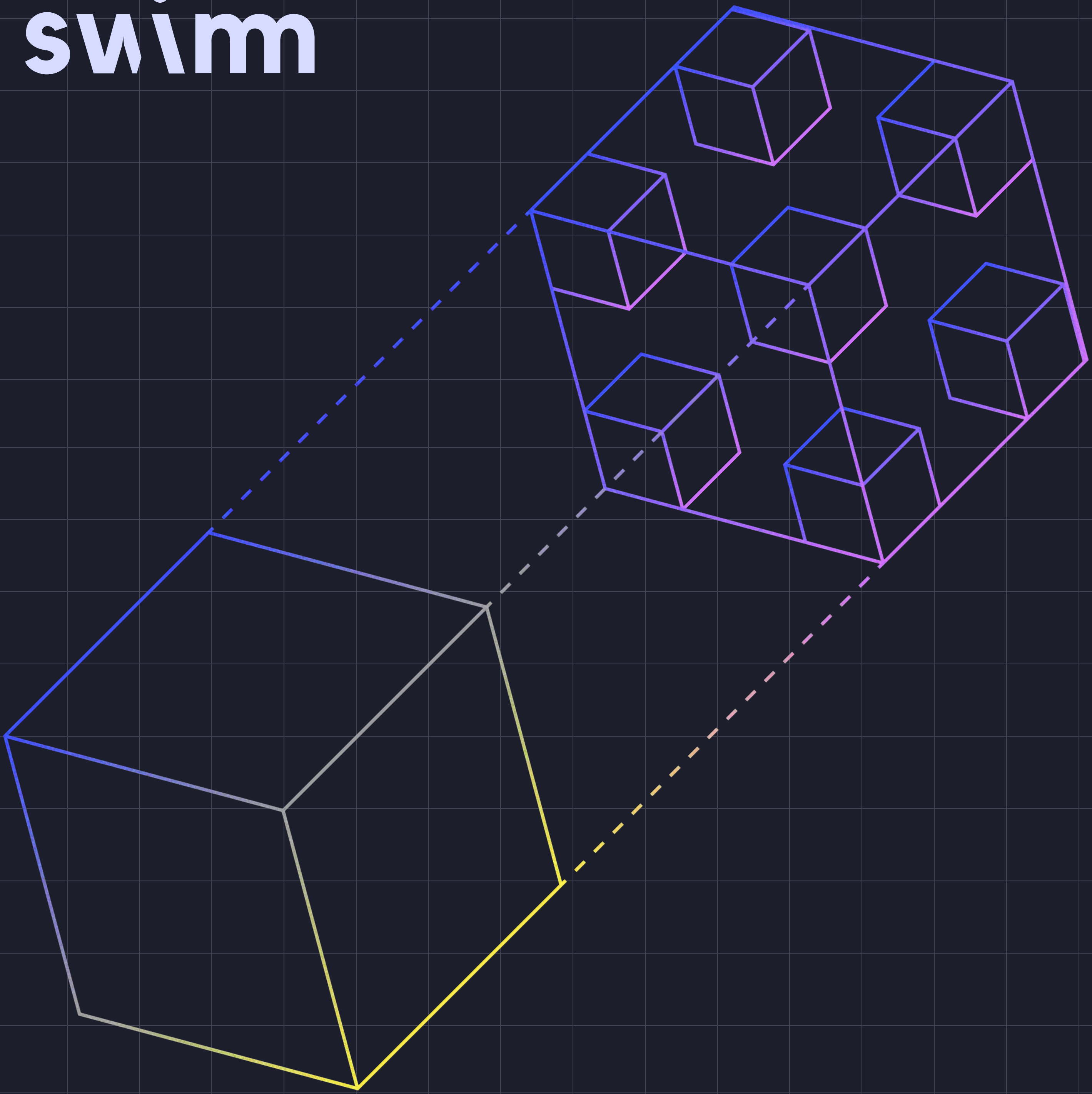
swim



Surviving the transition to microservices

Introduction

So you have decided to break your monolith into a microservices architecture.

Congratulations! A long path lies ahead. Lucky for you - it is a path that many have walked before. In this guide, you can learn from their mistakes, and even better - from their successes.

Now let's get the easy questions out of the way.

What is this guide about?

This practical guide will provide you with insights and actionable steps to survive the transition from a monolith to microservices.

What is this guide not about?

This guide is not a debate of whether you should or should not make the transition to a microservices architecture. It assumes you have already made up your mind, and aims to help you in the process, making it as painless and smooth as possible.

What do we define as microservices?

For the sake of this guide, we refer to "microservices architecture" as a specific architectural style of building software applications. These applications are composed of small, independent services, each of which runs in its own process and communicates with others using a well-defined API. Each microservice is responsible for a specific, isolated functionality and can be developed, deployed, and scaled independently. This contrasts with a monolithic architecture, where all functionalities are tightly coupled into one single, indivisible unit.

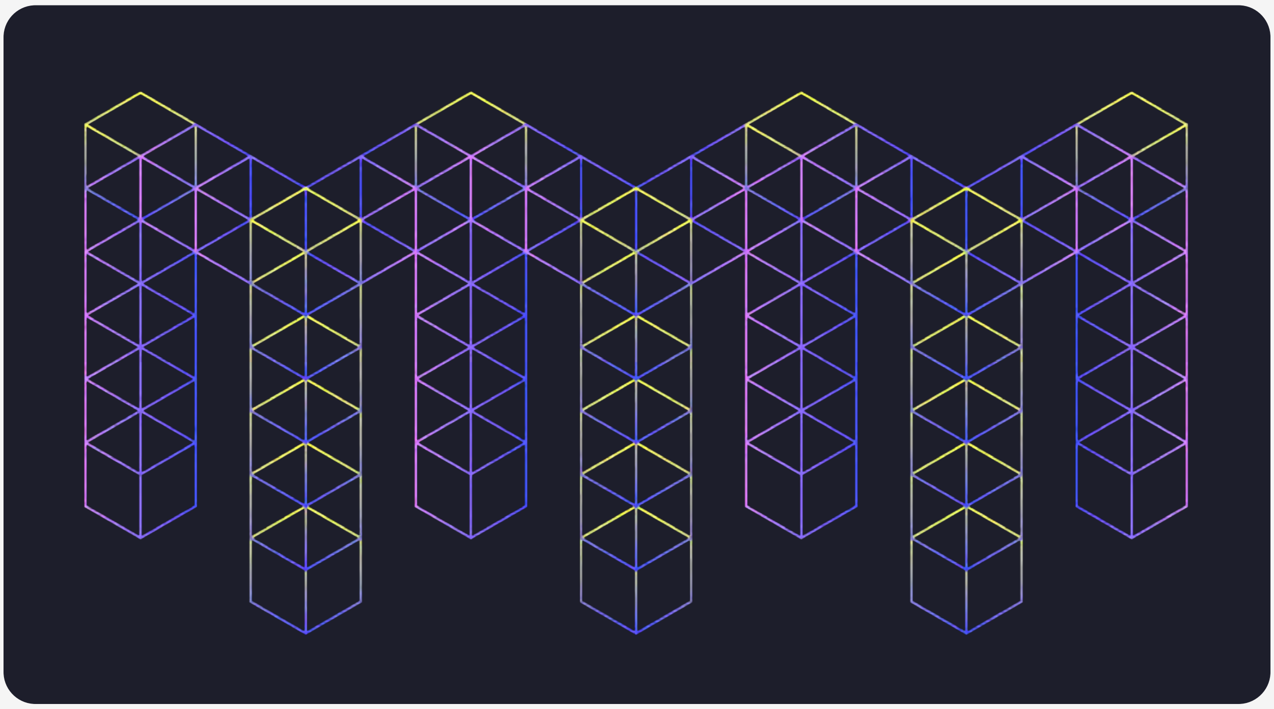
Table of Contents

Part 1

Core pillars for a smooth transition

The process of "breaking" the monolith is not just a decision, and it doesn't only encompass an isolated process. Shifting from a monolith to a microservices architecture is an organizational event. It affects the culture of your engineering organization, and how people work to accomplish their daily tasks.

We have interviewed experts from all walks of engineering who have transitioned to a microservice architecture and were kind enough to share their lessons, insights, and regrets.



To successfully perform the transition, three main components need to be addressed:

1. **Process** - how the transition will take place. We will describe a few key elements to a successful transition that help avoid many painful mistakes.
2. **Architecture** - what architectural patterns will help perform the transition. The architectural patterns need not only support the end result and the "ideal microservice architecture" you are striving for, but also support the process you committed to.
3. **Knowledge** - your engineering organization shifts to a new way of working. Close your eyes and imagine the day after the transition is complete. Yes, you have all the wonderful things you've strove to achieve when you decided to make the shift. But notice how different things are. Deployment is done differently, some code that people are used to having in one place in the monolith is now encapsulated behind a service with a new API they haven't used. How the services communicate with one another is new for many engineers. Finding the relevant code in the entire system is different than before. In other words, there is much knowledge that needs to be transferred.

Are you ready?

It's time to get into the details.

Part 2 The transition process

In this part we will describe the main stages your transition process should include. The following part will describe the architectural patterns that support this process.

A. Incremental steps

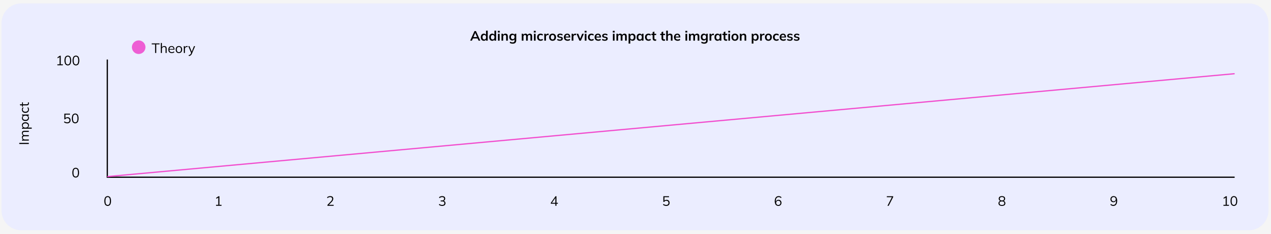
After interviewing many who have gone through this process, the biggest takeaway they share is to work gradually.

Sam Newmann captured an important insight in his talk Monolith Decomposition Patterns: "You won't appreciate the true horror, pain and suffering of microservices until you're running them in production."

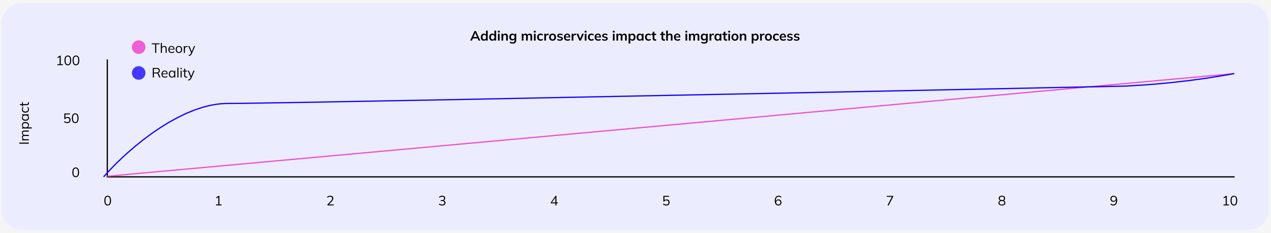
Drawing a practical take-away - you should extract one service from the monolith, and learn from that experience, before moving on.

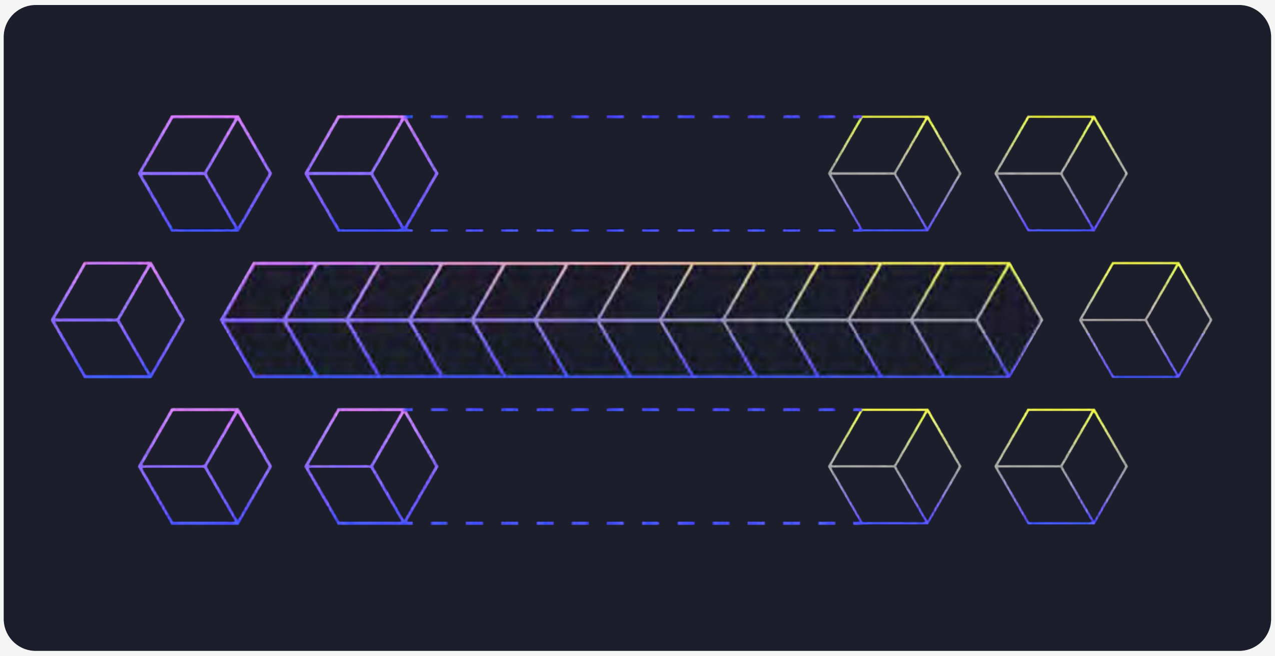
One extreme, of course, is to break the monolith into all microservices and then deploy them to production together. As one architect told us, he had seen that happen first hand, where his company realized their new microservices would actually get to work together for the first time - only in production.

A common mistake is to envision the transition process as follows:



That is, you work gradually, separate your modules into different services, one at a time. You picture a gradual, linear process, where the impact of every service is the same as that of the one preceding it. This assumption is wrong.





B. Your first microservice

The first microservice you extract from your monolith is by far the most important. Not because it is the microservice that everyone will count on, or the one that holds the most logic. But it is the first one to go through the entire iteration of separating from the monolith, and being deployed to production. It will teach you so much more than any guide, including this one, can ever do - as it will teach you what happens in your system, in your engineering organization, when you make that shift.

So, if the first microservice is the most important - how do we define successfully migrating it?

Actually, defining a success criteria here is pretty easy - **migrating the first microservice is complete when you've deleted its code from the monolith.**

This success criteria is something your architects or team responsible for your architectural changes must be aligned on. Though it may sound simple, breaking this first microservice will most likely result in some unexpected effects. For example, you may realize you need to separate some shared libraries that different modules in the monolith are relying on - in order to completely migrate even a single service from your monolith.

Once you have successfully migrated the first microservice, you should measure the deploy time for this first microservice. It may be substantially shorter than your current deployment times, as the deployment doesn't go through all of the other (infrastructure) code, tests etc. You may have a big win under your sleeve, one that can motivate you to move forward to the next microservice.

You may also realize that actually, you need to do some preparations before moving to the next microservice. For example, you might have realized that in order to successfully migrate the first microservice, you also had to migrate a shared library. You know that there are many other shared functions, modules, or libraries that would need to be migrated, or otherwise other developers would be dependent on them and unable to migrate their services. This might mean that the smarter thing to do would be to migrate the dependencies - your shared infrastructure, before migrating other services.

Migrating the first service shows your organization both the value (for example, shorter deployment time) and pain (with the unexpected dependencies).

You can try to anticipate those dependencies ahead of time, but nothing will teach you as effectively as migrating your first microservice. We have talked with professional, experienced architects that missed some of the dependencies when planning the migration. They all agreed that the next time around, they'd start with transitioning the first service, and build from there.

To migrate your first microservice successfully, consult the architectural patterns section of this guide.

C. Learn from your first microservice by conducting a retro

Once the first microservice is successfully migrated, it becomes a valuable learning opportunity for the entire organization. The process would have thrown up challenges, hiccups, and surprises, all of which provide important lessons for future migrations.

Conduct a retrospective on this initial migration: discuss what went well, what didn't, and what could be improved. Every aspect of the migration - from the planning and design stages to the implementation and post-deployment support - should be scrutinized, and insights should be derived.

See the knowledge section of this guide on how to ensure your organization actually applies its learning from this crucial stage.

Notice that while you will learn by far the most from the first microservice, you will probably have much to learn from migrating the second service, and a bit more to learn from the third service you are decoupling from the monolith.

Make sure you capture your learnings and update the migration guide to include them.

D. Staging environment

One of the main benefits of microservices architecture is the separation of tests: every service is deployed after running the tests that are relevant for this service, and for this service only. In order to enjoy this benefit, you would need to transition from system tests to service tests. To make that transition, you would need to know how to rollback, how to monitor each service, whether you run multiple tests in parallel, just to name a few of the decisions you would make.

After learning from migrating your first service, you will be well equipped to design your staging environment. A staging environment is crucial before moving forward. In this stage you will understand your CI/CD infrastructure and design for the tests.

Note that separating tests can start early, even before migrating the first service, that is, within the monolith.

E. Defining contracts

A "contract" refers to the set of rules that a microservice exposes to the world. This contract encapsulates the way the microservice interacts with others, providing a well-defined and predictable interface. It specifies the expected requests, responses, data formats, and error messages that the service can handle.

Contracts often come in the form of API specifications, which can be written in a language-agnostic format like OpenAPI or gRPC. Importantly, contracts must remain consistent even when the underlying service changes, to ensure compatibility with other services that depend on it. This concept is sometimes referred to as "contract-first" or "design-first" approach and is crucial for maintaining a robust, scalable, and maintainable microservice architecture.

Defining the contracts at this stage may sound too late in the process, but the fact is you will learn so much from the previous steps that it is hard to define the contracts accurately before performing them.

These are the best practices we recommend for defining your contracts:

1. **Determine request and response formats**

This usually involves defining the HTTP methods (GET, POST, PUT, DELETE, etc.), endpoints (URI), and any query or path parameters. For responses, define the status codes and the format of the response body.

2. **Define the data models**

Data models define the structure of the data your microservice will consume and produce - including data types, required fields, optional fields, and any constraints such as the length of a string or the range of numerical values.

3. **Specify error messages**

A well-designed contract will define what happens when things go wrong. This includes the error codes that your service will return and a description of what each error code means.

4. **Use a specification format**

Consider using a standard specification format to write your contract. OpenAPI (formerly known as Swagger) and gRPC are popular choices. These allow you to write your contract in a machine-readable format that can generate documentation, client SDKs, and server stubs, saving you time and reducing the risk of errors.

5. Create a service document

After defining the crucial parts of your service, you should create a service document as described in the knowledge section of this guide. This document would serve as the go-to resource for engineers that wish to interact with your service.

Remember, a well-defined contract reduces the risk of misunderstandings and conflicts, makes it easier for teams to work together, and can even enable automation of certain tasks like generating code and documentation. It's worth investing time to get it right.

F. Modifying the first microservice to serve as an example

Now that you have established your staging environment and what your contracts should look like, it is time to revise the first microservice to make sure it adheres to your guidelines. This service should serve as a great example for others to learn from when they migrate the next services. This is also a great time to create your migration guide for others to follow when migrating additional services.

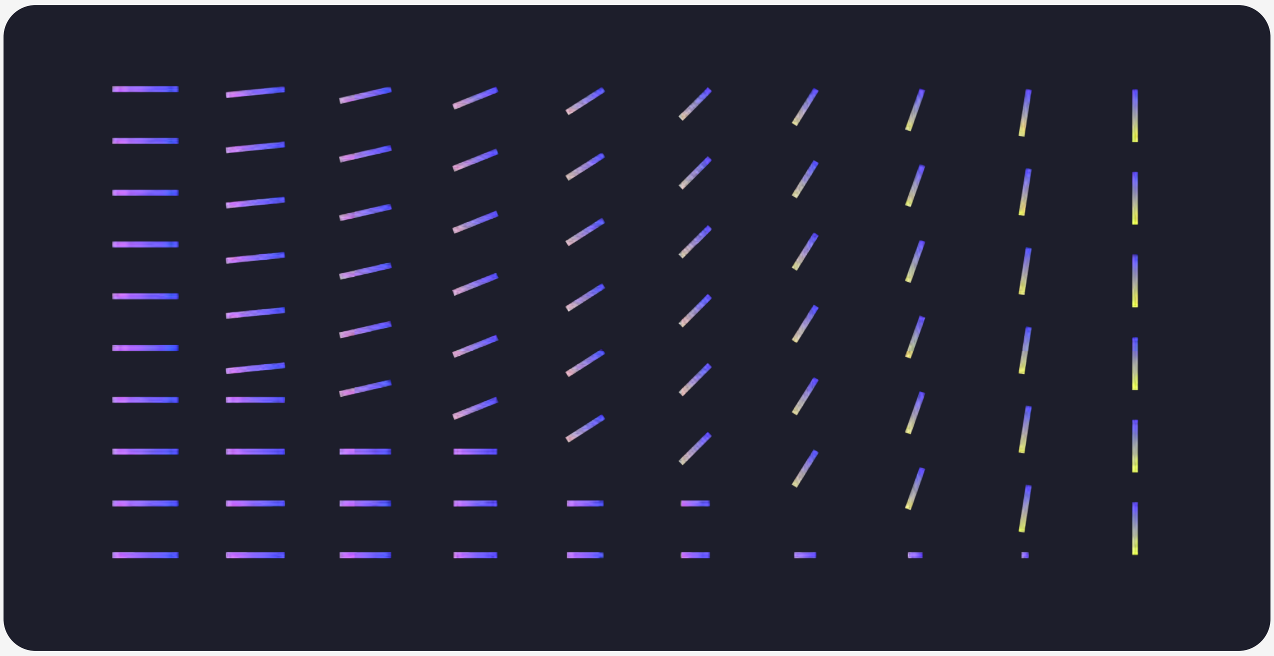
To summarize the process

- Separate your first microservice.
- Measure the deployment time of your first microservice.
- Learn from deploying the first microservice - what lessons should be learned for the next microservices?
- Migrate the shared infrastructure.
- Create your staging environment.
- Define contracts.
- Modify the first microservice to serve as an example.
- Create a migration guide.
- Migrate additional services gradually.

Part 3 Architectural patterns

In the previous section, we explained the importance of working gradually, and put a lot of emphasis on migrating your first microservice. But how do you actually work gradually? Don't you have to rewrite everything in order to have even a single microservice deployed into production?

Well, no. This section details practical patterns that can and should be followed to support the process described in the previous section.



A. Use a proxy

As you prepare to migrate your first microservice, a key challenge lies in conducting this separation without impacting your current production environment, while also avoiding a delay in deployment until all services are migrated. This task requires a careful and measured approach.

To migrate safely, we want to be able to allow the new architecture (that is, a monolith + a microservice) to work in parallel to the previous architecture (a monolith).

One pattern that greatly helps is to make sure all service calls are directed via a proxy (we would assume it's an HTTP proxy, though it doesn't have to be). This proxy can then be used to selectively divert some of the calls to the new implementation. To effectively implement this pattern, follow the next steps:

1. **Set up the proxy** - The first step would be to set up this proxy between the monolith and your upstream services, so that all calls are directed to the monolith. In other words, the functionality is exactly the same, but you've added an additional hop - specifically a proxy. After you deploy the proxy into production, make sure it directs all traffic directly to the monolith. At this stage, you want to make sure that everything works just as before. *Note that since you've added an additional network hop (proxy), this would inevitably result in some latency (that is, the traffic needs to go through another system, so it will take longer). This is a price you pay when you move to microservices - and you should feel this price and be willing to pay it, or understand at this early stage that you aren't.

2. **Deploy a new microservice into production** - notice that you are *deploying* this service, but not *releasing* it yet. This service lives in production, but the HTTP proxy does not direct any traffic to it. As Sam Newmann mentions in Monolith Decomposition Patterns, "we've too often combined these two concepts together in our heads - the idea of deployment into a production environment, and release of that software to our users".

3. **Test with a feature flag** - the fact that your microservice has been deployed to production does not mean it should affect your users. Since you have two implementations in production - you can have only specific users (in this case, your engineers - test users) experience the new architecture, while the rest (your actual users) still use the original implementation.

4. **Direct traffic to the new microservice** - after you have tested your microservice, you can redirect traffic to that service by changing the proxy's configuration.

5. **Any issues? Rollback** - testing is fine, but as you well know - when actual users get to play with your features, things tend to break. If you want to get back to the previous state, again - you can just configure your proxy to direct traffic to the monolith rather than the new service.

What you have achieved here is an elegant way to gradually migrate services. In no point is the monolith aware that there is anything going on.

B. Branch by abstraction

The proxy approach detailed in the previous section is elegant and efficient. Yet, it assumes that you can separate your functionality into services. This is often more easily said (or written) than done. In many cases, there is no single (HTTP) call that comes in that we can map to a specific functionality. Most functionalities are triggered as a side effect of processes within the monolith.

"Branch by abstraction" means creating abstraction over an existing functionality. This probably means refactoring - you would need to create a clear interface for this functionality you want to decouple from the monolith. Then, it would involve creating an abstraction point.

To implement branch by abstraction successfully:

1. **Isolate the current implementation** - This is the first and most important and delicate step of this pattern, where you isolate the current implementation from the rest of the monolith. At this step you take the logic of the current functionality you intend to replace, and move it to a single place (same package, module or folder) - still within the monolith.

2. **Create an abstraction point** - Create an interface that will behave as the contract to call that given functionality. Make sure your software calls a default implementation of this interface, which will then call your just isolated functionality

3. **Direct all calls to your functionality to the abstraction point** - Make sure no other parts of your code interact directly with your implementation. Rather, all calls should go through the new interface you've created.

4. **Start working on the new service implementation** - Your new service will receive requests and act upon them. The functionality that used to be inside the monolith needs to be copied into that new service.
5. **Deploy the new service into production and test with a feature flag** - Again, deploy but don't release. You can safely test your new service while the monolith preserves its capabilities.
6. **Switch over** - when you are happy with the new service, you can switch over to it. As explained in the previous section, rolling back is simple if necessary.
7. **Clean up** - Once everything is working as expected, you can remove the old functionality from your codebase.

C. Parallel run

Being a variation of "branch by abstraction", "parallel run" allows both implementations (the one within the monolith and the new shiny microservice) to co-exist at the same time. This is particularly useful for testing if the behavior has changed. When decomposing a monolith you are refactoring the system - you are changing implementation, but don't want to affect the behavior of the code.

You want to make sure the system is functioning the same way it did before, but with the new architecture in place. With both architectures running in parallel - you can compare the outcome of both functionalities, and look for discrepancies. This is a way to make sure your new service operates as expected.

GitHub created a tool called Scientist to manage parallel runs.

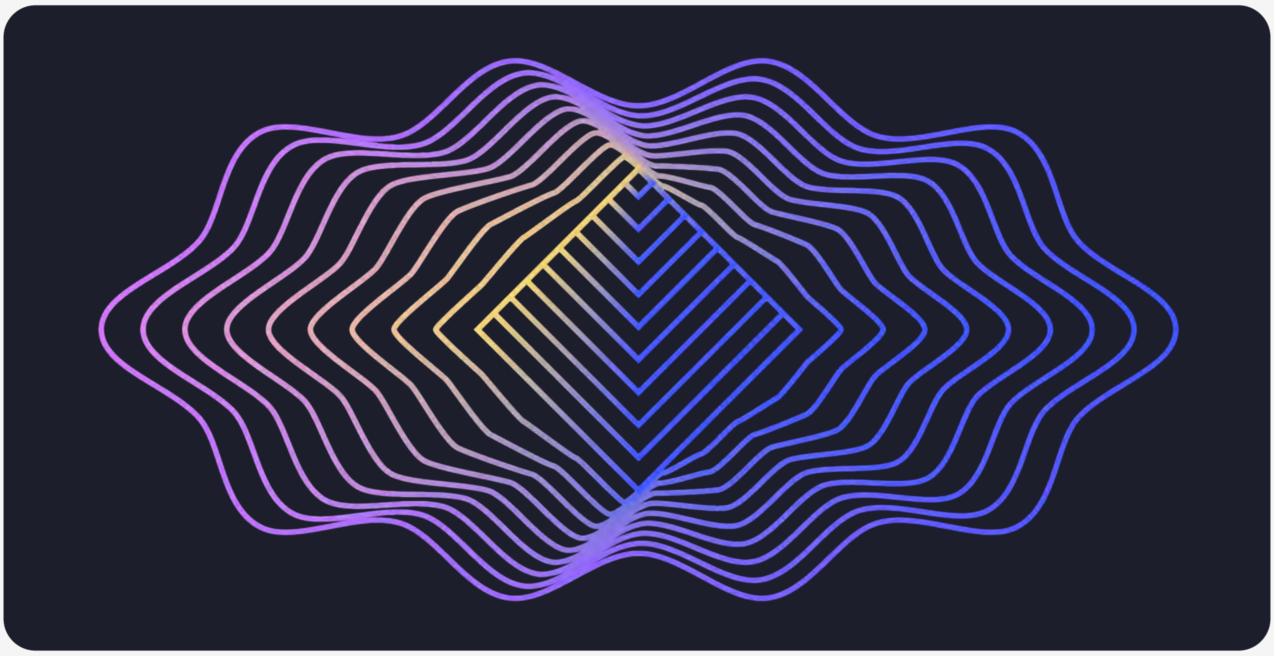
Part 4

Managing knowledge

Infrastructure changes, like transitioning from a monolith to a microservices architecture, usually happen once organizations mature and the code no longer matches the changing needs of the business. Maybe you need to scale, and perhaps you find yourself slowed down by inter-team dependencies.

Even though changing infrastructure can be super helpful in the long run, managing the transition is challenging especially as many developers will need to understand and work with the new infrastructure.

In the beginning of the transition, you are liable to have a few engineers who understand the new architecture, how to use it and how to extend it, and many engineers who are inevitably accustomed to the previous architecture. To effectively transition the architecture, add more services in the future, and use the new services correctly, requires careful attention and thought. This section outlines best practices to manage knowledge sharing to ensure the process goes smoothly.



A. To engineers performing the transition - a migration guide

This guide will be used by any engineer who is about to migrate an additional microservice.

I. When you should write a migration guide

In the process section, we understood why it is so important to migrate your first microservice, and learn from that experience, before moving on. The process would have thrown up challenges, hiccups, and surprises, all of which provide important lessons for future migrations.

We specifically discussed the importance of conducting a retrospective on this initial migration.

This is a good step to document your learnings into a standardized migration guide. This guide should be regularly updated as more services are migrated. This guide would serve as an invaluable resource for everyone involved in the process, reducing confusion and ensuring consistency. Over time, as more services are transitioned, the organization will continue to refine and optimize its migration strategy, paving the way for a smoother and more effective transition to a microservice architecture.

II. Creating an effective migration guide

1. **Service selection criteria:** Outline the criteria to determine which parts of the monolith should be turned into a microservice. This might include factors like modularity, coupling, cohesion, business domain, etc.

2. **Environment setup:** Detail the steps necessary to create a new service. This can include setting up the development environment, repository, configuration files, creating boilerplate code, etc. Also, document the standards to be followed, such as code conventions, directory structure, etc. Make sure it includes actual examples from existing repositories (see this section).

3. **Defining the service contract:** Provide a template and example for defining the service contract. This includes request and response formats, data models, and error messages, as described in the defining contracts section. In a way, this is the "contract of contracts" - explaining what a contract should include. Again, rely on an existing example - such as your first microservice.
4. **Data migration:** If the service has its own database, include steps on how to migrate data from the monolith's database to the new service's database. This could also include scripts, tools, or procedures used. See also accessing data.
5. **Testing:** Decide on the testing strategies to be used, including unit tests, integration tests, and contract tests. Also, specify the coverage requirements, and any testing frameworks to be used. See also the section on staging environments.
6. **Deployment:** Describe the steps to deploy the new service to different environments (development, staging, production). Include information on CI/CD pipelines, containerization, service orchestration, and other deployment tools being used. Importantly, detail the deployment progress - and why you separate deployment from releasing, as described in the architectural patterns section.
7. **Rollback plan:** Outline a strategy to rollback the changes in case the migration fails or the new service has issues in production. If you follow the architectural patterns section, rolling back should be simple.
8. **Monitoring and logging:** Give instructions on setting up monitoring and logging for the new service. Also, include steps to integrate the service with any existing monitoring/dashboard solutions.
9. **Communication with other services:** Explain how the new service will communicate with other services. This could include setting up APIs, message queues, event-driven communication, etc. Be sure to show real examples of how this is done by existing services.

B. To the entire engineering organization - Here’s what you need to know

In order to work effectively in a microservices architecture, your engineers will need to:

1. Understand the architecture.
2. Know what services exist.
3. Know how to use the relevant services correctly.

We recommend that you create the following documents:

I. Architecture overview document

First, create an overview document consisting of the following elements:

Motivation - you have spent a lot of time and effort introducing a new architecture. You know you are doing it for good reasons, and it is important that developers are aware of them.

Relationships and main services - explain the different entities and their relationship. If you introduce multiple new services, explain how they are placed in the existing ecosystem and how they interact. If you document a single, new library or service, explain how it interacts with other services, what services may communicate with it, and why. Remember that this is only a high-level description, so don't go into too much detail.

Diagrams are super useful for this kind of reference.

II. Individual services documents

There are two types of documents for a specific service:

1. **Internal-facing** - for the team/engineers that will contribute or maintain the code of this service.
2. **External-facing** - for teams/engineers that will use this service.

We will focus on the latter - documents that help other engineers use the service. This document would serve as the go-to resource for engineers that wish to interact with your service.

III. Create a good usage example

You have created your service for others to use. Even if you are not going to use it yourself - create a real example showing how you should use the functionality you are exposing. Make sure your examples adhere to best practices you would like to encourage when using your service.

What is a good example? A good example, in this context, has two main attributes:

1. It demonstrates all (or most) of the things/steps a developer should implement when using your service. If, for example, you expose a specific function through REST API, and you almost always need to validate the response in a certain way and then issue another REST API call - you should create an example that shows these steps.
2. It is the simplest, shortest example that demonstrates the pattern. Remember, the goal is to exemplify the pattern rather than understand the details of this specific example.

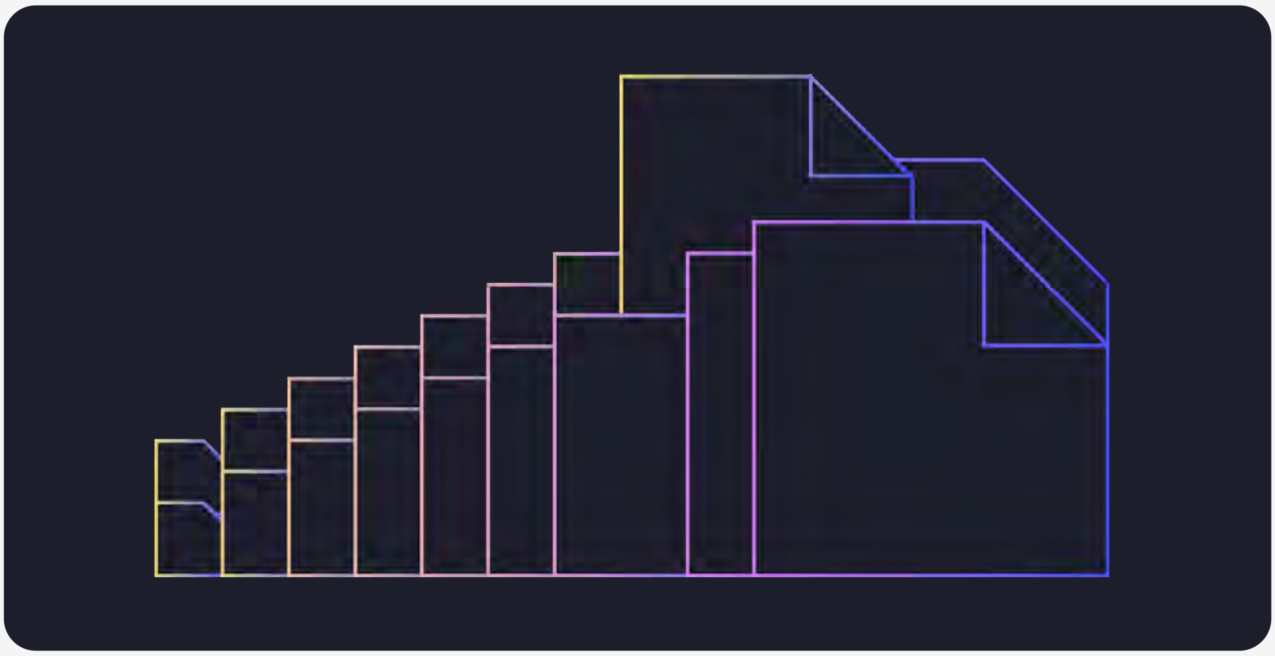
The best example is a "real" usage example, that matches the attributes enough. See the section on why you should use real examples.

IV. What your document should include

Your document should include the following components:

- A general explanation of the service/library/repository you are describing.
- Provide concrete usage examples from the codebase.
- Mention the important functions that are used. This should be the main example described above. If there are many different examples, we recommend splitting them up into different documents.
- Describe the best practices - dos and don'ts regarding infrastructure usage, input validation, assertions, etc. When there are specific assumptions or tweaks for specific use cases, mention them.
- A technical spec - include all information from your contract - as described under the defining contracts section of this guide:
  a. Request and response formats.
  b. Data models.
  c. Error messages.
  d. Specification format.

Part 5 — Why you should use real examples in your documents



In your documents, we advocate for using real examples, that is - code that exists in your codebase, rather than inventing examples or using pseudo code. This recommendation stems from multiple reasons:

1. **Digestibility.** It's easy to understand. As a developer, a real example is easy to relate to. It also provides a good basis to rely on when the developer would look to use the tool themselves.
2. **Creation.** It's easy to create such a document. If an example already exists in your codebase, there's no need to invent a new one. All you have to do is describe it.

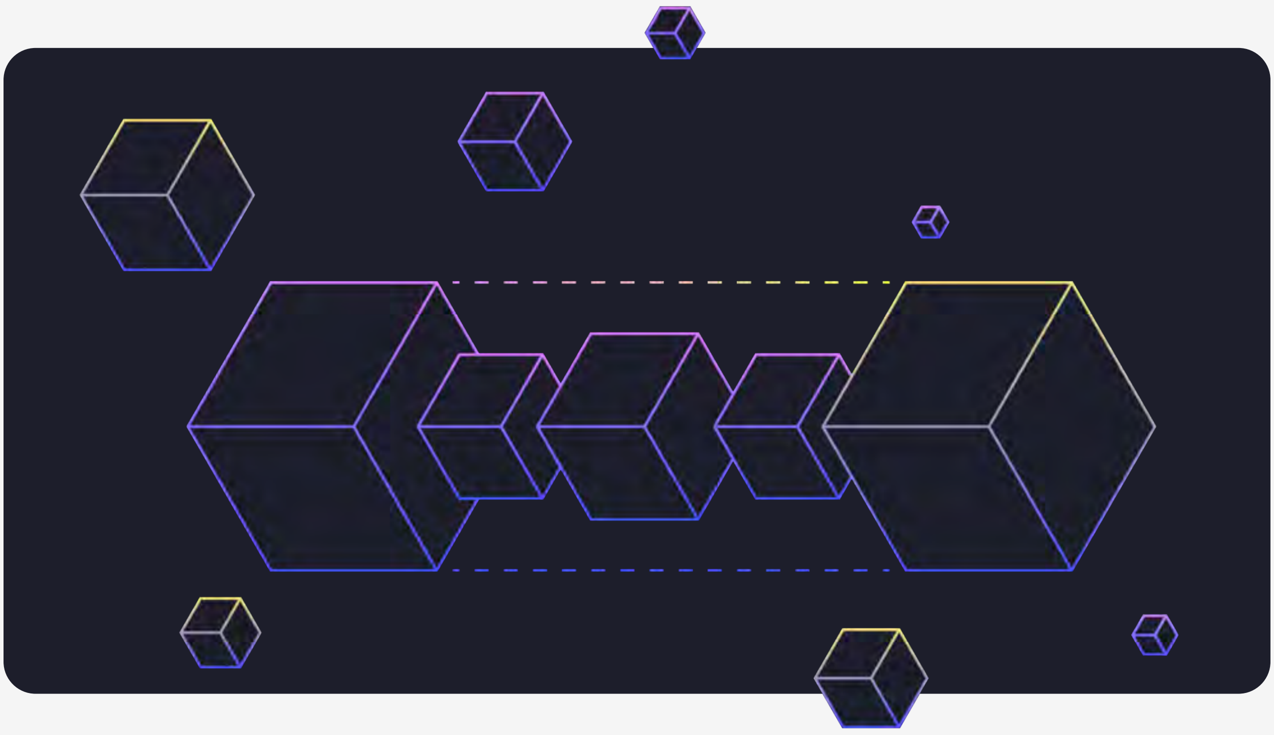
3. **Memorability.** It helps you remember. When you look at a concrete, real example - you see all the small implementation details. Not all of them are important to mention or explain, yet it makes sure you don't forget about those that are.
4. **Maintainability.** It's maintainable. If you create your document using Swimm - by code coupling to an existing example, if something ever changes in the system and the example changes, your document will be updated.
5. **Discoverability.** It's easy to discover. Thanks to the discoverability of Swimm documents, they are found when someone uses this tool. For example, when using a library for the first time, a developer may look for other usages of this library in the codebase, such as the wrapper function. Thanks to Swimm's IDE plugins, developers are likely to find the relevant document next to the wrapper function because it was referenced and code-coupled in the document.

Part 6

The importance of microservice transitions

Transitioning from a monolithic structure to a microservices architecture is a monumental step in software development, marking a departure from the traditional approach to building applications. Instead of a singular entity, applications in a microservices setup consist of various autonomous services, each having a distinct function and the capability to be developed, deployed, and scaled independently.

The transformation isn't merely procedural—it's a complete organizational metamorphosis affecting engineering culture and daily operations. To sum up what we've discussed, and to navigate this change effectively, bear the following in mind:



- **Core Pillars for Transition** must be established. These pillars encompass the methodology of the transition, the architectural patterns supporting both the end state and the transition, and the assimilation and transfer of knowledge.

- **The Transition Process** should be gradual. Starting with the extraction of a single service allows teams to learn and adapt. It's crucial to review and reflect upon each transition, especially after the first microservice migration. Part of this reflection involves understanding the role of a staging environment and how to segregate tests for individual services.

- **Managing Knowledge** during this shift is pivotal. Engineers should document lessons learned post each microservice migration, encompassing areas from service selection to deployment strategies. An overview of the new architecture, details of each service, and practical usage examples will benefit the entire engineering organization.

- **The use of Real-Life Examples** in documentation provides practical insights, aiding comprehension, ensuring maintainability, and facilitating easy discovery of essential components.

Drawing from the insights of those who've ventured this path before, one can mitigate risks, streamline the process, and ensure that the transition is not just about breaking down the monolith, but building a system that's future-ready.

As we've highlighted in this guide, the foundation of this successful transition lies in understanding the core principles, the nuances of the transition process, and the importance of managing knowledge effectively. With the right approach, tools, and commitment, along with the following knowledge, organizations can make the shift smoothly and harness the full potential of microservices.

Finally, embracing a microservices architecture is a transformative journey that promises scalability, flexibility, and independent service management.

However, the journey is replete with challenges that require meticulous planning, strategic decisions, and a commitment to continuous learning.

swimm

Intrigued and want to learn more about Swimm?

Sign up for a community demo today.

Get a personalized 15 minute demo