weeks before it gets approved. You can't release continuously but it would be great to be able to do it right away whenever you are asked.

Another thing to consider is the environment around you. You may have to adapt to current processes in your organization. If there is a release train and all teams have to wait for it, then there is a high chance that you will need to adapt to it, too.

Sometimes releasing multiple times a day is not worth the infrastructure overhead. Deploying once a day or three times a week may be sufficient.

Still, let's not forget about the excuses.

The most common one I hear when working with web applications is:

> *We have a special case where it is impossible to release often, so we do it six times a year.*

Releasing a non-critical application only a few times a year can cause several issues. You are exposed to longer feedback loops, larger and riskier releases, and compatibility issues such as dependencies on third-party libraries or systems. If you cannot release a quick fix for a critical bug, it is a sign of poor process. Additionally, developers might slowly become demotivated because there will be long lead times before they see the work they did. As a result, demotivation can reduce their productivity. Finally, competitors releasing more often and faster can implement new features and adapt to market trends better. When customers see that, they might perceive infrequent updates as a lack of innovation.

In the following sections, you will learn about various strategies, release automation, and a pragmatic approach to doing things that takes into account the specific context and constraints of your business.

Release versus deployment

Technically, every release is tightly coupled with deployment. Each time you deploy the new version of the application to production, it is also released to customers.

As a result, there are three scenarios:

1. Changes are immediately visible to all users.
2. Changes are visible only to a subset of users.
3. Changes are hidden from all users.

But why would you hide features that have just been deployed? There are a few reasons, each with its own implications.

First, you strategically choose to show it only to early adopters. Their feedback becomes a crucial part of the decision-making process, helping you determine whether to roll out the feature to other users or not. If not, you can improve or remove it, ensuring that your product evolves based on real user needs.

Additionally, hiding features can be part of an A/B testing strategy, where you compare user behavior and engagement between groups with and without access to the new feature.

There is also a chance that your company sends release notes to all customers each month in a digital brochure or a video, informing them about all new features. So, it might be obligatory to hide them, but it should not stop you from regular deployments.

Finally, you may want to hide new features to match the company's marketing plans. Sometimes a company wants to reveal new features at a specific time, such as during an industry event or as part of a marketing campaign.

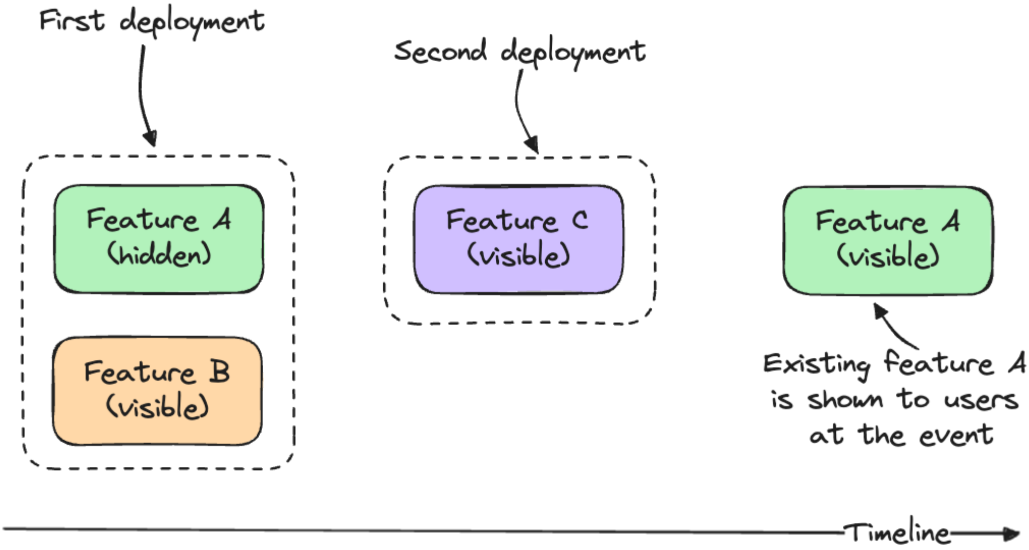


Figure 118. Feature A is hidden until the event while other features are immediately accessible

Deployment strategies

In addition to the deployment strategies you learned about in the previous step, this terminology is also used to define a strategy for releasing new application versions to the production environment.

While there are numerous strategies, I will focus on those I have encountered most often. Some require support for multiple versions simultaneously, meaning that the code you put into production must be backward compatible. It does not apply only to code-one of the most frequently discussed topics in this context is how to ensure compatibility for the database, especially in the case of a potential rollback to a previous version.

Basic

First, let's look at the basic deployment strategy, also known as the *Big Bang* deployment. This was once one of the most popular methods of releasing applications.

In a nutshell, there is only one version of the app in production. The process involves turning off the app, performing the update, turning it back on, and then directing all the traffic to the new version.

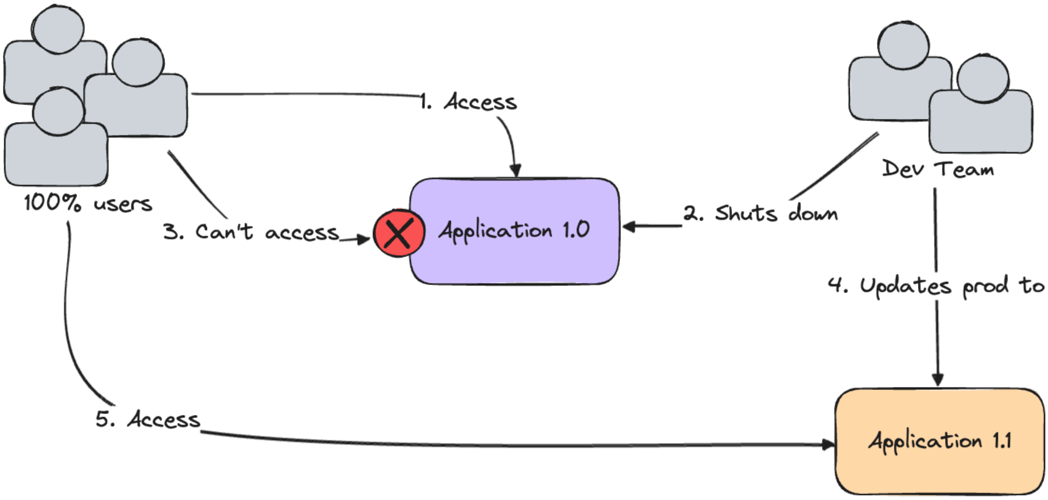


Figure 119. Overview of basic deployment

You have to plan it during off-hours to affect a minimum number of users (preferably none), as it requires shutting down the entire application. When you shut it down, it is unavailable to any user.

The next step is to proceed with the update. When everything goes smoothly and there are no errors, the application is ready to serve users again. As the previous version no longer exists, 100% of traffic will be forwarded to this new version.

Of course, there is a chance that the new version won't work as expected. You might encounter 4xx and 5xx errors, issues with the application's UI, or other problems. In such cases, you have two options:

- Rollback and deploy the old version.
- Fix the issue and deploy a new version that includes this fix.

Imagine that you planned a one-hour window for this switch. As you start fixing the issue, the time passes, and you find yourself under pressure. Typically, working under pressure leads to mistakes. The safer choice would be a rollback, but what if it is just a simple configuration change? Time is ticking, and you have to make a decision. Finally, you fix the issue, but it costs you a lot of stress.

The problem is that your application was unavailable for several hours. This can have unpleasant consequences, even if users have not noticed it. If you have signed a service level agreement with your customers and guarantee 99.5% availability throughout the year, it may be unavailable for a maximum of 1d 19h 28m 8.8s. Let's say the application was unavailable for 4 hours—you have used around 10% of the total allowable downtime.

> ⚠ That is why it is so important to choose the right strategy for the given requirements rather than to promise a level of service you cannot meet.

This approach has the advantages of a straightforward deployment process (if everything works out) and no need to support several application versions simultaneously. However, it requires comprehensive testing, backups, and a clear rollback plan if problems arise post-release.

It will work well in applications that are not critical and can afford not to work for some time. Leaving aside the old days, I most often encountered this approach in Intranet applications and those where the user base was small (up to a few thousand) or covered a single time zone (or several close to each other), so you have time to update it during non-working hours.

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.

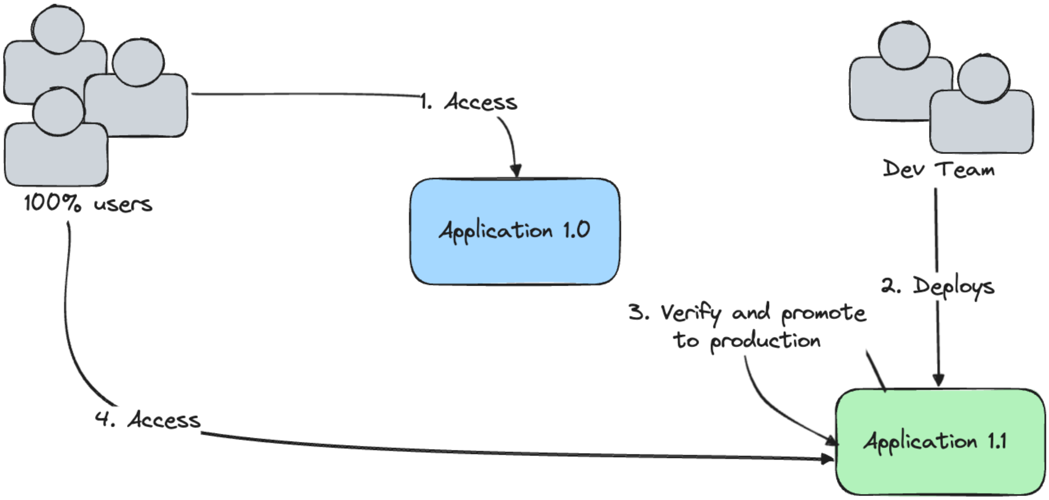


Figure 120. Overview of blue-green deployment

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.

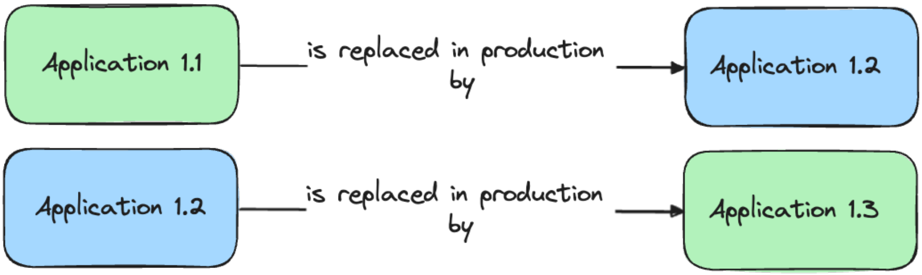


Figure 121. Blue and green environments replace each other in production

> ℹ 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 is my favorite approach to releasing apps but also the most difficult to set up. As a result, it is suitable only for certain applications, and I recommend it only when I am confident that my business can derive significant benefits from it.

In a nutshell, a current version of the application runs in production. At the same time, we prepare another instance with the new version and execute verification checks. If everything works fine, we switch a small part of production traffic to it. Then, gradually, we push more and more users to it until we reach 100%.

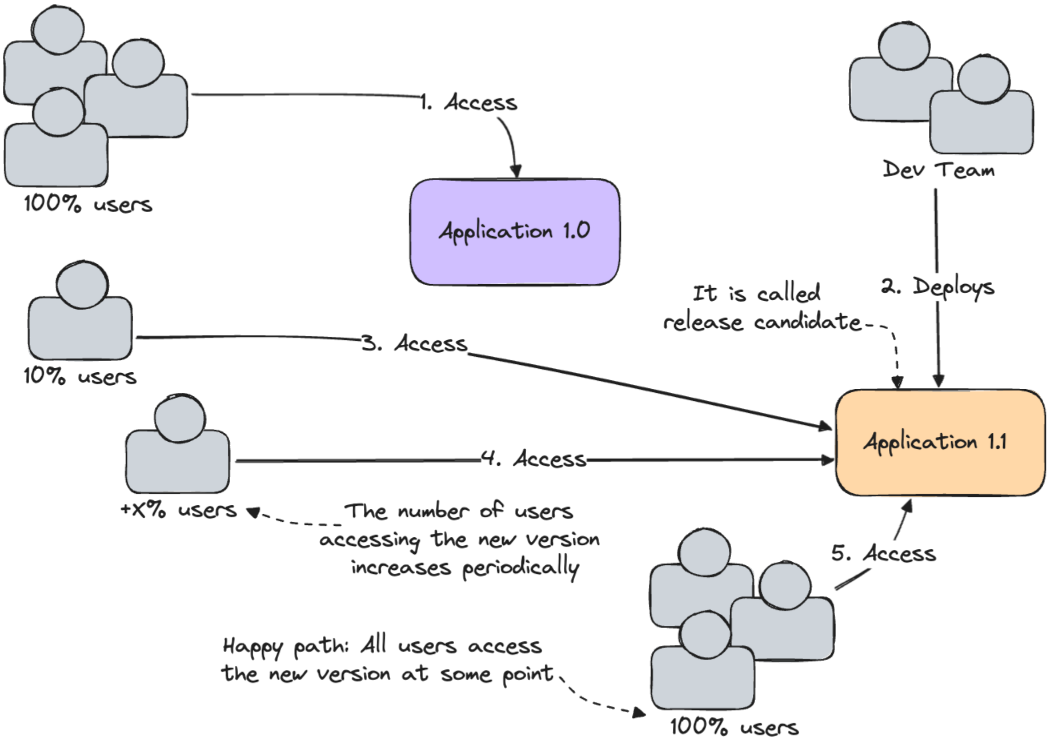


Figure 122. Overview of canary deployment

All users access the current application version. Next, the development team triggers the deployment of the new version. It can be triggered automatically on commit push or manually.

When verification checks are successful, this version is promoted to production next to the current one. Now, 100% of traffic goes to version 1.0 (diagram above). Next, a small amount of traffic is redirected to a canary release. It can be 5, 10, 15%, or any number you decide as a team that fits your needs. Then, you increase the percentage (manually or automatically). In the meantime, you will have to check the application state. There are plenty of observations you can make, including the following:

- **4xx and 5xx errors**. Observe if there are any errors. It might be acceptable if some errors of this type occur due to user behavior, such as attempting to access unauthorized content or non-existent features. However, if you notice a consistent pattern where every attempt to access a feature results in 4xx, or 5xx errors, this probably indicates an issue with your application.
- **Changes in frequency of feature use**. The current version of the product catalog is accessed 1,000 times per hour. This traffic is generated by 50% of users. On your canary release, it is only accessed several times during the same time (by another 50% of users). It might be an indicator that something is wrong with the canary.
- **Latency and response time**. Again, here you can compare both versions that run in production. If one differs significantly from the other in response time in the same areas, then something is wrong with the new version.
- **Customer feedback**. Collect and analyze feedback from users exposed to the canary release, directly or indirectly, to identify UI/UX issues or unexpected behavior.

If everything works fine, gradually increase the traffic to 100%. Then, wait for the next canary release. If significant issues arise, abandon the release and redirect all traffic back to the stable version.

While there is a possibility that some issues may not be detected during the gradual rollout, it is important to note that this is rare. In such cases, I recommend moving forward and releasing a new canary candidate with the necessary fixes as soon as possible instead of doing a rollback.

Are feature flags compatible with canary releases? Absolutely. Feature flags can be integrated into any deployment strategy because their functionality is independent of how you choose to deploy your code.

The key advantage of canary deployments is their incremental approach. By gradually introducing a new version to a small subset of users, many potential issues can be identified and mitigated before a full-scale release. It also allows for early feedback from real users. However, it requires an infrastructure that supports traffic routing, monitoring, and version management, which can be technically challenging.

Rolling

In rolling deployment, you gradually replace existing instances with new versions. Imagine that your application has multiple instances, all running on version 1.0. To deploy a new version, you update one instance first. Once verified as successful, you proceed to update the next instance, continuing until all instances are updated.

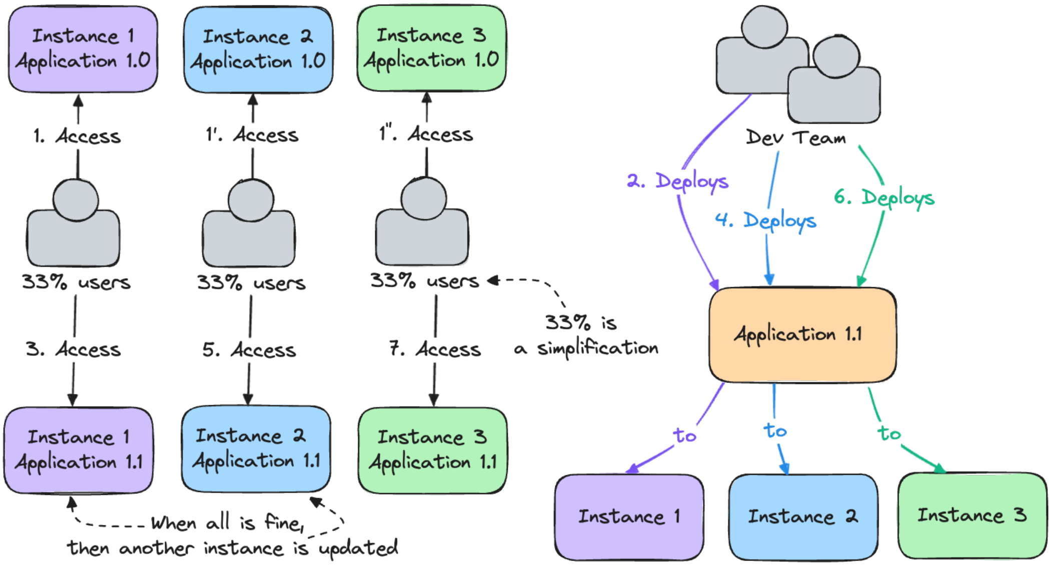

Instance 1 Application 1.0
Instance 2 Application 1.0
Instance 3 Application 1.0
1. Access
1'. Access
1''. Access
33% users
33% users
33% users
3. Access
5. Access
7. Access
33% is a simplification
Instance 1 Application 1.1
Instance 2 Application 1.1
Instance 3 Application 1.1
When all is fine, then another instance is updated
Dev Team
2. Deploys
4. Deploys
6. Deploys
Application 1.1
to
to
to
Instance 1
Instance 2
Instance 3

Figure 123. Overview of rolling deployment

All users use version 1.0 of the application. The traffic is split equally between three instances (e.g., using a load balancer). For the sake of simplicity, let's assume 33% of users’ traffic per instance.

The development team decides to start rolling out version 1.1 of the application. They don’t want to affect all users, so they replace one of the instances with the new version. Now, 66% of the traffic goes to the old version (1.0) and 33% to the new version (1.1).

If everything is fine, another instance is updated with version 1.1. The traffic is split into 33% to the old version (1.0) and 66% to the new version (1.1). Finally, the last instance is updated to the new version (1.1), and 100% of the traffic runs on the newest version.

Like canary, this strategy allows you to validate the newest application version only with a subset of users. If everything goes well, you can continue rolling it out. However, when you spot an issue, you still have an instance(s) running the old version.

You need multiple instances of the application to make this strategy effective. Furthermore, you do not have fine-grained control of the traffic, as in canary release, where you can start with 5% of users, for example, from one location, and then gradually increase the traffic.

Compatibility with previous versions

One of the questions I am asked every time I discuss blue-green, rolling, and canary deployments is how to ensure that the database is compatible with multiple versions.

It is an excellent and multidimensional question.

First of all, no matter which way you go, if multiple versions run simultaneously, you must have a plan for the entire change. I usually plan it this way:

1. Define the initial modification.
2. Plan the subsequent steps one by one.
3. Each step should contain a rollback plan.

I have prepared three scenarios to help you understand how to approach database migration while ensuring it is backward compatible.

Scenario 1

There is a table called *Drugs*. It contains *Name*, *Description*, *Type*, *Manufacturer*, and a few other columns. There is a new requirement. The *Manufacturer* column should no longer be a part of this table. You prepare the script to drop it:

```
ALTER TABLE Drugs DROP COLUMN Manufacturer;
```

You deploy the new version and forward 5% of the traffic to it. At the same time, you start seeing errors in the previous version. That is because you dropped the column. The old code still wants to use it, but the column does not exist.

Migration plan:

1. **Version 1.1**. Make column *Manufacturer* optional. Stop writing to it, but continue reading.
2. **Version 1.2**. Remove the code responsible for reading from the *Manufacturer* column. Make sure all references are removed as well.
3. **Version 1.3**. Remove the *Manufacturer* column. Before doing this, ensure you have a database backup, as this operation is irreversible.

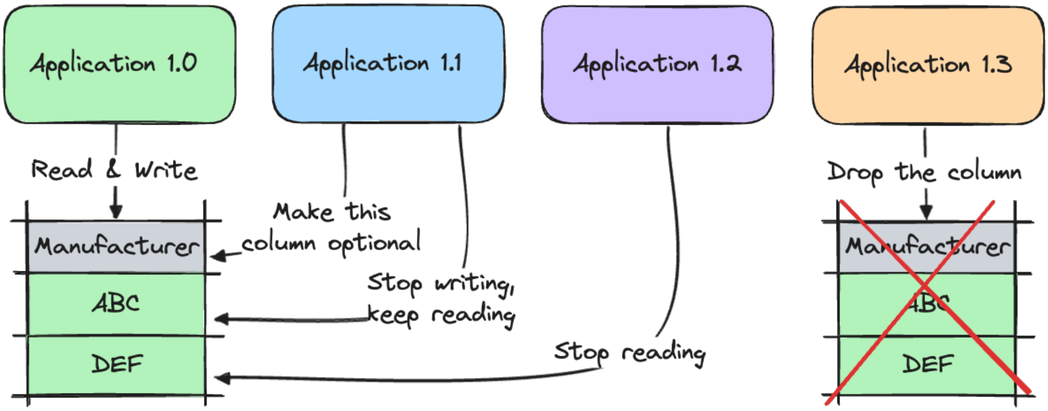


Figure 124. Scenario 1: Visualization of the incremental changes

Scenario 2

A table named *Customers* stores information like *Name*, *Email*, *Phone*, *Address*, and a few other columns. There is a new requirement—you need to move the *Address* column to another table. This column is required.

Migration plan:

1. **Version 1.1.** Add a new optional *Address* column to another table. Write to and read from both tables.
2. **Version 1.2.** Make the old *Address* column optional and stop writing to it. Still read from both tables.
3. **Version 1.3.** Migrate all entries from the old *Address* column to the new one. Make the new *Address* column required and stop reading from the old one.
4. **Version 1.4.** Drop the old *Address* column.

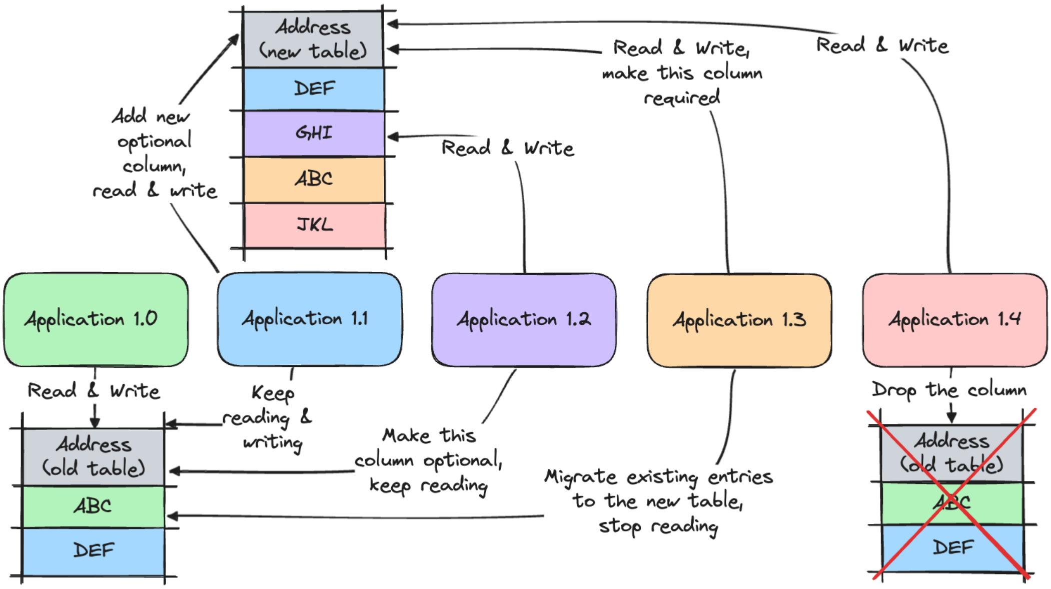


Figure 125. Scenario 2: Visualization of the incremental changes

Scenario 3

A table named *Patients* stores information like *FirstName*, *LastName*, *Phone*, *DateOfBirth*, *AssignedCode* (required), and a few other columns. There is a new requirement—you need to change the type of *AssignedCode* column from numeric to text because codes can also contain text.

Migration plan:

1. **Version 1.1**. Make the *AssignedCode* column optional. Add a new optional column *AssignedTextCode* to the table next to *AssignedCode*. Write only to the new column and read from both columns.
2. **Version 1.2**. Migrate all existing entries from the *AssignedCode* column to the *AssignedTextCode*. Make the new *AssignedTextCode* column required and stop reading from the *AssignedCode* column.
3. **Version 1.3**. Drop the old *AssignedCode* column.

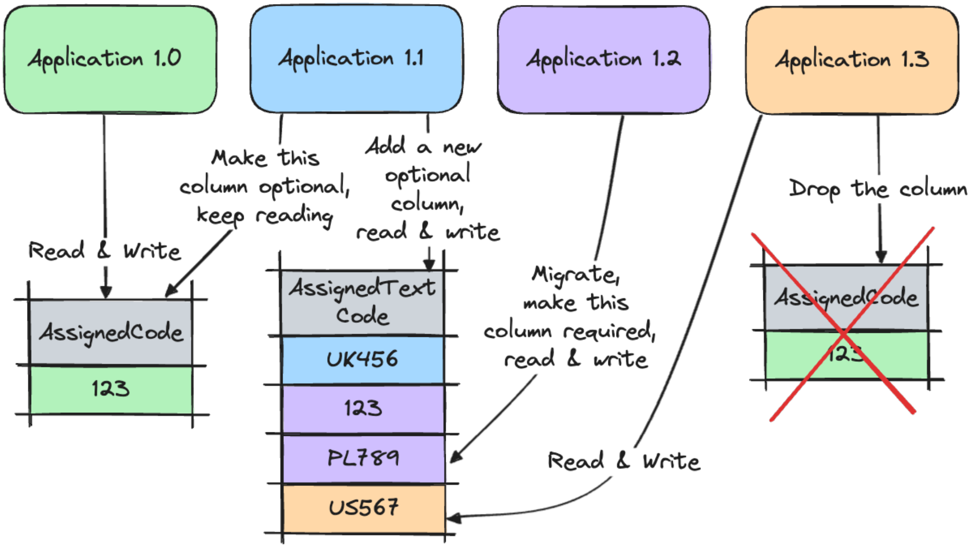


Figure 126. Scenario 3: Visualization of the incremental changes

If you look on the Internet, you will find that this method has a name: *parallel change*. It allows you to run multiple versions of your application simultaneously. I showed you how it looks from the database perspective, but if you are interested in how it works with code, you can read about it here[1].

Please note that there are various ways to plan migrations; I have shown just one possible approach. The key is to ensure that your changes are backward compatible.

Fully automated releases - Continuous deployment

Continuous deployment is considered the pinnacle of deployment practices. Complete automation, no human intervention after pushing the commit. If we look at the book definition, every change we make, every commit, triggers the deployment of a new version to production.

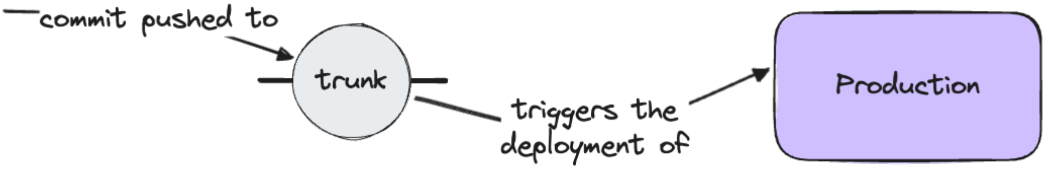


Figure 127. Each code change is deployed to production

We set up checks to automatically verify deployment, and if it is successful, our customers use the new version.

Is it something so amazing? I believe it is. Is it worth the sacrifice of everything and bearing any costs? I don't think so.

Before continuing, you must be aware that I am a big fan of this approach. I was infected with it years ago, and it has stayed that way ever since. I have battle-tested it multiple times in small and large-scale organizations. If you had asked me 3-4

[1] https://martinfowler.com/bliki/ParallelChange.html

years ago in which direction to go, I would have always told you to do continuous deployment.

Fortunately, this has changed. I no longer consider it zero-one. Before using this approach, I try to understand the environment around me. I do not pursue it at all costs if it does not make sense. Nevertheless, continuous deployment is one of the best things that has happened to me in all my years of implementing IT systems.

The change

Continuous deployment changes the way you work. This change will affect not only technical aspects—such as infrastructure and observability—but also the approach to product development, how you interact with your customers, collect feedback, plan new features, and more.

To illustrate this, let’s consider a hypothetical scenario:

1. You work for a company that provides software designed to assist customers in selling goods to their own customers.
2. For many years, this software was delivered twice per year. It was accompanied by the announcement of a planned release to all customers.
3. Each release contained changes from the last five months.
4. The testing phase with bug fixing was always planned and usually took around one month to complete. Customers were involved in the beta testing phase.
5. Finally, there was a big release day. All hands on deck, the next two weeks were marked by constant waiting to see if everything was okay.
6. After each release, you collected customer feedback and planned the development for the next few months.

You hired a new employee who came from your direct competitor. This competitor is doing great in the market—customers praise their software and are happy that the feedback they give is implemented within days. On top of that, the software is reliable. Upon inquiry, you discover they employ continuous deployment that is executed in the following way:

1. Software is released multiple times per day. Last year, it had more than 1,000 releases.

2. There are no pull requests. Developers work together on code (pair and mob-programming) and push it directly to the trunk.
3. All testing and other verification checks are automated and done in a build and deployment pipeline.
4. Non-critical bugs are acceptable artifacts. If an issue arises, a fix is deployed to production within minutes or hours. Furthermore, thanks to good observability, they can spot bugs before customers report them.
5. Based on continuous feedback from customers and observing their behaviors in the application, they can steer the product in the right direction while avoiding potentially misguided investments.

You are amazed by that process but start to get stressed. You make an instant decision to switch to continuous deployment, which turns everything upside down. Customers are unhappy—the software version was updated without their approval and might heavily affect their operations. Some people, such as manual testers and release managers, are no longer needed, and firing people is never easy.

Developers have to adapt to the fact that besides them and automatic checks, there are no dedicated teams of testers, and their changes are pushed directly into production. In addition, they need to learn how to work in groups, pair and mob programming, swarming, how to plan code changes, and more.

It is a long process. Changing the way we think takes time and requires a step-by-step approach. It is worth starting with changes in a development team. We can then extend these changes throughout the organization and our interactions with customers. Our customers must trust that we can deliver reliable software, listen to their constant feedback, and react quickly to potential bugs. It will only be possible if we first fix the processes within our company.

Development experience

With the teams I have collaborated with, our usual goal was to achieve what we termed “happiness.” We adopted practices that worked well for us and discarded those that did not. Of course, it varied between the teams, but some elements began to repeat. You will find them in this section.

Swarming

The following situation may seem familiar to you in terms of the approach to implementing tasks:

1. There are several tasks to be implemented.
2. Each developer picks a task.
3. They work on their tasks independently (in isolation).
4. There is more and more unpredicted work, causing the developers to feel overwhelmed.
5. They face problems alone. Maybe they ask for help, but it is a side task for someone else, often without full focus.
6. When the task is ready, it is sent for review (large amount of code).
7. There are several ping-pong rounds to address issues.
8. Finally, the task is approved and merged into the main.

As a result, there are a lot of open tasks, but almost none are finished. Everyone focuses on something completely different, and the level of cooperation is relatively low. Additionally, code reviews on large amounts of code by people who did not work on it might be a subject of *LGTM (Looks Good To Me)*, which unfortunately means a superficial review. This leads to knowledge silos, where only one person knows what is going on in a particular code area. What happens if that person leaves the company or is unavailable?

> I must admit something—when I have a PR in front of me containing 85 changes in 78 files, I intentionally pretend not to see it. It is much work to do the code review. At times, I will write LGTM without a deeper glance at it. I will certainly not gain an in-depth understanding of the subject. I feel overwhelmed.

The solution is to involve multiple people to analyze the feature, prepare the plan, work on it, and finally deliver it. This is defined as swarming. Let’s look at its meaning:

Swarming

involves the entire team focusing, analyzing, and planning around a single feature. In the end, the feature is broken down into smaller tasks that team members can work on simultaneously.

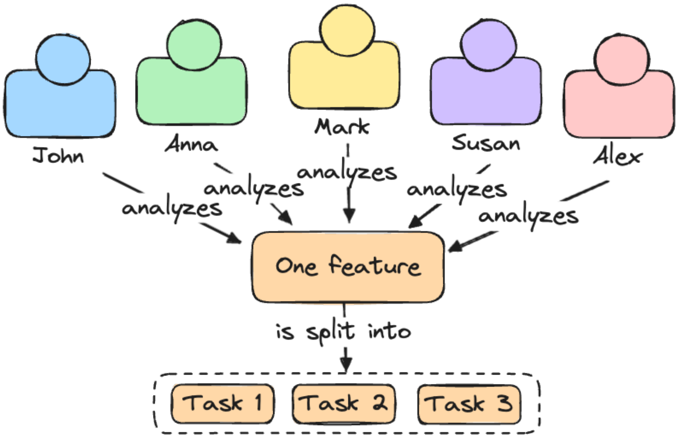


Figure 128. Example of swarming

Should the entire team be involved in just one feature, or do you need to organize a feature team? From my observations, involving at least three people tends to work well. With only two individuals, biases can arise, especially if one possesses exceptional skills or charisma. Ultimately, whether it is two, four, or the entire team, make the decision that feels most comfortable to you.

What might such cooperation look like?

Two team members focus on the frontend while the other two focus on the backend. They can switch roles seamlessly, especially on a team where everyone is versatile. This setup ensures that all four people are fully engaged in delivering the single feature, and that knowledge is shared among all of them, preventing silos.

Pair & Mob programming

This is my favorite way of collaboration. If you need to share the knowledge, do it while you gather it. What does this mean?

You should not wait to share the code and explanation until the creation of a pull request; you can implement it together with one or more colleagues. This way, you split the knowledge among at least two people without much effort—it will be a natural consequence of coding together.

Pair programming
: is where two people work on the same feature, sharing the same machine.

Mob programming
: is where more than two people work on the same feature, sharing the same machine.

If you look closely, both ways involve programming in a group (two or more people), focusing on the same task, and using a single machine. One person writes the code while another reviews it in real time, and they switch roles from time to time.

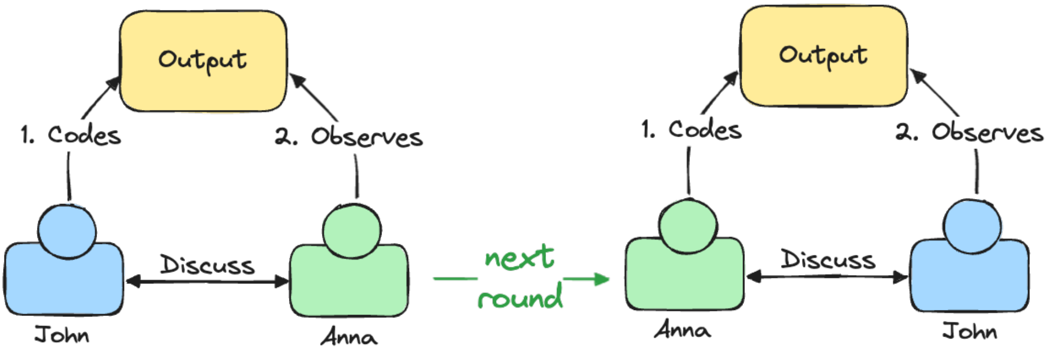


Figure 129. Example of pair programming

Nowadays, we are fortunate to have tools that enable us to collaborate remotely. This eliminates the need to share a single machine. After all, who enjoys working on someone else's machine?

It is an excellent idea to rotate pairs frequently within a team to not to create pair-silos. You can also extend the concept beyond just programmers. For example, pair a developer with a designer or product manager—sometimes the results are incredible.

ℹ Pair or mob programming can completely eliminate the need for pull requests. Code review takes place during implementation, and any commit can be pushed directly into production.

Trunk-based development

Trunk-based development is a software engineering practice where developers frequently integrate their code changes into the main branch, known as the trunk. While the core idea is to work directly on the trunk, some variations allow for short-living branches.

In its purest form:

- Developers commit directly to the trunk, without side branches.
- Code reviews happen in real-time through pair or mob programming sessions.
- Commits are small and frequent, often multiple times per day.
- Each commit triggers a continuous integration (CI) pipeline with automatic checks.
- Successful builds are typically deployed to production immediately.

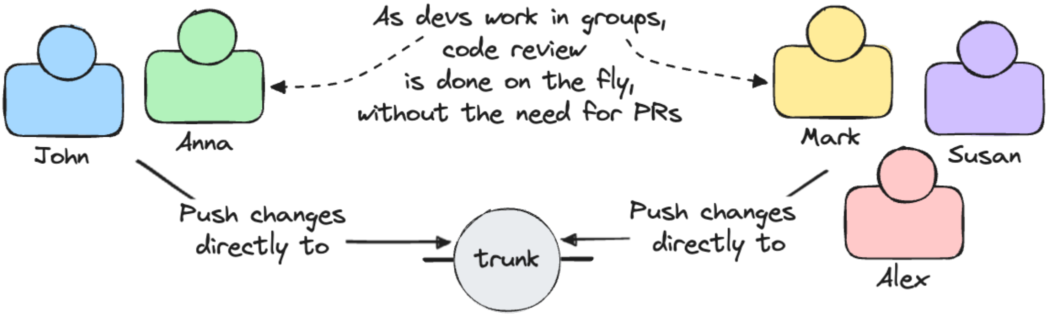


Figure 130. Integrating changes directly to trunk

Some teams use a modified approach with short-living branches. Such branches last no more than a day or two before merging back to trunk. This allows for quick code reviews via pull requests while maintaining frequent integration.

To manage incomplete features, teams often use feature flags, allowing new code to be deployed but not activated until ready.

Trunk-based development speeds up building and releasing software. It also helps improve quality by getting everyone to work together more closely. But it is not always easy to start doing. Teams often need to change how they think about their work and be more disciplined in their coding habits. This means they might need some time to learn and get used to this new way of working.

Example flow

Taking into consideration all that you have learned in this section, the flow for continuous deployment can look as follows:

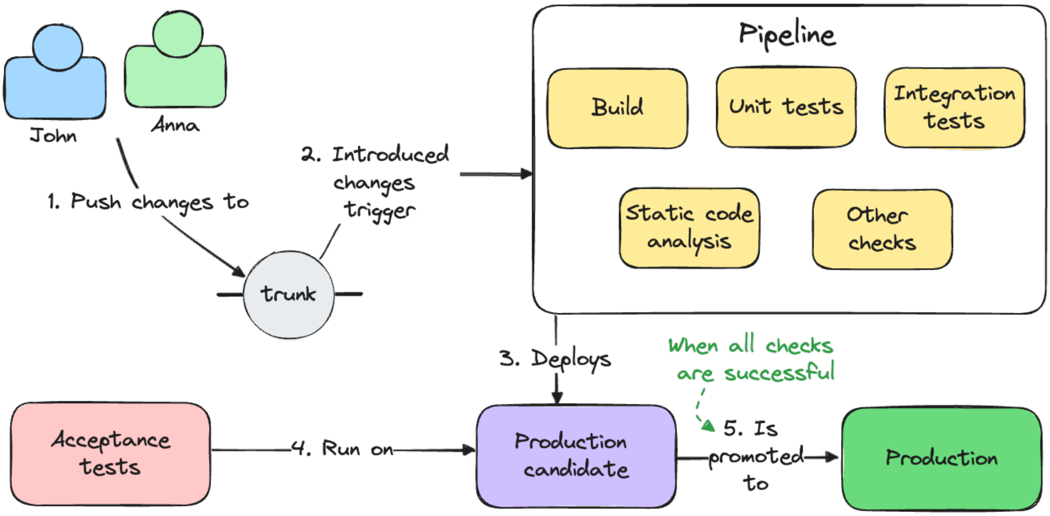


Figure 131. Example of continuous deployment flow

1. Two teammates, John and Anna, work together on a task. They do pair programming. They review their code as they write it.
2. When the change is ready, one commits the code and pushes it to the main.
3. The continuous integration pipeline is automatically triggered. When the build is successful, verification checks are run. These checks can be unit tests, integration tests, or static code analysis for application code and infrastructure (infrastructure-as-code).
4. The production candidate is deployed. At that moment, no traffic has been forwarded there yet.
5. Some acceptance tests are run, such as those that check critical business processes.

6. When tests are green, the candidate is promoted to production.

The entire process repeats each time there is a new change. Please note that this is just one approach to the continuous deployment process.

Semi-automated releases - Continuous delivery

Sometimes, it is not possible to fully automate the deployment process because of various reasons, including:

- **Compliance requirements**. There might be some compliance standards that need to be manually checked before each deployment, or every version of the software requires an external audit due to regulations.
- **Risk management**. Businesses might not accept going live multiple times per day because of the high risk of failure, and they prefer to do it once a day during non-work hours. Every company has a different risk threshold, and you have to accept it.
- **Organizational culture**. Transitioning to fully automated deployments requires a shift in thinking and time to get used to it.

> ℹ Sometimes, you may hear that these are just excuses, but life in IT is not black and white. We have to deal with trade-offs, people relations, politics, and many other issues. That is why we should always pragmatically look for solutions to problems in the surrounding environment.

Continuous delivery is quite similar to continuous deployment. The build pipeline with verification checks still runs after pushing to the main, but the production environment is not deployed automatically. Someone has to do it manually, like clicking a button, or schedule a job to do it, say, once a day.

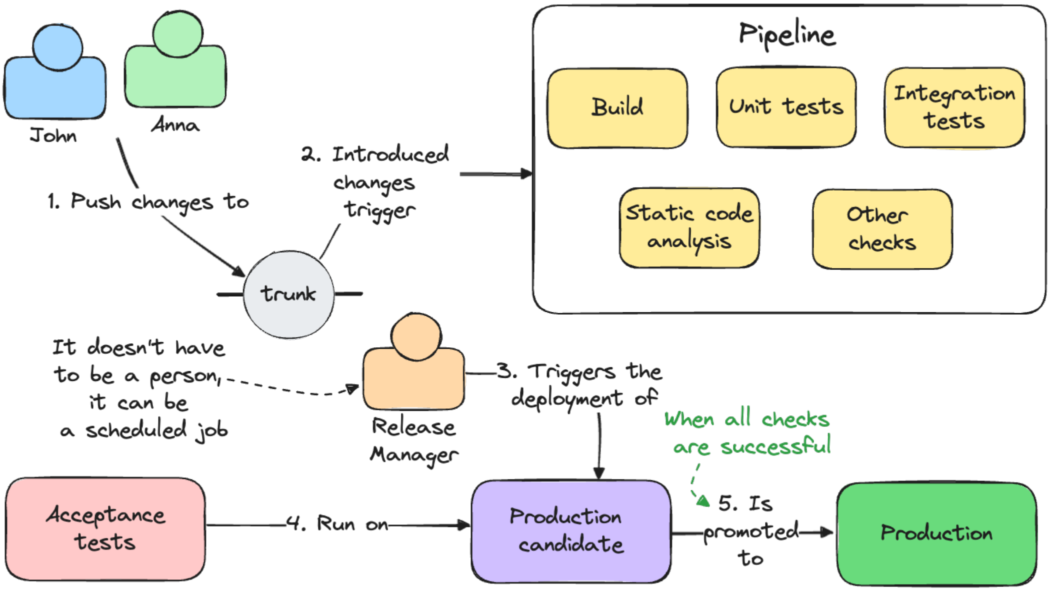


Figure 132. Example of continuous delivery flow

A core rule of this method is always to be ready to deploy the code to production, so it requires keeping the build continuously green. If you see a red build, you have to fix it.

An excellent idea is to deploy the code to a staging environment continuously. In this case, it will work exactly as continuous deployments with one difference—the code will be deployed to staging instead of production. This way, we can, for example, run acceptance tests on a real environment and get used to the deployments and rollback process.

Pragmatic approach

Some concepts sound great in theory, but implementing them takes more work because of our environment, existing processes and other factors. That is why I suggest a pragmatic approach to the release strategy.

It involves combining the benefits of various methods and adapting them to your needs. For example, when you work in a legacy environment, where each release takes one month from the last commit, do not try to change everything immediately. Follow the step-by-step approach. Try to reduce the time to three weeks, then two, then one, and it may already be enough in your case. Not every application has to be deployed multiple times per day. Doing it just because someone else does it makes no sense when it brings no value to your business.

You may like the idea of continuous deployments, but you can't get past the lack of pull requests. It is fine. Automate the entire deployment but use short-living branches.

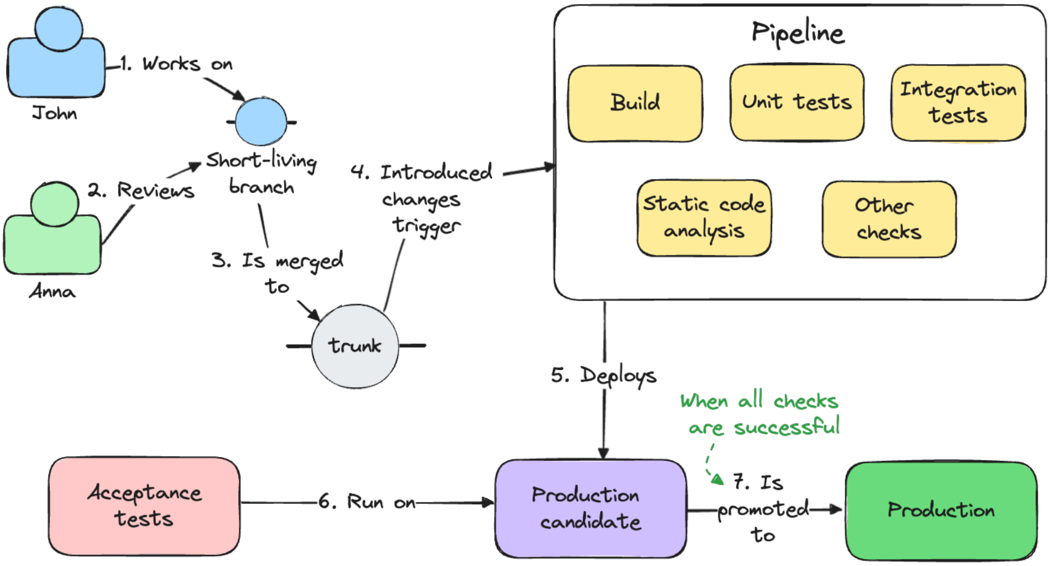


Figure 133. Example of deployment flow using short-living branches

If you feel that deploying code once a day or once every three days would be more suitable, implement it using continuous delivery or a similar approach.

Always look for ways to improve the development process. Observe what works and what doesn't. Continuously adjust and validate ideas. Be open to experimenting with new tools and techniques.

In this section, I describe several engineering practices that can help you improve the release process.

Short-living branches

Typically, in software projects, you will encounter two extremes:

- **Long-living branches**. Someone creates a branch, works on it for a week or more, then creates a pull request, and the code is reviewed. Usually, there are dozens or hundreds of files changed, and there is much back-and-forth in reviews and reworks.
- **Direct commits to the trunk**. Everyone commits directly to the trunk (usually main). There are no pull requests, developers work together on the task (swarming, pair, and mob programming), and there are usually multiple integrations into trunk per day.

Let's explore a middle-ground approach called short-living branches. This approach can serve as a stepping stone towards direct trunk commits, particularly for teams still developing their skills or working in contexts where pure trunk-based development isn't ideal (such as highly regulated environments, experimental feature development, or open-source projects).

Short-living branches, as the name suggests, live for a short time after the creation. What does a short amount of time mean? This is something your team should decide together. It might mean a few hours in some cases, or perhaps 1-2 days in others. If it tends towards the latter, aim to occasionally shorten this timeframe.

You should ensure the branch is merged and closed during this time. This approach can significantly increase the speed of change, leading to higher levels of motivation among team members.

> *Yeah, my code is merged!*
>
> *Great, there are a small number of changes, and I will try to grill it!*

Here are several recommendations:

- **Analyze and split**. Divide your work on a task into smaller pieces. Each subtask can be treated as a separate, short-living branch. This way, you can efficiently parallelize the work with your colleagues, with each person taking one subtask and creating their own branch. Spoiler: swarming helps here.

- **New work discovered = new branch created**. Whenever you find a new subtask to do after the initial split, treat it as a new, short-living branch (add a subtask and start your work).
- **The shorter, the better**. Go with at most 1-2 days for each branch. Even if you have to merge changes to or rebase your branch at the end of work, it will be less than it would have been after 1-2 weeks. I try to keep it alive for a maximum of 4-6 hours.
- **The less, the better**. Aim to keep your pull requests as small as possible. This approach encourages reviewers to examine changes thoroughly. From my experience, having fewer files in a PR often results in quicker feedback. Sometimes, it is ready within 10 minutes of the review request, and you can merge it.
- **One area of change, one branch**. For instance, while working on a short-living branch to add an avatar next to a username, you might find it appropriate to move 20 files to a different namespace and another 15 files to another (unrelated to the current change). In such cases, I recommend creating a separate short-living branch just for these refactoring commits.

How can you divide the work between multiple short-living branches? Imagine that you have to prepare the download of the file. You split it into various subtasks:

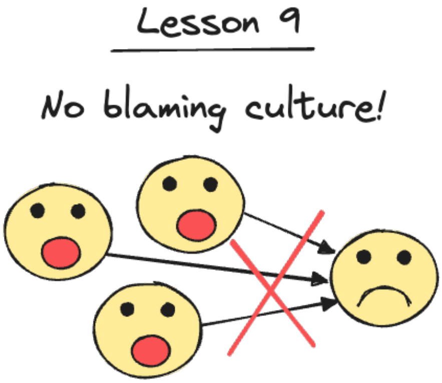
1. Prepare download icon design (approx. 1-2 hours).
2. Integrate the download icon into the file row without any action connected (approx. 2-3 hours).
3. Handle business rule number 1 required to generate a download link (approx. 2-3 hours).
4. Handle business rule number 2 required to generate a download link (approx. 2-3 hours).
5. Prepare technical implementation to generate download link (approx. 4-6 hours).
6. Connect technical implementation with business rules (approx. 2-3 hours).
7. Implement endpoints (approx. 2-3 hours).
8. Trigger download action on clicking the download icon (approx. 1-2 hours).

Now, create a separate short-living branch when you start each of the above subtasks. When you see a new subtask (it is impossible to plan everything during the initial analysis), create a separate short-living branch for it. That’s it!

All hands on deck

When there is a severe issue, everyone in the team should come together to help. It does not matter if you are a tester, DevOps, frontend developer, or product owner – anyone who can help solve the problem should join in.

The main rule is to focus on solving the problem rather than on who caused it. This way, you introduce a blameless culture without finger-pointing. You are there as a team, not as individual developers.



Document everything you find during the investigation and the steps to solve the issue. This way, you will be well-prepared with a detailed report for the post-mortem meeting.

Post mortem

After we fix a problem and make customers happy again, we often forget to have a meeting to talk about what went wrong. We just move on until the next problem happens, and then we do the same thing again.

After resolving an incident, it makes sense to gather again—this time without stress—and focus on preventing similar situations in the future. There should be no blaming, but everyone should be honest with their opinions.

How to organize post-mortem meetings:

1. Gather everyone who was involved in solving the incident.
2. Reconstruct the timeline of events that led to it.
3. Identify what steps were taken to resolve it.
4. Try to find the root cause of the incident. You can use techniques like *Fault Tree Analysis (FTA)* or *5 Whys*.
5. Evaluate the incident's impact on your business, such as how much money you lost or how many users were affected.
6. Prepare actionable steps to help you to avoid such situations in the future.
7. Finally, assign people who will be responsible for the execution of the actionable steps.

Sometimes, it might not be your fault as a development team. Occasionally, a critical issue may arise from an external provider.

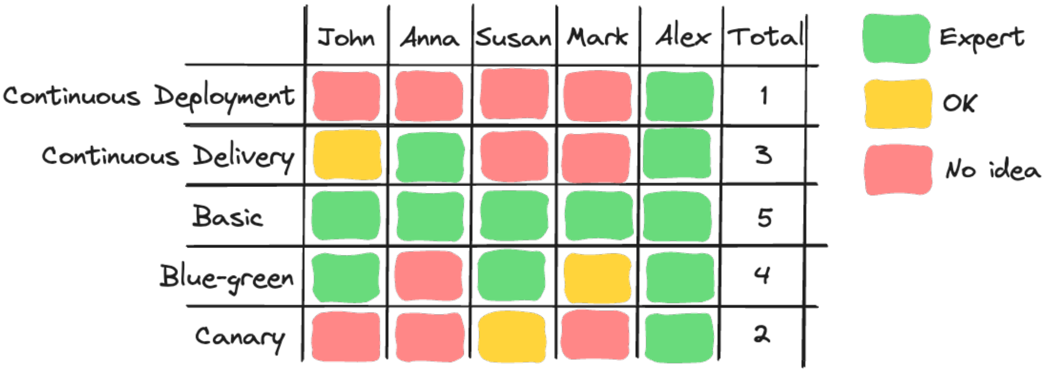
> Imagine running your application in the cloud. You deploy it in two regions to ensure that if one fails, traffic is directed to the other.
>
> However, there is a global issue, and your application does not work because both regions failed.
>
> If it is a big problem for your operations because every minute of a non-working application leads to severe financial losses, a solution may be to use two providers at once to reduce this risk in the future. The trade-off will be the operational costs of additional infrastructure and the need to build new competencies.

Finally, remember that the goal of a post-mortem meeting is continuous improvement. It is about building a culture of learning and resilience. Each incident is an opportunity to improve your processes and strengthen your service's reliability.

Our case

Looking at our case, we can see that there are several factors that we have to take into consideration when deciding on the release strategy.

First, we gathered together with all team members and conducted a quick session to assess our skills related to this area.



	John	Anna	Susan	Mark	Alex	Total
Continuous Deployment	🟥	🟥	🟥	🟥	🟩	1
Continuous Delivery	🟨	🟩	🟥	🟥	🟩	3
Basic	🟩	🟩	🟩	🟩	🟩	5
Blue-green	🟩	🟥	🟩	🟨	🟩	4
Canary	🟥	🟥	🟨	🟥	🟩	2

🟩 Expert
🟨 OK
🟥 No idea

Figure 134. Team skill assessment

Next, we looked at key points:

1. SLA is 99.5% availability. Our application can be offline for up to 1 day, 19 hours, 28 minutes, and 8.8 seconds annually. This is not bad; it gives us some flexibility while selecting a release strategy.
2. The expected number of patients is ~1,500, and the expected number of employees is ~100, so the total number of users is ~1,600.
3. Customers are in Europe, so working hours are almost the same across time zones.
4. The dev team is inexperienced in continuous deployments and canary releases.
5. There is no existing infrastructure.
6. We have a budget limitation of a maximum of $500,000.
7. The deadline is set for two months from now.

Finally, we focused on the following architectural drivers:

Consideration	Architectural driver
SLA	Quality Attribute (Availability)
Budget limitation	Business Constraint
Deadline	Business Constraint
Low experience in continuous deployments and canary releases	Technical Constraint
No existing infrastructure	Technical Constraint

And selected key ones:

- **Availability**. The agreed SLA is at 99.5% level. Our application can be unavailable for up to 1 day, 19 hours, 28 minutes, and 8.8 seconds during the year.
- **Cost-Efficiency**. Due to our budget limitations, we should not focus on a gold-plated solution that will do everything automatically.
- **Simplicity**. There is no existing infrastructure. This means that we should focus on delivering a solution that is as simple as possible. It still has to meet all the requirements.

We must also consider that we have already decided to use a modular monolith. As the number of users will be small initially and the expected availability level is not that high, we decided not to leverage the power of multiple instances. We want to start with a single instance and observe how it behaves. Therefore, we can remove rolling deployments from our list. Canary seems too complex and our team lacks experience in it. The same goes for continuous deployments.

As the application is not critical and will serve only European customers for around 12 hours a day, there is a window for updates without impacting users. After consulting with the business, we decided to first go with the basic deployment (everyone has experience with it) and then switch to blue-green after the MVP is released.

We want to be ready to deploy anytime, and our default choice is to deploy the new version to production once a day during non-business hours. Unfortunately, this approach requires selected people to work outside of regular business hours. That’s why it is so important to change this process as soon as possible.

Recap

In this step, you learned about various release strategies in software development. These strategies include continuous deployment and delivery, which ensure that your code changes are automatically released to production after passing all necessary verification checks.

We also covered canary releases, where new features are gradually rolled out to a small subset of users before a full deployment, helping to catch any potential issues early. Additionally, we discussed blue-green deployments, which involve running two identical production environments to reduce downtime during releases, and rolling deployments, which incrementally update instances to minimize user impact.

To support these release strategies, I introduced several engineering practices. Pair and mob programming fosters teamwork and knowledge sharing by having multiple developers work together on the same task. Swarming involves the whole team working together to analyze what needs to be done. Short-living branches keep changes small and manageable, and allow frequent integration into the main codebase, especially when direct commits to the trunk are not possible.

Finally, I described the importance of post-mortem which is critical for identifying what went well and what didn't during a release, enabling continuous improvement.

It is time to focus on testing.

STEP 6: Focus On Testing

Do any of these sound familiar?

> *Tests? We don't need them.*
>
> *We will add them later.*
>
> *Tests will be merged in another branch.*
>
> *The project manager will handle testing.*
>
> *We cannot afford to add automated tests; they are too expensive.*

Too often, testing is treated as an extra add-on to our software. The truth is, it is an integral part of software development. Please do not treat it as additional work, do not ask anyone for permission to write tests, and do not plan extra time for it. As a software engineer, you are responsible for the engineering process, and writing tests is one of its components.

The sooner you understand this, the better.

The purpose of writing tests is not to be fancy but to ensure that our software works as expected—both technically and according to business requirements. Tests give us peace of mind and help us sleep better at night.

It is worth noting that testing, even with 100% coverage, will not guarantee a bug-free system. There will always be bugs, but thorough testing will significantly reduce them.

In this step, I will focus on areas that I believe are most important from a software architect's perspective. We will explore the pyramid of tests and its variations. Additionally, we will cover how load, performance, and penetration testing can help you and your team keep your application in good shape.

Remember, you do not need to use everything you learn here. Stay pragmatic. Like the entire software architecture, this is a set of tools, not rules. A decision that works for one project might not work for another. Therefore, it is crucial to know when to use each tool.

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.

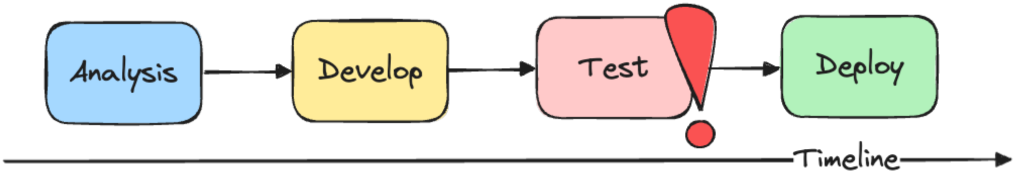


Figure 135. Testing at the end of the development process

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.

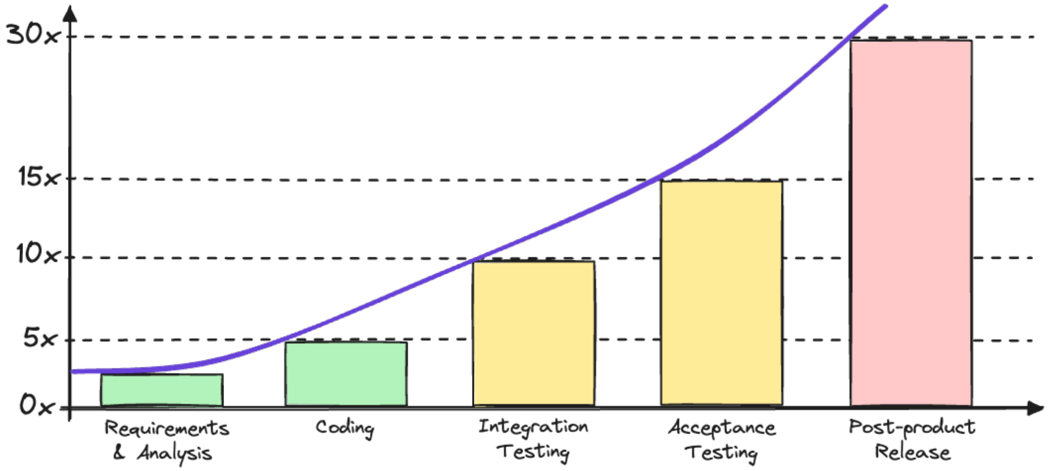


Figure 136. Relative cost of fixing defects at different stages of software development

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

Shift-left testing is a practice that integrates testing early and continuously throughout the development process. It is called "shift-left" because it moves testing activities to the left (beginning) of the process. Testing starts as soon as the first code is written and is performed regularly during development process, rather than as a separate final step. It involves everyone on the product team in the testing process and aims to catch problems earlier.

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

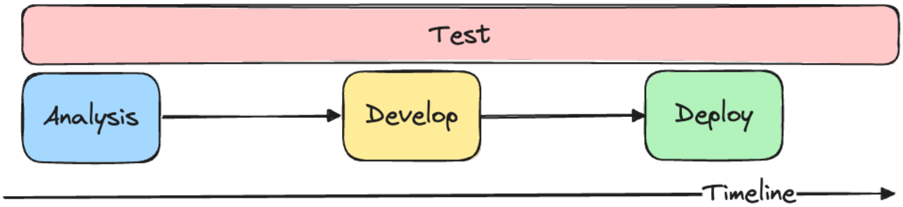


Figure 137. Testing throughout the development process

Testing software throughout the entire development process (shift-left) offers major benefits:

1. **Problems caught early.** Find bugs when they are cheap and easy to fix. This prevents costly last-minute rewrites.
2. **Cost savings.** Fix bugs early when it is cheap. A problem caught during planning costs far less than one found after release. Catching major design flaws early can lead to massive savings. On the other hand, discovering them at the end will cause extremely high costs. As Jose Luis Latorre, one of my book reviewers, says:

> *I was involved in two big projects:*
>
> *Project 1: As the UI team lead on a multi-year project using traditional testing, we discovered a critical issue near launch because the system had not been fully tested with the expected database load. Several design flaws were discovered when the system became unresponsive under full load and failed to meet the throughput requirements specified in the SLAs. This took over a year to mitigate and had a significant impact on staffing and the delay in the go-live date.*
>
> *Project 2: In contrast, as a performance test lead on another project, I implemented a shift-left strategy, where comprehensive end-to-end tests that tested the entire system (the part that had been developed so far) simulated a full day of full-load operation early in development. This approach uncovered two critical database design issues early on, which were fixed promptly—one took 5 days to fix and the other only 8 hours. We even stress-tested the system, compressing a full day's load into a single*

> *hour to ensure there were no surprises at the end of development. This approach worked like a charm and ensured no surprises.*
>
> —Jose Luis Latorre

3. **Less deadline stress**. Distribute testing throughout the development lifecycle. This will eliminate (or at least reduce) last-minute fixes and give you a chance to create more thoughtful, optimized solutions.
4. **Increased trust in code**. Continuous testing throughout development builds confidence. You know your code works as it should, reducing the risk of big surprises at the end.
5. **Better teamwork**. Shift-left gets everyone involved in testing. This improves communication and makes quality a shared goal. It also ensures the right product is being built.
6. **Enhanced code quality**. By testing continuously, you can address architectural issues early, leading to a more robust and maintainable codebase. The earlier that weaknesses in design are identified, the greater the opportunity to implement improvements or alternative solutions.
7. **Faster time to market**. With defects being caught and resolved earlier, development can proceed more smoothly, resulting in faster delivery of a stable product to market.
8. **Less manual testing**. By integrating testing throughout the development process, fewer issues are left for manual testing at the end, making final testing easier.

How can you achieve shift-left testing in your organization? Let me explain using my preferred setup.

I advocate for a cross-functional development team, moving away from siloed roles like frontend, backend, testers, and infrastructure specialists. Instead, I prefer when everyone is involved in all aspects of development (I know, it is not that easy). While team members may have areas of expertise, they are encouraged to contribute across the board. This approach means that regardless of your primary role, you are involved in testing (as well as frontend, backend, etc.).

Yes, even testers write production code in my teams.

This approach allows us to integrate testing seamlessly into the development process rather than treating it as a separate stage. As we implement features, we simultaneously add tests (even before writing production code).

When a commit is pushed, our CI pipeline executes validation checks, including unit tests, integration tests, and static code analysis. Once these checks pass, we deploy the production candidate and run smoke tests on critical processes.

If all looks good, we gradually move user traffic to this new environment using deployment strategies such as canary releases, rolling, or blue-green deployments. This approach allows us to monitor the new release's performance and roll back quickly if problems arise. This monitoring is also part of "testing."

You may ask: where is manual testing in this process? The answer is that traditional manual testing phases are largely eliminated. By shifting testing to the left and integrating it throughout the development process, we catch and fix problems earlier, reducing the need for extensive manual testing at the end. Of course, it may still be there, but limited.

This way, testing is with us at every stage of software development: before and during implementation, while running the CI pipeline, after deployment, and finally, while monitoring the production candidate with limited traffic. As a result, we improve code quality, reduce time to market, enable more frequent and reliable releases, and get early customer feedback.

Shift-left vs. traditional testing

	Shift-left	**Traditional (shift-right)**
Timing	Integrated throughout development	Occurs at the end of development
Issue detection	Early, reducing need for major changes	Late, often leading to extensive rewrites
Costs	Lower, due to early bug fixes	Higher, due to late detection

	Shift-left	**Traditional (shift-right)**
Time pressure	Reduced, allowing for thoughtful solutions	High, especially near project deadlines
Design opportunities	Continuous refinement of design and architecture	Limited by late-stage testing after major decisions
Manual testing	Reduced need through continuous testing	Extensive manual testing at the end
Team collaboration	Cross-functional with integrated responsibilities	Often siloed roles with minimal overlap
Quality confidence	Increased throughout development	Risk of late discovery of critical issues

As you can see, shift-left testing outperforms traditional testing in many areas. By testing early and often, you can catch bugs earlier, reducing costs and increasing quality. This approach also speeds delivery by eliminating the rush to make last-minute fixes that often occurs during final testing. Continuous testing throughout development also significantly reduces the risk of overlooking problems.

The result? Higher-quality software delivered faster and more cost-effectively than with traditional testing.

Testing pyramid and its variations

Now let's look at another core concept called the "testing pyramid." This concept is used in software development to balance different types of automated tests. There are three variations:

- **Pyramid**. There are enough tests of various kinds on each pyramid level. The fundament is formed by unit tests (fast and reliable), then the next one is integration tests (less than unit tests), and at the top are end-to-end tests (less than integration tests), which are slow, expensive, and often unstable.

- **Inverted pyramid**. As the name suggests, the pyramid is inverted, with disturbed proportions. There are many end-to-end tests, few integration tests, and almost no unit tests.
- **Diamond**. There is a similar number of end-to-end and unit tests, but the highest focus is on integration tests.

Most applications start with the pyramid approach, but they often end up being inverted. Why does this happen?

When building something new, we are often in a state of wishful thinking. We want to develop a great application that will be different from the ones we worked on in the past. We don't want to repeat mistakes, so we plan to pay special attention to testing and balance different types of tests.

However, as the project progresses, we continually add features and face pressure to meet deadlines. We are often encouraged to speed up by skipping tests, with the promise of addressing them after the release. With this, we end up accumulating technical debt.

Unfortunately, after the initial release, there is more work and new requests from customers, leaving little time to maintain our tests. New team members notice these patterns and may also neglect testing.

Over time, unit tests become less effective due to tangled code and complex relationships. The only option left is to rely heavily on end-to-end tests, despite their slowness and instability. This is how the pyramid gets inverted, and it is common in legacy systems.

Pyramid

You may have already heard about it—if not in testing, then perhaps in the context of the hierarchy of needs. The rule is simple: the most essential items (like water and food) are at the base, while less critical items (such as self-fulfillment) are at the top. This creates a balanced structure. A similar approach is used in the testing pyramid.

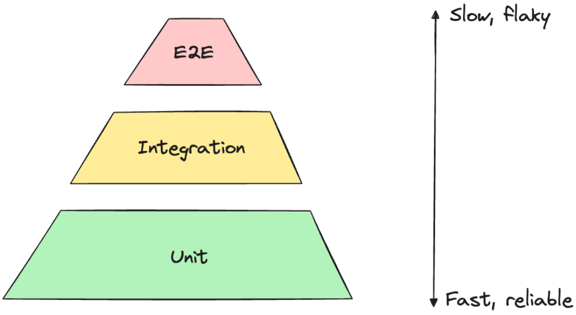


Figure 138. The pyramid

At the bottom, you will find unit tests that allow you to test a chunk of functionality in isolation, such as a single business rule that checks whether someone is an adult (>18) or not. These tests form the foundation of the application. Without proper (and sufficient) unit testing, you greatly reduce the chance of keeping your code well structured.

The next layer comprises integration tests, which allow you to check how several chunks interact to carry out a given process. For example, you might send a request to the API and validate that for given parameters, you receive a `201 Created` response. Behind the hood, the endpoint sends the command to the command handler, which stores the data in the database and, in the end, publishes the event to the event bus.

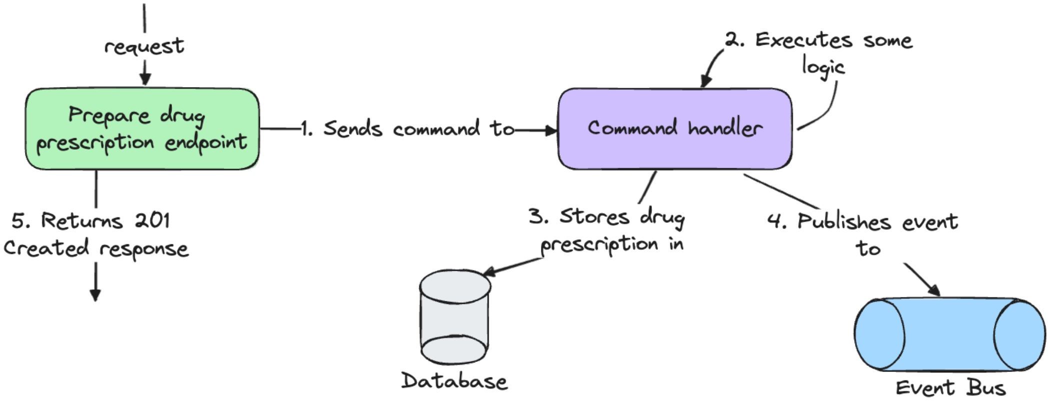


Figure 139. The example of what can be tested by the integration test

In integration tests, it is crucial to mimic the production environment without using mocks for infrastructure elements like databases. You want to check how the system performs in an environment that is as close as possible to the real one. Therefore, if you base your solution on Postgres (database) and RabbitMQ (message broker), you want to run the integration test against them, not their mocks.

This process used to be time-consuming, as test instances of infrastructure components had to be prepared, cleaned up, and maintained manually or by scripts created for this purpose. Furthermore, if the infrastructure component was shared between different environments or teams, there was a high chance that test results could be affected by others.

Today, it is much simpler, thanks to the *Test Containers*[2]. Instead of setting up a virtual machine with the database, cleaning it, and preparing for another test, you can use existing images of various components, run them in containers, and interact with them as needed. After running the integration test, the container is dropped, so you do not need to care about it anymore.

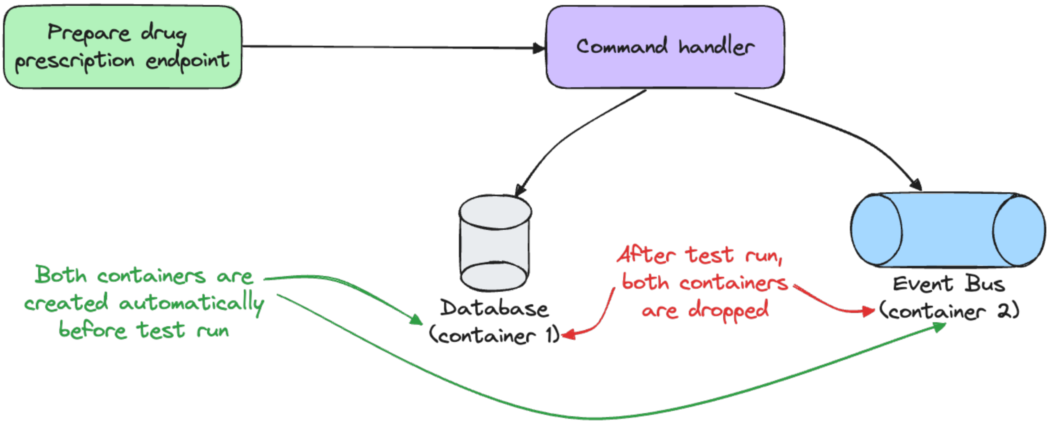

Prepare drug prescription endpoint
Command handler
Database (container 1)
Event Bus (container 2)
Both containers are created automatically before test run
After test run, both containers are dropped

Figure 140. Use a test container for each test, and it will be automatically removed afterward

At the time of writing this book, the Test Containers supported a wide range of integrations, including RabbitMQ, Kafka, Elasticsearch Postgres, MySQL, MongoDB, K3s, and many others.

[2] https://testcontainers.com

Although test containers allow us to run integration tests quite fast, they are still much slower than unit tests.

At the top of the pyramid are end-to-end tests. There are the least of them because they are slow and are prone to being flaky.

Flaky

means behaving in a non-deterministic way. In the context of tests, this means that when you run them, they may pass or fail at different points in a seemingly random way.

They are often run in the browser (web apps) or GUI (desktop). Because of this, they take a lot of time to execute—sometimes dozens of seconds or even minutes—as they have to click around or go through forms. Another issue is that when such a test fails, you only receive information that it failed, but not where exactly. In this situation, you need to examine the test, determine on which page it failed, identify the components involved in its execution, and then review the code that uses those components.

That is why, if you cannot avoid end-to-end tests, you should keep their number to a minimum. I recommend covering only business-critical processes and skipping the rest. What does a critical process look like?

Let’s say you work for an online shop where users search for products, place orders, and pay for them online. You want to ensure with 100% certainty that:

1. Customers can search for products (and find them).
2. Customers can successfully place orders and make payments.

It would make sense to verify that customers can add products to the shopping cart, proceed to the checkout page, and complete the payment. If these processes fail, you risk losing customers—who may turn to other shopping sites—and potentially losing revenue.

On the other hand, if a product review form does not work or advertisements are not displayed, these issues are likely not business-critical. Therefore, it is crucial to properly assess which processes are the key ones. Typically, the development team cannot make this decision alone; it requires close collaboration with the business.

Inverted pyramid

When you search for information about the inverted testing pyramid, you will often find it described as an anti-pattern. The advice commonly given is to avoid it because it can create a false sense of security.

Is this statement true? Well, as with many other topics in software development, the answer is: it depends.

First, let's check what it looks like:

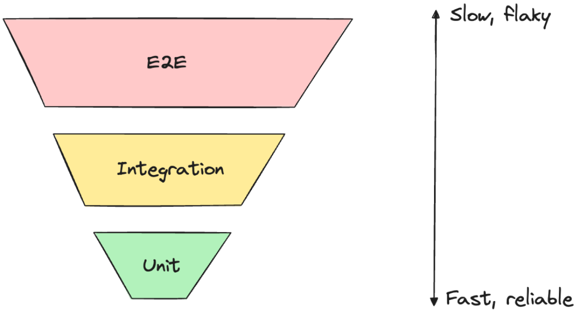


Figure 141. The inverted pyramid

Here, the majority of tests focus on end-to-end testing. As you already know, these tests are the slowest in the entire chain and tend to be flaky.

In the middle, there are integration tests. In practice, when dealing with a system built on top of an inverted pyramid, there are much fewer integration tests than end-to-end, and sometimes they do not even exist.

The smallest part of tests focuses on unit tests. This is usually due to spaghetti code, where testing smaller parts (units) of the system doesn't bring much benefit. In such cases, everything is so tangled that touching one function or class affects many others.

The inverted pyramid of tests most often results from years of neglect, leading to legacy applications. Of course, no one plans for this, but it just happens. By the time we realize what has happened, it is usually too late to make major changes.

In such cases, even though it is not an optimal solution, it is sometimes the only way to test the application meaningfully and increase confidence that it works as it should—and probably won't blow up.

The same situation arises if you join a company and realize that the legacy application has no tests at all. Before refactoring it, you will want to ensure that you don't break the functionality, and it might make more sense to add end-to-end tests as a verification check.

Of course, if you are dealing with an inverted pyramid of tests, the chance of stability is relatively low, as they tend to fail randomly from time to time. In legacy systems, tests might depend on each other. For example:

1. The first test adds several items to the basket to check adding functionality.
2. The second test uses these items to calculate and validate the total price of the basket.
3. The third test, uses these items to proceed to the checkout page.

Moreover, running all these tests can be costly and time-consuming—the entire test suite might take several hours to complete.

In summary, while the inverted testing pyramid is generally seen as an anti-pattern, it may sometimes be the only feasible approach, particularly with legacy applications.

Diamond

The third variation is called diamond, and it focuses mainly on the integration tests.

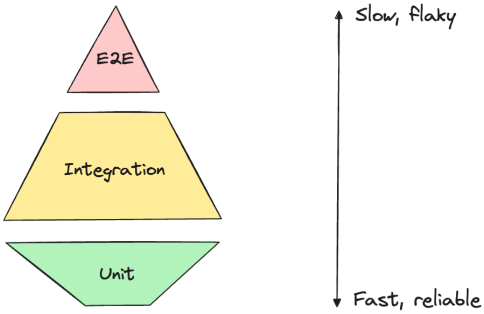


Figure 142. The diamond

As mentioned in the pyramid section, unit tests are the core of testing. However, their main limitation is that they test isolated pieces of functionality. This can result in individual components working correctly in isolation but failing to produce the desired result when integrated.

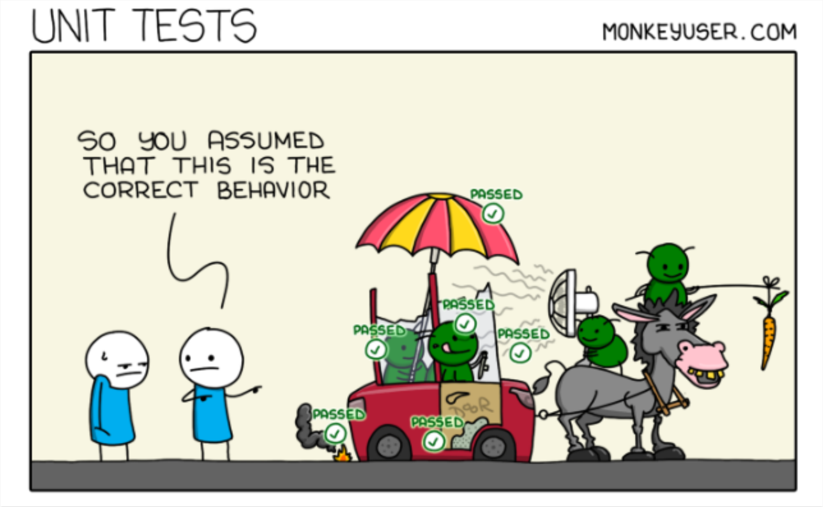


Figure 143. Unit test checks individual chunks

This is where integration tests are helpful. They allow you to test how several components interact with each other, making it possible to test entire vertical slices, such as the preparation of a drug prescription.

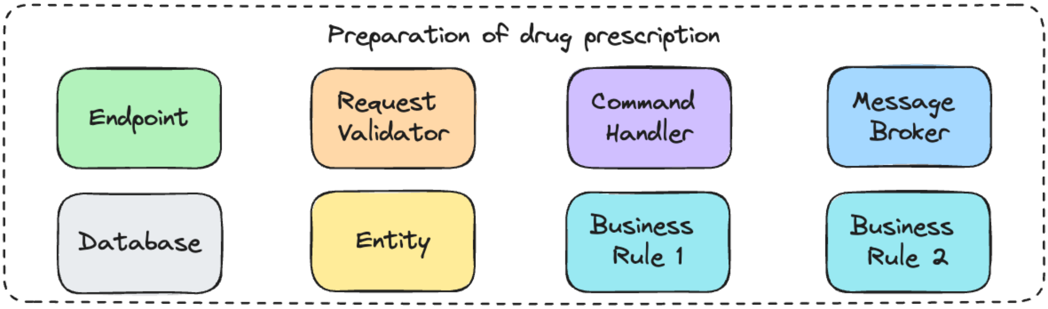


Figure 144. Testing the entire vertical slice

All the elements shown above create the business process. When calling the endpoint, they interact with each other, resulting in a response that is the outcome of their combined actions. Testing their interaction is a sensible approach.

You will most likely face a diamond in applications that are heavily based on API endpoints. For example, when developing a public API for external clients, you want to make sure that every endpoint call is reliable.

What about unit and end-to-end tests in such systems?

The number of unit tests will depend on the complexity of the business logic. For a CRUD application, there won't be too many unit tests. However, in more complex cases, the number of unit tests will be comparable to integration tests. If there are more unit tests than integration tests, you might be dealing with the pyramid.

End-to-end tests may take the form of smoke tests, which are run on a production candidate before it is promoted. As mentioned earlier, if you can avoid using these tests, it is advisable to do so.

From my diary

Theory is one thing; reality is another. You can blindly follow concepts, but what really matters is what works well in your specific context. Over years of working with various systems, I have gathered key observations that I would like to share.

Every time we prioritized integration tests over unit tests, we observed increased endpoint reliability and stability. However, this approach inadvertently resulted in a decline in code structure and its quality. The focus on overall functionality came at the expense of maintaining well-designed code at the chunk level.

When we focused too much on unit tests and too little on integration tests, we were able to keep our code well structured and it was easy to modify. The downside was that our endpoints often failed to perform as expected due to insufficient integration tests. Even though unit tests passed, we had many issues integrating multiple chunks.

Focusing mainly on end-to-end tests in greenfield applications almost always led to huge technical debt and, eventually, an inverted pyramid of tests. The plan was to start with end-to-end tests and gradually add other types of tests. However, this approach often failed because there was never enough time to add additional tests due to the constant addition of new features.

End-to-end tests have always been the most helpful for us when working with legacy projects. When no tests were in place, the easiest approach before refactoring was to cover key business processes with E2E tests. And when such tests did exist, they were the only way to understand how the process worked, especially in the absence of domain experts.

Flaky end-to-end tests often caused frustration and neglect. When there were hundreds or thousands (ouch!) of such tests, running them took several hours. Sometimes, some tests would fail randomly, so we had to rerun that part of the suite. Then they would pass, but we couldn't be sure if other tests had affected the results. So we had to run them again, and then other tests failed. This led to a repetitive cycle of testing and retesting.

The amount of stuff we had to mock was a good indicator of how well our test was designed. Each time we needed many mocks, the test was unstable and useless. It was a trigger for us to review and redesign the code we were testing.

Shifting our focus to behaviors rather than all possible combinations resulted in the creation of more effective tests. By concentrating on the expected outcomes and interactions, we were able to develop tests that were well-structured, reliable, and easier to maintain. This behavioral approach allowed us to better align our testing with customer requirements.

Performance

One of my favorite and most important parts of testing is performance evaluation. It is fascinating to see how an application behaves when loaded with the expected traffic before it goes into production. I also enjoy discovering the limits of the current architecture, as it provides insight into potential issues we might face.

In the upcoming sections, you will learn about performance testing. We will examine the differences between load and stress testing and discuss when and how often each should be performed.

Load testing

Load testing helps you check how your application performs under expected traffic for a given period.

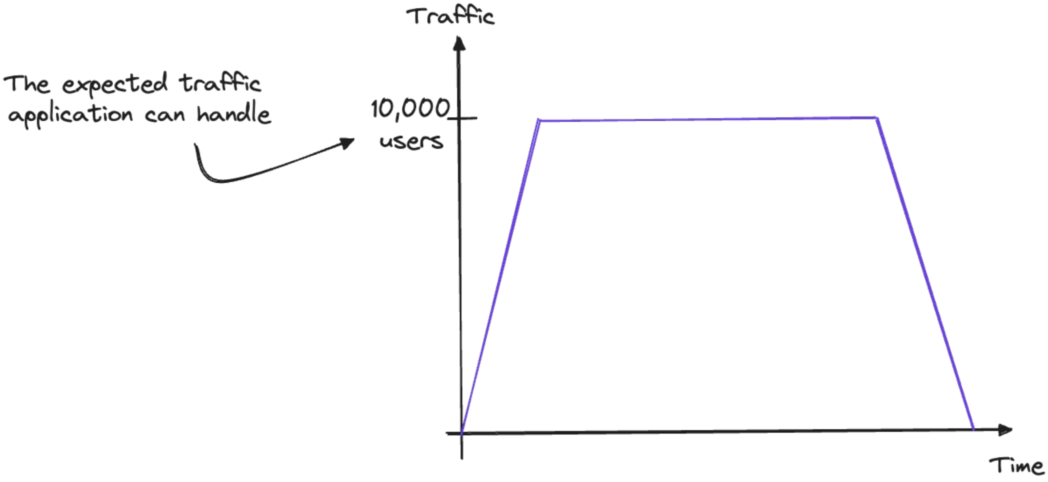


Figure 145. Load tests check how the application behaves under the expected load

What does this mean in practice?

Let's assume you have developed an MVP that hasn't been released yet. Based on your analysis, you expect around 10,000 concurrent users in the first month.

It is a good idea to test whether your application can handle this level of traffic before you face any surprises in production. Load testing allows you to proactively adjust your architecture if needed, rather than reacting to issues in a live environment and dealing with the pressure to fix them quickly.

One of the first questions I am usually asked is what should be tested with load tests. This is a good question!

Your application will not be used evenly. Some areas will be used less often, such as registration, changing passwords, or changing permissions. However, other areas, such as buying tickets for an event or using the payment gateway, will be accessed heavily. That is why you shouldn't treat all areas the same and plan your tests accordingly.

The first step is to identify the areas where you want to perform load tests. As a rule of thumb, start by selecting the areas that are expected to be used most frequently. For an application that has not yet been released, this can be challenging, as it involves making predictions. While you can estimate which areas will be heavily used, you can't be certain. The accuracy of these predictions will vary depending on the environment.

For example, in a large-scale organization that produces hardware, you might predict that 5,000 devices will always synchronize simultaneously at a specific time, let's say 3 pm, and the application must be able to handle this load. On the other hand, if you work in a startup, you usually don't know which parts of the application will be used most heavily, and your assumptions might change completely if the business pivots.

In the case of an application that has already been released, this process is easier due to existing data on user behavior. This data helps you easily identify which areas are bottlenecks.

Once you have identified the areas with the heaviest usage, you need to prepare a plan that includes:

- **The scenarios you want to test**. Thanks to the discovery, you have a list of areas you want to subject to load testing. Now, you need to develop specific scenarios for these areas. These scenarios will outline the actions and conditions you want to test. I recommend preparing scenarios that imitate the regular user flow. For example, in the case of buying tickets, your test should mirror the user's journey from accessing the home page, searching for the event, opening the event page, selecting the number of tickets, proceeding to the checkout page, and completing the payment.
- **The number of concurrent users**. It defines how many users you want to imitate to generate the expected traffic. This number may vary between different areas; for example, one area might typically handle 10,000 concurrent users while another might handle 25,000. I also recommend adding 20-30% for each load test because it gives you a certain reserve in case the traffic increases fast in the production.
- **The time limitation to perform load tests**. Load tests should not run indefinitely, as this can incur costs and delay feedback. It is advisable to set a time limit, such as running each load test for approximately 30 seconds, with all tests completed within 10-15 minutes.

One common pitfall is assuming that users will follow a predictable pattern. For example, a 70% discount on a popular item may not be spread evenly over 15 minutes for 15,000 users (1,000 users per minute). Instead, you might see most of the users in the first few seconds. Be cautious of such scenarios!

The last aspect to consider is where and when to run load tests. This is a nuanced question with no single answer; it depends on your specific context.

Load testing should not be a one-time action but a repeatable process. Ideally, it should be conducted for every change made to the production code. However, this may not always be feasible (as discussed in previous steps). Therefore, there are various approaches to scheduling load tests.

If you deploy the application multiple times per day (like in continuous deployment), it makes sense to run load tests each time the commit is pushed to production. This way, load tests will be a part of your deployment pipeline.

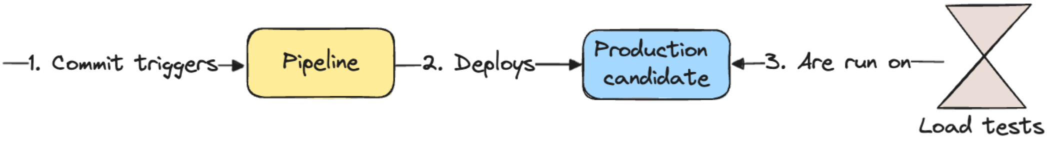


Figure 146. Run load tests on every commit

Based on my experience, this can work only when your code is correctly modularized. In such a case, you do not need to run the entire test suite, which would take much time. Instead, you only need to run the tests related to the modified areas.

A similar approach applies to continuous delivery. However, in this case, the load tests are executed after the deployment is triggered (not automatically on every commit) and when the production candidate is ready. So, if you deploy the application once a day at night, the tests will be executed only at that time.

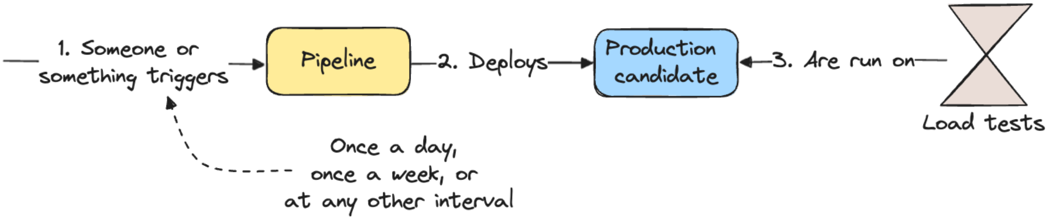

1. Someone or something triggers
Pipeline
2. Deploys
Production candidate
3. Are run on
Load tests
Once a day, once a week, or at any other interval

Figure 147. Run load tests at a given point in time

In environments where applications cannot be released at any time and must undergo rigorous testing phases, it is common to conduct load tests prior to scheduled releases. Since each release requires preparation time, you will typically perform load testing multiple times a year in staging or UAT environments.

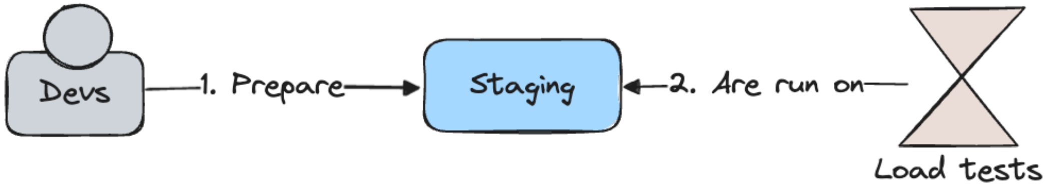


Figure 148. Run load tests before scheduled release

As you can see, it is impossible to find a one-size-fits-all solution, and you always have to consider the context and environment around you.

Stress testing

Stress tests are often confused with load tests. Although they are in the same category—performance—they differ significantly.

The primary goal of stress testing is to assess how an application behaves under extreme conditions, such as a sudden spike in traffic (e.g., double or more than usual).

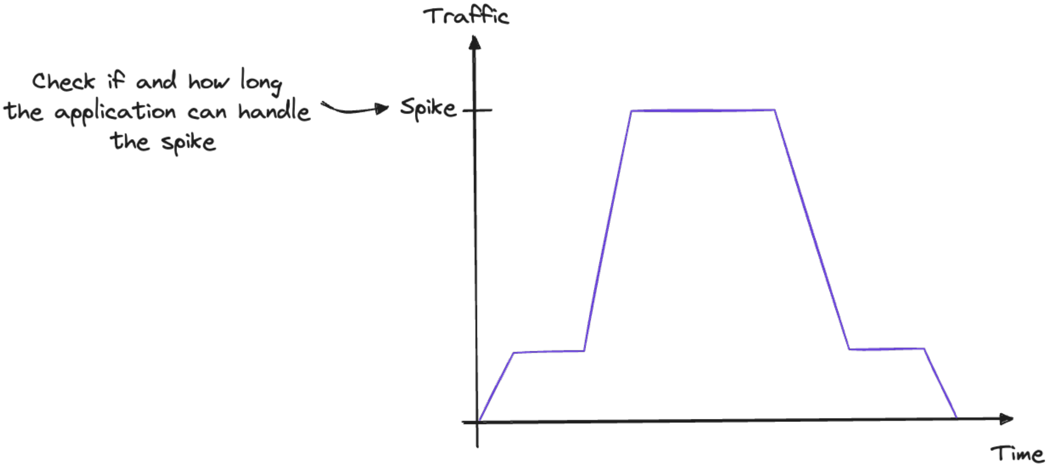


Figure 149. Stress tests can assess how the application performs during a spike in traffic

This way, you can spot bottlenecks and verify whether the application can handle the anticipated user load, such as during high-demand events like ticket sales.

Additionally, stress testing can help determine when you might need to replace or adjust the current architecture by pushing the application to its limits.

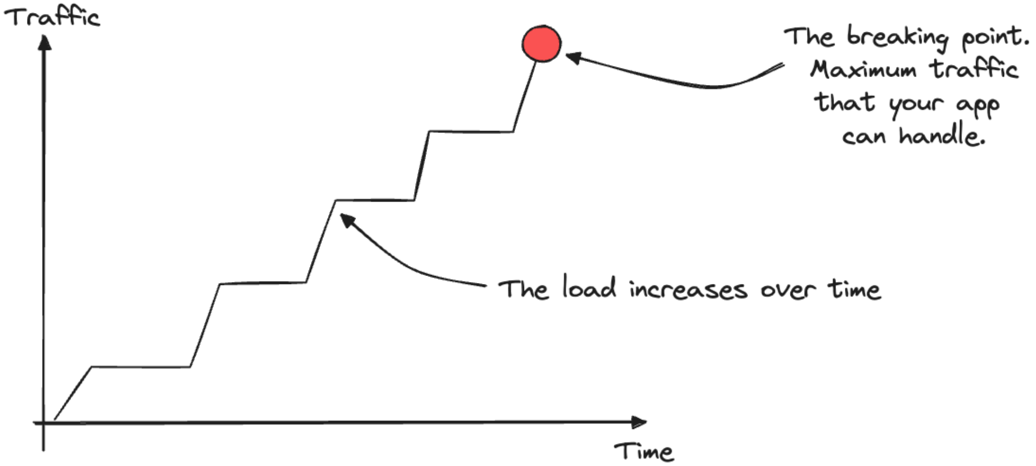


Figure 150. Stress tests can check the limits of your application architecture

> Suppose you have determined that your application can handle a maximum of 30,000 concurrent users. Currently, you serve 10,000 users, with a growth rate of 1,000 users per month. This gives you nearly two years before you need to consider changing the current architecture. In this case, you should begin planning for an upgrade when you approach 23,000 to 25,000 users or if the growth rate accelerates significantly (e.g., 5, 10, or more times).

You can also monitor how fast your application scales and the cost it incurs over a given period of time. This helps you avoid situations where your application cannot handle the production traffic volume, potentially leading to downtime or unexpected expenses. All in a controlled environment.

You can also leverage everything you prepared for load tests—such as identified bottlenecks, scenarios, and so on. The location for running stress tests will depend on the specific context, just as it does for load tests.

One significant difference is when to run these tests. Unlike load tests, stress tests should not be conducted too frequently, as they are too expensive and they validate the application against specific circumstances (not common). To be pragmatic, I recommend running stress tests in the following cases:

1. **Before releasing the MVP**, to check the limits of your application. Depending on how quickly your product gains popularity, you may need to consider alternative solutions in the future.
2. **Before events like Black Friday, Cyber Monday**, or other highly anticipated events, like performances of popular singers. I can’t count how many times the systems I used to buy various tickets couldn’t handle the traffic on the site.
3. **After introducing a feature** you know will be heavily used.
4. **Periodically**—this could be once or twice a month, several times a year, etc. Observe what works best in your case.

Properly conducted stress tests can provide many benefits to the business and prepare us for situations involving extreme use of our application. However, it is important to remember that we cannot predict every possible scenario, so we can only minimize risks and potential impacts, not eliminate them.

Penetration testing

The main purpose of penetration testing (also known as pentesting) is to find security holes in your application. Ultimately, everyone should ensure that data is safe, that there are no harmful scripts on pages, and that the application is always available to customers.

However, it is not uncommon for this step to be completely forgotten or skipped when planning a testing strategy. The former occurs due to a lack of experience building resilient systems, and the latter because of additional costs. Unlike other tests, penetration testing is often outsourced to an external company that specializes in checking application vulnerabilities.

Why outsource? Not every company has a team of cyber-security experts. In fact, this is rare, even in larger organizations. The advantage of hiring external consultants is that, since they are unfamiliar with your system, they can explore it in ways you might not have considered.

How do penetration tests work? It depends. An expert or a group of experts might request a link to your application, or you may be asked to provide more detailed information, such as API documentation or login credentials.

With this information, they will start exploring the application, testing it against tons of various vulnerabilities, including the following:

- Insecure Direct Object References (IDOR)
- DoS and DDoS attacks
- Passing undesired values through endpoint parameters
- Lack of encryption on sensitive information
- Exposure of sensitive information in logs
- Supply chain attacks
- Cross-Site Scripting (XSS)
- SQL Injection
- Misconfiguration of infrastructure
- Exposure of passwords

They are all described in detail in the security section.

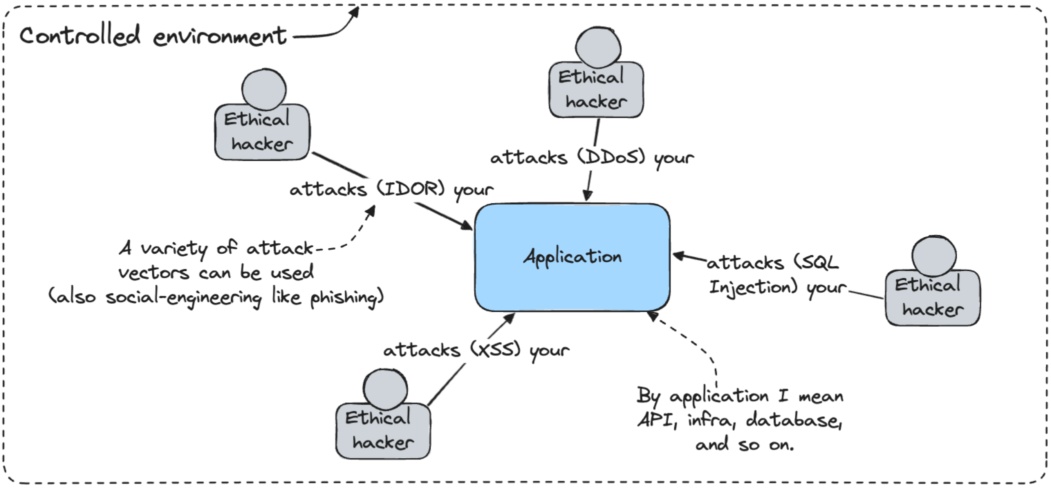

Controlled environment
Ethical hacker
attacks (DDoS) your
Ethical hacker
attacks (IDOR) your
A variety of attack vectors can be used (also social-engineering like phishing)
Application
attacks (SQL Injection) your
Ethical hacker
attacks (XSS) your
Ethical hacker
By application I mean API, infra, database, and so on.

Figure 151. Penetration testing examines the application's susceptibility to security vulnerabilities

Finally, they will prepare a document listing all identified vulnerabilities, categorized by their severity—low, medium, critical, etc. From that point forward, it will be your responsibility to address and resolve these issues.

🔑 When developing an application, I recommend using automated tools to validate your code (static code analysis) and infrastructure and monitor security. You can integrate these tools into the CI/CD pipeline. The sooner you set it up, the better—this is always one of the first things I do in my projects. Please note that while these tools are helpful, they do not replace penetration testing; they are just an additional layer of security.

🔑 When dealing with a legacy app or audit software, perform a security scan using automated tools (like in the above case). After fixing critical issues, plan for penetration testing with an external company. This way, you will validate the most important part (IMO) of the software—its security.

Penetration testing should not be a one-time action. It is a good idea to:

1. **Repeat it periodically**. For example, once a year. Usually, this is enough, but if necessary, you can repeat it more often.

2. **Repeat it before entering an entirely new market**. For example, when planning to serve customers on another continent. New users, new traction, potential interest of hackers.
3. **Repeat it after switching cloud providers or hosting in general**. For example, when moving from AWS to Azure.

If you want to sleep well, I recommend not skipping this kind of testing. If you were to ask me about one type of test I couldn't live without, it would be penetration tests (followed by load tests). It is always better to identify security holes in a controlled environment rather than in production, where customer data could be compromised.

Always ask yourself what the worst-case scenario would be if customer data leaked or your app went down for an hour. A data breach could result in millions of dollars in compensation, while application downtime could lead to significant financial losses due to halted transactions.

Given these potential consequences, is it wise to skip penetration testing?

Our case

Let's look at our case. We are implementing a greenfield application with little expected traffic (1,000-1,500 users). Additionally, we have complete control over our API and clients that integrate it. Some areas (modules) of our domain are complex, some are complicated, while the rest are simple.

Depending on the area, we can focus more or less on different tests:

1. In simple modules (CRUD), there will be almost no behavior to test, while in complex and complicated modules, it makes sense to keep the balance between unit and integration tests. We should keep the end-to-end tests as small as possible and only smoke test critical parts.
2. Since our application is not expected to have many users, the predicted number of requests is low, and the deadline is approaching, we can decide to skip load and stress tests. We accept the risk that the application might not respond if traffic exceeds our predictions. Alternatively, we can perform load tests once and, after the MVP, connect them to our pipeline (we do not have time to do it now = a technical debt).

3. To minimize security risks, we connected automated tools to our pipeline and hired a penetration testing partner.

Recap

In this step, you discovered areas of testing that I consider most important from a software architect's perspective. Of course, one might say that there are other important topics, like regression or acceptance testing, and I accept that.

You have learned about the traditional vs. shift-left testing, testing pyramid and its variations, and I have shared a practical approach to dealing with them. I have also discussed what worked well and the challenges I encountered while working with the unit, integration, and end-to-end tests.

Another key piece of knowledge you have gained is related to performance testing, particularly in the areas of load and stress tests. It is crucial to be able to distinguish between these two and know when to use each.

Finally, I introduced my favorite part of testing, which I consider the most important—penetration testing. This allows white hat hackers to attack the application in a controlled environment and find security holes before you release it to production.

Now, it is time for the core part of this book: the evolution of your software architecture.

STEP 7: Evolve Your Architecture

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

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

Is it possible? Yes.

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

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

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

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

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

Observe and adapt. Evolution, not revolution.

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

Project paradox

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

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

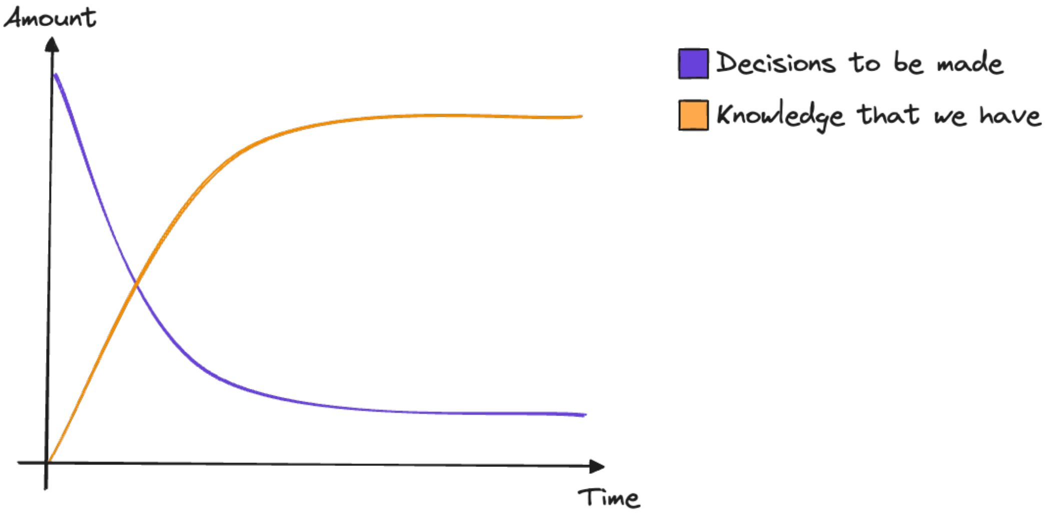


Figure 152. Relation between the amount of knowledge and the number of decisions to be made

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

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

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

Common architectural pitfalls

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

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

Let’s look closer at each of them.

Too complex architecture from the start

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

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

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

- Go with microservices.

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

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

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

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

There are a few problems:

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

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

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

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

Too trivial architecture for too long

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

However, you face three major challenges:

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

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

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

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

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

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

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

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

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

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

You add special logic to handle all these cases.

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

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

Congrats! You have created another big ball of mud:

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

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

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

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

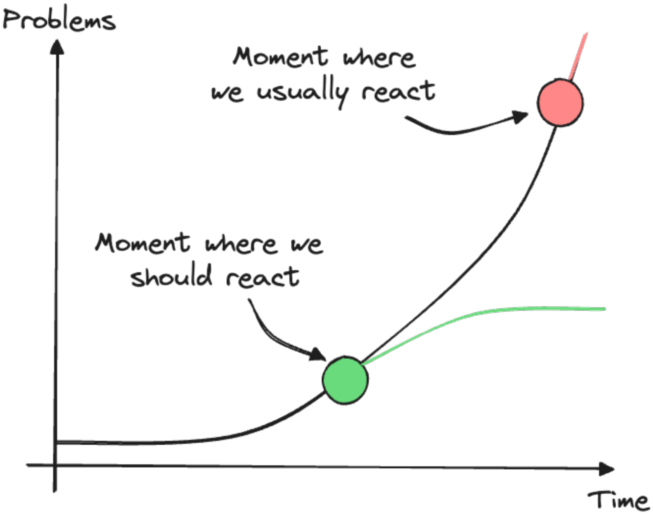


Figure 153. Increasing problems of maintaining trivial architecture for too long

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

Evolve together with your business

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

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

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

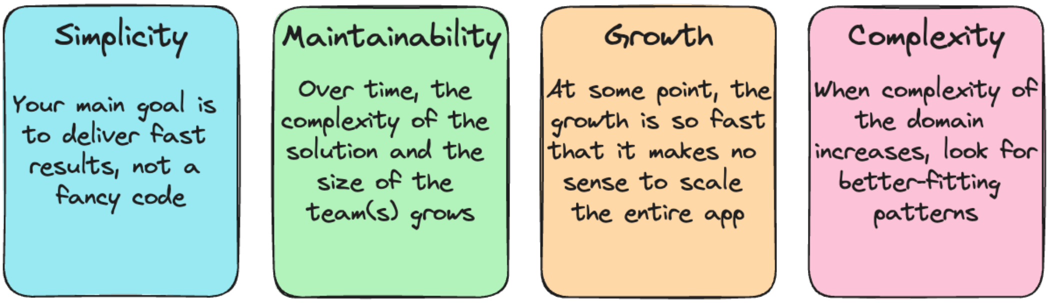

Simplicity
Your main goal is to deliver fast results, not a fancy code
Maintainability
Over time, the complexity of the solution and the size of the team(s) grows
Growth
At some point, the growth is so fast that it makes no sense to scale the entire app
Complexity
When complexity of the domain increases, look for better-fitting patterns

Figure 154. Four steps of evolution

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

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

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

No more theory; let's get to work.

First: Focus on simplicity

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

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

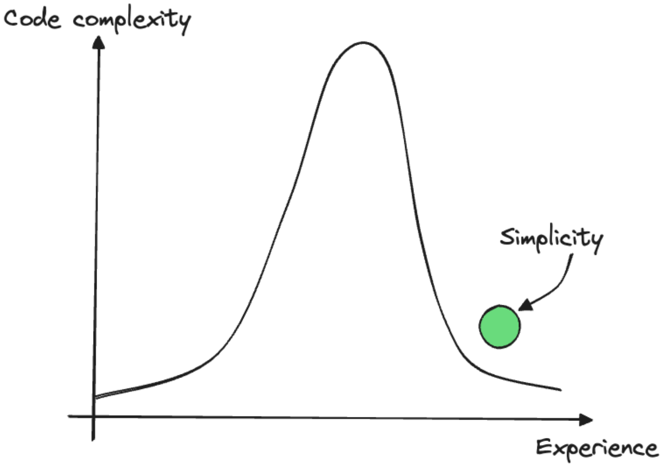


Figure 155. Code complexity depending on experience

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

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

Requirements

We identified eight main functional requirements for the MVP that we need to address:

1. Registering a new account and logging in.
2. Scheduling an appointment for the patient.
3. Scheduling a follow-up appointment for the patient.
4. Handling patient visits, including treatment plan preparation.
5. Prescribing drugs for the patient.
6. Finishing patient treatment.
7. Updating medical records with the details of the finished treatment.
8. Preparing the invoice and sending it to the patient.

Main considerations

Based on the entire analysis, we noted down all important considerations:

1. We expect 3-5 medical clinics to use the application in the first three months.
2. Each private medical clinic has different opening hours (some are open 6, some 8, and some 12 hours a day).
3. We operate exclusively in Europe, so time differences are minimal.
4. We expect 1,000-1,500 patients to use the application. Our forecast is that each patient will create about 100 requests per month (20 working days). This means that all patients will generate around $1,500 \times 100 = 150,000$ requests.
5. The five clinics we consulted have around 100 employees (front desk staff and doctors). We assume each employee will generate around 1,000 requests per day, translating to $100 \times 1,000 \times 20 days = 2,000,000$ requests per month.
6. The total traffic we will need to handle is $150,000 + 2,000,000 = 2,150,000$ requests. Assuming that the application will be used for 12 hours a day, it gives $20 days * 12 hours * 60 mins * 60 seconds = 864,000$ seconds. It translates to $\frac{2150000}{864000} \approx 2.5$ requests per second. There is no need to have a complex

infrastructure. **Note:** For the purpose of this book, I am assuming that we have an even distribution of requests. In the real world, you may end up in a situation where 80% of the requests come between 3 pm and 4 pm, with the remaining 20% spread over the other hours.

7. We can handle the application's availability using one of the standard cloud solutions, which guarantees 99.5%.
8. Doctors making home visits will use their smartphones, so we need a mobile application.
9. Doctors and the front desk staff in the clinic will use the web application, so we need to implement it as well.

> Remember that the systems we want to build are usually based on assumptions since we do not know how the market will react to our idea.

Decisions

Before we start, let's look at the key architectural drivers we selected to help us make decisions.

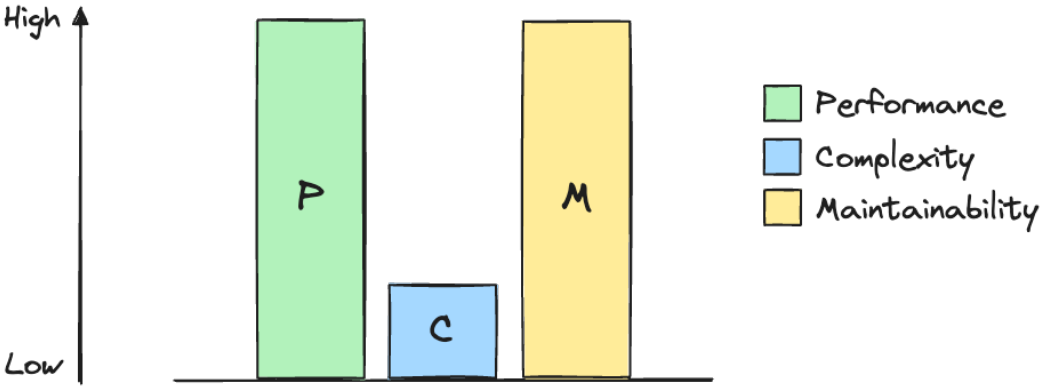


Figure 156. The state of selected architectural drivers

In the first phase of the application, we want to keep:

- Performance at the highest possible level.
- Complexity at the lowest possible level.
- Maintainability at the highest possible level.

We are going to observe how they will change over time.

Of course you can take into consideration any architectural driver. The three selected here are quite popular, but you may also focus on others. It is your decision.

As we forecast extremely low traffic of about `2.5` RPS, we decided to use the simplest platform-as-a-service (PaaS) solution offered by the cloud provider of our choice. Even if traffic increases fivefold, this PaaS solution will still be sufficient to handle the load. This way, we can be confident that the application will perform well.

In one of the previous sections, we decided to use the modular monolith for the backend to keep the solution's complexity low.

To simplify the frontend, we will use a *Progressive Web App (PWA)*. This way, we can handle both mobile and web applications. There is no need to build a native mobile app based on the current requirements. Additionally, we have decided to outsource this development due to the lack of expertise within our team.

Our decisions regarding the backend and database were based on the team's competence matrix. Most team members have extensive experience with C#, REST API, and relational databases (Postgres). After long discussions, the team decided to go in this direction and use the existing experience due to the lack of time to learn new technologies.

In the end, the components of our application look as follows:

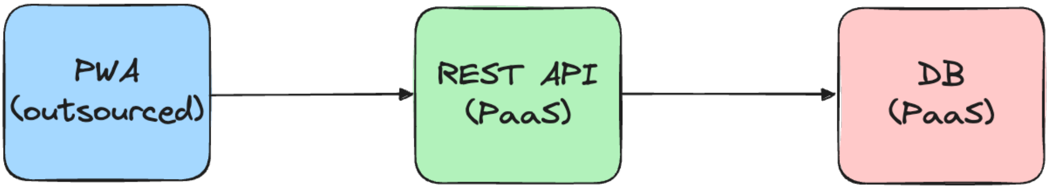


Figure 157. The overview of the simple architecture

- PWA is outsourced to an external company.
- Both the API and the database (Postgres) run on a separate platform-as-a-service.
- They can be easily scaled if needed.

> You can also consider using infrastructure-as-a-service (IaaS), which would mean running the API, and database on virtual machines. However, since our team lacks prior experience with infrastructure management, leveraging the capabilities of platform-as-a-service (PaaS) is a more practical choice for us. It is always a matter of trade-offs: if PaaS costs become too high and you are comfortable managing virtual machines—including system updates, patching, etc.—IaaS might be a better option. Additionally, a hybrid approach, combining both IaaS and PaaS, is also a viable option.

Code structure

Once you have identified the application components, it is important to consider how to structure the code. It should be as simple and understandable as possible.

Keep in mind that at this stage, you may not fully grasp how the application will behave, the complexity of each module, how it will evolve with new features, or how your customers will interact with it. Essentially, you are operating in a realm of uncertainty.

It can be beneficial to defer many technical decisions (project paradox) until you have more information.

> I will show you one way to approach this step. Feel free to do it differently, but remember to keep it simple.

For our case, I recommend creating a single project for production code—it does not matter if your code is implemented using JavaScript, C#, Java, or any other language.

Next, add new folders and name them as you did for bounded contexts during the analysis. Each folder will represent a module.

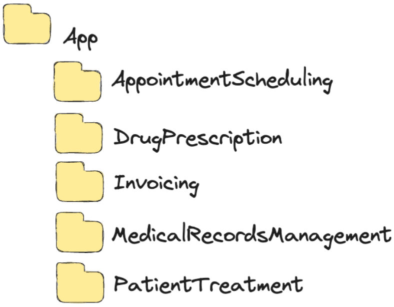


Figure 158. Modules represented as folders

Take each process and create a folder for it in the matching module.

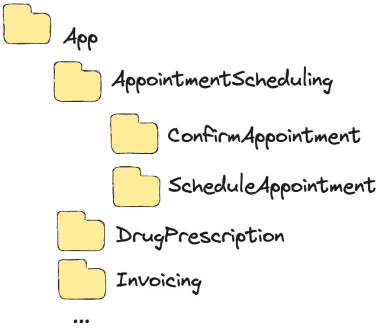


Figure 159. Processes represented as folders inside modules

Put all the logic related to each process in its own folder. By “all,” I mean:

- Endpoints
- Validators
- DTOs
- Database access
- Entities

And everything related to this business process.

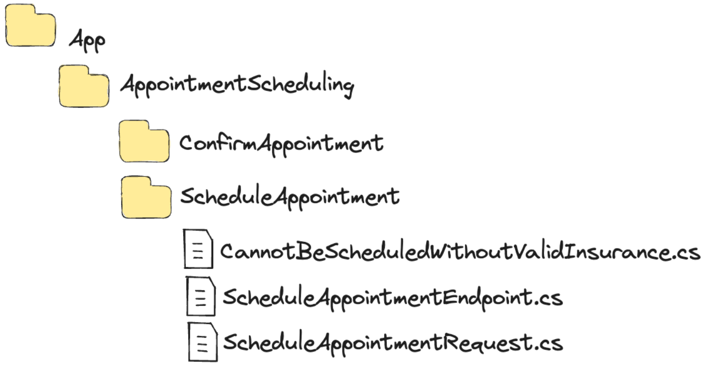

App
AppointmentScheduling
ConfirmAppointment
ScheduleAppointment
CannotBeScheduledWithoutValidInsurance.cs
ScheduleAppointmentEndpoint.cs
ScheduleAppointmentRequest.cs

Figure 160. All code related to the selected business process

This way, each process is encapsulated within its own folder. Instead of technicalities and layers, you focus on the entire logic related to the business process.

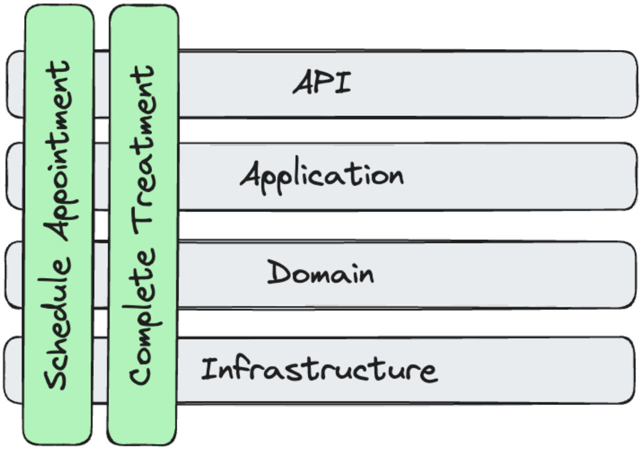


Figure 161. Focus on the business process instead of layers

If you want to remove the process, you remove the folder. That’s it.

This approach is a variation of the *Vertical Slice Architecture* which was described by Jimmy Bogard. You can read more about it here[2].

If code needs to be shared between two or more processes within a module, e.g., an *Appointment* entity or *AppointmentsPersistence*, you can put it in the module root folder.

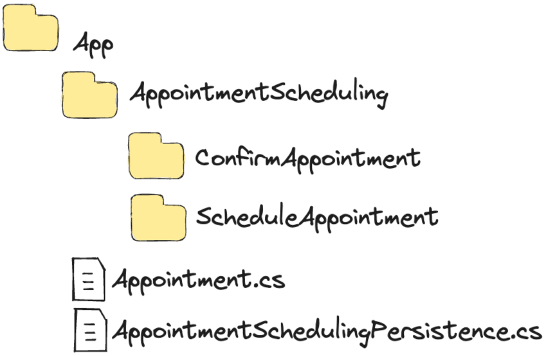


Figure 162. Code shared by multiple processes within a module

If you need to share code between various modules, create another folder next to all modules (root level) and name it *Common*, *Shared*, *BuildingBlocks*, or something similar. You can put reusable things like API error handling, business rules validation mechanisms, or endpoint filters in this folder (wrapped into their own folders like *ErrorHandling*, or *EndpointFiltering*). It should not contain any business logic specific to any of the modules.

Behavioral entities

My recommendation is to treat entities not just as a collection of properties, but also to incorporate behaviors. Alongside purely informational properties such as:

- ConfirmedAt (date and time when the appointment was confirmed)
- ScheduledAt (date and time when the appointment is scheduled)
- AssignedDoctor (object that contains details of the assigned doctor)

You could also include functions that represent behaviors like:

[2] https://www.jimmybogard.com/vertical-slice-architecture/

- Confirm
- Schedule
- AssignDoctor

Each function allows you to validate related business logic and, upon successful validation, modify the state of the object accordingly.

```
class Appointment:
    properties:
        Id: uuid
        ConfirmedAt: date
        AssignedDoctor: doctor
        ScheduledAt: date

    methods:
        Confirm(confirmedAt, assignedDoctor):
            // Validate against business rule
            check(AppointmentMustBeConfirmedWithDoctor(assignedDoctor))

            // If success, then confirm the appointment
            AssignedDoctor = assignedDoctor
            ConfirmedAt = confirmedAt

        Schedule(scheduledAt):
            // Validate against business rules
            ...

            // If success, then schedule the appointment
            ScheduledAt = scheduledAt
```

This method allows you to implement behaviors directly on the entity level using business rules defined in each process. There is no need for an external service or manager to handle business logic.

⚠ There is a risk that, due to increased complexity, such a class will be extended with more and more properties and behaviors, creating what is known as a "God class." You need to react beforehand and apply other, better-fitting patterns.

API endpoints

Besides the code structure, it is a good idea to design the API endpoints in a way that makes them easy to extend. You can start with:

1. POST: `api/appointments` to create an appointment.
2. PUT: `api/appointments/1` to schedule an appointment.
3. GET: `api/appointments` to get all the appointments.
4. GET: `api/appointments/1` to get the details of the selected appointment.
5. DELETE: `api/appointments/1` to delete the selected appointment.

Remember that adjustments might be needed when new functionality is introduced. In our case, in addition to creating and scheduling the appointment, at least one doctor must confirm it. Therefore, the API could look slightly different if we treat it as an update (`PUT`) operation.

1. PUT: `api/appointments/1/status` to confirm an appointment (with status `Confirmed` passed in the request body).
2. PUT: `api/appointments/1/status` to schedule an appointment with status `Scheduled` passed in the request body).

If there is a new requirement to include the option to study notes from the appointment, you can add another `GET` operation.

- GET: `api/appointments/1/notes`

And so on. Adopting this approach allows you to easily add or modify endpoints without affecting the overall structure, making it easier to maintain and update in the long run.

Another common approach is to determine endpoint paths based on whether the actions are singular or plural.

1. POST: `api/appointments`. Only if you can create multiple appointments with a single request, if not, then it should be `api/appointment`.
2. PUT: `api/appointment/1`
3. GET: `api/appointments`
4. GET: `api/appointment/1`
5. DELETE: `api/appointment/1`

Personally, I prefer the first approach (without distinguishing between singular and plural), but I also accept this one. Ultimately, the choice is up to you and your team.

API versioning

Another critical aspect of designing the APIs is versioning. With versioning, you can introduce breaking changes without immediately affecting all clients that use your API.

Let's say that we introduced an endpoint responsible for scheduling the appointment:

> PUT: `api/appointments/1`

At some point, we were asked to add appointment confirmation. To do that, we have to handle two `PUT` operations on appointments, so we decided to distinguish them by defining the following endpoint:

> PUT: `api/appointments/1/status`

We also dropped the existing `PUT api/appointments/1`. We deployed the new API to production, and suddenly, frontend clients (web and mobile apps) stopped working because they were still calling the old endpoint.

That is why you have to introduce API versioning sooner rather than later. With versioning, you can give clients time to migrate to a newer version. For example, you can notify them that old endpoints (v1) will be dropped in 90 days and that they need to switch to the newer version (v2) before then.

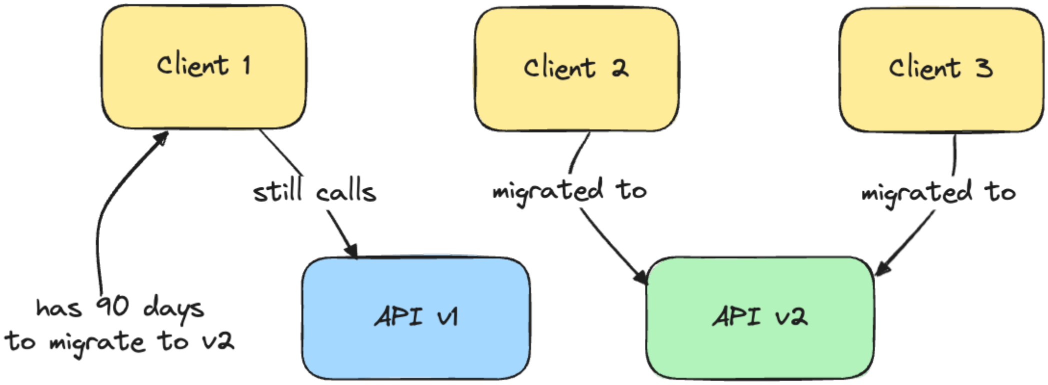


Figure 163. Multiple API versions running at the same time

There are several ways of handling API versioning, but I will focus on two of the most popular methods.

The first method is versioning through the path. If you choose this approach, you must include the API version in the URL. This means that the API path will look like this:

```
api/v1/appointments/1
api/v2/appointments/1/status
```

If you only change the parameters of the endpoint, it will look like this:

```
api/v1/appointments/1
api/v2/appointments/1
```

It is also important to inform people responsible for clients' development that they must migrate to the newer version within a given time period.

> In large-scale organizations with poor internal process execution, such changes might take more than one year. This means that after the new API version with breaking changes is introduced, you will need to maintain the older one for quite a long time.

Pros	Cons
The version is included in the URL, making it evident that it changed.	Changing the version requires changing the URL on the client side.
Caching systems can easily distinguish URLs.	It might become complex to deal with more than two API versions.

The second method is versioning through headers. In this case, you require clients to include the information about the version they call in the header. It will look similar to:

```
somename-api-version: 2
```

When the path stays the same, it is easy to handle. Client calls:

```
api/appointments/1
```

With the version in the header, the rest is handled on the backend side. However, when the path changes, for example:

```
api/appointments/1/status
```

Passing the information about the version will not be enough. It will also require changing the path on the client side to migrate to the newer version or handling complex routing in the backend.

Pros	Cons
The URL remains clean and free from versioning details.	Less obvious which version of the API is being used.
Separates the versioning from the resource identification.	Requires additional logic on the backend side to inspect headers and route requests appropriately.

Most often, you will encounter semantic versioning[3]. This way, you mark the version of the API as Major.Minor.Patch, e.g., 2.4.1. Patch is changed when you fix bugs in a backward-compatible manner. Minor is changed when you add the functionality that is backward compatible. Major introduces breaking changes, and clients have to migrate if they want to use a newer version.

Workflow

When executing a given action, all the calling code is wrapped inside an endpoint. Let's take the example of scheduling an appointment.

```
// PUT api/appointments/1/schedule
class ScheduleAppointmentEndpoint:
    properties:
        Persistence: AppointmentPersistence
    methods:
        Request(request: ScheduleAppointmentRequest):
            appointment = Persistence.Appointments.Get(request.Id)

            if null appointment:
                return 404

            existingAppointment.Schedule(request.ScheduledAt)

            Persistence.Appointments.Save()

            return 204
```

To keep the code simple, there are no additional services, managers, or command handlers:

1. The request comes to a *ScheduleAppointment* endpoint.

[3] https://semver.org/

2. It contains an existing appointment id (appointment is first created, then confirmed with one of the doctors, and then scheduled—that is why appointment with the given id should exist before scheduling it).
3. If appointment is not found, then we return `404: Not Found`.
4. Else, appointment is scheduled. Before it is scheduled, it validates the action against business rules.
5. Next, all changes are stored using the persistence that is defined for the entire module of *AppointmentScheduling*.
6. In the end, we return `204: No Content`.

Architecture tests

It is easy to break such a code structure. Therefore, it makes sense to ensure that it does not break over time and that the entire development team follows earlier-defined rules.

One popular method is to write unit tests that check various areas:

- Code from one folder (*DrugPrescription*) cannot be referenced by code inside another (*AppointmentScheduling*)
- Naming conventions
- Classes that reside in one namespace cannot depend on the service
- Interfaces should not contain the word “Interface” in their names
- Each class that implements a concrete interface should be sealed

In some languages, you can find ready-to-use solutions (like ArchUnit[4]); in others, you may need to develop them yourself using the unit test library. Here is an example of how such tests might look:

[4] https://www.archunit.org/

```
class AppointmentSchedulingTests:
  methods:
    ShouldPass_WhenNoDependencyOnDrugPrescription():
      Module("AppointmentScheduling")
            .ShouldNot()
            .DependOnModule("DrugPrescription")

    ShouldPass_WhenAllClassesThatImplementISchedulerAreSealed():
      Classes
            .FromModule("AppointmentScheduling")
            .ThatImplementInterface("IScheduler")
            .ShouldBeSealed()
```

According to the above example, if any part of the *Appointment Scheduling* module references *Drug Prescription*, architecture tests will fail. The same will happen if any class from the *Appointment Scheduling* module that implements the `IScheduler` interface is not marked as sealed.

This powerful mechanism will help you and your team keep the code structure clean, especially while having the entire production code in a single project.

What I dislike is the naming of such tests, as they check the structure of the code rather than the architecture itself. As you have learned from this book, software architecture is complex and touches many areas. That is why I think a more fitting name would be "Solution Structure Tests." :)

Database

Similar to how code is divided into modules, it is recommended to apply the same approach to the database. Starting from the beginning, it makes sense to create a separate connection string for each module, even though, at this moment, they will all point to the same place.

```
"ConnectionStrings": {
    "AppointmentScheduling": "Host=postgres;Database=healthcare;...",
    "DrugPrescription": "Host=postgres;Database=healthcare;...",
    "Invoicing": "Host=postgres;Database=healthcare;...",
    "MedicalRecordsMgmt": "Host=postgres;Database=healthcare;...",
    "PatientTreatment": "Host=postgres;Database=healthcare;..."
}
```

Note that if you want to use a different database for one of the modules, it is very simple to configure. You only need to change one of the connection strings, for example, *AppointmentScheduling*, and that's it—from then on, it will connect to a separate database.

When considering splitting data between modules, one of the first ideas might be to use separate databases. However, the chosen provider might offer a more straightforward approach. Since we selected Postgres as our relational database engine, we can divide the data logically into schemas.

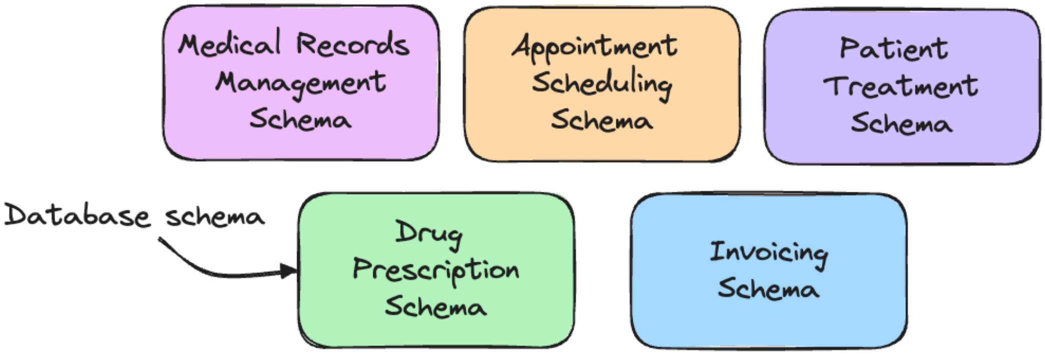


Figure 164. Each module has its own schema in a database

With this approach, each module data will be encapsulated in its own schema.

Communication

How do you communicate in the step that focuses on simplicity? As the name suggests, it should be simple!

There are three main ways to handle it:

- Synchronously through the public interface or through the gateway
- Asynchronously using in-memory queue
- Combination of both

The first option is to go fully synchronous. The entire modular monolith runs in the same process, so it makes sense to take advantage of that and define either a facade that acts as a public API on each module, or a gateway. Then call it from another module.

You can also communicate fully asynchronously using events and an in-memory queue. Since all modules will have access to the same memory, this is also an acceptable choice. This approach builds a good foundation for the future if you decide to introduce an external component like a message broker.

The pragmatic approach uses a combination of both methods—synchronous and asynchronous. In some cases, it would make the most sense to communicate via events, while in other places, it is better to call another module synchronously.

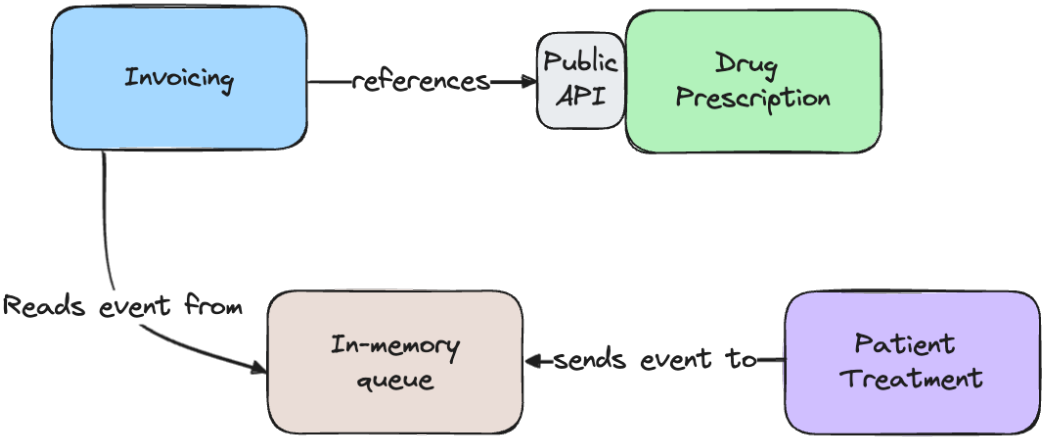


Figure 165. Example of communication in our application

For example, when patient treatment is completed, the *Patient Treatment* module publishes an event *Treatment completed* to the in-memory queue. The *Invoicing* module reads it and starts preparing the invoice for the patient. Since it also needs to get information about prescribed drugs to charge the patient, it calls the *Drug Prescription* module through its public API.

> What about inbox and outbox patterns? You can implement them and ensure that events are delivered to other modules. However, you might also accept the fact that even in the case of failure, you can manually fix the issue. If my application faces dozens of such failures in a day, then I would prefer to implement these patterns. If it fails once every two weeks, then I am fine with handling it manually.

Architecture Decision Log

I recommend documenting architecture decisions from day one of the implementation. This approach helps you to understand why a decision was made, its pros and cons, and what alternatives were considered.

Since the beginning of my programming career, writing documentation has been my weakest point. I am not particularly fond of documentation, as it usually gets obsolete faster than you can blink.

I remember having to describe architecture in a Word document and updating it every time it changed. It included everything from diagrams to concept descriptions and comments from others. By the time I finished it, it was already outdated.

Over time, I tried a multitude of alternatives, and the result was always the same—a lot of unstructured knowledge. Someone had to go through all the content just to find the relevant information obsolete.

Unstructured and obsolete information are two of the most significant issues in knowledge management.

Another problem is finding that component X or pattern Y is present but makes no sense in your application.

> *Who the heck decided on this? It makes no sense!*

Maybe it was a mistake, or maybe it solved a problem at the time it was added. However, you have no context other than the repository history.

This is where the *Architecture Decision Log (ADL)* can help.

It records key architectural decisions made during the development of a software project and explains why certain choices were made and when. The best part is, you don't need extra tools to create it.

How do you create it?

I recommend keeping it as close to the code as possible. A good idea is to create a folder next to your application (which can be in the root folder) named, e.g., `architecture-decision-log`.

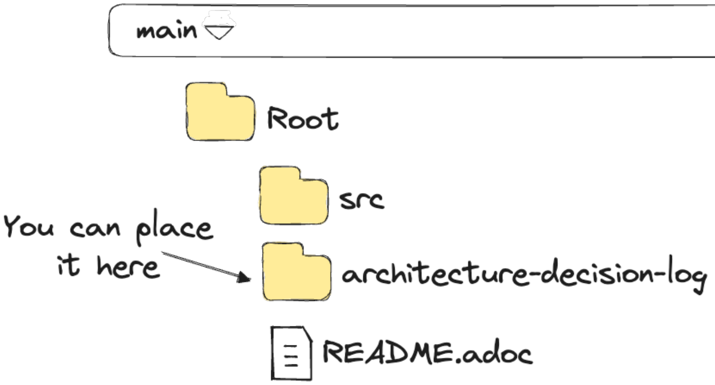


Figure 166. Architecture decision log stored as a part of the repository

When working with multiple repositories, for example, in a distributed system, it is a good idea to split the ADL by repository and its unit (e.g., microservice). That is because your microservices might be:

- Implemented in various technologies.
- Using different components—one can use a non-relational database (MongoDB), while another can use a relational database (MySQL).
- Based on different architectural patterns.

Sometimes, I am asked where to put ADL when there are shared decisions for all parts of the system, like the one that claims that all services should run in

Kubernetes. What you can do is create a repository that is the entry point to your other repositories. There, you can keep the general information about the application as well as the ADL for shared decisions. However, a trade-off of this approach is that someone has to monitor it as such repositories can quickly become outdated.

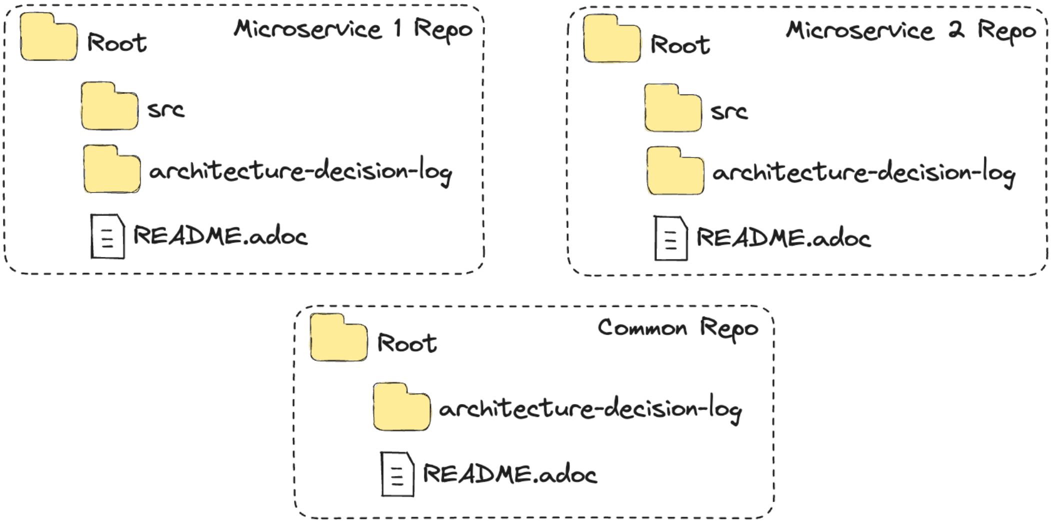


Figure 167. Architecture decision log per microservice plus one shared repository

Next, inside this folder, you will be adding a new *Architecture Decision Record (ADR)* each time you decide on anything related to architecture:

- Adding, modifying, or replacing infrastructure components.
- Deciding on another architectural pattern.
- Using another logical pattern.

And so on.

Each record is stored in a separate file (in markdown, adoc, txt, or any other format). It is an excellent idea to number the files starting from 0001 because it will keep the order when sorted alphabetically.

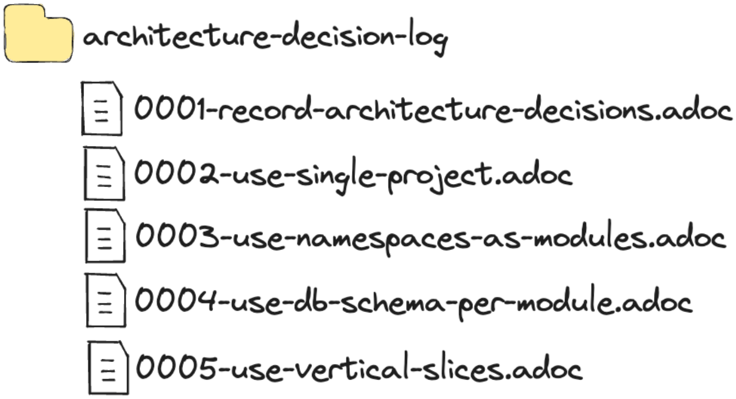

architecture-decision-log
0001-record-architecture-decisions.adoc
0002-use-single-project.adoc
0003-use-namespaces-as-modules.adoc
0004-use-db-schema-per-module.adoc
0005-use-vertical-slices.adoc

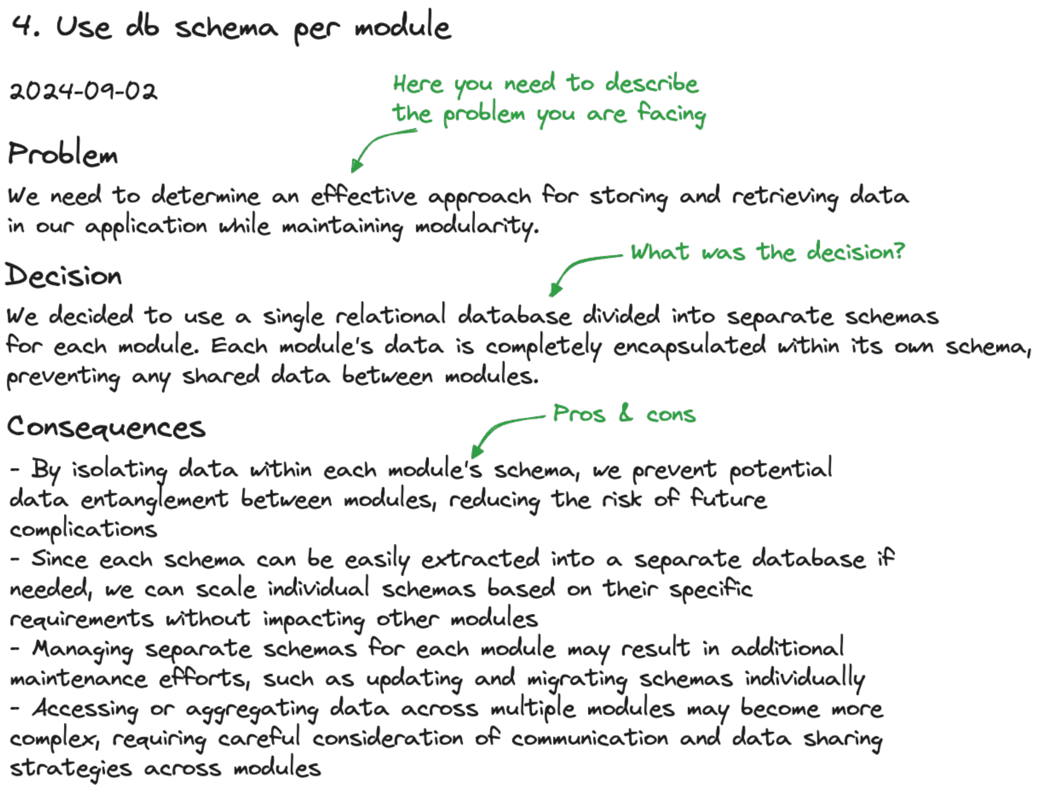
Figure 168. Examples of architecture decisions records

Each record is immutable. If a decision needs to be changed, add another record with +1 numeration. This way, you can structure the log, and anyone reading it will know it is ordered according to the timeline.

Each record contains a complete description of the architectural decision:

- **Title**. It should be clear and concise as it simplifies the search process, making it easier to find specific decisions quickly.
- **Date**. The date when the decision was made, providing a timeline.
- **Problem Description**. A detailed explanation of the issue that triggered the need for the architectural decision.
- **Decision**. A clear statement of the decision made by the team to address the problem.
- **Consequences**. An outline of the pros and cons resulting from the decision. This helps in understanding the implications and trade-offs involved.

Here is an example of the ADR for splitting a database for modularization using schemas:



4. Use db schema per module

2024-09-02

Here you need to describe the problem you are facing

Problem

We need to determine an effective approach for storing and retrieving data in our application while maintaining modularity.

What was the decision?

Decision

We decided to use a single relational database divided into separate schemas for each module. Each module's data is completely encapsulated within its own schema, preventing any shared data between modules.

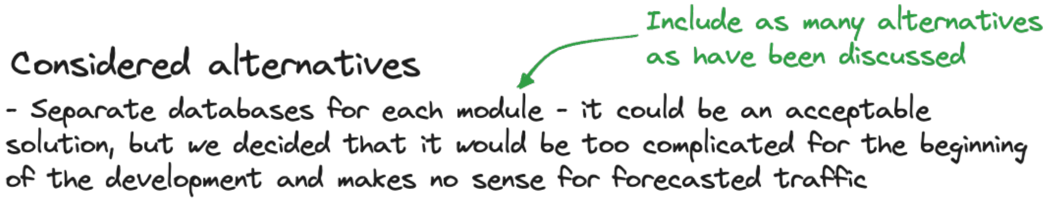
Pros & cons

Consequences

- By isolating data within each module's schema, we prevent potential data entanglement between modules, reducing the risk of future complications
- Since each schema can be easily extracted into a separate database if needed, we can scale individual schemas based on their specific requirements without impacting other modules
- Managing separate schemas for each module may result in additional maintenance efforts, such as updating and migrating schemas individually
- Accessing or aggregating data across multiple modules may become more complex, requiring careful consideration of communication and data sharing strategies across modules

Figure 169. Example content of selected architecture decision record

It is also a good idea to include an additional field called "Considered alternatives." In this field, you can describe other options that were discussed and explain why they were not selected.



Include as many alternatives as have been discussed

Considered alternatives

- Separate databases for each module - it could be an acceptable solution, but we decided that it would be too complicated for the beginning of the development and makes no sense for forecasted traffic

Figure 170. Example content of considered alternatives

This information can help in the future to understand not just why the decision was made but also why other options didn't work.

The cool thing about ADRs is that they can be stored in the repository next to your code. You don't need to search for them in Jira, Confluence, Sharepoint, or anywhere else—just you and your repository, nothing in between.

When you implement a new feature that requires applying a pattern or changing a component, I recommend adding a new ADR in the same commit that applies the change.

At some point, it becomes natural for everyone to think about ADRs whenever there is a new architectural decision. This helps keep your architecture documentation clear, structured, and up-to-date.

In this log, you can record all small and large decisions. Once you see how useful it is, you won't want to stop it—I guarantee that! :)

Result

Area	Decision
Deployment strategy	Single unit (Modular monolith)
Production code projects	1
Code structure	Variation of Vertical Slice Architecture
Communication	Synchronous using public API (via facade, no HTTP) and asynchronous using in-memory queue
Database	Single, modules represented by schemas

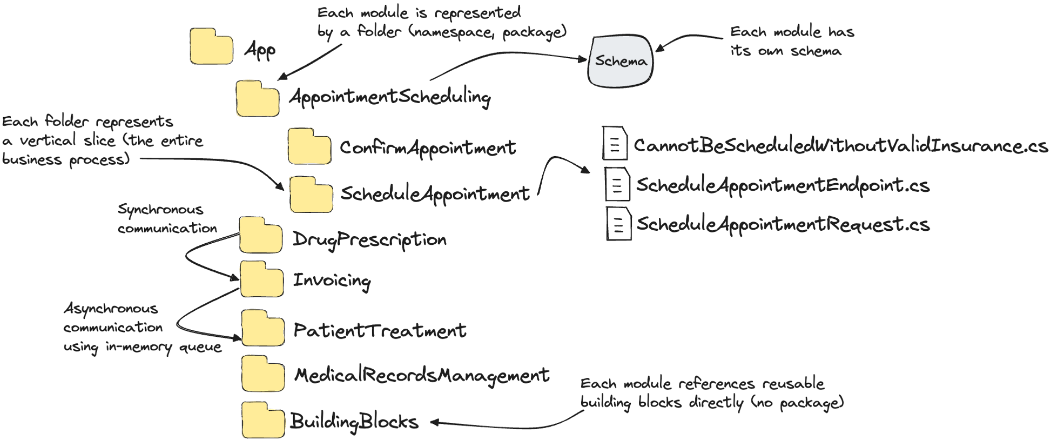

Each module is represented by a folder (namespace, package)
App
AppointmentScheduling
Schema
Each module has its own schema
Each folder represents a vertical slice (the entire business process)
ConfirmAppointment
ScheduleAppointment
CannotBeScheduledWithoutValidInsurance.cs
ScheduleAppointmentEndpoint.cs
ScheduleAppointmentRequest.cs
Synchronous communication
DrugPrescription
Invoicing
Asynchronous communication using in-memory queue
PatientTreatment
MedicalRecordsManagement
Each module references reusable building blocks directly (no package)
BuildingBlocks

Figure 171. The current state of the application

Second: Focus on maintainability

Over time, and thanks to the MVP release, the initial assumptions were validated:

- Some modules have grown rapidly as many new features were added, while others have remained untouched since the MVP phase.
- Two modules are used more frequently than others.
- A module we initially considered simple has become complicated.

Our application now serves an increasing number of patients and clinics.

Additionally, the team has grown to 14 people, making it quite difficult to work on a single project due to conflicts and merging issues. More candidates are in the pipeline.

We are slowly coming up against a brick wall. We must react.

This step requires product validation with the market. We learn from user behaviors, application maintenance, bug fixing, and other activities that can only be observed with real traffic.

What changed?

Compared to the previous step, the environment around our application has changed in the following areas:

1. Our customer base has grown from 3-5 clinics to 20 and continues to increase, with three new clinics added per month. In total, these 20 clinics hire 500 employees, each generating 1,000 requests per day. It translates to `500 * 1,000 * 20 days = 10,000,000` requests per month.
2. The application now serves 10,000 patients, around 7-10 times more than expected (1,000-1,500). Each patient creates around 300 monthly requests. This means all patients generate around $10,000 \times 300 = 3,000,000$ requests per month.
3. The entire traffic we handle is $3,000,000 + 10,000,000 = 13,000,000$ monthly requests. Assuming that the application will be used 12 hours a day, it gives $20 days * 12 hours * 60 mins * 60 seconds = 864,000$ seconds. It translates to $\frac{13,000,000}{864,000} \approx 15$ requests per second. We can still handle it with a single instance, as each request takes around 50 milliseconds to finish.
4. *Patient Treatment* module has become quite complex, and we keep adding new functionality.
5. *Drug Prescription* module is used more often than others.
6. Since the MVP release, we have added ten new features.
7. We can observe that some modules are CRUDish while others are complicated.
8. The new requirement is to enable reporting for several modules.

CRUD

it represents four basic operations: *Create*, *Read*, *Update*, and *Delete*. Applications of this type are usually characterized by uncomplicated business logic or, in extreme cases, its complete absence.

Let’s look at architectural drivers, as they will form the basis for our subsequent decisions.

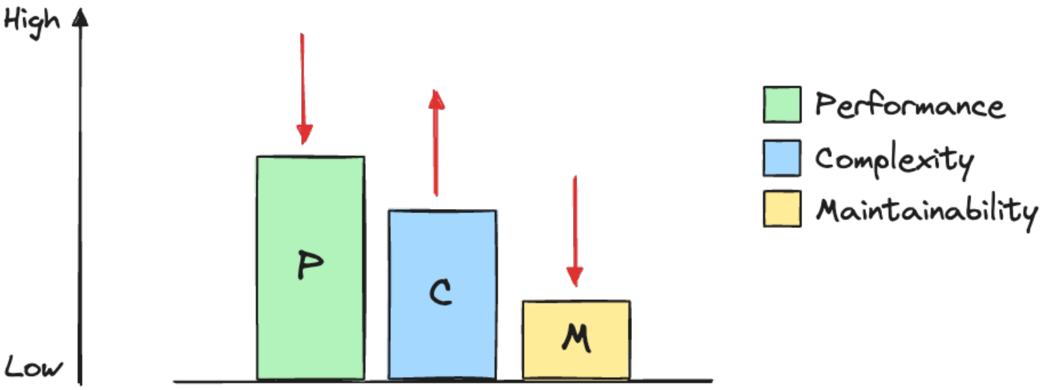


Figure 172. The current state of architectural drivers

Compared to the previous step:

- **Performance decreased.** This is due to the increased usage of the application. The number of patients and employees increased, generating around seven times more requests than at the beginning.
- **Complexity increased.** Due to new features and extension of existing business logic, the total complexity of the solution increased.
- **Maintainability decreased.** The maintainability was significantly affected as more people worked on the same code base wrapped in a single project and added new features. This led to problems with synchronization, merging, and bug-fixing.

The first two problems are already affecting us, especially performance, but the major problem is related to maintainability. If we do not improve it, we cannot effectively provide new functionality and fix bugs.

We will mainly focus on this driver in this step.

Maintainability: Problems to address

While looking for solutions to maintainability issues, we must be cautious to ensure that our decisions do not negatively impact other drivers. Although it is unlikely that they won't be affected at all, we can still try to minimize the impact.

The most challenging problem we face is having 14 developers working on a single project. Even though we try to split the work, avoiding all problems while extending and adding new features is impossible. When someone updates the version of one of the third-party packages, others are affected, so all teams have to plan and agree on such changes. We also observe communication issues—too many people cannot communicate effectively.

Another problem is that due to heavy traffic, our application generates quite a lot of events. When an event is lost, for example, in the case of an application failure, we must ensure that the state is fixed. The more events generated, the higher the impact of the failure. Previously, when the traffic was low, we could address it manually, but it has slowed down the development team.

In the MVP phase, we applied the same patterns to all modules. Over time, we gathered more domain knowledge and observed how each module changed and behaved. In the *Invoicing* module, we keep adding new types of invoice exporters. Besides PDF, there is JSON, XML, CSV, and docx, and the codebase is growing. There is also a new requirement to address reporting for several modules, and we can check if we can apply better-fitting patterns there.

Furthermore, the amount of code in endpoint classes has increased, making it harder to understand what is happening there. Addressing this issue would be a good idea as well.

Problem	**Category**
All code changes applied in the same project	Code structure
Same patterns applied everywhere	Code structure
Too much code in endpoint classes	Code structure
Problems with communication inside the team	Team structure
Planning changes in code requires much work	Team structure
In case of application failure, we lose a lot of events	Communication

Code structure

Having the entire production code in a single project can become problematic at some point. Usually, this happens sooner rather than later, even if the code is split into modules wrapped into folders.

That is why it makes sense to start dividing it into multiple projects. These projects can be maintained separately, reducing the risk of accidental changes by other teams. However, it is still a monolithic structure and requires some attention from the teams that maintain it.

Over time, we observed how modules behave, so we can now focus on patterns that better fit some of them. For example, in a complex module like *Patient Treatment*, looking at *Command Query Responsibility Segregation (CQRS)* could be helpful. We can use *Transaction Script* in the reporting module.

> For a greenfield application being released for the first time, it makes little sense to decide on various patterns while knowing little about numbers, user behaviors, or how the module will evolve. That is why I recommend observing all these factors before deciding on specific patterns.

Multiple projects

When your team experiences difficulties maintaining a module or modules due to them being part of a single project, it may be time to start dividing them into separate projects. The challenge could come from a single problematic module, several modules, or from maintaining each module individually.

There is no single, perfect solution for this kind of issues. You must evaluate the situation and decide on the best course of action. If it makes sense, check how much it will cost and compare it with the available budget.

I recommend handling migration to separate projects step-by-step, precisely as in the case of legacy system refactoring. Start with the most problematic one, extract it, and move forward. If you take enough care while starting the application's development and modularize it correctly, such extractions usually take a few days of work for two people.

Previously, all modules were represented as folders inside a single project. In each folder, we placed code related to each business process (vertical slice). There are two candidates to extract:

1. *Patient Treatment* module that gets more and more complex.
2. *Drug Prescription* module that is used most often.

Both have a high chance of becoming separate deployment units soon. We see the potential, but we do not want to increase the complexity of the entire solution that much yet.

Since we have to add another module for reporting, it also makes sense to add it as a separate project (we will eventually migrate all existing modules to separate projects).

We decided to start by extracting the *Patient Treatment* and *Drug Prescription* modules. We also decided to keep vertical slices there (no changes). Both projects are referenced from the main one as it is responsible for all modules registration and run.

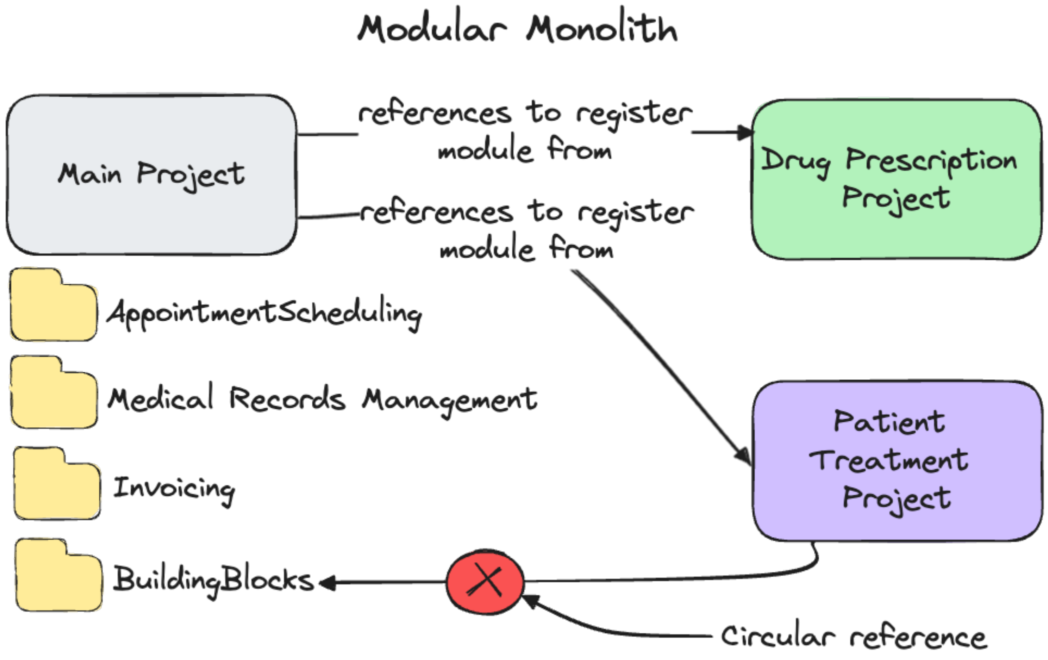

Modular Monolith
Main Project
references to register module from
references to register module from
Drug Prescription Project
AppointmentScheduling
Medical Records Management
Invoicing
BuildingBlocks
Patient Treatment Project
Circular reference

Figure 173. Extraction of two projects that are still a part of modular monolith

We rebuilt all projects and encountered errors. Since we extracted both modules, we cannot access building blocks (circular reference), so we cannot use mechanisms from there. Before, this was possible because all modules were part of a single project.

The simplest solution is to extract the building blocks to a separate project and reference it in all the others—*Main*, *Drug Prescription*, and *Patient Treatment*.

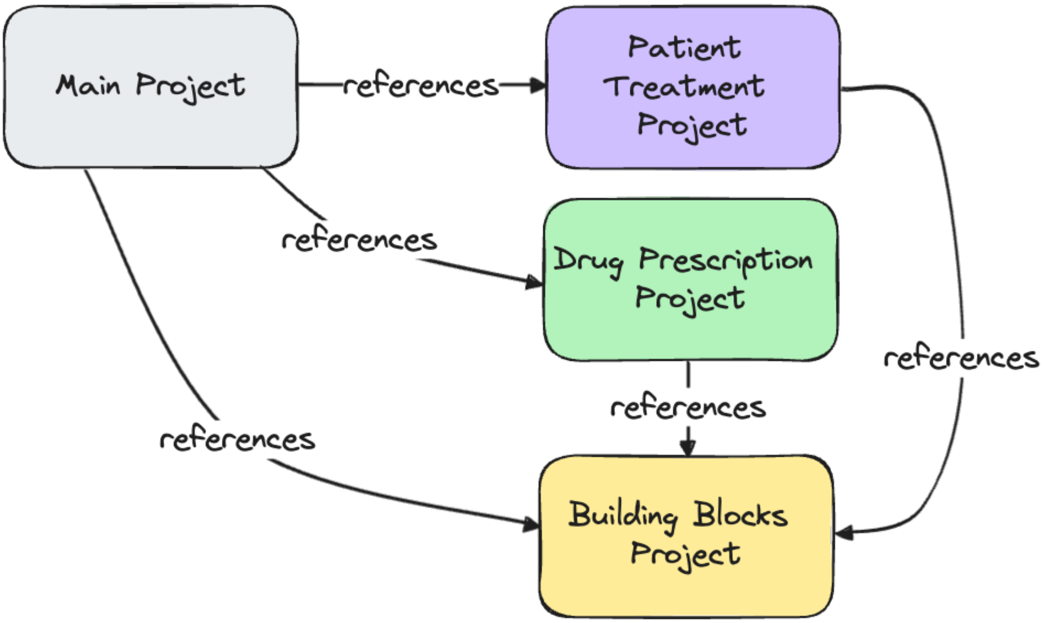


Figure 174. Extraction of building blocks to a separate project

From now on, whenever *Main* starts, it registers all modules (those still in *Main* and those extracted to separate projects) and starts the entire modular monolith. With this approach, all modules still run in a single process.

When dividing into separate projects, you may be tempted to split their structure into several layers. However, you should think twice before doing it—only do it if necessary.

In this case, there are usually four layers:

- **API**. Contains endpoints, request objects, request validators, and other API-related code.
- **Infrastructure**. Contains infrastructure-related code such as message bus implementation or repositories.
- **Application**. Contains commands, queries, handlers, repository interfaces, and more.
- **Core/Domain**. Contains business logic and business rules. If you decide for tactical Domain-Driven Design, it will be represented by aggregates, entities, value objects, domain services, and domain events.

This approach is often a wrong decision. It is counterproductive to enforce this structure across all modules, particularly for those with simple, straightforward logic. Be pragmatic—always ensure you have a sound rationale for adopting such a setup. The modules should be sufficiently complex so that the migration brings benefits rather than additional complications.

Next, we have to think about handling reports. Our customers want to know the number of appointments scheduled in the last month (based on *Appointment Scheduling* module data) and the number of patients examined in a given year (*Medical Records Management* module data).

We have three options:

- Use a third-party solution.
- Add reports in each module.
- Add another “module” called *Reporting*.

After further investigation, we decided to use a new module and create a separate project. If there are more reports, we will migrate to the third-party solution (it is not worth using now because of its high pricing).

There is only one validation check when executing reports—the user must have an administrator role. Someone proposed applying a transaction script to the new module to simplify the flow.

Transaction Script

You might have read that the transaction script is considered an anti-pattern because it operates on an anemic model or becomes messy when more complex business logic is involved.

I wouldn't call it an anti-pattern, but it should be applied where it can help. As always, it depends.

Let's have a look at the definition[5] proposed by Martin Fowler:

> *A Transaction Script organizes all this logic primarily as a single procedure, making calls directly to the database or through a thin database wrapper. Each transaction will have its own Transaction Script, although common subtasks can be broken into subprocedures.*
>
> —Martin Fowler, *Transaction Script*

Transaction scripts are a perfect fit for CRUD applications where there is almost no business logic or where the logic is very simple. You do not need to introduce any layers and can call the database directly (executing scripts) or through a database wrapper (e.g., ORM).

Here is an example flow of a transaction script:

```
// We read the data from the database using a thin DB layer
confirmedAppointment = getConfirmedAppointmentForPatient(patient.Id)

// When a confirmed appointment does not exist, it cannot be scheduled
if null confirmedAppointment:
    return 404

// Schedule an appointment using a thin DB layer
scheduleAppointmentForPatient(patient.Id)
```

This way, you can write simple code that will be enough. However, there is a chance that the validation checks and the business logic will be more complex.

[5] https://martinfowler.com/eaaCatalog/transactionScript.html

```
// We read the data from the database using a thin DB layer
confirmedAppointment = getConfirmedAppointmentForPatient(patient.Id)

// When a confirmed appointment does not exist, it cannot be scheduled
if null confirmedAppointment:
    return 404

if isInThePast confirmedAppointment:
    return 409

if isAlreadyScheduled confirmedAppointment:
    return 409

...5 more checks

// Check if the patient has valid insurance
hasPatientValidInsurance(patient.Id)

...3 more actions

// Schedule an appointment using a thin DB layer
scheduleAppointmentForPatient(patient.Id)
```

As complexity increases, transaction script face several drawbacks: they become harder to read, more difficult to maintain, and present significant testing challenges. In this case, looking for a better-fitting pattern would make sense.

For the reporting module, we do not want to use any layers and will call the database directly from the endpoint.

```
// PUT api/appointments/1/schedule
class GetScheduledAppointmentsInTheLastMonthReportEndpoint:
    properties:
        database: Database
    methods:
        Get(customerId: number):
            // Check if the current user has an appropriate role
            if not Administrator context.User.Role
            return 400

            // Prepare parameterized query
            query = $@"SELECT..."

            results = database.Execute(query)

            return 200 (results.ToMapped())
```

The same approach can be applied to other reports.

CQRS

Command Query Responsibility Segregation (CQRS) allows you to divide a single model that is responsible for both write and read operations into two separate ones:

- **Write**. Represented by commands, which instruct your system to perform operations, thereby changing the system state.
- **Read**. Represented by queries, which retrieve data from your system without changing its state.

There is a common misconception that CQRS requires two databases—one for read operations and another for write operations. However, this is not true. Using two databases can be a step in the system's evolution, but it is not required.

Let's first look at the simple explanation[6] from Greg Young:

[6] https://web.archive.org/web/20211124134459/http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/

> *CQRS is simply the creation of two objects where there was previously only one. The separation occurs based upon whether the methods are a command or a query (the same definition that is used by Meyer in Command and Query Separation, a command is any method that mutates state and a query is any method that returns a value).*
>
> —Greg Young, *CQRS, Task Based UIs, Event Sourcing agh!*

Did you notice that the definition does not mention a database, separate tables, or separate databases? That's because it does not matter if you store the data in:

- A single database
- Read and write databases
- Relational or non-relational databases
- Text file(s)
- Excel

You can store it anywhere you want—it is just a piece of information that you are going to modify (command) or read (query).

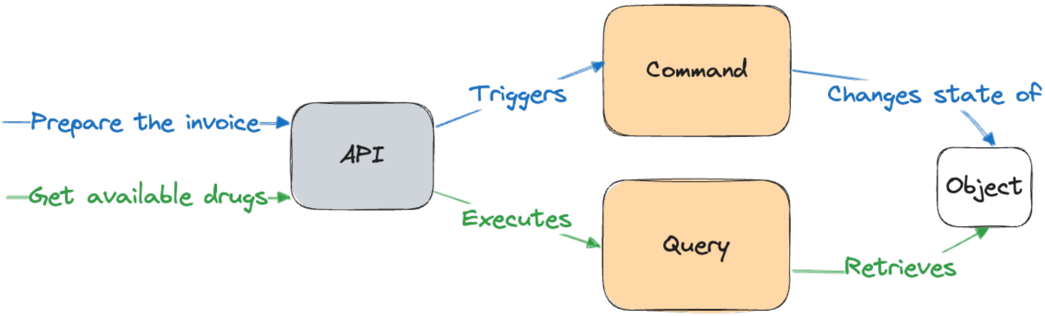


Figure 175. Division into command and query in CQRS

Write operations will usually be dependent on various validation checks. Here are some examples:

- **Schedule an appointment.** An appointment cannot be scheduled if the patient has no valid insurance.

- **Select drug**. A drug cannot be selected if there are any contraindications for the patient.
- **Prepare invoice**. An invoice cannot be prepared if the treatment has not yet been completed.

As there are endpoints that handle heavy logic in *Patient Treatment*, it makes sense to extract the logic from there and use commands for write operations and queries for read operations. This is the current state of treatment plan preparation:

```
class PrepareTreatmentPlanEndpoint:
    properties:
        Persistence: PatientTreatmentPersistence
    methods:
        Prepare(request: PrepareTreatmentPlanRequest):
            treatmentPlan = Persistence.TreatmentPlans.Get(request.Id)

            if not null treatmentPlan:
            return 409

            ...some other checks

            treatmentPlan.Prepare(request.PlanDetails)

            Persistence.TreatmentPlans.Save()

            return 204
```

We can create a new command and a handler that will handle it:

```
class PrepareTreatmentPlanCommand:
    properties:
        Id: PatientTreatmentPersistence
        PlanDetails: PatientTreatmentPlanDetails

class PrepareTreatmentPlanHandler:
    properties:
        Persistence: PatientTreatmentPersistence
    methods:
        Handle(command: PrepareTreatmentPlanCommand):
            treatmentPlan = Persistence.TreatmentPlans.Get(command.Id)

            if not null treatmentPlan:
            return 409

            ...some other checks

            treatmentPlan.Prepare(command.PlanDetails)

            Persistence.TreatmentPlans.Save()
```

Next, we trigger the command from the API:

```
class PrepareTreatmentPlanEndpoint:
    properties:
        executor: CommandQueryExecutor
    methods:
        Prepare(request: PrepareTreatmentPlanRequest):
            executor.ExecuteCommand(request.ToCommand())

            return 204
```

The handler then handles that. Let’s visualize it:

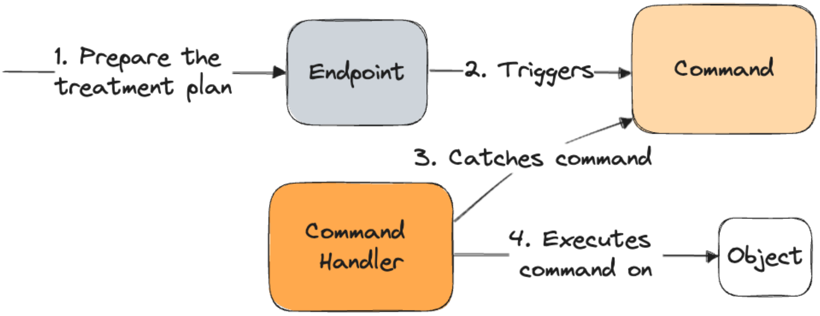


Figure 176. Handling preparation of the treatment plan

A similar approach can be used when querying the system. Imagine that there is the following operation to retrieve the patient's treatment plan:

```
class GetPatientTreatmentPlanEndpoint:
    properties:
        executor: CommandQueryExecutor
    methods:
        Get(request: GetPatientTreatmentPlanRequest):
            treatmentPlan = executor.ExecuteQuery(request.ToQuery())

            return 200 (treatmentPlan)
```

Then, the query is handled by handler.

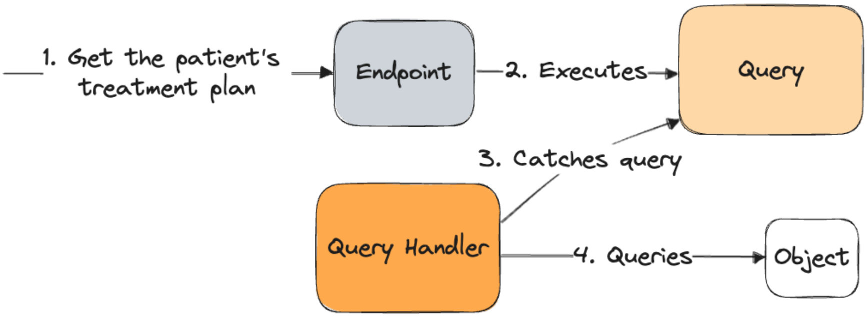


Figure 177. Retrieval of patient's treatment plan

Moreover, the division between writing and reading models allows you to optimize each. Perhaps the best way to handle writes is to use ORM, but using a plain SQL query to retrieve the data while querying could be faster?

Team structure

The problem of having a team that is too large has to be solved. We started with a small team, and the development was pretty straightforward. There were some synchronization issues and occasional merging conflicts, but in general, everything worked as expected. Over time, it grew to 14 people—too many for a single team.

First, we started observing communication issues. Someone would discuss something with one person but fail to inform another. As a result, people were unaware of the changes, and we missed the knowledge from specific areas. To illustrate the problem, let's look at the picture below. It shows how the number of communication lines increases as the team grows[7].

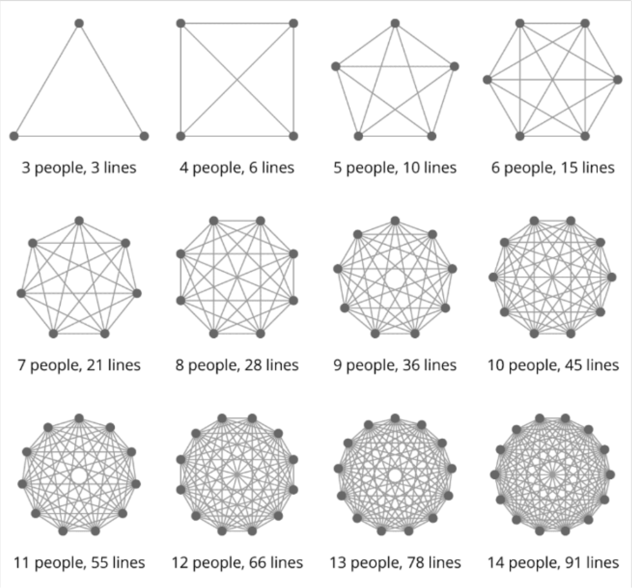


Figure 178. Number of communication lines in a team

[7] https://getlighthouse.com/blog/developing-leaders-team-grows-big/

There were four developers at the beginning of the development, and ensuring that all the relevant information was shared between everyone (there were only six communication lines) was quite simple. Over time, it got more complex, with 91 communication lines (14 people), making it impossible to ensure that everyone was equally informed.

If we were to base the split solely on communication lines, the ideal setup would be four teams:

- Team A: 3 people.
- Team B: 3 people.
- Team C: 4 people.
- Team D: 4 people.

But before we make the split, there are at least three more questions we need to answer:

1. **How can we split the application between teams?** One method is to assign a team to a selected bounded context(s). This way, the team is responsible for all business processes within that context. Our application is divided into modules that are 1:1 mapped with bounded contexts, meaning we can assign them to each team.
2. **What is the competency matrix in each team?** Similar to what we did before the start of the development, we can use the Competency Matrix to describe the strengths and weaknesses of the entire (14-person) team. This helps us split people into balanced teams. At this point, it is impossible to create four teams because of missing competencies. It makes more sense to have three teams of 5, 5, and 4 people.
3. **How will all teams communicate and synchronize?** Even if the application and teams are divided based on bounded contexts, they will still need to communicate with each other. They will work on one code base (still a modular monolith) using shared code (like middleware). Additionally, a team assigned to one of the bounded contexts might need information from other contexts or be triggered by them. Using Context Map is beneficial here.

After reviewing the competency matrix and considering all the information, we decided to split our team into three groups.

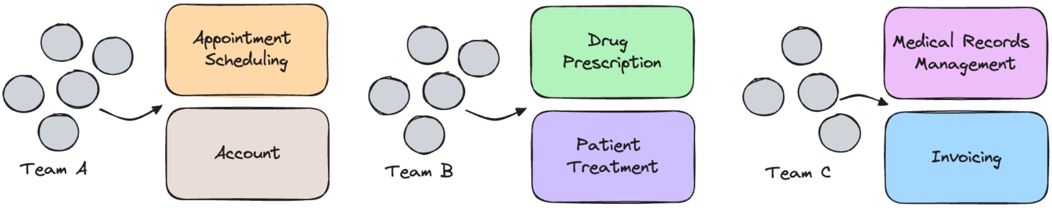


Figure 179. Teams structure after split

- **Team A (5 people)**. Responsible for *Appointment Scheduling* and everything that will be related to appointments in the future. Additionally, they will maintain account registration and login processes (*Account*).
- **Team B (5 people)**. Responsible for *Patient Treatment* and *Drug Prescription*. We assigned these two modules to this team because they often communicate with each other.
- **Team C (4 people)**. Responsible for *Medical Records Management* and *Invoicing*. These modules are not related, but they are both relatively small, so we decided to assign them to one team.

This division is not permanent and is bound to change. However, this is not a cause for concern. The important thing is to react every time we see a problem. Over time, boundaries will no longer change so much, which will also contribute to the stability of the teams.

Communication

All synchronous communication is still handled in the same way as before.

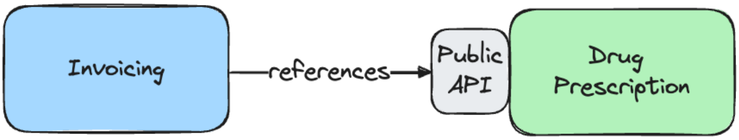


Figure 180. Synchronous communication in our application

In an asynchronous case, you must decide whether to migrate to an external component or keep the communication via an in-memory queue. The first makes sense when:

1. The application starts experiencing an increased load and cannot efficiently handle the volume of messages.
2. Managing the in-memory queue is becoming complex and time-consuming. If there are many complaints, it might be the perfect time to consider an external component.

Whether you decide to keep using an in-memory queue or message broker, it is a good idea to consider using outbox and inbox patterns.

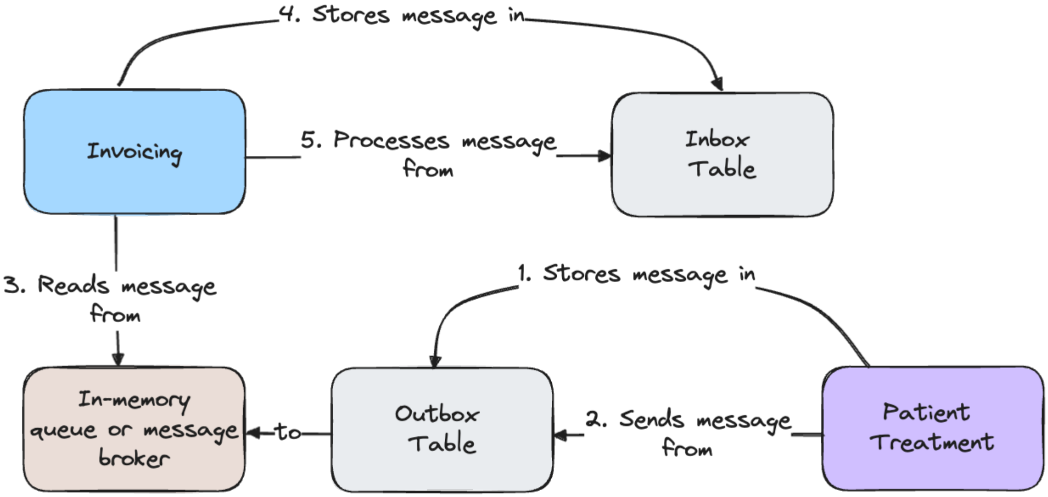


Figure 181. Asynchronous communication in our application using outbox and inbox patterns

In the previous step (*Simplicity*), I mentioned that having an external component was optional because the application's traffic was low. Now, we face 7-10 times more traffic than in the beginning, with more expected in the future. Minimizing the risk of losing messages is a really good idea, as manually checking and fixing issues each time would be too time-consuming.

Please note that adding an external component or outbox and inbox patterns will increase the overall complexity of the solution. You will need to deal with network issues and extra application logic. However, using outbox and inbox patterns improves the application's maintainability by eliminating the need to manually handle each lost event.

Result

We made the solution easier to maintain, but it also became more complex. We now have to manage several projects, including building blocks project. We used patterns like CQRS and transaction script in some parts. We have different teams handling specific modules, which has improved communication and sped up development.

Because of these changes, the entry threshold is unfortunately higher than before.

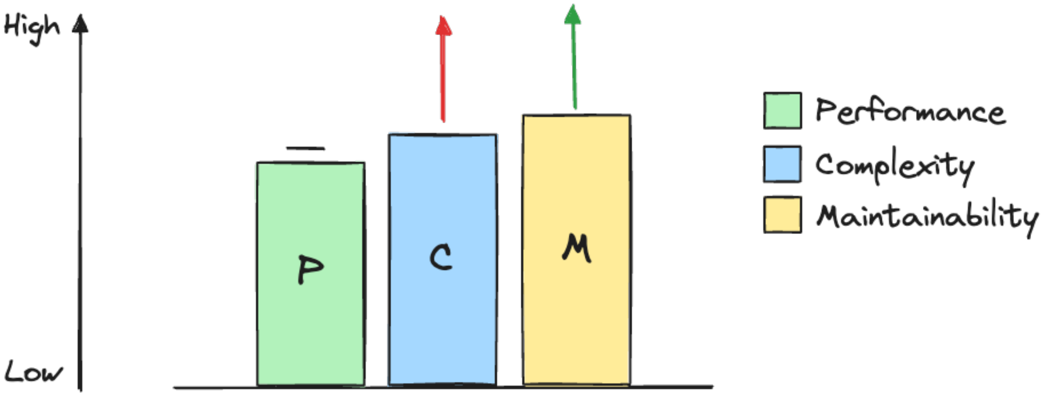


Figure 182. Architectural drivers after focusing on maintainability

Area	Decision
Production code projects	5 (main + 2 modules + Building blocks + Reporting)
Code structure	CQRS in *Patient Treatment* module, transaction script in *Reporting*

Area	Decision
Communication	Synchronous (as before) and asynchronous using an in-memory queue, inbox, and outbox patterns

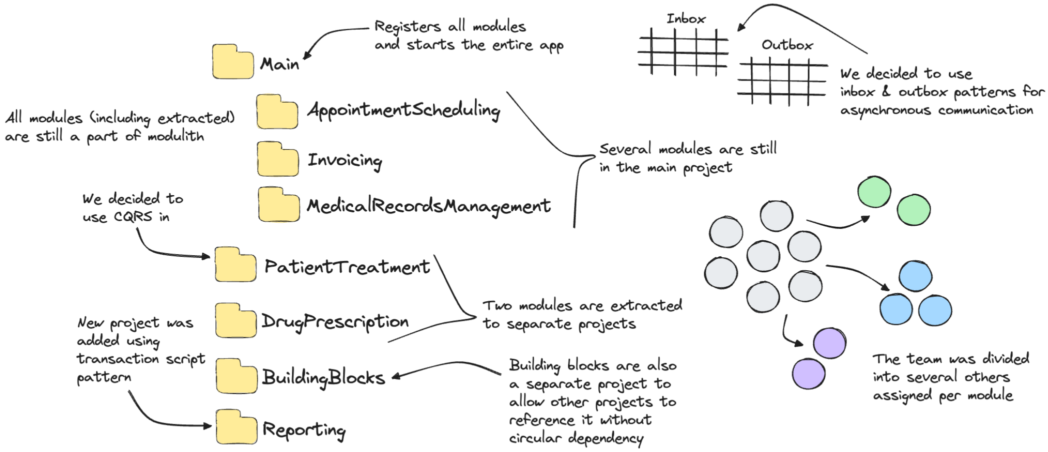

Registers all modules and starts the entire app
Main
Inbox
Outbox
We decided to use inbox & outbox patterns for asynchronous communication
All modules (including extracted) are still a part of modulith
AppointmentScheduling
Invoicing
MedicalRecordsManagement
Several modules are still in the main project
We decided to use CQRS in
PatientTreatment
DrugPrescription
Two modules are extracted to separate projects
New project was added using transaction script pattern
BuildingBlocks
Building blocks are also a separate project to allow other projects to reference it without circular dependency
Reporting
The team was divided into several others assigned per module

Figure 183. The current state of the application

Third: Focus on growth

After several months, our application has become the most popular for private medical clinics in Europe. We currently have almost half of the market and plan to grow further. We aim to reach 60% in the upcoming year, a target we are confident we can achieve.

However, we are beginning to have serious problems. Customers are starting to complain about the performance of some modules, and the cost of scaling is overwhelming—we are spending more than 15% of our revenue on infrastructure.

What changed?

Compared to the previous step, the environment around our application has changed in the following areas:

1. Our customer base grew from 20 clinics to 300, with 30-35 new clinics joining each month. These 300 clinics hire 30,000 employees who generate 1,000 requests per day each. It translates to $30,000 * 1,000 * 20days = 600,000,000$ requests per month.
2. The application now serves 100,000 patients, ten times more than before. Each patient creates around 300 requests per month. This means all patients generate around $100,000 \times 300 = 30,000,000$ requests per month.
3. The entire traffic that we handle is $30,000,000 + 600,000,000 = 630,000,000$ requests per month. Assuming that the application will be used 12 hours a day, it gives $20days * 12hours * 60mins * 60seconds = 864,000$ seconds. It translates to $\frac{630{,}000{,}000}{864{,}000} \approx 730$ requests per second.
4. The *Patient Treatment* module generates 40% of the traffic.
5. The *Patient Treatment* module is also the most frequently updated.
6. Due to the extensive work required for the *Patient Treatment* module, we have decided to build another team to support the existing one.

Let's look at architectural drivers:

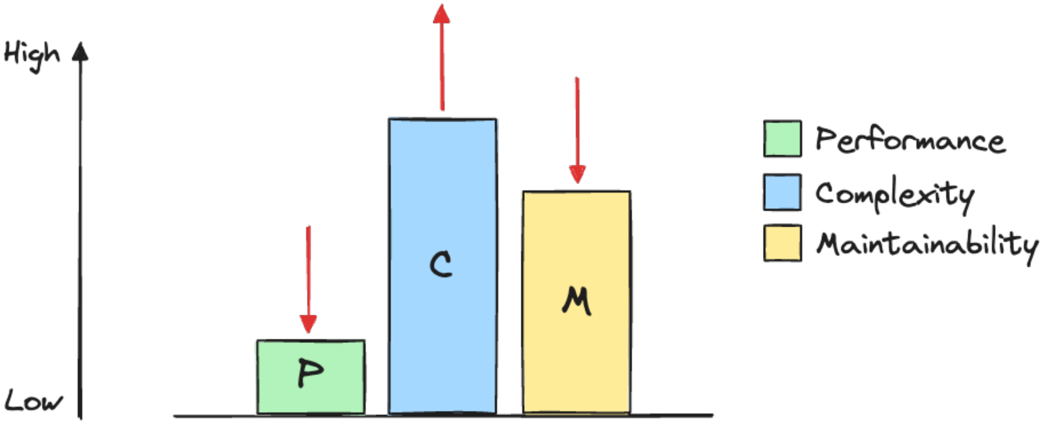


Figure 184. The current state of architectural drivers

Compared to the previous step:

- **Performance decreased.** It was significantly affected by almost 50 times greater traffic than in the previous step. Additionally, more complex business logic increased the response time to each request.

- **Complexity increased**. Due to new features and the extension of existing business logic, the total complexity of the solution increased, but not to the level we should be really worried.
- **Maintainability decreased**. With more features than ever and the increased complexity of the solution, the level of maintainability is not as satisfactory as it should be.

This time, the major problem is the solution's performance. It is poor, and many customers complain about it. If we don't improve it, our customers will search for alternatives.

We will focus on this driver in this step.

Growth: Problems to address

The most critical problem is the performance. There are over 130,000 users generating a lot of traffic, and we must address this issue immediately.

In optimizing performance, we must balance improvements with simplicity. Our goal is to enhance speed without sacrificing maintainability, which is crucial for our team's long-term efficiency.

It would also be good to do something with the *Patient Treatment* module, which is responsible for handling 40% of all requests to the application and is changed more often than any other module. We scale our application mainly because of it.

Problem	Category
Slow application	Performance
Scaling the entire application to handle traffic from one module	Deployment strategy
Every time *Patient Treatment* module code is touched, it requires redeployment of the entire app	Code structure

Scaling

Our application has gained a lot of traction, with over 130,000 users generating heavy traffic—around 730 requests per second. We have been continuously scaling our modular monolith, but the costs are rising, and it takes increasingly longer to scale to additional instances.

The rule of thumb is to scale the existing architecture to its limits before considering alternative solutions. By "limits," I mean too high costs, time needed to scale, maintaining too many instances, etc.

> Imagine this situation: The current cost of scaling is around $30,000 per month. You know that you can extract part of your application to separate deployment units and scale them independently. This will reduce scaling costs to $15,000 per month.
>
> Your team has been tasked with estimating the costs of this migration. Given that you are working with a legacy application, it is important to consider the significant amount of refactoring required to wrap cohesive logic into separate modules and then extract them to microservices. The initial estimation for this effort is around $500,000.
>
> It means that you would need to wait approximately three years (33 months) until the savings associated with scaling cover the migration costs. And remember that you have yet to consider that with distributed systems, there are higher maintenance costs due to the complexity of the solution.

That is why the decision to migrate the existing architecture to a different one should always be based on a profit-and-loss calculation.

> *Numbers, numbers everywhere!*

Before thinking about solutions, find the root cause of your performance problems. It might be the API, the database, or some infrastructure component. This way, you might avoid high migration costs if it turns out that it will be enough to use only another component and leave other parts of the architecture as they are.

Vertical versus horizontal

Before we go any further with scaling, you need to be aware that you can scale your application in two ways:

Vertical scaling. This approach is based on increasing the power of the existing machine by adding RAM, storage, or replacing the GPU or CPU with the one with higher performance and more cores. It is straightforward and does not require changes to the application code. However, at some point, you cannot add more resources because of hardware limitations. Additionally, the more resources you add, the more expensive it gets, and it makes no sense to upgrade it anymore.

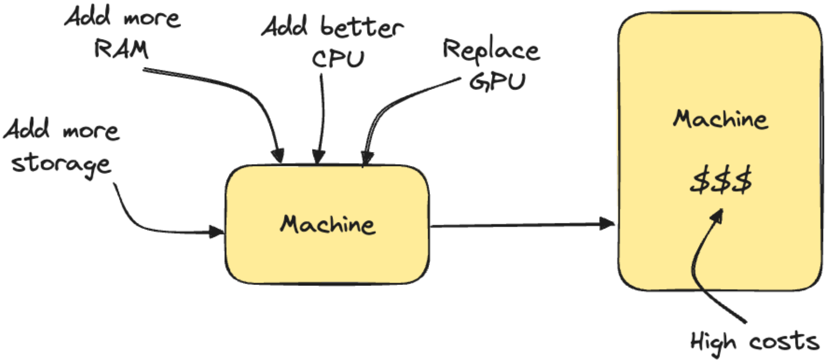


Figure 185. Vertical scaling requires hardware upgrades

Horizontal scaling. In this approach, you add more machines to your pool of resources. Instead of making one machine more powerful, you increase the number of machines handling the load. This way you can handle large traffic by distributing the load across multiple instances. It is more complex to implement than vertical scaling, as it often requires modifications to the application to ensure it can run on various instances, including load balancing, data replication, and ensuring fault tolerance.

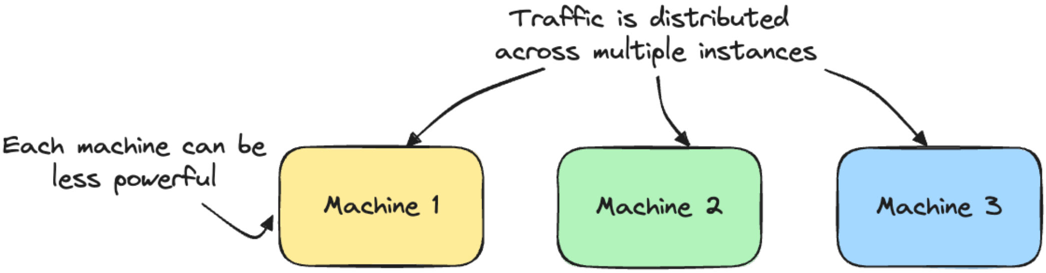

Traffic is distributed across multiple instances
Each machine can be less powerful
Machine 1
Machine 2
Machine 3

Figure 186. Horizontal scaling is based on balancing traffic across multiple instances

API

First, let's have a look at the API. We know that it handles 730 requests every second.

We measured the average time needed to handle a request and return the response, which is approximately 100 ms. So, to handle 730 requests, we will need the following number of processing threads:

$$ProcessingThreads = \frac{730 \times 100ms}{1,000ms} = 73$$

What does this mean in practice? If one instance has only one processing thread, you will need 73 instances to handle the traffic. If it has four processing threads, you will need 19 instances.

When using async and await, the situation changes significantly. It will allow you for concurrent execution without necessarily using multiple threads.

With async and await, a single thread can handle multiple requests concurrently. While one request is waiting for I/O (such as database queries), the thread can switch to processing another request.

This means that you may need far fewer than 73 processing threads to handle 730 requests per second, depending on the nature of the work being done.

If the 100 ms of processing time is mostly I/O-bound, you could potentially handle all 730 requests per second with just a few threads. However, if the 100 ms is CPU-bound computation, async and await won't provide as much benefit.

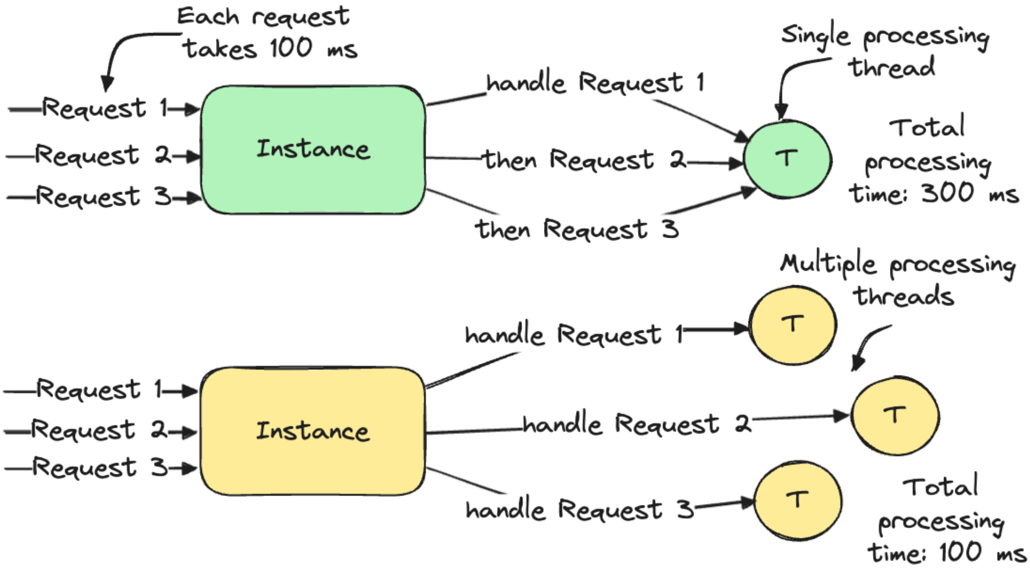


Figure 187. The difference between a single thread and multiple threads

As you can imagine, each environment will be based on different characteristics, so it is impossible to say:

> *Take X machines and you will be able to handle 20 requests per second.*

Of course, this is not the end of planning scaling—life would be too beautiful if it were that simple:

- **Average is often useless**. When one request takes 10 ms, and another one 100 ms, the average is going to be 55 ms. Is this information helpful? I do not think so. Optimizing scaling based on average time could lead to excessive resource usage and increased costs.

- **API is not the only component**. There might be other components, such as message broker or a database, in your architecture setup. There is a high chance that the request will take only 3 ms in the API but 97 ms to process it in the database. That is why it is important to find bottlenecks before doing any optimizations.

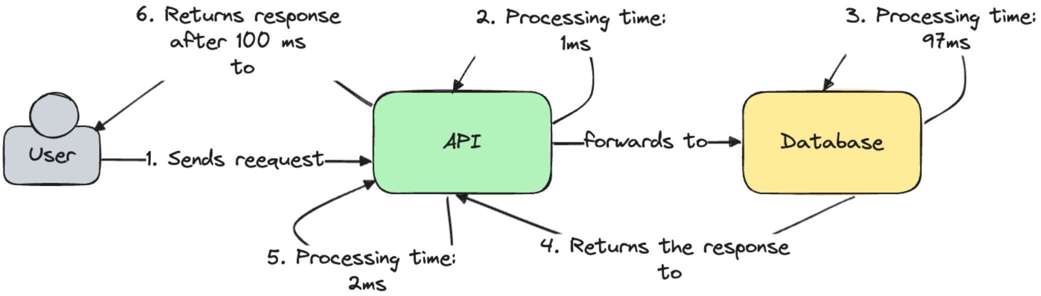


Figure 188. Different request handling times depending on components

Database

Scaling a database is a complex task that requires careful analysis to identify where performance issues arise. Problems typically fall into one of three categories:

- Read operations
- Write operations
- Combination of read and write operations

Based on the specific optimization needs, various strategies can be employed, such as index improvement, read replicas, partitioning, or sharding.

Review indexes and queries

One of the first actions to take when you notice that your database cannot guarantee enough performance on read operations is to check if your tables are correctly indexed.

> *Thank you, Captain Obvious!*

I know, I know. But I cannot count how many times this has happened, and starting with other optimizations was a bad choice. That is why I always check it as a first step.

Next, there is a chance that the database query you are using is not optimal. This could be due to the ORM or a minor mistake made by someone. Review it with your colleagues and try to improve it together.

Make sure that none of these are the issues before scaling.

Read replicas

When you are certain that the problem is neither related to indexes nor queries, it is time to check whether it is due to read or write operations. Sometimes, there will be way more reads than writes, while other times, it will be the other way around.

If you are experiencing poor performance due to read operations, you can try to replicate (make a copy) your database and forward part of the read traffic to it. This process creates an additional working instance.

These instances are called read replicas and can help offload the primary database in case of heavy traffic.

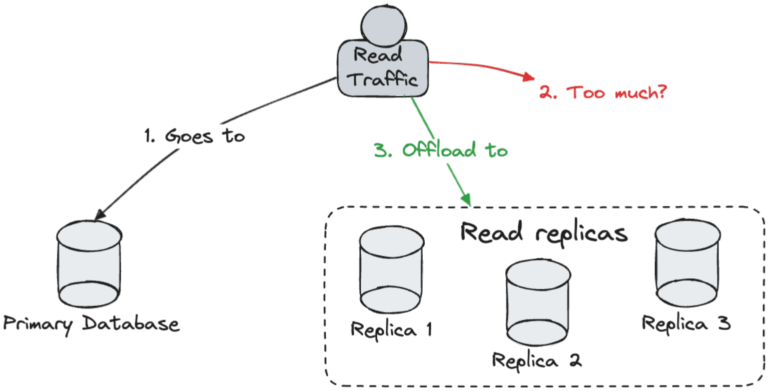


Figure 189. Read replicas offload the primary database in case of heavy traffic

How do you replicate the data? Well, it depends on whether you need strong or eventual consistency.

Strong consistency
ensures that any read operation immediately reflects the most recent write, providing a consistent view of data across all nodes at any time.

Eventual consistency
allows for temporary differences between nodes, guaranteeing that, given enough time, all nodes will eventually converge to the same data state.

In the latter case, replication is relatively easy to achieve. Replicas are asynchronously updated, and we accept the fact that users might see different results.

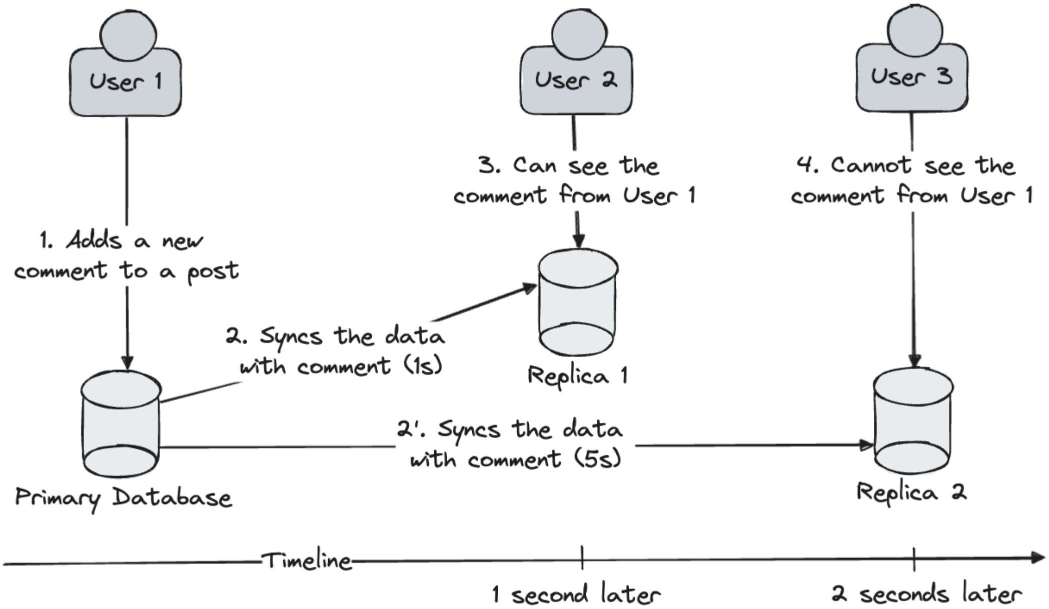

User 1
User 2
User 3
1. Adds a new comment to a post
2. Syncs the data with comment (1s)
3. Can see the comment from User 1
4. Cannot see the comment from User 1
2'. Syncs the data with comment (5s)
Primary Database
Replica 1
Replica 2
Timeline
1 second later
2 seconds later

Figure 190. Eventual consistency while reading the data

The first scenario presents an entirely different situation. Achieving strong consistency is tricky. You either accept a considerable delay and lack of access to data for a certain period, or you increase the complexity of your solution. Delays are often unacceptable, so you have to find another way.

There are worse and better solutions to this problem, but I won't focus on them here. Instead, here are two that I think are worth reading:

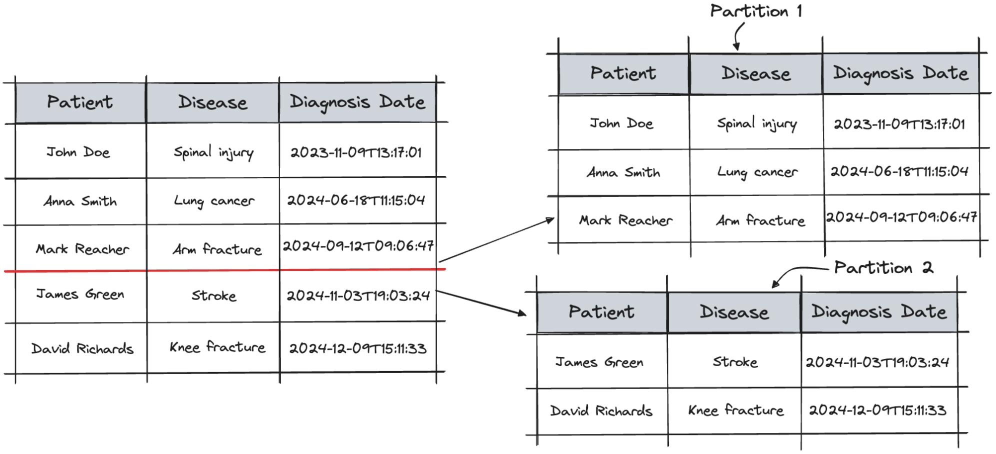
1. How We Learned to Stop Worrying and Read from Replicas[8]
2. Read Consistency with Database Replicas[9]

Read replicas are available in many databases, such as Postgres, MySQL, and SQL Server. Since we are using Postgres and have problems with read performance, we will leverage this mechanism.

Partitioning

If you have problems with reads due to the amount of data stored in a single table, you should consider another solution called partitioning.

With this mechanism, you can divide a single table into multiple partitions, each storing a specific data set. All partitions are stored in the same database.



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
James Green	Stroke	2024-11-03T19:03:24
David Richards	Knee fracture	2024-12-09T15:11:33

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

Patient	Disease	Diagnosis Date
James Green	Stroke	2024-11-03T19:03:24
David Richards	Knee fracture	2024-12-09T15:11:33

Figure 191. Splitting the database table into two partitions

Before implementing any database solution, make sure that the technology you choose supports future migration to partitions. Lack of support for partitions can lead to significant challenges down the road.

[8] https://medium.com/box-tech-blog/how-we-learned-to-stop-worrying-and-read-from-replicas-58cc43973638
[9] https://shopify.engineering/read-consistency-database-replicas

When the table is divided, the query will operate on less amount of data. You can have partitions that are identified by:

- Customer number—numbers 1-100 in partition 1, then 101-200 in partition 2, etc.
- Product category—sweaters in partition 1, jeans in partition 2, t-shirts in partition 3.
- Hash-based function.

The first two methods can lead to a situation where one partition is significantly larger than another due to the amount of data for a particular customer or product. One customer or product may have more data than others, resulting in an uneven split.

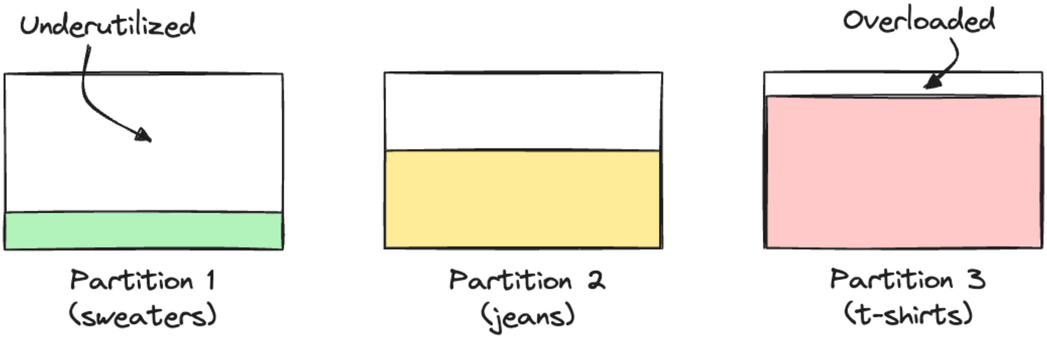


Figure 192. Unequal distribution of data between partitions

That is why I suggest using a hashing function that can be run on a selected column, such as an identifier. This way, the data is randomly and evenly divided between multiple partitions. Let me explain how hashing works.

You can skip this section as database providers offer built-in partitioning by hash (e.g., MySQL has `PARTITION BY HASH(column_name)`), so this is more of a theoretical walkthrough to understand the fundamentals.

First, you have to select a hashing function or create your own. For the sake of this example, I will create it from scratch.

$$hash(x) = x \times 34$$

This function can multiply any column (`x`) by `34` to create a hash.

The next step is to define the number of partitions you want—you can select any number. I choose three partitions in this case. To calculate which partition should be used, I use the following formula:

$$partition = hash(x) \mod numberOfPartitions$$

Now, let's look at the table with customers data.

CustomerId	Name	Country
1	Tutti Frutti	Italy
2	Hikers	England
3	Vignoble	France

You must apply the hash function on each row to divide it between partitions. I will use `customerId` column for this purpose.

First row:

$$hash(1) = 1 \times 34 = 34$$

$$partition = 34 \mod 3 = 1$$

Second row:

$$hash(1) = 2 \times 34 = 68$$

$$partition = 68 \mod 3 = 2$$

Third row:

$$hash(1) = 3 \times 34 = 102$$

$$partition = 102 \mod 3 = 0$$

As a result, the table that represents the partitions looks like this:

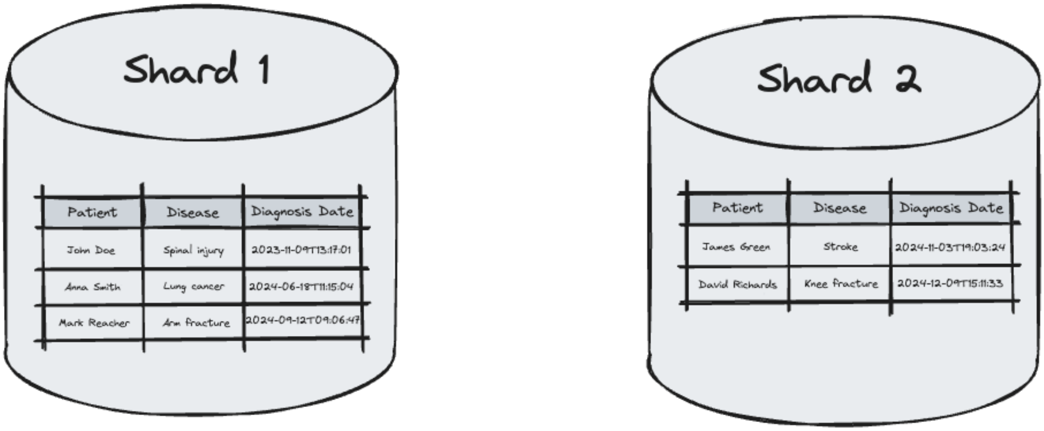
CustomerId	Hash	Partition
1	34	1
2	68	2
3	102	0

Each time a new row is added to the customers' table, a hash is calculated for it, and a partition is assigned. This ensures that the data is equally distributed.

Sharding

Partitioning works well until the point where using a single database node is insufficient due to excessive data and operations. That is where another concept comes in handy—sharding.

Sharding is simply horizontal partitioning. This means that your tables are still partitioned, but instead of keeping them on one database node in the form of partitions (partition 1, partition 2, partition 3), they are split across multiple nodes (shard 1, shard 2, shard 3).



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

Patient	Disease	Diagnosis Date
James Green	Stroke	2024-11-03T19:03:24
David Richards	Knee fracture	2024-12-09T15:11:33

Figure 193. Splitting the database table into two shards

The distribution between shards can be handled similarly to partitioning, using methods such as a hashing function. However, this time, instead of the partition number, you will operate on the shard number.

$$shard = hash(x) \mod numberOfShards$$

This approach allows for increased throughput and theoretically infinite scalability. It is particularly beneficial when managing millions of users, as it overcomes the limitations of adding power to a single instance, which would eventually become ineffective.

Moreover, if one or several database nodes experience an outage, the remaining nodes continue to operate, ensuring uninterrupted service for some customers.

The disadvantage is that it brings additional complexity, and in most cases, you will need to handle the distribution at the application level.

Cache

> *We have a performance problem that we need to solve. OK, let's add cache. Now we have two problems.*

This is one of my favorite quotes; unfortunately, I can't find the source.

When you add an additional external component to your application—regardless of its type—you increase its complexity. The entry threshold will be higher, and the team will have one more thing to care about.

It is no different for cache. It is a great mechanism that can help you with some problems, but it is often overused. That is why I recommend considering it only after you have exhausted all options with your current setup (e.g., a database) and when scaling costs begin to kill your business.

Using a cache can significantly improve the speed of read operations in your application. It stores data in memory, making it much faster to access than fetching data from a disk or a database.

For instance, when a user requests information from the API, the API first checks the cache for the data. If found, it returns the result immediately. If not, it then queries the database.

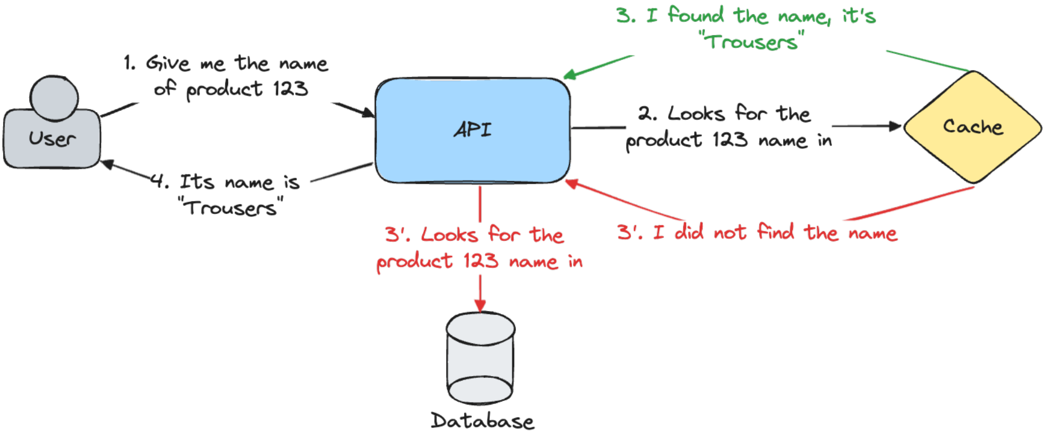

3. I found the name, it's "Trousers"
1. Give me the name of product 123
User
API
2. Looks for the product 123 name in
Cache
4. Its name is "Trousers"
3'. Looks for the product 123 name in
3'. I did not find the name
Database

Figure 194. The flow of using the cache to find the value

This process helps minimize round trips to the database (reducing traffic) and therefore the need to scale it.

When deciding which data to store in a cache, look for the one that is read frequently but changed infrequently. This way, you minimize the potential drawbacks of caching, such as stale data and consistency issues.

There are a few other things you need to be aware of. One of the most important is to watch out for cache limits:

1. **Memory**. If you have 10 GB of data but the cache instance can handle a maximum of 6 GB, you will need to use two instances.
2. **Requests per second**. If the limit is 100,000 requests per second (per instance) but your traffic generates 1,000,000, you will need at least ten instances.
3. **Connected clients**. Usually, the maximum number of connected clients is also limited. For example, if a cache instance supports a maximum of 20,000 clients simultaneously and you have more, you may need a higher tier or additional instances.

Additionally, you must carefully manage the cache expiration policy to ensure that outdated or stale data does not remain in the cache for too long.

Another thing to remember is to maintain consistency between the cache and the underlying data source; otherwise, your application might use outdated or incorrect data.

Finally, not all cache providers offer the option to back up stored values, such as saving them to a file on disk. In the event of a server restart or a cache component failure, all data will be dropped (as it is located in memory). If this happens, you will need to recreate it (e.g., sync with the database).

Microservice extraction

We analyzed our current environment and identified two issues:

1. The *Patient Treatment* module generates too high costs, as we have to scale the entire application.
2. We need to redeploy the application each time we change the code in the *Patient Treatment* module (several times per day).

After a deeper analysis, we decided to create a separate deployment unit for this module.

First, we need to extract the code related to the *Patient Treatment* module to a microservice. This means that the modular monolith will no longer be aware of the *Patient Treatment* module, which will run independently.

When modules are built around cohesive areas, the change is pretty straightforward because you only care about the extraction. There will be no refactoring as everything is already loosely coupled.

> When moving the code, I recommend having a separate repository for the new microservice. This is not compulsory, but if mismanaged, it can cause headaches. If you are not comfortable with this approach, you can use the monorepo approach instead.

After successful extraction, we still have to do one more thing. Do you remember that in the previous step, we had to extract building blocks to a separate project within a modular monolith to allow it to be referenced by modules from various projects?

Well, it won't be enough in this step. Building blocks is a project located inside a modular monolith. Our new microservice knows nothing about the modular monolith, so it cannot reference it.

We have two options:

1. Duplicate the logic from building blocks in both a modular monolith and a microservice (not optimal).
2. Extract the building blocks project from the modular monolith and create a package that can be referenced.

The second option will be a better fit for us. This way, if we have the subsequent deployment units, we can easily reference building blocks package in each. There are now three independent units.

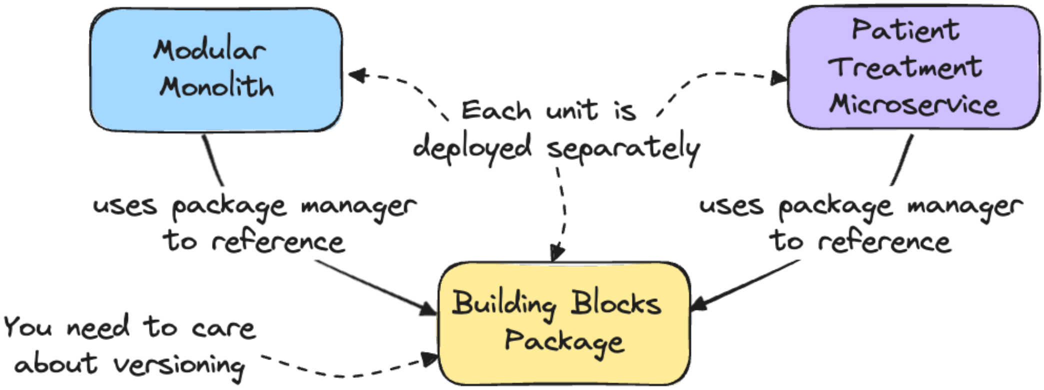

Modular Monolith
Patient Treatment Microservice
Each unit is deployed separately
uses package manager to reference
uses package manager to reference
Building Blocks Package
You need to care about versioning

Figure 195. Three independent deployment units

You can think of it as an NPM package in JS or a NuGet package in C#. Of course, this adds additional complexity, as you now need to take care of its versioning and update references (on each change) to it in both modular monolith and microservice.

Communication

As we have extracted one of the modules to a separate deployment unit, we are now dealing with two units:

1. Microservice, which represents *Patient Treatment*.
2. Modular monolith, which represents all other areas.

In the previous step, I mentioned that asynchronous communication via an external component such as a message broker is optional because you lose the power of communication within a single process. However, with the new architectural setup, each unit now runs in its own process. This requires the use of an external broker for communication, as they now need to communicate over the network. Of course, we still keep using inbox and outbox patterns.

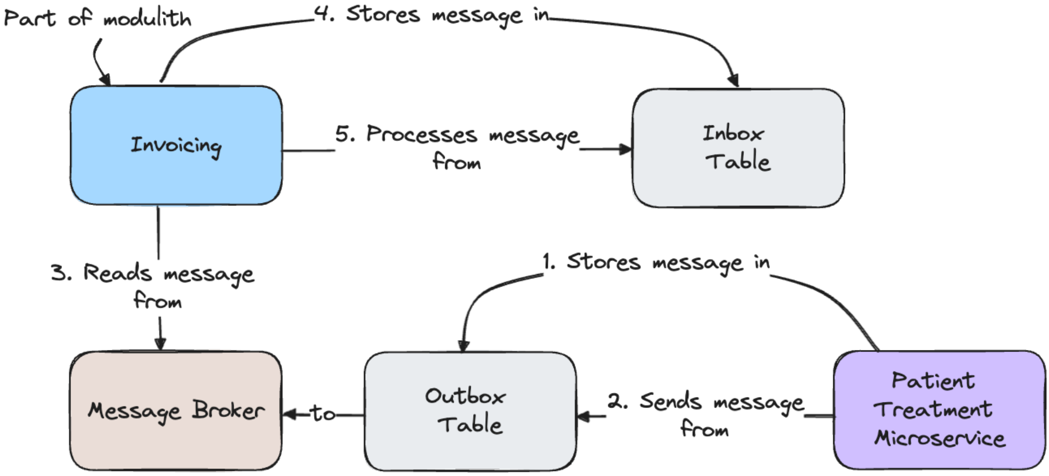


Figure 196. Asynchronous communication in our application using external message broker

In the case of synchronous communication, we can leave it as it is. This means that modules inside modular monolith can still reference each other using their public APIs (no network traffic, pure code reference).

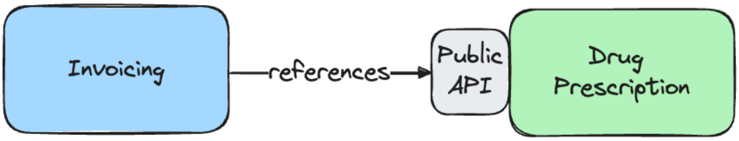


Figure 197. Synchronous communication in modular monolith

After several rounds of brainstorming, we also decided that we do not need to communicate between the modular monolith and our new microservice using direct calls to REST API over HTTP. We want to keep them loosely coupled.

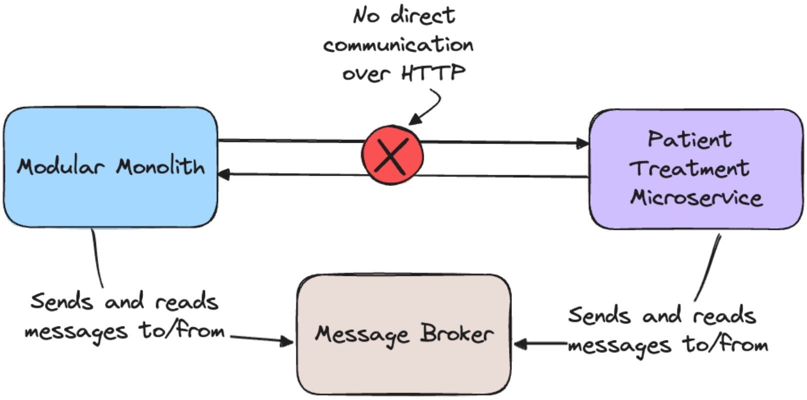

No direct communication over HTTP
Modular Monolith
Patient Treatment Microservice
Sends and reads messages to/from
Message Broker
Sends and reads messages to/from

Figure 198. No direct communication between modular monolith and microservice

Use direct calls only when an immediate response is crucial for critical operations. For all other scenarios, including ours, I recommended maintaining loose coupling between deployment units.

Result

Thanks to all the steps that we took, we were able to reduce running costs. Instead of scaling the entire application, we can now scale the *Patient Treatment* microservice that generates the highest traffic.

Introducing a separate deployment unit does come with challenges, particularly in handling communication between units over the network (network errors, network partitions, higher latency). We decided to use an additional external component—a message broker. As a result, it increased the complexity of the entire solution and decreased its maintainability.

The solution is more complex than before, but it performs well again.

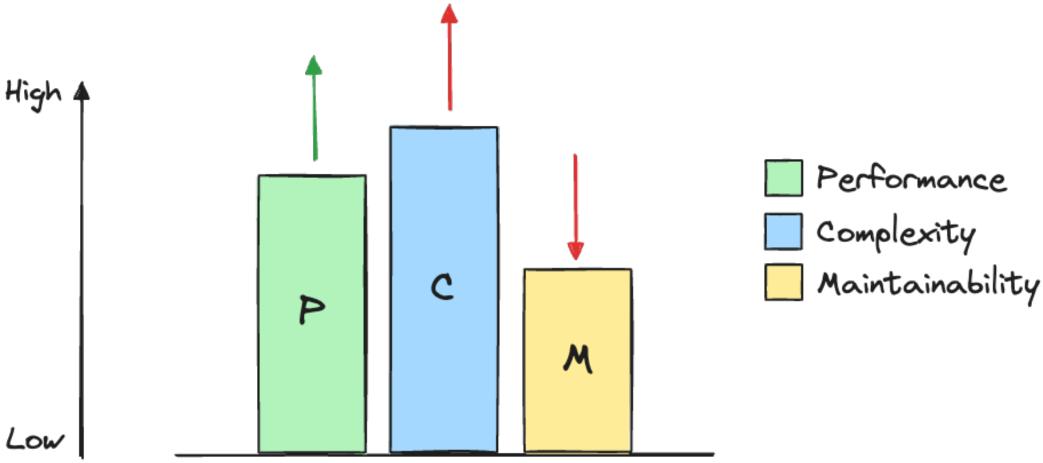


Figure 199. Architectural drivers after focusing on growth

Area	Decision
Deployment strategy	3 units (modular monolith, microservice, and building blocks package)
Code structure	*Patient Treatment* module extracted from modular monolith to the microservice
Communication	Synchronous (as before) and asynchronous using an external message broker (including inbox and outbox patterns

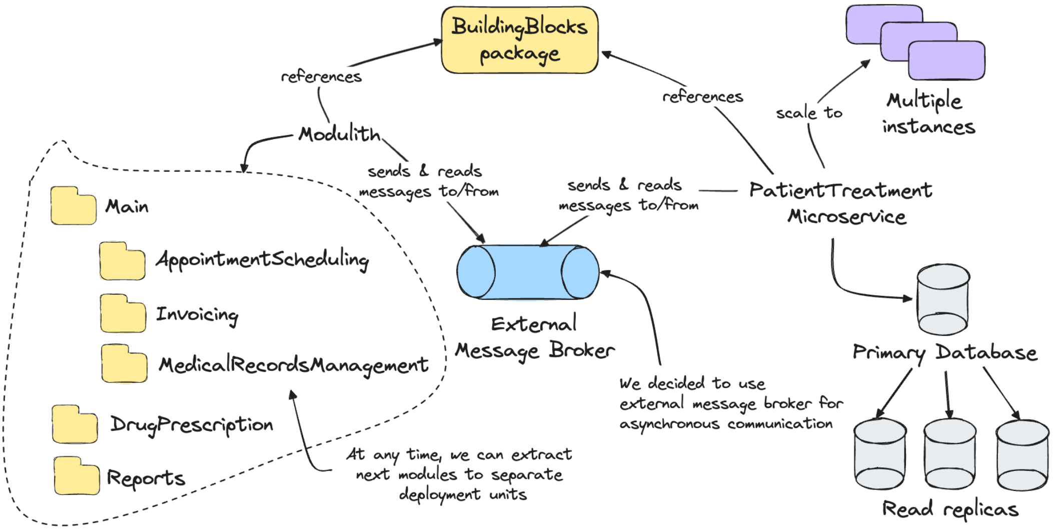

BuildingBlocks package
references
references
Modulith
sends & reads messages to/from
sends & reads messages to/from
scale to
Multiple instances
PatientTreatment Microservice
Main
AppointmentScheduling
Invoicing
MedicalRecordsManagement
DrugPrescription
Reports
External Message Broker
Primary Database
We decided to use external message broker for asynchronous communication
At any time, we can extract next modules to separate deployment units
Read replicas

Figure 200. The current state of the application

Fourth: Focus on complexity

A year has passed. The application is stable, and we have captured much of the market. Growth has slowed compared to before, performance issues have been resolved, and infrastructure costs are now acceptable.

However, a new problem has emerged. The government has issued new regulations, and medical clinics have reacted by extending their processes. As a result, the business logic in the *Drug Prescription* module has become quite complex and continues to be extended with new features.

We have observed that behavioral entities we used have gradually become too large to maintain. In a nutshell, there are too many if statements and cross-checks.

What changed?

Compared to the previous step, the environment around our application has changed in the following areas:

1. The government announced many new regulations (for example, prescriptions can only be canceled within one hour of provisioning). Each regulation in the *Drug Prescription* module increases the complexity of its business logic.
2. Due to new features, the *Drug Prescription* module has become increasingly complex, making it more challenging to maintain with the current code structure.
3. The team responsible for the *Drug Prescription* module has started to express concerns about its maintainability.

Let's look at architectural drivers, as they will form the basis for our subsequent decisions.

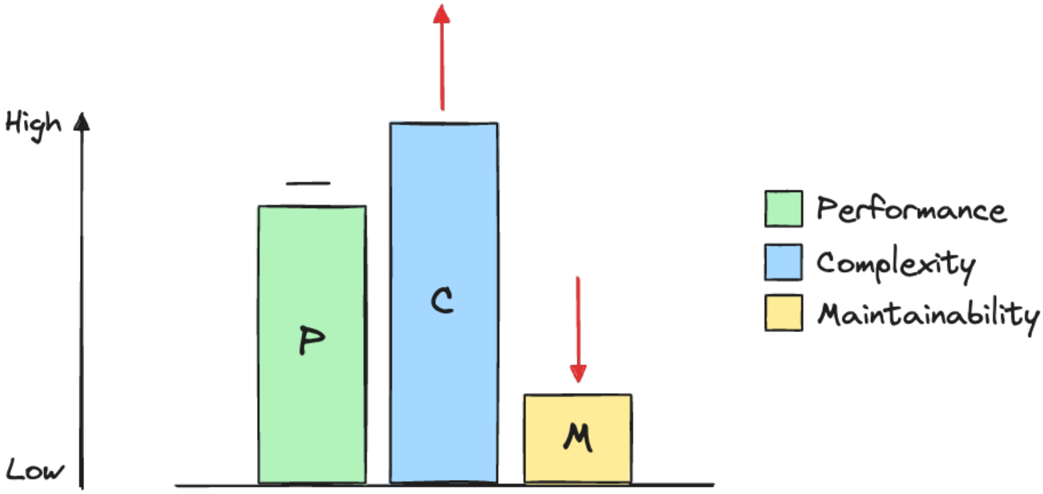


Figure 201. The current state of architectural drivers

Compared to the previous step:

- **Complexity increased**. New features and regulations affecting the *Drug Prescription* module have made the application more complicated. These additions require new features, checks, and adjustments to existing processes, making the system more complex and challenging to understand.
- **Maintainability decreased**. The increased complexity of the application has made it harder to keep running smoothly. It takes more effort to fix problems

> and make updates. Small changes in the *Drug Prescription* module can cause significant issues, so updates take longer and cost more to implement.

Performance is not noticeably affected, so it does not require our focus at this time. Reducing the complexity of the code will positively impact maintainability, so this will be our primary focus moving forward.

Complexity: Problems to address

There are new processes and regulations, so we must extend the existing business logic.

The problem is that the code is already complex with many if statements. When we touch one part, it causes issues in other areas. If we don't address this, the codebase will quickly become unmaintainable.

Problem	Category
Increased code complexity	Code structure
Ineffective design	Code structure
Complex business logic	Functional requirements

Redesign the module

The model we prepared in the first step of simplicity proved to be too naive for the *Drug Prescription* module. We didn't know how it would behave in the future, so we took some risks. Now, with multiple services checking the business logic, things are starting to spiral out of control. We need to redesign it.

What can help in this case is using the third level of Event Storming, called the *Design Level*. This type of workshop is typically conducted by developers. There will be one new type of note:

Aggregate

- **Color**: Usually yellow

- **Purpose**: It groups related events, commands, actors who execute them, and rules inside cohesive boundaries, guaranteeing consistency by enforcing invariants and transaction boundaries. Treat it as a consistency guardian.

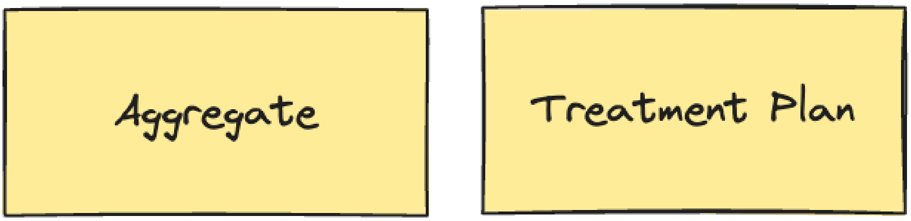


Figure 202. An example of an aggregate

Before we start modeling, let's examine the drug prescription process (I skipped sending the drug prescription copy).

> ⚠ It differs from what we modeled at the beginning of this book, as we are now in the future. If you do the *Design Level* in the greenfield application, I recommend not modifying existing diagrams from higher levels but creating another one from scratch.

The process:

1. The doctor looks for drugs for treatment.
2. The doctor selects drugs. At most, ten drugs can be selected per prescription. Also, the doctor must ensure that selected drugs do not adversely interact with other medications the patient is taking.
3. The doctor can prepare the prescription once the drugs are selected. A unique number is assigned when a new prescription is registered, and the prescription cannot be deleted. It can be modified until it is provided to the patient.
4. The doctor provides the prescription to the patient only if the patient has valid insurance.
5. Once provided, the prescription cannot be changed but can be canceled by the issuing doctor within one hour.

As you can see, it has become quite complex. There are more rules than in the beginning.

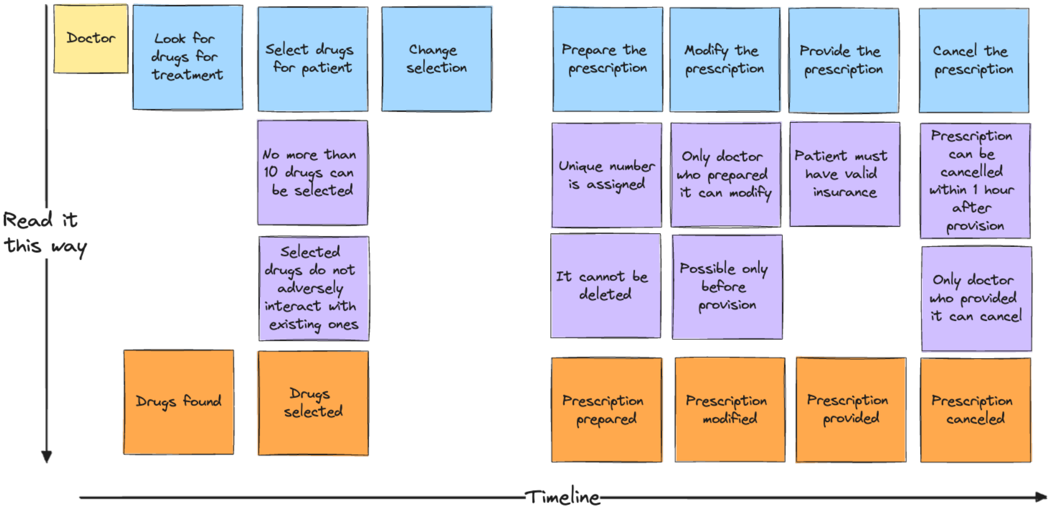

Doctor
Look for drugs for treatment
Select drugs for patient
Change selection
Prepare the prescription
Modify the prescription
Provide the prescription
Cancel the prescription
No more than 10 drugs can be selected
Selected drugs do not adversely interact with existing ones
Unique number is assigned
It cannot be deleted
Only doctor who prepared it can modify
Possible only before provision
Patient must have valid insurance
Prescription can be cancelled within 1 hour after provision
Only doctor who provided it can cancel
Drugs found
Drugs selected
Prescription prepared
Prescription modified
Prescription provided
Prescription canceled
Read it this way
Timeline

Figure 203. The result of the model design

The diagram above presents two steps: the first pertains to drug selection, and the second pertains to drug prescription. What can we do with this knowledge?

Let's start with the second step, which is more complex. Several operations can be performed, depending on the set of rules. The state of the prescription can change (e.g., prepared, provided, canceled). The prescription can also be modified, but only before provisioning.

The fact that its state can be changed makes it prone to corruption, especially if modifications are allowed from the outside. Think of it as having public property in prescription that anyone can modify. This way, the business logic can be split around multiple places, and a change in one place can negatively impact other places—a recipe for disaster in the long run.

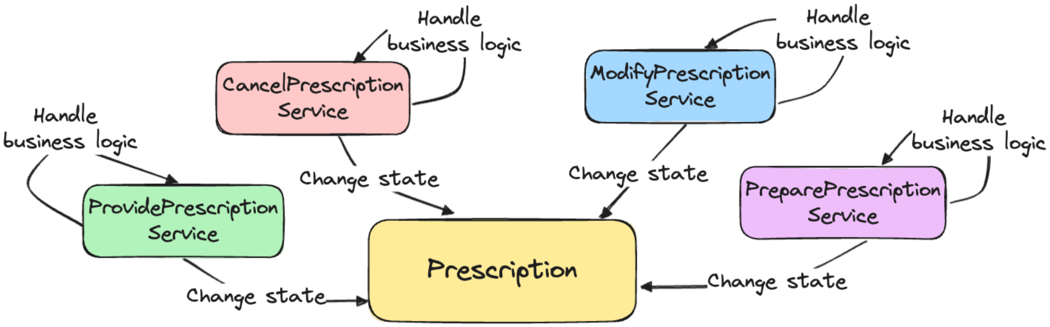


Figure 204. Business logic and state modifications distributed across services

Instead, you could introduce the *Prescription* aggregate that encapsulates the entire business logic within its boundaries.

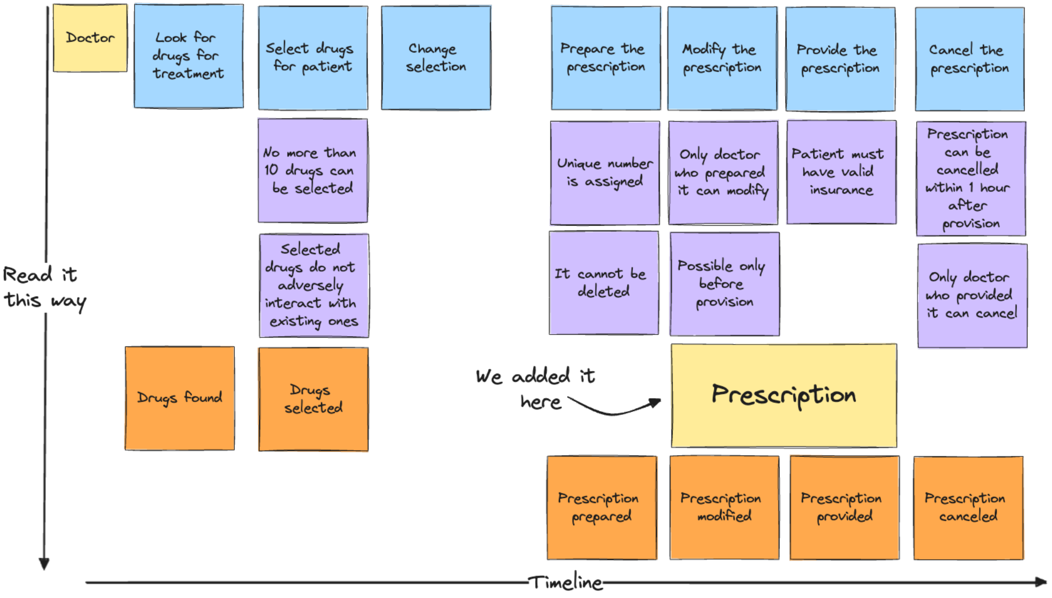

Doctor
Look for drugs for treatment
Select drugs for patient
Change selection
Prepare the prescription
Modify the prescription
Provide the prescription
Cancel the prescription
No more than 10 drugs can be selected
Unique number is assigned
Only doctor who prepared it can modify
Patient must have valid insurance
Prescription can be cancelled within 1 hour after provision
Selected drugs do not adversely interact with existing ones
It cannot be deleted
Possible only before provision
Only doctor who provided it can cancel
Read it this way
Drugs found
Drugs selected
We added it here
Prescription
Prescription prepared
Prescription modified
Prescription provided
Prescription canceled
Timeline

Figure 205. Prescription aggregate

As the business logic is fully handled inside the aggregate, it does not leak to other places or layers, and aggregate can guarantee consistency during the lifetime of the prescription. Thanks to this approach, someone who wants to modify the prescription must use aggregate's public interface. When the modification request comes, the aggregate will validate it.

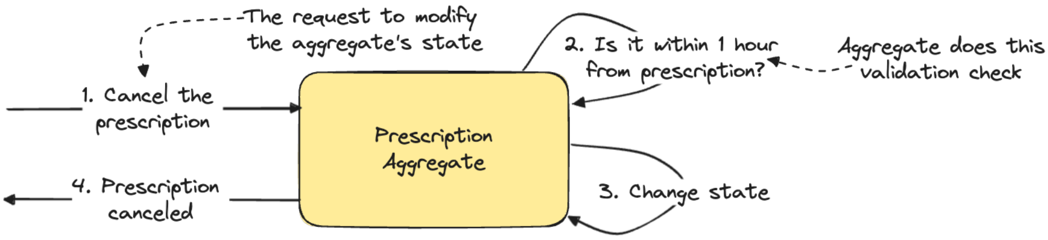


Figure 206. Changing state of the Prescription aggregate

Each time the doctor needs to perform an operation (such as canceling the prescription), the *Prescription* aggregate will first check whether the operation can be handled and then execute the related business logic. Magic, isn't it? :)

Coming to the drug selection step. It could be handled in two ways, depending on the context. If you do not modify the state, consider moving the entire logic related to it to the frontend.

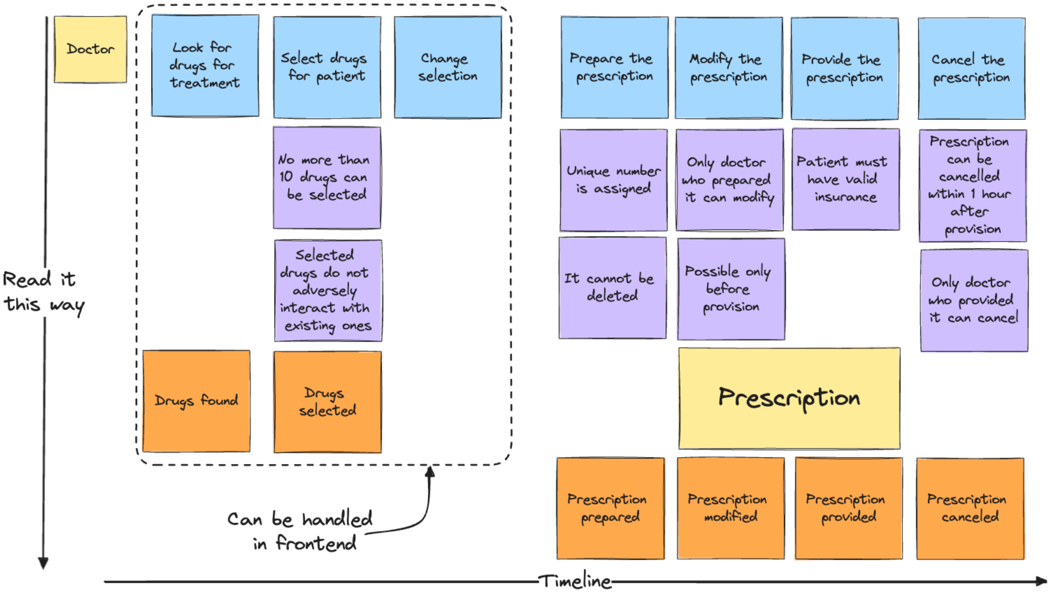

Doctor
Look for drugs for treatment
Select drugs for patient
Change selection
No more than 10 drugs can be selected
Selected drugs do not adversely interact with existing ones
Drugs found
Drugs selected
Can be handled in frontend
Read it this way
Prepare the prescription
Modify the prescription
Provide the prescription
Cancel the prescription
Unique number is assigned
Only doctor who prepared it can modify
Patient must have valid insurance
Prescription can be cancelled within 1 hour after provision
It cannot be deleted
Possible only before provision
Only doctor who provided it can cancel
Prescription
Prescription prepared
Prescription modified
Prescription provided
Prescription canceled
Timeline

Figure 207. Possibility to move the selection logic to the frontend

ℹ It would be nice to keep the selection's state (different from the aggregate's state) on the frontend, for example, by using the browser's local storage. The selection won't be gone when the user closes the browser or when there is an unexpected error.

However, if the process is complicated and requires state modifications, similar to prescriptions, you could introduce the *Drug Selection* aggregate to ensure consistency.

Such workshops can be conducted at any time. In the companies where I have worked, we did this quite often—sometimes multiple times per week— whenever we needed to design something.

Domain Model

Following our design sessions, we began to consider how to transfer what we modeled into code. Given the complex business logic, someone suggested using the *Domain Model.*

Domain Model[10]
an object model of the domain that incorporates both behavior and data.

There was an awkward silence in the room. Finally, someone spoke:

> *OK, but what does it mean? How can we transfer what we modeled to the domain model*?

Luckily, the person who made the suggestion had the answer. He explained that by modeling the domain in objects which encapsulate both data and behavior, we ensure that business rules are consistently applied. Additionally, the domain model can facilitate communication between technical and non-technical team members, and it makes the application easier to maintain and extend as business requirements evolve.

We also learned that we can achieve this by leveraging the mechanisms from tactical Domain-Driven Design:

[10] https://www.martinfowler.com/eaaCatalog/domainModel.html

- Value objects
- Entities
- Aggregates
- Domain events

After further clarifications, we were ready to start.

Value objects

When you are not concerned with object identity, value objects come into play. These objects are immutable—once created, they remain unchanged.

First, let's start with a straightforward example. Suppose you sell boxes, and each box has its own dimensions. An object with the following structure can represent each dimension:

```
class Dimensions:
    values:
        Length: double
        Width: double
        Height: double
```

In this case, changing one of its values, such as `Length`, changes the entire dimension and may lead to a change in price. If two `Dimensions` objects have the same length, width, and height, they are considered equal.

If `Dimensions` had included a random identifier:

```
Class Dimensions:
    Values:
        Identifier: uuid
        Length: double
        Width: double
        Height: double
```

And if two objects with the same length, width, and height were compared, the result would be unequal (because of a different identifier).

What examples can be found in our application?

When a doctor prescribes a drug, they first look for it in the list of available drugs. Each drug on this list is a value object—it is defined by its properties (name, description, dosage) rather than by a unique identity. Consider two bottles of the same drug: they are interchangeable because their value lies in what they are, not in being a specific instance. This is the essence of a value object. It is similar to choosing a color for a car: you pick "red" from a predefined palette, not a unique "red instance #123".

```
class Drug:
    values:
        Name: string
        Description: string
        Dosage: string
```

During the appointment, the doctor examines the patient. When he is done, he must find the matching diagnosis in the registry. The code describes each diagnosis and has a corresponding description. Suppose there are three patients with the same diagnosis. In that case, the doctor does not need to say that patient number 1 has flu with identity ABC, patient number 2 has flu with identity DEF, and patient number 3 has flu with identity GHI. No, they all have a flu.

```
class Diagnosis:
    values:
        Code: string
        Description: string
```

Thanks to value objects, you can avoid the primitive obsession[11]. By encapsulating data in meaningful objects rather than relying on primitive types like string, int, or double, you create code that is more expressive and self-documenting. This improves readability and clarifies the intent and behavior of your code.

To illustrate the problem, let's create a `PatientMedicalRecord` class based only on primitive types:

[11] https://refactoring.guru/pl/smells/primitive-obsession

```
class PatientMedicalRecord:
    properties:
        Id: uuid
        PatientName: string // first, last or full?
        PatientEmail: string // where is the validation?
        PatientSocialSecurityNumber: string // what format?
        DiagnosisCode: string // what format?
        DiagnosisDescription: string // maximum length?
        PrescribedDrugs: collection<string> // names, codes?
```

Take a look at the comments. You will need to include this logic in either `PatientMedicalRecord` class or extract it to some `PatientMedicalRecordValidator` class. You must repeat this validation for all other classes using any of these properties. It will spread across many places, and if something changes in one place, you must remember to fix it in other areas. This approach does not seem very reliable.

It will be much easier if we close the logic within the scope of each value object and define `PatientMedicalRecord` on top of it:

```
class PatientMedicalRecord:
    properties:
        Id: uuid
        Name: PatientName
        Email: PatientEmail
        SSN: SocialSecurityNumber
        Diagnosis: Diagnosis
        PrescribedDrugs: collection<Drug>
```

Let’s first check the `PatientName` value object:

```
class PatientName:
    values:
        FirstName: string
        LastName: string
```

Now, when someone creates a new instance of `PatientName`, they do so by using `new PatientName("John", "Doe")`. Inside this class, you can have validation, such as checking the maximum length of the first and last names, or verifying that they are not null or whitespace. Any other class that uses `PatientName` will depend on its internal logic.

In `PatientEmail` you can include a `.Validate` method that checks if the email is valid.

You can also include methods like `.ConvertTo`, `.ConvertFrom`, `.Parse` and any other methods you find useful. The only thing to remember is that you operate on values.

Finally, `SocialSecurityNumber` will make sure that the number passed is in the correct format, e.g., `countryCode-4digits-5digits-3digits`. So when you call `new SocialSecurityNumber("UK-1234-56789-432")`, the value object itself (no external services) will be responsible for checking the passed string against the above format. If someone passes an incorrect value, such as `new SocialSecurityNumber("42-12-232")`, it will return an error.

Entities

Unlike value objects, entities have a distinct identity that runs through time and different states (they are mutable). An entity is defined not only by its values but also by a unique identifier.

For example, in our application, the patient can be an excellent example of an entity:

```
class Patient:
    properties:
        Id: uuid
        Name: PatientName
        Email: PatientEmail
```

It is important to note that we used here value objects—`PatientName` and `PatientEmail`. By using them instead of primitive types, we are re not just storing data; we are capturing the semantics and rules of our domain.

Each patient has a unique identity. It is physically impossible for two patients to have the same identity (maybe in another world). If there are two people named John, they will be completely different individuals. Imagine that you decide to use a value object instead of such a structure:

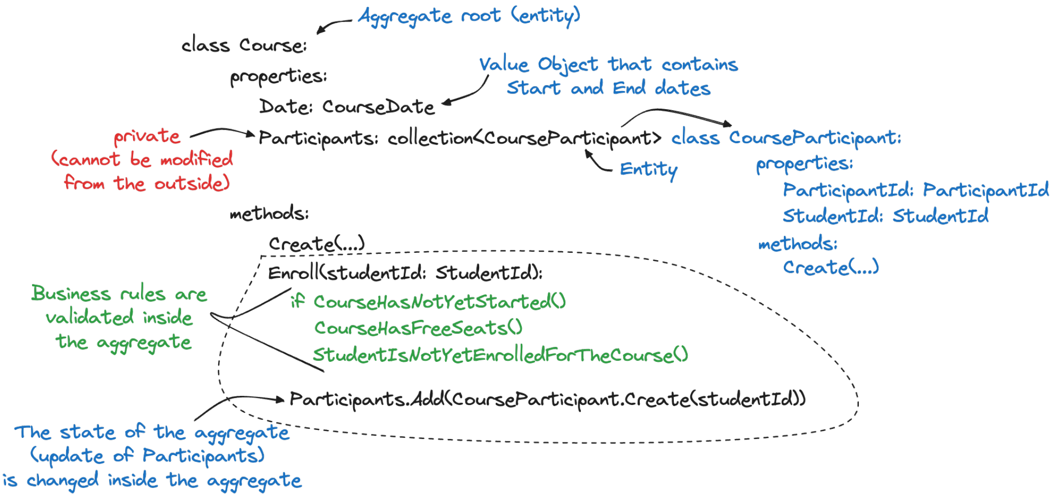
```
class Patient:
    properties:
        Name: PatientName
```

Without unique identifiers, it would be impossible to distinguish two or more patients with the same name.

Aggregates

Aggregate has its own identity and business rules, and it executes business logic within its boundaries. It guarantees the consistency of the data that belongs to it. You already know this from one of the previous sections.

Each aggregate will contain at least one entity that will act as the aggregate root. The aggregate root will act as the public interface through which you can execute actions, thereby changing its state. Aggregates can also contain other entities, but these entities will never be modified directly.



```
class Course:
    properties:
        Date: CourseDate
        Participants: collection<CourseParticipant>
    methods:
        Create(...)
        Enroll(studentId: StudentId):
            if CourseHasNotYetStarted()
               CourseHasFreeSeats()
               StudentIsNotYetEnrolledForTheCourse()

            Participants.Add(CourseParticipant.Create(studentId))

class CourseParticipant:
    properties:
        ParticipantId: ParticipantId
        StudentId: StudentId
    methods:
        Create(...)
```

Aggregate root (entity)
Value Object that contains Start and End dates
private (cannot be modified from the outside)
Entity
Business rules are validated inside the aggregate
The state of the aggregate (update of Participants) is changed inside the aggregate

Figure 208. The example course aggregate

As you can see, executing enrollment action for the course is based on different rules. Some rules are related to `Course` entity, while others are related to `CourseParticipant` entity. While enrolling for the course, the aggregate ensures that it performs all the required actions for course enrollment within one operation.

> *Since an aggregate's state can only be modified by its own business logic, the aggregate also acts as a transactional boundary. All changes to the aggregate's state should be committed transactionally as one atomic operation. If an aggregate's state is modified, either all the changes are committed or none of them is.*
>
> Vlad Khononov, *Learning Domain-Driven Design (p. 150). O'Reilly Media*

Keeping the structure of the above aggregate, if a student would like to enroll in a course, this is the flow:

```
// First find the course
course = courseRepository.GetCourse(id)

// When not found then return 404
if null course
    return 404

// Enroll for course (the entire logic is encapsulated within aggregate)
course.Enroll(studentId)

// Save the state of the aggregate to the repository
courseRepository.Save()

return 200
```

If you need to touch more than one aggregate to, e.g., enroll a student for a course, this is a clear indication that something is wrong with the current aggregate boundaries, and you need to review them.

Now that we understand the concept clearly, we can proceed to the final step: building the `Prescription` aggregate.

```
class Prescription:
    properties:
        Number: PrescriptionNumber (Value Object)
        PrescribedFor: Patient (Entity)
        PreparedBy: Doctor (Entity)
        Drugs: collection<Drug> (Value Object)
        PreparedAt: PreparationDate (Value Object)
        ProvidedAt: ProvisionDate (Value Object)
        ModifiedAt: ModificationDate (Value Object)
        CancelledAt: CancellationDate (Value Object)
    methods:
        Prepare(doctor: Doctor, preparedAt: datetime):
            PreparedBy = doctor
            PreparedAt = PreparationDate.Parse(preparedAt)

        AddDrug(doctorId: DoctorId, drug: Drug):
            if OnlyDoctorWhoPreparedItCanModify(doctorId)
               CanModifyOnlyIfNotYetProvisioned(ProvidedAt)
               Drugs.Add(drug)
            else Error

        Provide(patient: Patient, providedAt):
            if PatientMustHaveValidInsurance(patient.Id)
                ProvidedAt = ProvisionDate.Parse(providedAt)
                PrescribedFor = patient
            else Error

        Cancel(doctorId: DoctorId):
            if PrescriptionCanBeCancelledWithin1HourAfterProvision()
               OnlyDoctorWhoProvidedItCanCancel(doctorId)
               CancelledAt = now
            else Error
```

Domain events

Do you remember the Event Storming sessions where we focused on finding business events?

- Appointment scheduled
- Prescription provided
- Treatment started
- Treatment completed
- Invoice prepared

These are valuable facts that indicate something important has happened in the application. All of the above can be translated into domain events (which is why Event Storming sessions are so helpful).

When an action is taken, we can raise an event. For example, in `Prescription` aggregate, we can add it to the `Prepare` method.

```
class Prescription:
    properties:
    ...
    methods:
        Prepare(doctor: Doctor, preparedAt: datetime):
            PreparedBy = doctor
            PreparedAt = PreparationDate.Parse(preparedAt)
            RaiseDomainEvent(PrescriptionPrepared)
```

There can be several effects of raising such an event:

1. It can be stored within the module database (here, *Drug Prescription*) for information purposes.
2. It can trigger other actions within the module.
3. It can trigger other actions within other modules or external systems.

Domain events should always be written in the past tense, similar to business events from Event Storming sessions.

Result

Whoa! What a journey it was.

We conducted the *Design Level* (Event Storming) session where we discovered the prescription aggregate. Then, we went through the world of tactical Domain-Driven Design, including value objects, entities, and domain events.

Thanks to the new pattern—the domain model—we were finally able to get the *Drug Prescription* module under control. Now, we are confident that we will be able to maintain it, as the complexity has slightly reduced. No more "ifology," and the business logic is now wrapped inside aggregates that guarantee consistency.

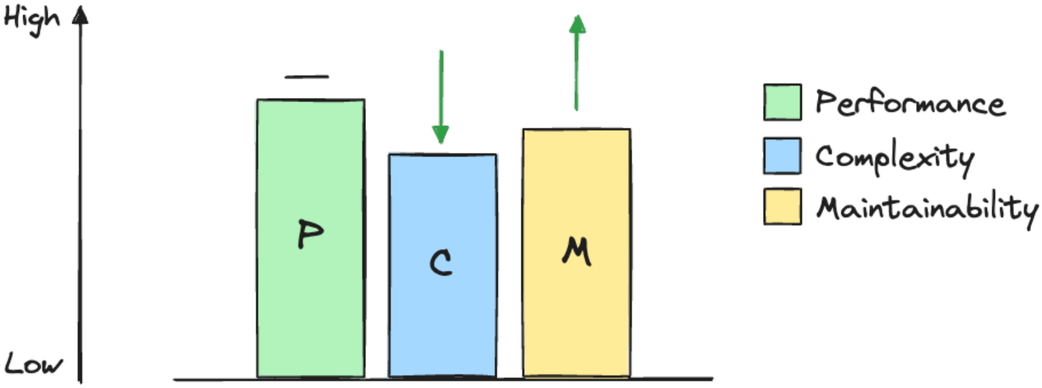


Figure 209. Architectural drivers after focusing on complexity

Area	Decision
Code structure	*Drug Prescription* module using the domain model pattern

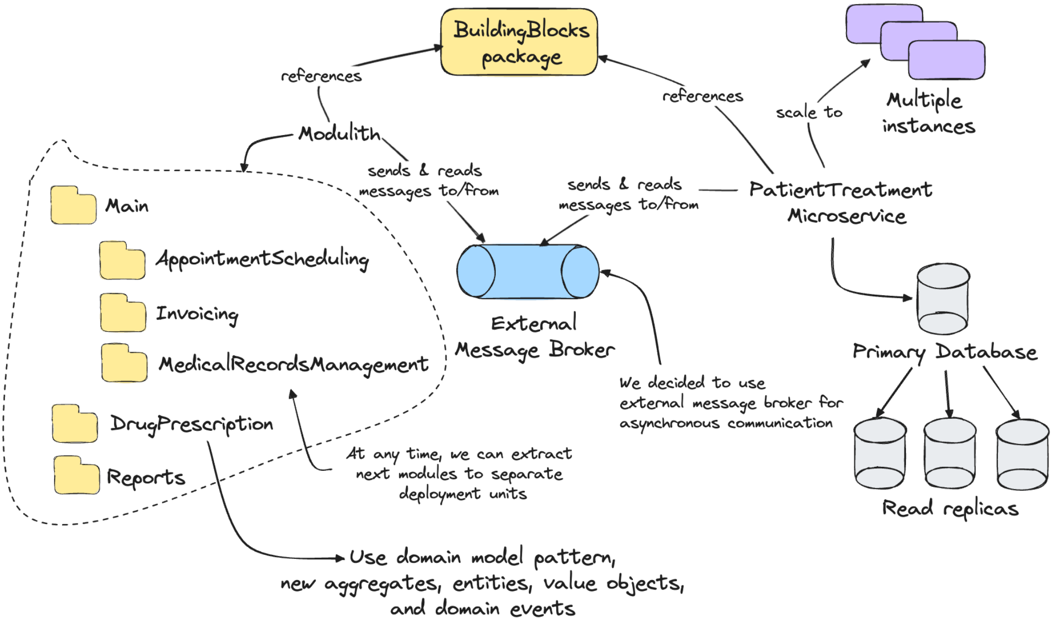

BuildingBlocks package
references
references
Modulith
Main
AppointmentScheduling
Invoicing
MedicalRecordsManagement
DrugPrescription
Reports
sends & reads messages to/from
sends & reads messages to/from
External Message Broker
PatientTreatment Microservice
scale to
Multiple instances
Primary Database
Read replicas
We decided to use external message broker for asynchronous communication
At any time, we can extract next modules to separate deployment units
Use domain model pattern, new aggregates, entities, value objects, and domain events

Figure 210. The current state of the application

Recap

An evolutionary approach to software architecture is an excellent way to make your application stable and minimize the risk of it becoming a legacy system over time.

When implementing a greenfield application, try to make it as simple as possible. Focus on mechanisms that solve your current problems, not those that might solve future issues. Build what you need today; otherwise, you will fall into the trap of too complex architecture.

Next, ensure that the application is easy to maintain. At some point, the simple solutions you chose might no longer be optimal. When that happens, consider adopting a different approach.

As your application gains traction, you may need to scale it. When focusing on growth, observe if it makes sense to scale your current setup (e.g., a single deployment unit) or replace parts of it with another solution (e.g., extraction of a microservice). Always assess the costs and benefits to ensure it pays off to make such changes.

Finally, parts of your application might become too complex to maintain. That is the time to consider different patterns that might better fit the problem you are trying to solve. Always consider the current context and choose solutions or tools that will help you and your team at that moment.

The application's evolution described is just one example. Architects encounter countless scenarios, making it impossible to cover every case. However, with the knowledge gained from this step, you should find it easier to make wise decisions.

STEP 8: Don’t Forget About Security

We all like to think about software architecture in terms of code and components, which is good because a well-structured application is one of the keys to success. But let’s not forget about security. A lot has been written about it in books, articles, and blogs.

Search for *OWASP*, and you will find tons of results describing the most common security risks in web applications. If you talk with your colleagues, you will notice that almost everyone has at least heard of it.

Yet, all too often, we overlook the critical role of security. We realize the seriousness of the situation only when we experience our first data breach or our application becomes unavailable. Why does it take such a catastrophic event to remind us of the importance of security? I have some thoughts on this.

When we start building an application, we tend to focus first on implementing as many features as possible. We want our product to rock the market, so we keep up a fast pace until we finally release the MVP (Minimum Viable Product).

Next, customers provide feedback, which usually results in developing new features and correcting existing ones. Again, we are extremely busy with the implementation.

By the time we release the application to the public, someone is already looking for ways to break it or steal sensitive data. The more attractive the data, the sooner this will happen.

While some factors are beyond our control, we still have the power to minimize risks. Not every company has a dedicated cyber security team, so as software engineers, it is our responsibility to educate ourselves in this area. Let’s not make life easy for hackers by making basic mistakes.

In this section, I will draw your attention to the security holes I frequently encounter. I can’t cover all the threats in a few pages, so I have picked the most common ones.

If you want to learn more about security, the best place to start is the OWASP page[1].

Insecure Direct Object References (IDOR)

Suppose you have built an application that helps customers manage invoices. They can create new invoices and duplicates and view their history. The application is well-rated, fast, and modern, so it received much attention. To use this application, customers must register and provide sensitive information, such as their bank account number, which will be used later to generate invoices.

One of the API endpoints (`https://invoicingApp/companies/{companyId}/invoices`) is designed to retrieve all the invoices from a given company. For the sake of simplicity, let's assume that the `companyId` parameter is an integer. To access their invoices, customer `123` clicks the link in the UI, and is redirected to the page with all invoices: `https://invoicingApp/companies/123/invoices`.

Unfortunately, a hacker registers a new account to steal other customers' data. The attacker prepares a plan to check for the most common vulnerabilities. First, they try manipulating the URL to access unauthorized data. They replace their `companyId` `124` with `123`, which belongs to another customer.

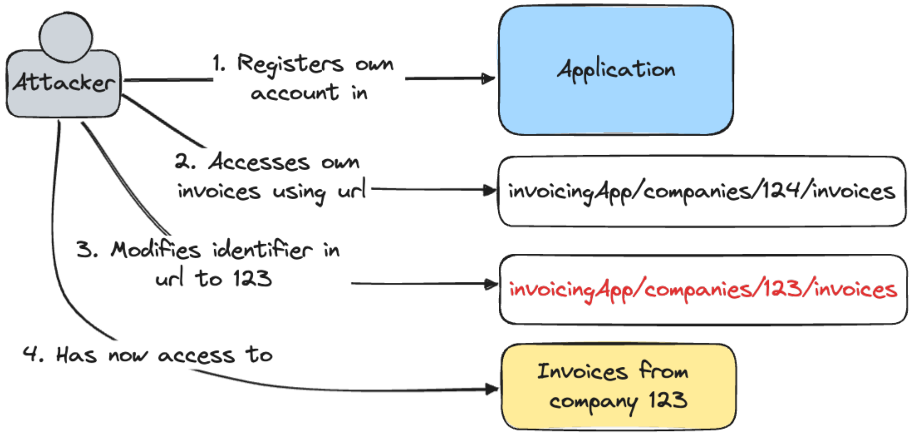

Attacker
1. Registers own account in
Application
2. Accesses own invoices using url
invoicingApp/companies/124/invoices
3. Modifies identifier in url to 123
invoicingApp/companies/123/invoices
4. Has now access to
Invoices from company 123

Figure 211. The attacker replaces his own company id in the URL with the id of another company

[1] https://owasp.org/

Voila! The hacker can now see all invoices from the `123` company.

Why does this happen? Mainly because of the lack of authorization checks. You may think—no, no, there is no way such a thing could happen. You would be surprised how many apps have problems with this. I believe this is often due to the following reasons:

1. **The rapid pace of change**. We are adding many features and trying to satisfy our customers.
2. **Release as fast as possible**. We want to validate the MVP, and there is no time for proper testing. However, we must understand that neglecting authorization checks is not an option, and we need to find a way to balance speed and security.
3. **Lack of skills**. The development team might lack skills in this area, and no one has considered this a threat.

Such vulnerabilities can cause serious problems, including loss of customer trust and potential company bankruptcy.

How can you prevent this? Implement proper access control. Validate each request that comes to the API against the user's permissions. If there is no authorization for the selected resource (another `companyId` data), return `401 Unauthorized`. This way, you can ensure the data is inaccessible to unprivileged people.

A common misconception: Using a more complex identifier like UUID is enough to keep customers safe. While it is hard to guess a UUID, you might still expose it in another area without realizing it.

Imagine that you build a job board. Companies can post job offers and collect applications for job positions. When companies log in, they operate in their context (authorized access), but their `companyId` is exposed in the public portal (anonymous access) when someone accesses a job offer as a potential candidate. This way, a person can read the id, create an account, and then manipulate the URL of another endpoint.

Exposure to DoS and DDoS attacks

When building an API, you need to consider that someone will try to play around with it sooner or later by sending lots of requests. This might happen out of curiosity to see how the API reacts to it or with the intent to make the application unavailable to your customers. Regardless of the intention, it is wise to be prepared and implement defenses against such activities.

If you design the API to handle up to 200 requests per second, it will become immediately unavailable when someone sends 2,000 requests. The system might attempt to auto-scale if configured to do so, but since scaling takes time and another 2,000 requests arrive each second, the API will remain overwhelmed and unavailable.

This kind of attack is known as Denial of Service (DoS) when an attacker sends requests from a single source (e.g., one machine), or Distributed Denial of Service (DDoS) when requests are sent from multiple sources (e.g., multiple machines or devices).

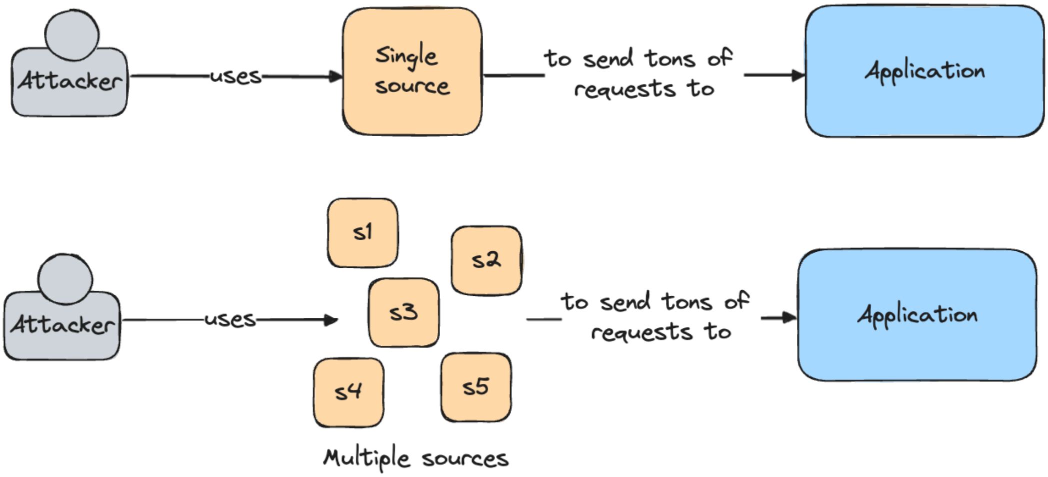


Figure 212. DoS and DDoS attacks

A DoS or DDoS attack causes two significant consequences:

1. **Application unavailability**. Customers cannot access the application or perform any operations. If your application processes financial transactions and earns $10,000 per minute thanks to provisions, you can imagine how high financial losses a 10-minute downtime can generate.
2. **High infrastructure costs**. If you set auto-scaling on your infrastructure without limits, it can significantly increase costs. If someone sends thousands

of requests per second, the system will scale to hundreds of instances or trigger the creation of additional infrastructural components. This could lead to a bill so large it might even shut down your company.

It is worth remembering that denial-of-service attacks are not limited to your API. Hackers can also target other parts of your infrastructure. For example, public storage solutions such as Amazon S3 buckets or Azure Storage can be exploited. Malicious actors may upload vast amounts of data, potentially reaching petabytes, which can severely impact your storage costs and service performance.

Why does this happen? It is usually because there is no defense mechanism against such attacks. In my experience, this often occurs in startups and smaller companies. Large organizations typically have a dedicated infrastructure team(s) that handles these issues. They enforce safety protocols for every new application, and these protective measures are usually already in place.

In smaller companies, only one team might be responsible for everything, from code to infrastructure, security, and observability. Due to this heavy workload, it is easier to overlook setting up proper defenses during the development process.

How can you defend against these attacks? The first step is to set up monitoring in your application to detect an attack in progress. Without proper monitoring, you will only see that the application is down—no details about the characteristics of requests, their number, etc.

Next, you can add rate limiting. It can be done using an additional component, such as an API gateway, and setting it up there. The rate limiter can be set to any value of your choice. For example, if you configure it to allow 100 requests per second, other requests will not be handled. With this, users can still access the application and perform operations. Even if you limit it, requests will continue to execute, but slower. This will give you more time to react, but you will still need other mechanisms.

Finally, use third-party services like Cloudflare[2]. It is super easy to configure and gives you great defense in case of such attacks. An example:

[2] https://www.cloudflare.com/

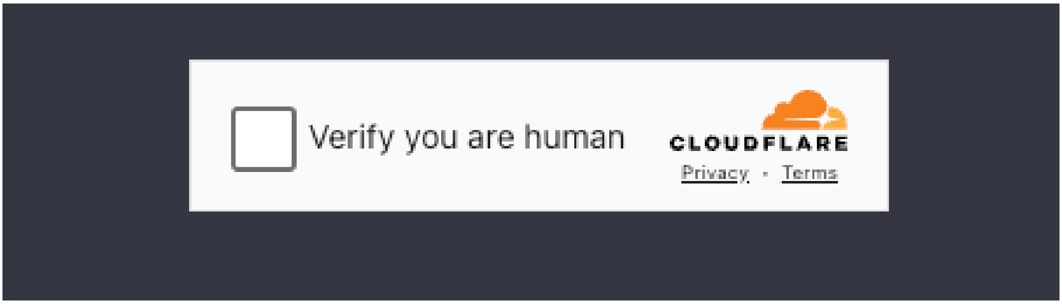


Figure 213. Example protection using Cloudflare

You can also use a Web Application Firewall (WAF) on on-prem servers to filter and block HTTP traffic to and from a web application.

A common misconception: Thinking, *This does not concern us; we are a small company, and our products are not widely known. Who would decide to attack us?*

Unfortunately, this often ends with an unpleasant surprise. No company is too small to be targeted. In recent years, I have seen cases where the victims were niche Internet creators running a blog read by a few hundred people a month and ended up with enormous bills for bandwidth usage.

Unnecessary public endpoints

Suppose you have just built a job board API. Some endpoints are authorized and require a registered account. Some allow anonymous access as they should be reachable by nonregistered users who use the frontend application, e.g., to see the job listings.

The backend handles long and heavy operations on each job list request to gather all the data and prepare the response. Since this endpoint is publicly accessible, the attacker might overuse it and generate extremely high infrastructure costs, especially if your infrastructure scales automatically.

While it is sometimes necessary for endpoints to be publicly accessible (e.g., weather API), leaving them unprotected is usually a significant security risk. Even if an endpoint is intended to be accessed by anonymous users, you can ensure that it can only be called from authorized callers, such as your web or mobile applications.

Why does this happen? My opinion is that our mind works on the principle that public means accessible without any limits, so we don't have to worry about it. I talked to my friends about this, and they confirmed that if something has to be authorized, they focus on it heavily, and if it is public, they treat it as a second-class problem. As a result, we unconsciously create a security hole.

While it may not be as threatening as other vulnerabilities, the risk of not securing public endpoints should not be underestimated. Failure to secure them can lead to significant business disruptions, such as performance issues and increased infrastructure costs, eventually impacting customer satisfaction.

How can you prevent it? One of the simplest ways is to add an API key validation for the public endpoints. This way, each client who wants to connect to this endpoint has to send the key with each request. It can be included in the header as `somename-api-key: your_key`. When the request reaches the API, the key from the header is validated against the one stored, e.g., in a key vault. If they match, the client is allowed to proceed. Otherwise, the request is rejected with `401 Unauthorized`.

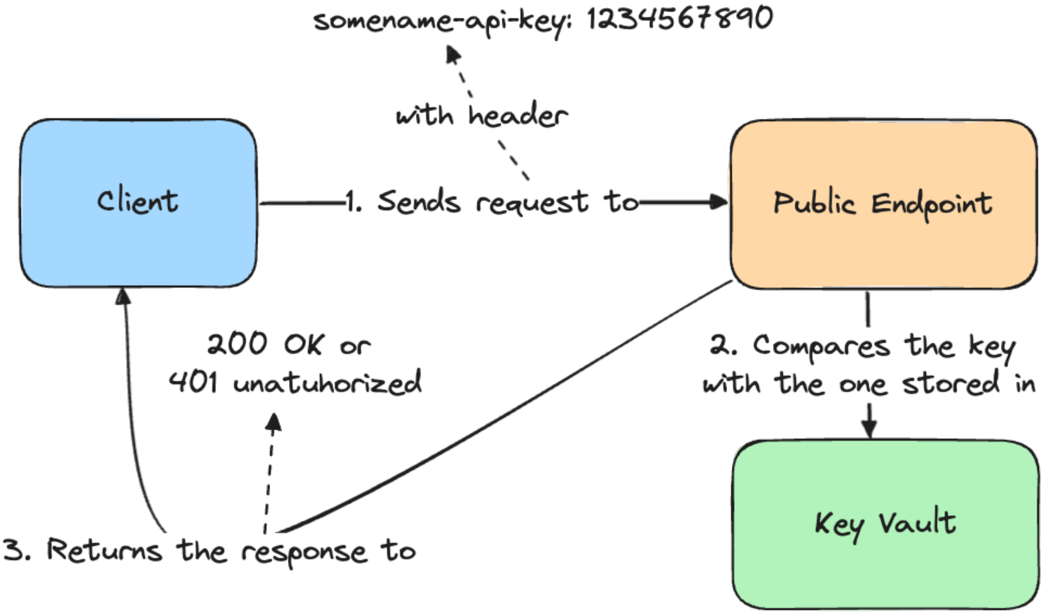


Figure 214. Authorizing access to the public endpoint using API key

A common misconception: Some believe attackers only want to access secured or private endpoints containing sensitive information. However, public endpoints are often targeted for DoS attacks or to exploit business logic flaws. Attackers can use public endpoints to create excessive load, causing performance issues and increased infrastructure costs.

Full path as endpoint parameter to download files

Suppose you have built a product catalog API that allows clients to retrieve a list of available products along with their basic information and photos. Both the API and the photos are hosted on a virtual machine.

One of the endpoints contains a parameter that allows sending the full path to download the photo:

```
products/1/photos?fullpath=C:\products\1\mainphoto.jpg
```

When a GET request is sent to this URL, it downloads the file.

In the meantime, you may have noticed that we have given the entire path to the file. This allows an attacker to attempt to download any file from the virtual machine where the application is located, including configuration files:

```
products/1/photos?fullpath=C:\anypath
```

This, again, has further consequences, as the files may contain information that must not be shared.

Why does this happen? It is caused by a lack of proper validation and restrictions on the file paths accessed through this parameter. Allowing users to directly dictate the file path opens up the possibility for directory traversal attacks, where an attacker can manipulate the input to access files outside the intended directory.

How can you prevent it? There are several measures you can take. First, you can configure the endpoint to return the photo using an identifier without passing the path as a parameter:

```
products/1/photos/1
```

This way, the request can only query the existing photos. Retrieving files from paths other than the one configured for photos is impossible.

Additionally, you can store photos and other files in an external storage service such as Amazon S3, Google Cloud Storage, or Azure Blob Storage rather than directly on the virtual machine. This not only separates the file storage from your application server but also provides built-in access control and various security features.

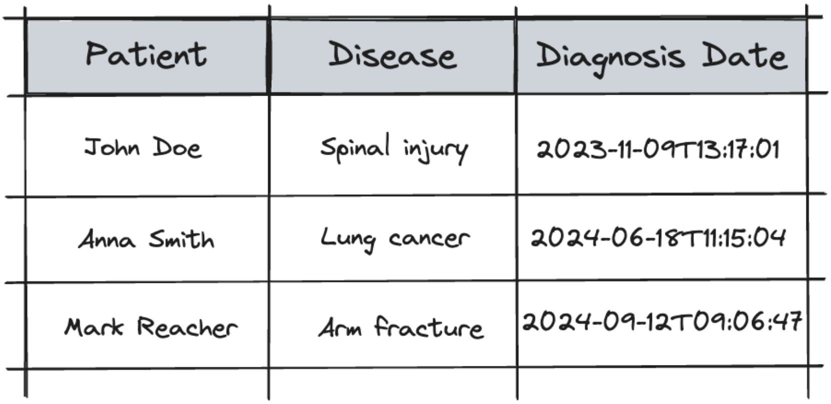
A common misconception: Assuming that the file retrieval mechanism can be universally implemented for all components. On the one hand, this approach introduces logic that can be used anywhere to retrieve the file. On the other, it creates a security hole that can be used to download any file from the host.

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.

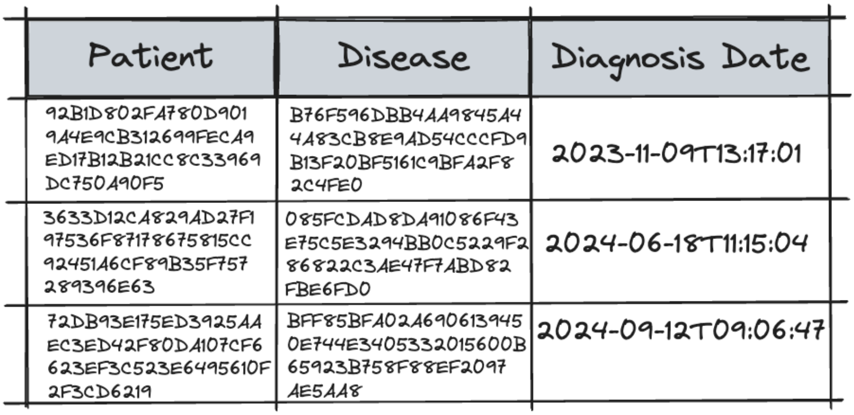


Patient	Disease	Diagnosis Date
John Doe	Spinal injury	2023-11-09T13:17:01
Anna Smith	Lung cancer	2024-06-18T11:15:04
Mark Reacher	Arm fracture	2024-09-12T09:06:47

Figure 215. 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	B76F596DBB4AA9845A44A83CB8E9AD54CCCFD9B13F20BF5161C9BFA2F82C4FE0	2023-11-09T13:17:01
3633D12CA829AD27F197536F87178675815CC92451A6CF89B35F757289396E63	085FCDAD8DA91086F43E75C5E3294BB0C5229F286822C3AE47F7ABD82FBE6FD0	2024-06-18T11:15:04
72DB93E175ED3925AAEC3ED42F80DA107CF6623EF3C523E6495610F2F3CD6219	BFF85BFA02A6906139450E744E3405332015600B65923B758F88EF2097AE5AA8	2024-09-12T09:06:47

Figure 216. 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.

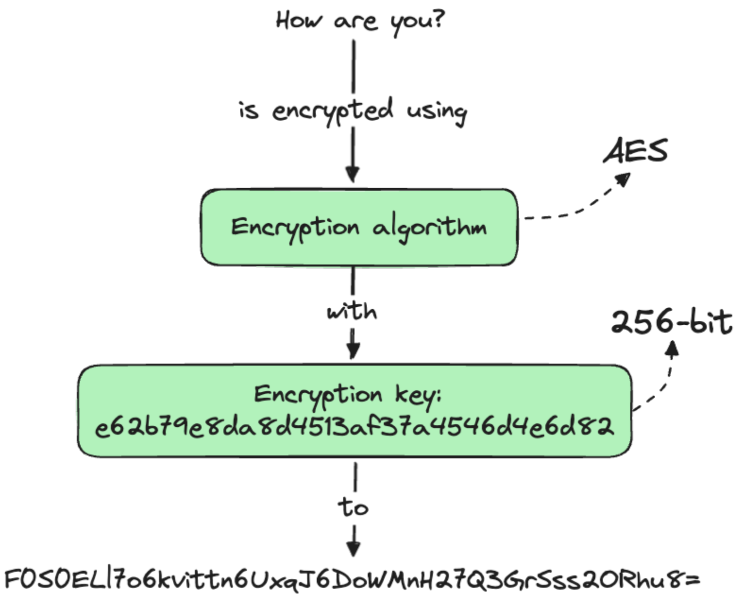


Figure 217. Encryption of a plain text to a cipher text

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.

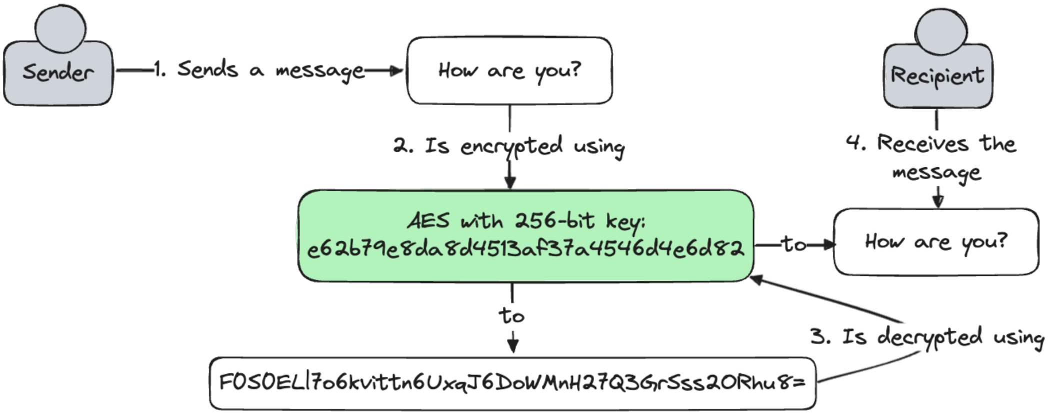


Figure 218. Example of the flow of encryption and decryption using AES and 256-bit key

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
```

ℹ 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.

ℹ 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

The vast majority of applications you are likely to deal with contain logs. We incorporate them throughout the application to help monitor performance and find issues.

Consider this scenario: During the implementation of an application, we ensured that our code was secure and adhered to industry standards. We made the endpoints inaccessible from the outside and encrypted sensitive data so that even if someone gained access to the database, they could not read the information. When returning readable information to an authorized user, we decrypted the data. Everything worked seamlessly.

However, several weeks later, someone encountered an issue with the application and added additional logging. Unknown to them, this logging captured sensitive decrypted data. The logs were archived in a file and stored on the server.

It turned out that the server was not well-secured. An attack occurred, and all log files were stolen. The attacker reviewed the stolen logs and discovered valuable information about payments, including credit card numbers and CVV codes:

```
{
    "Timestamp": "2024-05-27T12:34:56.789Z",
    "Level": "Information",
    "MessageTemplate": "Payment {PaymentId} for user {UserId} completed\
          successfully.",
    "Properties": {
        "PaymentId": "fd3c8269-95f6-40d8-836f-8813b7212c1c",
        "UserId": "428152aa-e3e1-4241-873a-6ba86df67744",
        "Email": "johndoe@example.com",
        "CreditCardNumber": "4111111111111111",
        "ExpirationDate": "12/26",
        "CVV": "123"
    }
}
```

From that point on, the attacker could use the card numbers and CVV codes to make unauthorized purchases. This is a serious issue!

Why does this happen? Often, it is due to a lapse in attention. When a problem occurs in production, especially a critical error, we aim to fix it as quickly as possible. Stress can lead us to take shortcuts, such as temporarily exposing sensitive data.

Once the problem is fixed, the immediate stress may subside, but if the exposed data is forgotten, it can have severe consequences. Even though the data is encrypted in the database, any operation involving sensitive details could be logged, leading to potential data breaches and compromising the security of your system.

How can you defend against it? Ensure you do not log any sensitive data. If logging such data is unavoidable, make sure to encrypt it before storing it.

Additionally, while adding new logs, having another person verify them is a good practice. This can be done during pair programming sessions or code reviews.

A common misconception: Only the development team and potential auditors are interested in logs. In reality, logs are a valuable source of information for potential attackers. They can provide insights into your customers’ data and your system. It is important for everyone to understand this and take the necessary precautions.

Often, logs are treated as a secondary concern—we collect and record them but neglect their security. This oversight makes it easier for attackers, as they only need to steal the log files to gain access to sensitive information.

Supply chain attack

When building software, like it or not, you depend on third-party packages because you do not want to develop everything from scratch. These packages are part of your software supply chain, which consists of your code, its dependencies, and so on. Typically, you add them using package managers like NPM, NuGet, or others.

However, using third-party packages introduces a potential vulnerability in the form of a supply chain attack. Unlike the code you develop, which you track and monitor, you have limited control over the code in the packages you reference unless you actively participate in their development. This reliance on external vendors—whether open-source or commercial—means that changes in their code may go unnoticed, increasing the risk of security breaches.

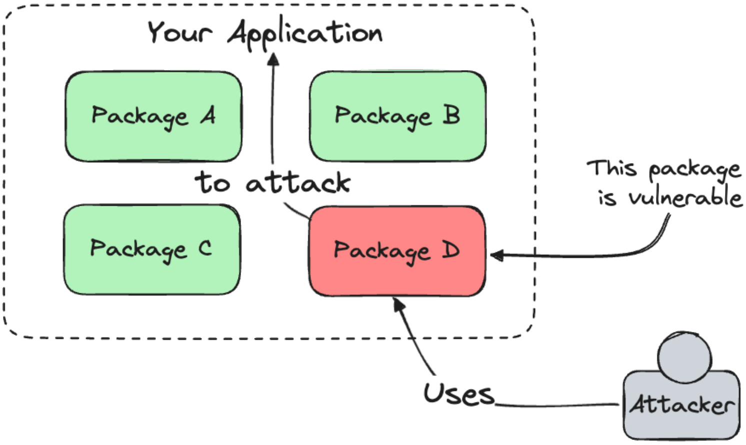


Figure 219. An example of the supply chain attack through a vulnerable package

For example, suppose you decided to use an open-source library for API endpoint parameter validation instead of implementing it from scratch. At the time of integration, the library was at version 1.0.0, which you validated and found to be reliable. It was used by hundreds of thousands of projects and appeared to function correctly.

Imagine a scenario where a major security hole is discovered in version 1.0.0 of the library. This vulnerability is patched in version 1.0.1, but if your development team fails to notice the update, your application remains exposed. Consequently, an attacker could exploit this known security hole to breach your system.

To mitigate such risks, it is essential to regularly update your packages. Providers frequently release updates to fix bugs, optimize performance, and introduce new features. However, new versions can also introduce vulnerabilities. For instance, an update to version 1.9.1 might inadvertently introduce a new security hole. If these problems go unnoticed, your system remains at risk.

It is also important to watch out for "dead projects"—libraries that are no longer actively maintained or quickly become obsolete. These libraries are more likely to contain security vulnerabilities. If you identify a dead project, replace it with an alternative as soon as possible. If no suitable alternatives are available, consider implementing the functionality yourself to maintain full control. In some cases, you might also fork the dead project and continue its development internally within your company.

> Be cautious if a package that hasn't been updated for several years suddenly receives a new update. This could indicate potential malicious intent. In such cases, it is wise to observe the situation and verify the legitimacy of the update before incorporating it into your system.

Why does this happen? Such attacks occur because the security of the software is only as strong as the weakest link in the supply chain. Without proper monitoring, we might miss key updates and security patches. Sometimes, even with monitoring in place, we might miss notifications about updates or underestimate the significance of changes. Additionally, we sometimes delay updates, particularly when a major version change requires extensive code adjustments.

How can you defend against it? One effective way to monitor external dependencies is to use automation tools like Dependabot in GitHub. Dependabot can track dependencies, notify you of updates, and even create pull requests with new package versions.

This approach helps keep your packages up-to-date, but it is still important to review each package individually to ensure it won't break your code. If an update is known to cause issues, plan and execute the update as soon as possible, especially if vulnerabilities were present in the previous version.

A common misconception: Once a stable version of a package is installed, it is secure forever. But what was safe at the time of installation may no longer be safe in the future. This is why regular updates and monitoring are so important for maintaining the security of your software.

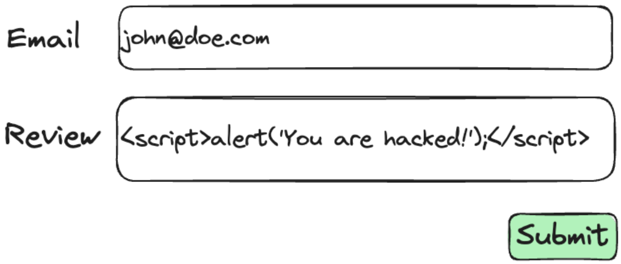
Cross-Site Scripting (XSS)

Imagine you have built an e-commerce web application that allows customers to buy products and leave reviews for their purchases.

The review can be typed into a text field and, once submitted, is stored in a database. However, the text is neither validated nor encoded.

Other users can view these reviews immediately on the product page, where many potential customers rely on them to decide whether to purchase the product.

An attacker exploits this by entering a malicious script into the review text field instead of regular text. For example, they might insert a simple script that displays an alert on the page.

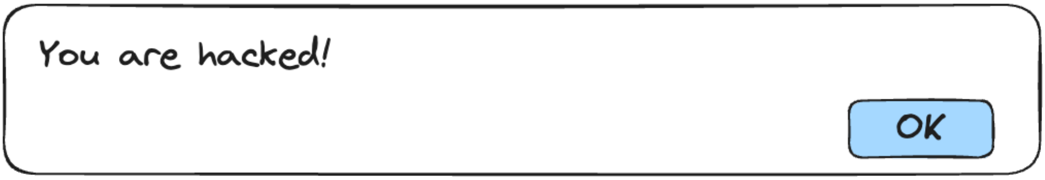


Email john@doe.com

Review <script>alert('You are hacked!');</script>

Figure 220. Attacker adds a malicious script as a review

The attacker submits the review with the malicious script. Since the review text is neither validated before storage nor encoded when displayed to customers, the script is executed every time someone views the product reviews page. Every customer will see an alert box.



You are hacked!

Figure 221. The simplest example of XSS attack with alert box

Of course, this is a trivial example, and it is unlikely that anyone will use this attack to show the alert box. What else can happen?

The attacker could embed a script that secretly interacts with the user’s browser to redirect them to malicious pages. These pages might prompt the user to enter sensitive information, such as banking details or social media credentials. The victim would only need to visit the page with the malicious script—such as the legitimate product reviews page—without any obvious indicators or alerts that something is wrong.

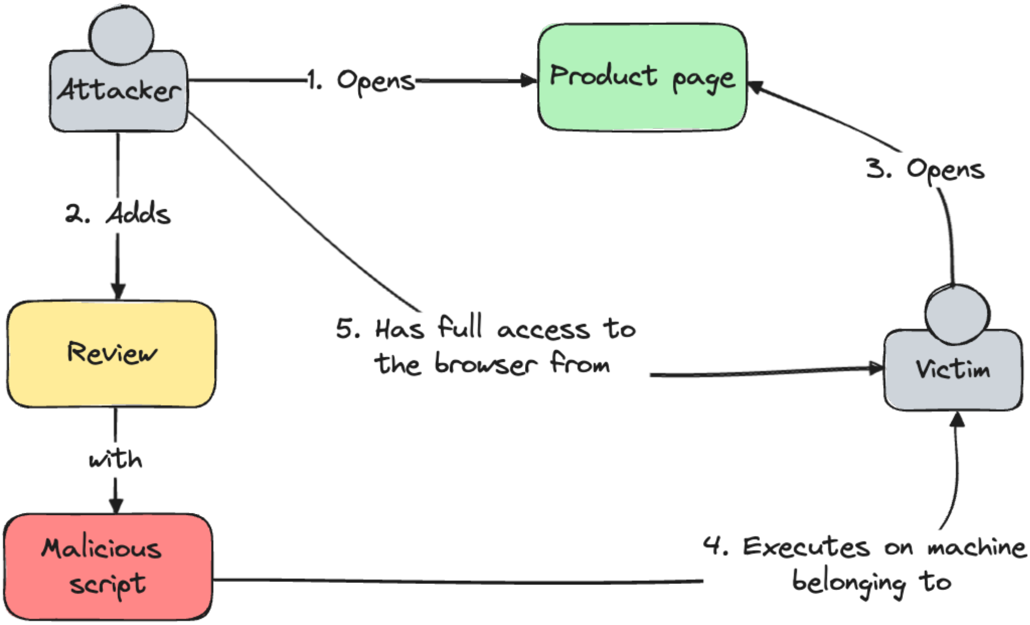


Figure 222. The flow of adding harmful script and executing it in the victim’s browser

From then on, the attacker has complete control over the victim's browser. After the user enters their data, the attacker immediately sees it and can use it to wipe out all the money from the account or steal their identity.

It is important to understand that XSS (Cross-Site Scripting) attacks come in different categories. The most common types are:

1. **Reflected XSS**. The attacker prepares a malicious link that contains the XSS payload like `http://yourApp/search?query=<script>alert('You are hacked!')</script>`. This link might be sent to the victim via SMS or social media with a message like *Your package was lost. Please click on the link to take action.* Alternatively, the link could be embedded in a page, enticing the victim to click it.
2. **Stored XSS**. Similar to the review example described earlier, the attacker injects a malicious script into a data field (e.g., a review), which is then stored on the server. The script executes every time a user visits the page containing the stored script. This type of XSS is more dangerous because it doesn't require the victim to click anything; simply visiting the compromised page is enough.

Why does this happen? There are several reasons for XSS attacks. First, end users often trust the information they see. For instance, if they receive a message stating that a package has been lost and are asked to click on a link, they may do so without hesitation. This trust can be exploited by attackers to gain control of their browsers.

The second reason is the presence of insufficient security features in the application. This could be due to a lack of expertise or penetration testing on the vendor side. These gaps expose users to unnecessary risks and can lead to severe consequences.

How can you defend against it? There are several ways to protect your customers from XSS attacks.

First, you should validate all the inputs the user can define. The validation rules should be as strict as possible in each case.

Next, encode the output before rendering it on the web page. This way, your browser won't interpret it as a code. For example? The following text was stored in the database:

```
<script>alert('You are hacked');</script>
```

Luckily, it was encoded before rendering the page to the user, and the text looks like this:

```
&lt;script&gt;alert(&#39;You are hacked&#39;);&lt;/script&gt;
```

As a result, no malicious script will be executed, as the browser won’t be able to interpret it as such.

A common misconception: Placing too much trust in users. It is especially noticeable when building internal applications for organizational use.

After all, all users are our employees, aren’t they?

Of course, they are. However, their accounts can be compromised, laptops stolen, or they may not have the best intentions. Therefore, adopting a zero-trust policy in security practices is key to protecting against potential threats.

SQL Injection

If you were to ask a random person about the most common security vulnerabilities, SQL injection would probably be among the first they mention. This type of attack has long been one of the most widespread and destructive.

Suppose you have developed an API for a banking system. One of the endpoints allows retrieval of all transactions associated with a specific customer. This endpoint is designed to accept a parameter that represents the customer’s unique identifier, defined as `customerId`: `GET /transactions?customerId={customerId}`. Inside, it calls a function that concatenates the SQL query with a given `customerId`:

```
function getCustomerTransactions(customerId) {
 var query =
 "SELECT * FROM transactions WHERE customerId = '" + customerId + "';";

 var results = executeQuery(query);

 return results;
}
```

Everything works properly for several months with no issues. Suddenly, an attacker tries to break into your system. They can see the endpoint to retrieve customer transactions and the possibility of passing their id. Since the `customerId` is a string, the attacker decides to pass the following value:

```
"12345'; TRUNCATE TABLE transactions; --"
```

What happens in this case? Here is the breakdown:

1. `12345'` closes the original string literal.
2. `; TRUNCATE TABLE transactions;` executes an additional SQL command to remove all entries from the transactions table.
3. `--` comments out the rest of the query to avoid syntax errors from the trailing ’.

As a result, the passed value alters the SQL command structure (not only the query parameter):

```
SELECT * FROM transactions WHERE customerId = '12345';TRUNCATE TABLE tr\
ansactions; --'
```

This malicious action wiped all transactions from the database, which is a critical issue, especially if you don’t have a backup.

But that’s not all. The attacker could also attempt to pull another customer’s data or inject a query like:

```
"12345'; SELECT * FROM otherTable; --"
```

This would allow them to read data from another table, further compromising your system’s security.

Why does this happen? One of the most common mistakes is taking a parameter from the input and concatenating it directly with an SQL query. Unfortunately, this is usually caused by a lack of knowledge rather than a simple oversight.

How can you defend against it? One of the simplest and most effective ways to protect against SQL injection is to parameterize the query. This ensures that the input is always treated as an SQL parameter rather than executable code. Here’s how you can do it:

```
function getCustomerTransactions(customerId) {
 var query = "SELECT * FROM transactions WHERE customerId = ?";

 // Use a function that supports parameterized queries
 var results = executeQuery(query, [customerId]);

 return results;
}
```

Thanks to a parameterized query, the passed parameter cannot alter the SQL structure. The SQL execution will look as follows:

```
SELECT * FROM transactions WHERE customerId = '12345'';TRUNCATE TABLE t\
ransactions; --';
```

Since the input is appropriately escaped, the semicolon and double dash are not treated as SQL command separators or comments, and the input is just a string value inside the query.

A common misconception: Using stored procedures is enough to protect against SQL injection. If they also concatenate user input with the SQL code, the same problem will happen as in the case of standard queries. Therefore, it is important to remember to use parameterization in stored procedures as well.

Misconfiguration of infrastructure

Incorrect infrastructure configuration is a common issue, whether the environment is fully automated (infrastructure as code) or involves manual or semi-automated processes.

Imagine you have built a cloud-based application that functions like a drive, allowing users to store various files—images, videos, and more. This application uses cloud storage services, which are configured with default public access. The only requirement for connecting to this storage is an access key.

No one notices the public access setting. One day, someone copies the access key and sends it to a colleague over chat. Unfortunately, this colleague is traveling by train, and an attacker sees the access key on the laptop screen and takes a photo. Later, the attacker uses the key to upload a mass of heavy files, generating high costs.

In another instance, you and your team decide to create a virtual machine for internal purposes. However, no one notices that Remote Desktop Protocol (RDP) access is not restricted, meaning anyone on the internet can access it. Depending on the information stored and the privileges granted, unauthorized access to the virtual machine can cause significant harm.

The same risks apply to every component of your infrastructure. One misconfiguration can create a security hole that attackers will exploit.

Why does this happen? If the infrastructure is set up manually or semi-automated, the risk of errors is much higher than in automated environments. This is because human errors, such as forgetting to click something, entering incorrect configurations, or ignoring certain steps, are common and can go unnoticed.

When issues arise with automated infrastructure misconfiguration, it might be because static code analysis and other infrastructure-watching tools that can be connected to the pipeline are missing.

How can you defend against it? In a fully automated environment, one of the easiest ways is to connect an analyzer to your pipeline. This way, if any infrastructure configuration is incorrect—note that it might change over time—it will break the build, and you will get information about it.

For manual infrastructure setup, I recommend planning and gradually implementing step-by-step automation. Until then, involve at least two people to work in a pair programming (four-eyes checking) manner when applying any changes. While this is not a perfect solution, it reduces the likelihood of mistakes.

A common misconception: Assuming that default configurations from providers are secure. The problem is that default settings often simplify infrastructure usage as much as possible instead of focusing on maximum security.

Bonus: Exposure of passwords

Yes, I know—there has already been a lot of discussion about passwords. However, I still encounter this issue, particularly with legacy apps, which is why I have decided to include it in the security section of this book as a bonus. :) There are two common cases that I encounter with passwords:

1. Stored as a plain text.
2. Stored as an encoded text, e.g., using Base64.

The first problem is obvious. The user sets the password during registration, which is stored in the database as plain text. Anyone with access to the passwords table can easily read it—not only attackers but also maintainers. If the user uses the same password for different sites, an attacker can use this information to access other portals that the user frequents.

The second one is related to using encoding instead of hashing. Encoding is a great way to transform a text from one format to another, simplifying communication between different systems. The problem is that it is designed to be reversible without needing a key. One well-known encoding method is Base64. Here's an example of an encoded password:

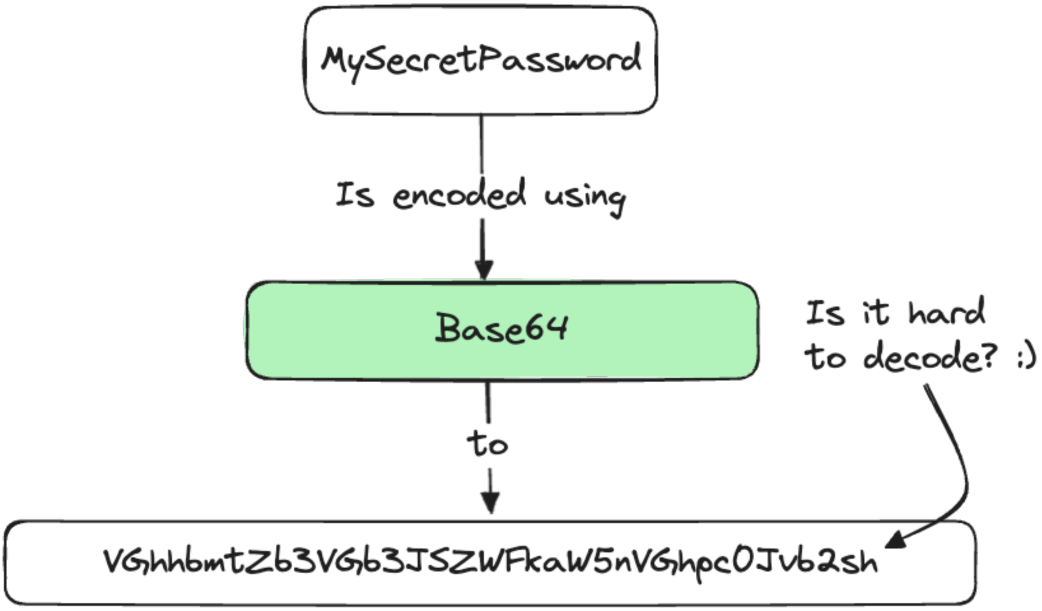


Figure 223. Example of the encoded password - it is super easy to decode

The main problem is that encoded data can be easily decoded without any secret key. When an attacker sees a password that is Base64 encoded, they can decode it within seconds. So, it does not improve the situation.

It does not matter if the password is stored using plain text or encoded. The attacker can easily read both.

Why does this happen? It is often due to a lack of knowledge. Many people do not understand the differences between encoding, encryption, and hashing. As a result, they use inappropriate methods for securing sensitive data, such as passwords.

How can you defend against it? You can either use an off-the-shelf authentication solution (recommended), where you delegate the entire registration and login processes (like Auth0 or the ones offered by cloud providers), or make sure that your passwords are hashed with an algorithm that is currently considered safe (this changes over time).

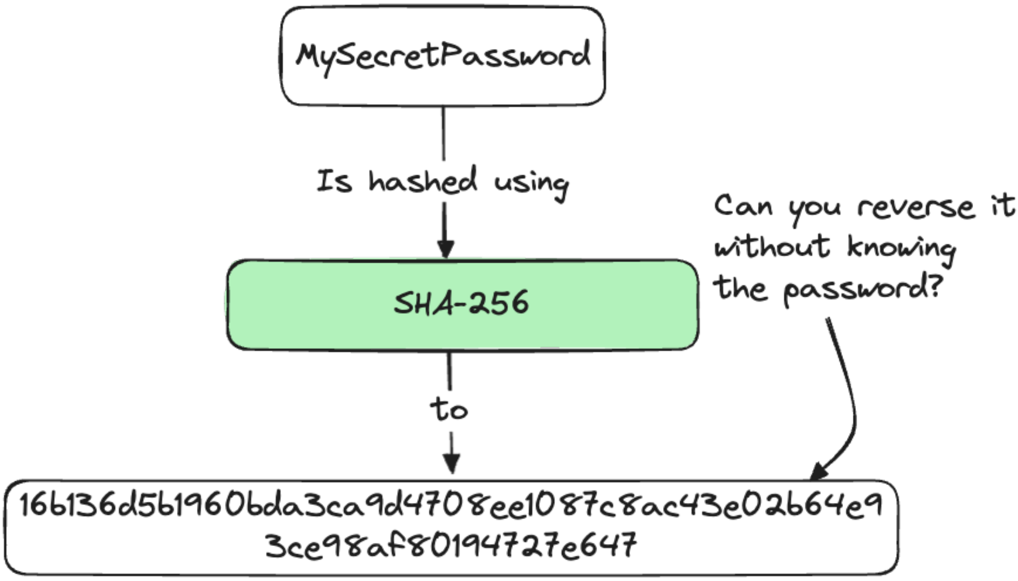

MySecretPassword
Is hashed using
SHA-256
to
16b136d5b1960bda3ca9d4708ee1087c8ac43e02b64e9
3ce98af80194727e647
Can you reverse it without knowing the password?

Figure 224. Example of the hashed password using SHA-256

Still, you must remember that someone might guess a password hashed with the above SHA-256 algorithm. How is this possible if it is irreversible?

While hashing is irreversible, attackers can still attempt to guess the original text through various methods:

1. **Dictionary attacks**. Hackers use lists of common words, passwords, and phrases.
2. **Rainbow tables.** Pre-computed tables of hash values for common strings.

When a database of hashed passwords is compromised, attackers compare these hashes to each word (hashed before comparison) from the dictionary or to their pre-computed values. If a match is found, they have discovered the user’s password.

Simple or common passwords (such as “qwerty,” “password1,” or “admin1”) are particularly vulnerable because their hashes are often included in these pre-computed lists. In addition, if only basic hashing is used without additional security measures, finding one user’s password allows attackers to quickly identify other users with the same password by searching for matching hashes.

To address this issue, a good practice is to use salts when hashing passwords. This involves generating a unique value, called a salt, and appending it to the password before it is hashed. This approach ensures that each password has a unique hash, even if the original passwords are the same.

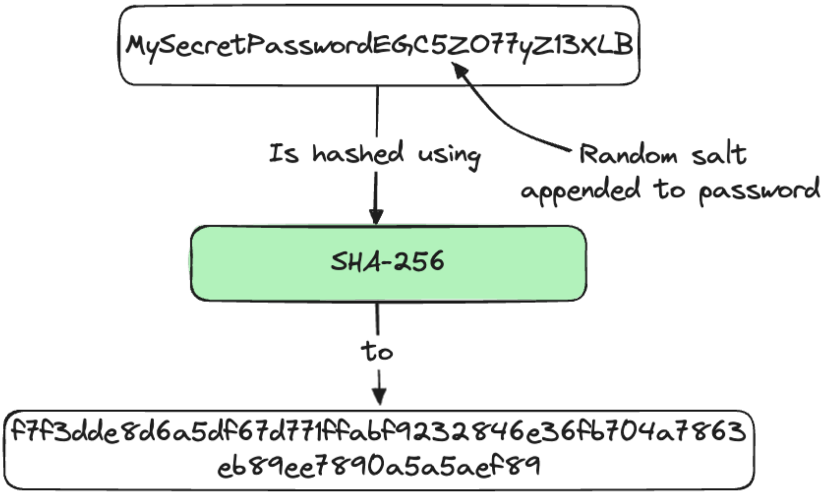

MySecretPasswordEGC5Z077yZ13XLB
Is hashed using
Random salt appended to password
SHA-256
to
f7f3dde8d6a5df67d771ffabf9232846e36fb704a7863
eb89ee7890a5a5aef89

Figure 225. Example of salting the password

Suppose that two users, John and Anna, have the same password: `password1`. Without salt, when the attacker gets the database, the passwords will be the same:

John: `0b14d501a594442a01c6859541bcb3e8164d183d32937b851835442f69d5c94e`

Anna: `0b14d501a594442a01c6859541bcb3e8164d183d32937b851835442f69d5c94e`

However, if you generate a unique salt for each user, for example, `BhBuNist41F6PXa` for John, and `w7ROQzE9dniLkgX` for Anna, and append it to the password: `password1BhBuNist41F6PXa` and `password1w7ROQzE9dniLkgX`, after hashing (using SHA-256), the hashes will look as follow:

John: `f4153a042e672a42304563940df6b1ad600859b703fc6c6f1fdbab0977267ab2`

Anna: `4b78b417c69bbea2df5b7405c6ecc78cce920244333f2b03cdae06f6462deff0`

Even though both users have the same passwords, the attacker cannot know this. This method also adds an additional security layer, even if users choose weak passwords found in publicly available dictionaries.

> If one of the most common passwords is hashed using SHA-256, the result is `65e84be33532fb784c48129675f9eff3a682b27168c0ea744b2cf58ee02337c5`. Can you guess what the original text is? This exercise highlights the potential for password guessing.

A common misconception: There is confusion between encoding, encryption, and hashing. Too often, we trust that it is already safe if something has turned into a magical string of characters. In the case of hashing, we create an irreversible set of characters; in the case of encryption, we can decrypt the cipher using a secret key. In the case of encoding, text is easily reversible.

Luckily, these issues are becoming less common due to the availability of off-the-shelf solutions, but they are still a big problem in some legacy systems.

Recap

As you can see, there are numerous potential vulnerabilities, and it is impossible to address them all single-handedly. That is why sharing knowledge about these vulnerabilities within development teams is so important.

Sometimes, the pressure to implement new features and the demand for quick releases can be intense. However, you must remember that security should never be treated as a second-class problem. Take the time to pause and reflect when necessary.

Finally, I recommend automating processes like infrastructure setup, integrating security tools into existing pipelines, conducting regular security scans, and occasionally performing penetration testing. This can significantly reduce the risk of vulnerabilities. Remember, though, that no system is entirely invulnerable.

Consider exploring additional relevant resources to deepen your understanding of security, such as:

1. OWASP for mobile appliactions (MAS)[3]
2. Common Weakness Enumeration (CWE)[4]
3. National Institute of Standards and Technology (NIST)[5]

[3] https://owasp.org/www-project-mobile-app-security/
[4] https://cwe.mitre.org/
[5] https://www.nist.gov/cybersecurity

Epilogue

Endings are tricky, and I have never been good at them. After sharing all these thoughts and experiences, it is tempting to simply say, “That’s all, folks!” and call it a day. But let’s give this epilogue a proper go.

In the fast-paced world of tech, software architecture stands out as a field of endless innovation and change. If I had attempted to cover every detail, this book would have been a never-ending story and still somehow incomplete. New technologies emerge, fresh ideas sprout, and best practices shift with each passing day.

So, what was the real purpose of this book? It certainly wasn’t to provide an exhaustive encyclopedia of software architecture. Instead, my goal was to equip you with fundamental knowledge and offer a practical view of a software architect’s work. I wanted to pull back the curtain and show you what this role really entails.

The insights, patterns, and experiences shared here were meant to build that foundational understanding that is frequently missing. Think of this book as your map and compass.

Also, remember that finding valuable information is not easy. It takes time and effort. That’s why I have mentioned various experts throughout the book. These people really know their stuff:

- Vaughn Vernon and Vlad Khononov have a lot of knowledge about DDD.
- Oskar Dudycz is a recognized authority in the field of Event Sourcing.
- Sam Newman is the go-to guy for microservices.
- Kamil Grzybek is an expert on modular monoliths.
- Milan Jovanović knows tons about .NET and C#.

However, always approach what you read or hear with a critical mind, regardless of the source. Even experts can have biases or blind spots, and the field of software architecture is too dynamic for any single perspective to be definitive.

Throughout the book, I shared ten important lessons with you. These were not just random ideas. They are principles that I try to follow every day.

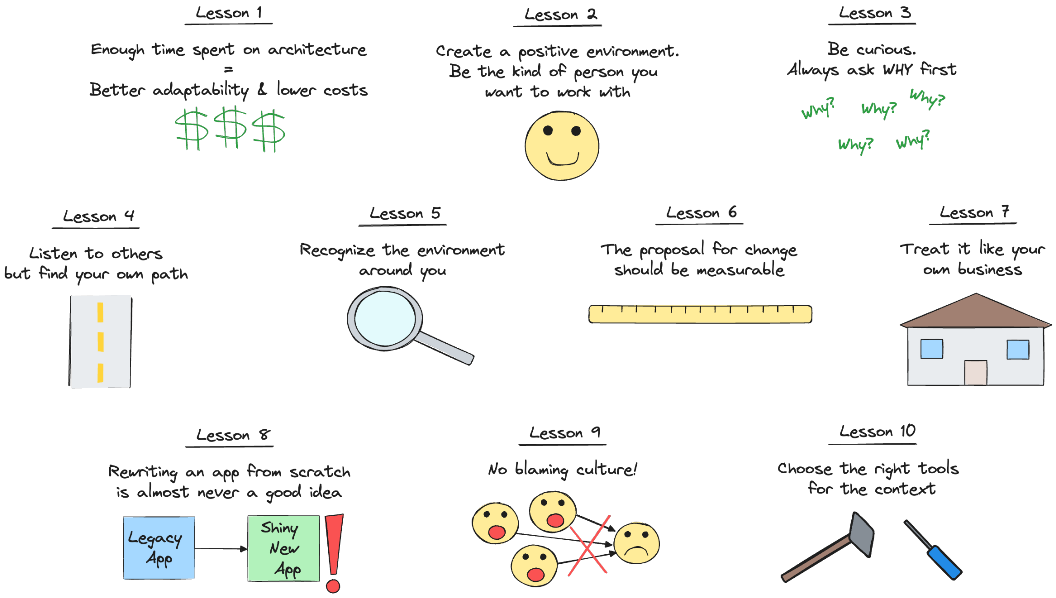

Lesson 1
Enough time spent on architecture = Better adaptability & lower costs
$$$
Lesson 2
Create a positive environment. Be the kind of person you want to work with
Lesson 3
Be curious. Always ask WHY first
Why? Why? Why? Why? Why?
Lesson 4
Listen to others but find your own path
Lesson 5
Recognize the environment around you
Lesson 6
The proposal for change should be measurable
Lesson 7
Treat it like your own business
Lesson 8
Rewriting an app from scratch is almost never a good idea
Legacy App
Shiny New App
!
Lesson 9
No blaming culture!
Lesson 10
Choose the right tools for the context

Figure 226. The summary of all of the lessons (my code of conduct)

1. **Enough time spent on architecture = better adaptability & lower maintenance costs**. Think of the software architecture as the backbone of everything. Sure, it may feel like you spend much time on it upfront, and you might face some pushback or time pressure. But trust me, it is worth it. A well-thought-out architecture makes your software way easier to adapt and way cheaper to maintain down the road. It is like building a house—get the foundation right, and everything else becomes much smoother. So stand your ground and make time for suitable architecture. Your future self (and team) will thank you.
2. **Create positive environment. Be the kind of person you want to work with**. Be the colleague you would love to have—the kind of person who is always ready to lend a hand. Stop acting grumpy and micromanaging—nobody likes that! Instead, create a space where everyone can flourish, from the newbie to the senior. Treat everyone as equals, regardless of their title or experience. Remember, your role is not about bossing people around but lifting others up.

3. **Be curious**. **Always ask WHY first**. Even if a decision seems set in stone, your questions might uncover something crucial, so don't be afraid to ask. Think of yourself as a friendly interrogator. Your curiosity is not just welcomed; it is your secret superpower. Who knows? Your "why" today could be the company's "whew" tomorrow.
4. **Listen to others but find your own path**. The people around you have probably been through a lot. They have stories and lessons that could save you a lot of trouble. So, take advantage of their experience and learn from their wins and oops moments. It is easier than making all those mistakes yourself. But here is the thing—while you are soaking up all this wisdom, remember who you are. Use what you learn as a starting point, and then put your own spin on it.
5. **Recognize the environment around you**. Take a moment to look around. Who are the people you work with? Grab a cup of coffee with them and chat around the water cooler. Try to understand how things work in your organization. How are product decisions made? How is the infrastructure set up? Do you push code into production every day or once in a blue moon? What is the budget situation? The more you know about your work environment, the better equipped you will be to navigate it.
6. **The proposal for change should be measureable**. It is exciting to join a new team and make a difference. You have fresh ideas and want to show what you can do. But wait a minute - change for change's sake is not helpful. Instead, ensure that any proposed change has a clear, measurable benefit. Want to replace an old component? Great, but be prepared to explain precisely what you will gain. Maybe it will reduce maintenance costs by 15% per month? Having hard numbers helps everyone understand why the change is worthwhile.
7. **Treat it like your own business**. When you build an application for yourself, it changes the way you think, right? When you consider adding that fancy new component, you probably ask yourself, "Is it really worth doubling the infrastructure costs?" You will care more about each decision because it feels personal and touches your own wallet. It is all about making smart choices. This mindset will help you balance cool technology with your current needs.
8. **Rewriting an app from scratch is almost never a good idea**. We have all been there. You are working with a legacy application, and it feels like a tangled mess. The temptation to throw it all out and start over is real. "This time," you think, "we will use the coolest technology, and it will be perfect!" But hold

on a second. Writing from scratch is like opening Pandora's box-you never know what will happen. The thing is, these old systems have years of hidden knowledge baked in. The people who built them may have left, taking their insights with them. And let's not forget those sneaky "it's always been done this way" processes that no one bothered to write down. Put these all together, and you have a recipe for disaster.

9. **No blaming culture**. Finger-pointing doesn't fix anything. In fact, it usually makes things worse. Remember, we are all in this together. When something goes wrong, it is not "Mike's fault" or "Susie's mistake" - it is our challenge as a team. If things go wrong in production, don't waste time playing detective to find the culprit. Instead, get everyone together for a post-mortem and figure out how to prevent it from happening again. Focus on solutions, not on who messed up.
10. **Choose the right tools for the context**. When choosing tools for your project, focus on what you need right now, not some distant future. Going for fancy tech that promises to handle 100,000 users is tempting, but if you only serve 1,000, that is overkill. Instead, pick tools and components that work great for your current situation. But here is the key: while solving today's problems, keep an eye on flexibility. Choose solutions that won't box you in later.

These lessons have helped me throughout my career, and I hope they can help you, too. Think of them as friendly advice from someone who has been around the block for a while. Feel free to use what works for you, adapt what doesn't quite fit, and always trust your own judgment.

As you finish this book, remember that this is not the end of your learning. Software architecture changes, and there is always something new to discover. Stay curious, be willing to adapt, and don't be afraid to ask questions.

Your journey as a software architect is unique. I hope this book helps you grow and succeed. The future of software architecture needs curious people like you.

Get out there and build amazing things!

Extra 1: Other Engineering Practices

Working with metrics

I still remember my first days at one of my early jobs—the team, the application I worked on, and my first cup of coffee. But one thing that really stuck with me was the large TV screen in the office of a nearby team. They used it to display everything from environment monitoring and football games to metrics showing the average time it took them to resolve ticket requests. I was genuinely impressed by the latter.

A few weeks later, things took a turn. Customers began reporting that their tickets were taking too long to resolve. However, thanks to their metrics, this was the only team in the organization that had a solid defense. In just three seconds, they could prove that their tickets were regularly resolved, and they could show the average resolution time clearly.

What they did was quite innovative then and was later copied by other teams.

As software engineers, one of the worst things we can do is base our decisions solely on gut feelings. Sometimes we may succeed, but other times we may fail. Moreover, if we need to persuade others, our decisions need a solid foundation. Metrics provide that foundation.

Numbers don't lie. They are facts, and we can draw conclusions based on them. Metrics are a priceless tool that we should fully utilize.

An example of such metrics is *DevOps Research and Assessment (DORA).* These metrics help evaluate the effectiveness and efficiency of software development and delivery practices. You can think of them as the KPI for software development. They are divided into four key areas:

- **Deployment frequency**. Measures how often you deploy your code changes to your production environment.

- **Lead time for changes**. Measures the average time from when a developer commits code changes to when the code is released into the production environment.
- **Time to restore service**. Measures the average time required to restore service after a system failure.
- **Change failure rate**. Measures the percentage of failed deployments out of the total number during a specific period.

These four metrics can provide concrete numbers to support your development practices. However, focusing on a single metric can be misleading. Let's take into consideration two teams:

> **Team A**. They release their application to production four times per year.
>
> **Team B**. They release their application to production 1,000 times per year.

If we only look at the change failure rate:

> **Team A**: Achieved four successful deployments, resulting in a 0% failure rate. All planned deployments were successful, indicating 100% effectiveness.
>
> **Team B**: Had 40 failed deployments out of 1,000, resulting in a 4% failure rate. With 960 successful deployments, Team B achieved a 96% effectiveness rate.

It indicates that Team A was more successful, right?

Other metrics can provide additional context. Let's look at deployment frequency:

> **Team A**: Releases to production four times per year, one deployment per quarter.
>
> **Team B**: Releases to production 1,000 times per year, which averages around four deployments per day, assuming they work about 250 days a year.

Team B released their code 250 times more than Team A. While this might seem like a dramatic difference, it doesn't necessarily impact the business outcome—both teams successfully released features.

So, why should a team release more frequently?

Releasing more often allows for incremental updates and immediate feedback. Instead of implementing a complete feature, which might take 1.5 months, Team B could introduce parts of it and validate them based on customer feedback and observations. This iterative approach could reveal that two-thirds of the initial feature scope needed revision, saving half of the planned budget.

The next metric to consider is lead time for changes. There you go. It is starting to work in favor of Team B and, above all, brings tangible benefits:

> **Team A**: Average lead time is one week as most of the tasks are not automated, including resources of testers, release, and system engineers. Additionally, someone from product management has to approve each change.
>
> **Team B**: Average lead time is 15 minutes. Assuming all tasks are automated, it includes only the resources of the infrastructure to run tests, checks, and validations.

One week (multiplied by different resources) vs. 15 minutes. High costs vs. low costs.

The last metric is time to restore service:

> **Team A**: Average time to restore service is three hours. This includes investigating the issue, checking logs, finding the root cause, manually running scripts, and double-checking everything. Such delays can significantly impact customer satisfaction, especially during critical times like Cyber Monday, and can have serious financial repercussions.
>
> **Team B**: Average time to restore the service is three minutes. With full automation, there is no manual intervention, allowing the team to restore service almost immediately, often without any noticeable disruption to users.

Thanks to DORA, we can operate based on facts rather than gut feelings. In discussions with business stakeholders, we can showcase our speed and efficiency, resulting in reduced operations costs. Within the development team, we can clearly identify areas that need improvement.

Vertical slices

What is a vertical slice? Before we look at the definition, let's consider a typical software development process.

Imagine you have a feature to implement—say, a file upload function. Estimates and planning are done for the next few weeks. After 1.5 months of work, you complete the entire feature, which includes:

- Upload of a single file.
- Upload of multiple files.
- Fifteen supported formats, including images and videos.
- Modal for the selection of files to upload.
- Drag and drop for upload of files.
- Support large file uploads (> 1GB).

In theory, everything seems fine.

Analyzing user traffic reveals that almost no one uses the modal to upload files. Additionally, 95% of files are smaller than 1GB, and only a few formats are commonly used (PNG, JPG, JPEG, MOV, and MP4).

What does this mean? You have overdelivered.

The market didn't need all those extra features. Now you have two options. First, keep these features in the application, which means higher maintenance costs. Second, drop them, which means you have wasted resources and money on unnecessary development.

Instead, the upload feature can be approached with vertical slices. Each vertical slice provides an entirely usable portion of the functionality—imagine it as a slice of cake. When you translate each layer of the cake into software development, this piece contains everything needed for the end customer to start using it:

- UI
- API
- Infrastructure (like database, queue, etc.)

This contrasts with the horizontal approach, where we first provide the database, then the API, then business logic, and so on.

Let's use a real example based on the upload feature. Market signals indicate that users want to upload a profile picture within your application. So, you decide to define the first vertical slice:

- Upload of a single, small file (< 100MB)
- Only .PNG, and .JPG formats
- Upload is done through modal.

After development, you are ready to release and validate it. Different scenarios might happen:

1. **Success**. The first users love it, and everything works perfectly. No further action is needed.
2. **Partial success**. In general, it works fine, but uploading through the modal is not user-friendly. Users suggest a drag-and-drop interface instead. It is time to decide what to do. One solution is to replace the modal with drag-and-drop, and another is to extend the current solution. In either case, you have another vertical slice.
3. **Failure**. Market signals were false, and almost no one uses the feature. Just drop it.

In scenarios 1 and 3, you have saved time and resources, allowing you to focus on other features. Scenario 2 requires the development of an additional vertical slice. After implementing the changes, you will need to release the updated product and repeat the validation process. There might be other vertical slices like:

- Upload multiple files (vertical slice number 3)
- Upload .MP4 files (vertical slice number 4)

And so on. This step-by-step approach allows you to incrementally build fully functional pieces of software. Remember, each slice you add should enable the end user to complete a specific task or process.

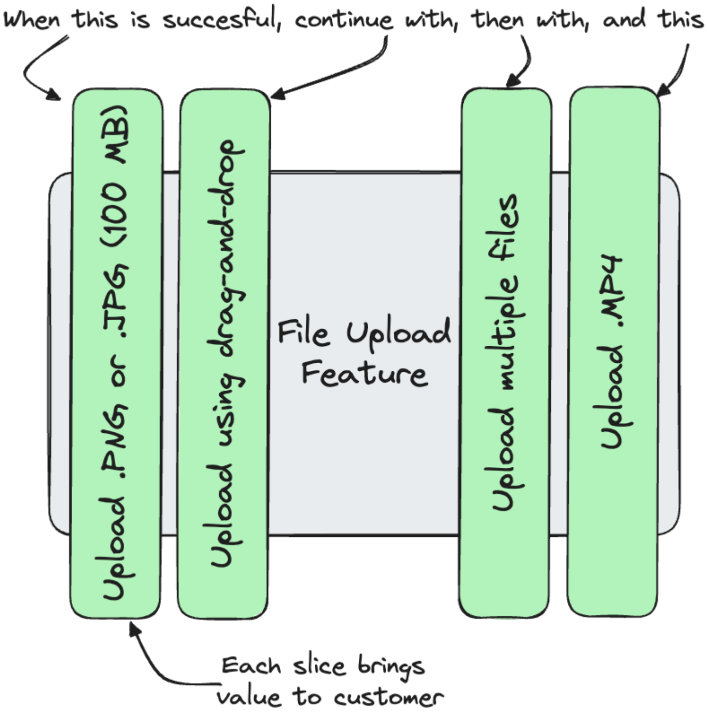

When this is succesful, continue with, then with, and this
Upload .PNG or .JPG (100 MB)
Upload using drag-and-drop
File Upload Feature
Upload multiple files
Upload .MP4
Each slice brings value to customer

Figure 227. File upload feature split to vertical slices

One of the easiest ways to split tasks into vertical slices is to think about the user manual—define the minimum process that is understandable and doable by the end user.

There are many benefits of using vertical slices. Here are a few:

- You validate your ideas with real users.
- You do not overwhelm your users with tons of new features at once.
- You save money.
- Quality can improve due to the limited scope of focus (fewer edge cases) and iterative test coverage.

- The time needed to release a feature is decreased as you release it in parts.
- You might limit the number of bugs due to limited scope.

To develop your skills in creating vertical slices outside of your application, try this exercise: Open any web application you are familiar with, select an area of interest (e.g., shopping cart, training module, etc.), and analyze how you would split the work if it were your application.

Developer carousels

During development, various problems related to a lack of communication can arise. These issues often lead to increased costs due to late-stage changes and the progressive demotivation of team members.

A common problem is the prolonged processing time for pull requests (PRs). For example, suppose person A works on a feature alone (not recommended) for a week and then creates a PR. Another team member starts the review, but the solution is unacceptable, requiring a rewrite from the author. This leads to multiple rounds of review and rework, significantly delaying code merges. PRs become bottlenecks, slowing down development.

Another issue is not enough involvement in the idea validation process. For instance, I once implemented a solution that was reviewed and merged by a team member. However, three weeks later, another team member noticed a security vulnerability I had introduced, and we had to fix it quickly.

Delayed problem resolution is another concern. For example, if someone notices that test execution times on each PR are very long, they might plan to discuss it in a retrospective meeting in two weeks. However, more critical issues can overshadow this discussion, and it might be forgotten. Two months later, the build timeout due to additional tests can halt the entire team's progress for a day or more.

Relying on specific individuals for domain or technical knowledge also creates challenges. For instance, "Jimmy knows this area better, so it would be best to assign it to him," or "Sarah is the only one who has worked on this area." This practice unknowingly builds silos, making it challenging to replace these key people later.

When looking for ways to solve the above problems, besides other engineering practices like swarming, pair, and mob programming, there is a place for a developer carousel.

A developer carousel is a flexible, daily meeting attended exclusively by technical people. It serves as a forum to discuss technical issues related to architecture, structure, problems, patterns, strategies, implementation, and more, in a casual, conversational manner.

What does it look like in practice?

When? The meeting should be held daily, preferably in the morning (7 or 8 am), before any other meetings. Everyone is fresh, and it is easier to focus.

How long? The meetings usually lasts from half an hour to an hour.

Who should attend? Only technical team members. There is no leader in this meeting; everyone has an equal voice. Participation is not mandatory. You can join whenever you want, and if others need you, they can call you in. You can also leave at any time, especially if the topic is not relevant to you.

How do you start it? Anyone can start the discussion by saying something like:

> *Folks, I have a problem with the database design and want to consult it.*
>
> *Have any new problems occurred since yesterday?*
>
> *I found a security vulnerability that we must address as soon as possible.*

If initiating discussions is challenging initially, you can appoint a "meeting leader" to start things off. This role can rotate among team members. If there are no pressing issues (which is rare), you can leave the meeting or use the time to drink coffee and chat about non-work topics.

What topics can be discussed?

- You are working on a feature and considering using an Abstract Factory pattern but would like to discuss it with others.
- You are unsure if the direction of your feature implementation is correct and prefer to validate it before you continue.

- You have spent half a day on a problem and still cannot find an optimal solution.
- You spotted a security leak and want to brainstorm a solution.
- You read an article about test strategy in big tech companies and want to share it with the team. Based on the discussion, you can plan actions to integrate it into your environment.
- There are two different approaches in the application code for the same functionality, and you want to vote with others on which one to follow.

Discussions, discussions, but what next? It is essential to plan an action immediately when a topic is raised. For example, if someone starts with problem X, the team should find a way to solve it before moving to another issue. Each team member can share their opinions and propose potential solutions. Once potential solutions are collected, the team can vote on each solution.

Solution	Number of votes
A	3
B	3
C	2

Since solution C clearly received the fewest votes, it can be eliminated. Now, a second round of voting is necessary. If one solution receives a majority, the next step is straightforward—plan how and when to implement it. However, if the votes remain tied, further discussion and a new approach may be required.

Solution	Number of votes
A	4
B	4

In this case, you must present the arguments again—both for and against each option. In most cases, one of the options will prevail. If not, the team can decide how to proceed. Options include trusting a key team member's judgment, conducting a spike to try option A with fallback to option B, or bringing in someone from another team to cast a deciding vote. You can also roll the dice. There are many options and no limitations!

There is one essential aspect of the above voting. If you do not participate in the meeting, you accept the vote of others. In most cases, the selected solution will be good enough, and you should not have anything to complain about. However, if you identify a critical issue, bring it up in the next carousel meeting.

How do you end the meeting? When there are no more technical topics to discuss, you can end the meeting (or stay for a chit-chat). If you run out of time, you can agree to discuss the problem in the next meeting. If the issue is critical or a blocker, decide on the next steps—either extend the meeting or meet with a smaller group (you can still discuss it with the larger group the next day).

The developer carousel meeting addresses various issues that plague teams by shortening the feedback cycle. Thanks to it:

1. We optimize the duration of PRs. Misunderstandings are caught quickly. Of course, PRs can be entirely replaced by pair programming, but that is outside the scope of this discussion.
2. Many eyes look at the same problem. If a team of eight agrees on a solution, it is usually more reliable than if only two people do.
3. Problems that may seem trivial but are serious are caught immediately.
4. Potential improvements are discussed and agreed upon in a timely manner.
5. Knowledge about specific areas in the application is spread across the team.

Developer carousels also help bring remote teams together by giving them a chance to chat about coffee, basketball, or weekend plans. No matter how the project is run, there is always room for this kind of meeting.

Please keep in mind that this is not a silver bullet. It is just a way to make certain team processes smoother. Adjust it as needed, and don’t make it a requirement or a standard—this should be a meeting you look forward to, not one you must attend.

Give it a try. If you and your team find it unnecessary, feel free to drop it.

Addressing technical debt

Throughout my years in IT, I have encountered countless situations where technical debt was mistaken for bugs within our systems.

> *We have many bugs, and our technical debt has grown because of them! We must fix them.*

The confusion around technical debt is similar to the misunderstandings surrounding bounded contexts. Various interpretations can make the concept seem quite blurry. I remember struggling to grasp it for a long time. That is why I set out to find a clear explanation that everyone could understand.

Did I succeed? I am not sure. However, I did find an excellent analogy online that explains it well:

> *Taking on technical debt is like purchasing with a credit card. You are borrowing against productivity to get something done quicker or with less effort now.*
>
> *Just like a credit card, this can provide short-term benefits, but it also incurs interest in the form of additional work needed to fix or improve the quick solution in the future.*

You take shortcuts to focus on more immediate priorities, such as adding new features for an upcoming release. You consciously take risks with the idea of paying off the debt later. As a result, the solution works as expected but is not optimal.

The key point is this: If you manage to pay off the debt regularly, it is acceptable. However, if you accumulate so much debt that repayment becomes impossible, this will lead to the creation of a legacy system.

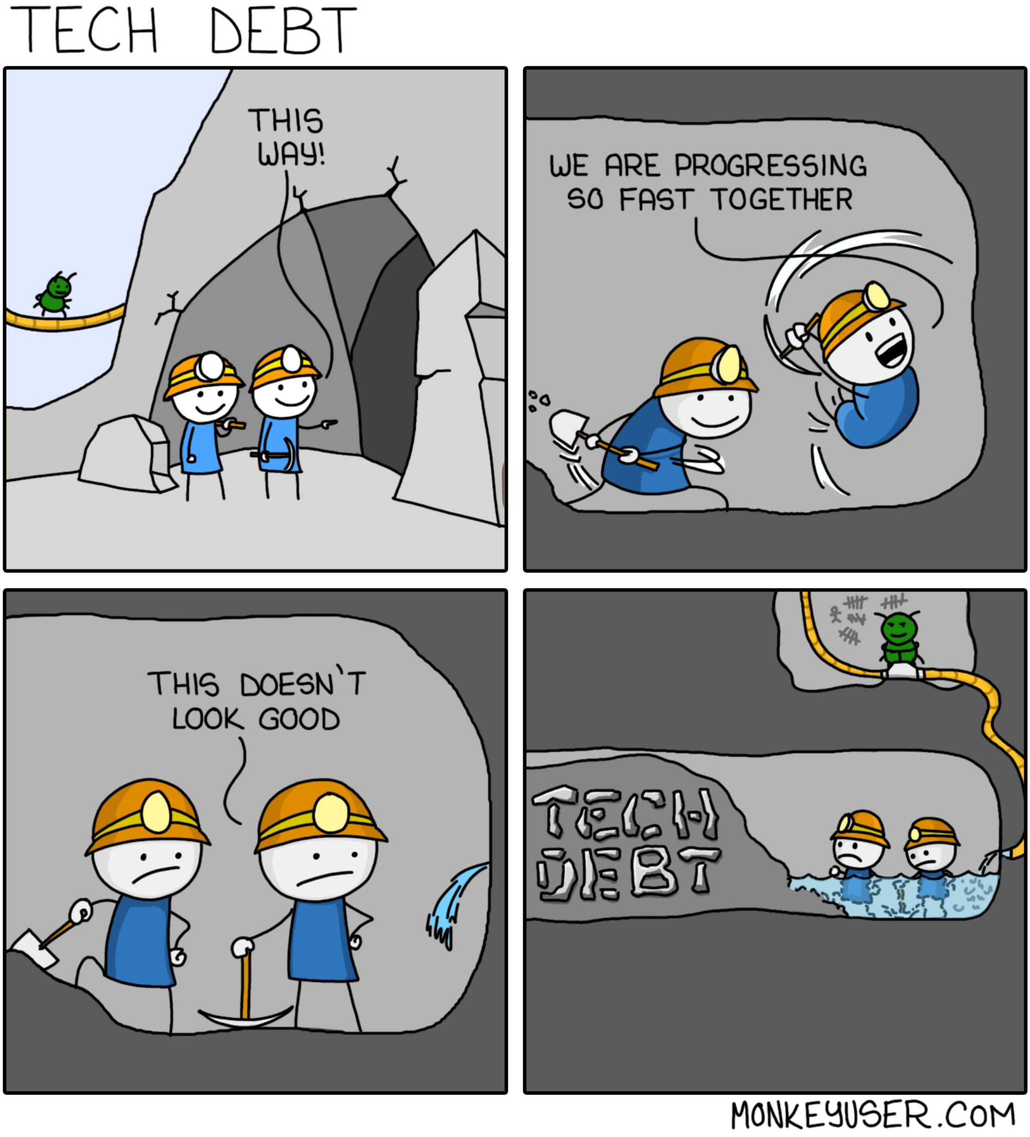


Figure 228. Technical debt

Here are some examples of technical debt:

- **Hard-coded values.** You know that writing values directly into the code is not a good idea, but you do not have time to extract them to a separate configuration and set it up for the pipeline. The team acknowledges this and plans to address the issue after the MVP release.

- **Manual step during automated deployment**. The team has automated most of the deployment process. However, one step requires manually copying and adjusting a script before the application is visible to the public. This step is prone to human error, but there isn't time to automate it now. The team will automate this step in the coming weeks.
- **Lack of high cohesion**. You notice that one of your modules combines unrelated functionalities. Additionally, you are asked to extend this module and release a feature quickly, worsening the cohesion. The team agrees to fix the incorrect modularization immediately after the release.
- **Code duplication**. When building a distributed system, you may find during the implementation of a new feature that some parts of the code are repeated across multiple microservices. These parts, unrelated to specific business logic (e.g., middleware for error handling), could be extracted into reusable building blocks.
- **Lack of proper monitoring**. The team is aware that proper monitoring will not be in place for the first three months. This increases the risk of undetected errors affecting users, but everyone understands and accepts this risk.

Now, imagine that none of the above issues were addressed. Sooner or later, you will face problems:

1. **Hard-coded values**. A hacker attacks your company and steals code from the repository. The attacker now has access to secrets that allow them to connect to the database and access customer data or remove your application from the hosting environment.
2. **Manual step during automated deployment**. As you did not address this issue, someone else added another manual step because of an already existing one. Other people did the same. After a year, there are so many manual steps that releasing the new application version in less than a week is impossible.
3. **Lack of high cohesion**. You continuously add information that does not belong to a given module. The database table grows and now has 100 properties of different kinds. The module itself contains dozens of checks and validations, often mutually exclusive. This results in poor performance and high maintenance costs.

4. **Code duplication**. A change in one place requires changes in dozens of others. Multiple teams are assigned to different microservices and need to adjust the same, repeated code. It does not matter if this is a code change or a package version update. This makes coordination difficult.
5. **Lack of proper monitoring**. You are unable to react quickly to problems in production because no one spots them. Instead of a quick reaction, you wait for feedback from customers who notice errors. Customers are not satisfied with the quality of the solution, and they start to search for other options.

Great. You now know what a technical debt is and the consequences of not addressing it.

But how do you deal with it? How do you plan to eliminate it?

I often hear stories about management blocking actions that could help tackle technical debt. I have also experienced the exact same situation many times: adding new features, then other features, then others, and so on.

There is no time for technical debt that grows each month.

If you work in a hierarchical environment, I recommend first explaining technical debt to your supervisor. Sometimes, we assume nothing will help, but it turns out the topic is simply not understood. Once you successfully explain it, describe the potential risks of not addressing it.

> ⚠ Do not focus on technical benefits. Try to find arguments that can convince business people. You can still talk about potential security vulnerabilities (a technical topic) but in the context of potential customer data leak (a severe issue that impacts your business).

Once you are ready, it is important to remember that managing technical debt is a continuous process. It doesn’t matter if you follow Kanban, Scrum, XP, or any other methodology. You can set a ratio, such as 1:5, to represent handling one technical debt item for every five new functionalities. This way, you are consistently working towards reducing it.

Technical debt is a regular part of software development, often needed to meet deadlines or business goals quickly. But, like borrowing money, it can cost more over time if you don’t manage it properly.

It is highly recommended that you regularly pay it off by fixing and improving temporary solutions. This will keep your software in good shape.

Extra 2: Architecture Exercises

Throughout this book, you have read about various software architecture concepts, calculations, and patterns.

But what would all the theory be without some practice? That's why I have prepared three real-world cases similar to the one we reviewed in the book so you can practice what you have learned.

Each case is built in the following way:

1. **Application description.** First, there is a description of the application given in the form of a story that you might hear when talking with business people. You will find a description of the main areas, some numbers, the business model, customers, and more.
2. **Information extraction**. Next, you must distill all the relevant information from the description. You can do this as it was done for the book's case, in the form of a table.
3. **Complex process modeling**. Each case will describe one complex process. Model it using Event Storming or Domain Storytelling.
4. **Architecture design**. Try to figure out the architecture—deployment, release and testing strategies, infrastructure, etc. It should fit the numbers and requirements from the description.
5. **Architecture evolution**. The next step will be to evolve your architecture through the changes described. Focus on architectural drivers that can help you throughout the process.
6. **Sharing results.** Finally, publish the result in your GitHub repository and share it with others :)

Case 1

Situation: You work as a software architect for a software house that has been hired by a real estate agency to build an application for their agents. You attend the first meeting with stakeholders, where they describe their requirements.

> We want to build an application for our real estate agents to help manage their property listings and allow customers to search for apartments to buy. A mobile app would be particularly beneficial since agents are almost never in the office—they go from property to property to show them to potential buyers.
>
> The application will cover all aspects of real estate, including preparing and publishing offers, listing offers, marketing, managing potential customers and buyers, managing contracts, and tracking transactions.
>
> We have been in the market since 1990 and manage all properties using spreadsheets. With 100 agents and over 5,000 new properties each month, managing these via spreadsheets has become increasingly difficult.
>
> For each property advertisement, we receive around 100 calls, of which 30% are interested in looking at the property. We store information about all interested customers in one of our spreadsheets.
>
> Each offer is first prepared and then reviewed and approved by someone else before it is published.
>
> We are ready to invest in this project without any budget limitations. Our goal is to release the MVP in the next 3-4 weeks. The application needs to operate 24/7 as our offering is directed towards customers from all continents. It must be available all the time. Your commitment to this project is highly valued, and we trust in your ability to deliver.
>
> We have no prior experience with software development in our company, and we plan to use your resources—as many as needed—for the implementation.

Task 1: Mark all crucial information.

Task 2: Categorize the information using a table similar to the one in the book's case.

Key area to model: Preparation and publishing of the property offer.

> An agent prepares a draft offer for a property, which includes the title, description, and other information. They set the initial price based on the market value of other properties. Once the draft is ready, it is sent for review.
>
> Any agent other than the creator can review the draft. When a new draft appears in the to-be-reviewed list, all agents are notified and can begin the review process. Only one reviewer can be involved. Agents receive a bonus of $100 for reviewing each draft.
>
> The draft can either be rejected or approved. If rejected, the creator must revise it and resend it to the same reviewer until it is approved. Once approved, the draft becomes an offer and is immediately published in the portal and visible to all users.
>
> An offer can be unpublished at any time, which means it will no longer be visible on the offer listing. It can also be republished (without additional review) at any time.

Model the above process using either Event Storming (Process Level) or Domain Storytelling.

Task 3: Select the most suitable deployment, release, and testing strategy based on the categorized information.

Changes: Several months after the successful MVP, the agency faces massive traffic on listing properties, causing performance issues. There are approximately 100 million views for 10,000 properties in a month. Additionally, there is a new feature—each property can have up to 25 images, with a maximum size of 5MB per image.

How would you react to this?

Case 2

Situation: You work as a software architect for a company that has built a thriving social media platform. The current user base is 300 million. The company plans to build another platform and expects at least 100 million users in the first three months.

> We want to build another social media platform where users can publish photos and engage with them through comments and reactions. Users will be able to upload an unlimited number of photos, each up to 5MB in size.
>
> Our revenue model will focus on advertising; advertisers will be charged for publishing ads on the platform. After payment, an invoice will be sent to the advertiser, whether they are a business (B2B) or an individual (B2C).
>
> Users can have friends or followers. A friend represents a mutual connection between two users, while a follower is someone who only observes another user. Data from our existing platform shows that the most popular users have millions of followers and approximately 100,000 friends.
>
> We anticipate traffic levels similar to or greater than those of our current platform. Each user on the existing platform generates around 5-10,000 requests per day. As this is a global platform, it is hard to define the most demanding time of the day. Usually, it is equally split.
>
> The platform must be operational 24/7, with a maximum allowable downtime of up to two days per year. We plan to release the first version within 6-9 months. While we are not under time pressure due to the existing platform, an earlier release would be great. There are no budget limitations.

Task 1: Mark all crucial information.

Task 2: Categorize the information using a table similar to the one in the book's case.

Key area to model: Interactions with published photos.

Users can publish new photos, which will then appear on other users' timelines. Eighty percent of the content on a timeline comes from the people the user follows or is friends with, while the remaining 20% comes from random users with similar interests.

Users can leave comments on photos. The author of a comment can edit or remove it, and the author of a photo can remove any comments on their photos. Comments are monitored and automatically removed by the application if they contain harmful content. The application has a filtering mechanism that checks each comment for harmful content.

Additionally, users can react to photos with one of three reactions: heart, like, or smile. The user who added the reaction can remove it by clicking on it again.

The photo's author can also block specific users or everyone from interacting with the photo.

Model the above process using either Event Storming (Process Level) or Domain Storytelling.

Task 3: Select the most suitable deployment, release, and testing strategy based on the categorized information.

Changes: The user base is smaller than expected several months after the release, with only 100,000 users. However, the decision is to keep the application because it grew from 10,000 users last month. The problem is that the current infrastructure generates excessively high costs and isn't needed for this number of users.

There is one more change—the demand is to be able to add videos, with each video having a maximum size of 5GB.

How would you react to this?

Case 3

Situation: You work as an independent software architect and have been asked to advise a company that sells tickets for various events. They want to build an application to support their ticket sales. They were using a third-party application, but it failed to meet their needs. Whenever sales started, the application would crash due to high traffic. They could not find another suitable solution, so they decided to build their own.

> Two years ago, we started offering tickets for events. Since our company lacks technical competence, we decided to look for an off-the-shelf solution, which has generally worked well. However, when we release tickets for highly anticipated events, the application crashes due to high traffic, preventing customers from purchasing tickets. This results in lost sales and reduced revenue.
>
> We have decided to implement a custom-built solution, but we must build an entire development team from scratch.
>
> The first version of the application should allow us to create new events and edit them. Next, we need to be able to create a ticket pool and assign it to each event. Tickets will be categorized into standard and premium. Standard tickets grant access to the event, while premium tickets provide entry to a VIP area with catering and other perks.
>
> Once the ticket pool is assigned to an event, it must be scheduled for a given day and time. The system should prevent ticket selection once all tickets are sold. When a ticket is sold, an invoice should be automatically sent to the customer.
>
> As mentioned before, the crucial part of the system is being able to handle peak traffic. The highest traffic that we have experienced is around 100,000 requests per second, according to our application provider. We only have one such high-traffic event per day, which we refer to as a key event. Other events do not generate this level of traffic.
>
> This application is intended for Australian customers only. It must be accessible when a key event is published; in other cases, we do not care much about its availability. There is also no deadline, but it would be great if we could start serving

> our customers a year from now. Our budget for development is $2 million, with a maximum running cost of $20,000 per month.

Task 1: Mark all crucial information.

Task 2: Categorize the information using a table similar to the one in the book's case.

Key area to model: Selling tickets to customers.

> There is a ticket pool for each event. It starts at one ticket and can be set to a maximum of 100,000 tickets.
>
> When an event is published, each customer can purchase up to 10 tickets, which can be either standard or premium. Tickets can be added to the basket and bought later, but they may no longer be available by then.
>
> Customers can pay for their tickets using a credit card. After a purchase, an invoice is automatically sent to them.
>
> Tickets can be returned within 48 hours of purchase, and a refund will be issued to the customer within seven days.

Model the above process using either Event Storming (Process Level) or Domain Storytelling.

Task 3: Select the most suitable deployment, release, and testing strategy based on the categorized information.

Changes: After the successful release, there's a new request to offer automatic ticket purchasing. Customers are willing to pay for this feature, so the company plans to introduce monthly and yearly subscriptions.

How would you adjust the application to accommodate this new feature?