**Introduction**

In this lesson, we'll get a walkthrough of what this chapter holds for us.

**We'll cover the following**

* [Chapter walkthrough](https://www.educative.io/module/lesson/introduction-to-microservices/q2qmvxOYgPG#Chapter-walkthrough)

Migration **from a deployment monolith to a microservices architecture** is the common case for introducing microservices. Most projects start with a deployment monolith that the team wants to split into microservices later because the deployment monolith has too many disadvantages.

Of course, it is also possible to implement a new system directly with microservices from scratch.

**Chapter walkthrough**

This chapter **provides an overview of the challenges involved** in migrating to a microservices system.

* The chapter discusses possible **reasons for migration**. In this way, readers can assess whether a migration makes sense in their context. The approach to migration depends on your objectives. Therefore, knowing possible reasons is helpful for choosing a migration strategy.
* The chapter then shows a **typical strategy for migration**, with some alternatives. In this way, readers can choose a migration approach suitable for their scenario.

**Reasons for Migrating**

In this lesson, we'll discuss a few common reasons for migrating to a microservices architecture.

**We'll cover the following**

* [Microservices offer a fresh start](https://www.educative.io/module/lesson/introduction-to-microservices/q2xk1EwkQB2#Microservices-offer-a-fresh-start)
* [The reasons are already known](https://www.educative.io/module/lesson/introduction-to-microservices/q2xk1EwkQB2#The-reasons-are-already-known)
* [A typical reason: speed of development](https://www.educative.io/module/lesson/introduction-to-microservices/q2xk1EwkQB2#A-typical-reason:-speed-of-development)

When migrating to microservices, it is **important to know the objectives for taking this step**. Depending on the reasons that led to this decision, the procedure for implementing them may vary.

**Microservices offer a fresh start**

Especially when replacing legacy systems, microservices have several advantages. Let’s discuss them.

The code of the legacy system no longer needs to be used in the new microservices because the microservices are implemented separately from the legacy system. The **microservices offer an unencumbered restart**.

The code of a legacy system is often no longer maintainable, and the technologies are frequently outdated. Therefore, reusing the old code would hinder the development of a clean new system.

Microservices thus solve the most important challenges when dealing with legacy systems, because otherwise, a restart is difficult.

Migration to microservices has the potential to solve the problem with the legacy system. After migration to microservices, further migrations can be limited to one or a few microservices. A migration of the entire system will probably not be necessary again.

A typical **reason for a system migration is outdated technology**. In a microservices system, such a migration can take place step by step – that is, microservice by microservice.

Another migration reason is an **unmaintainable system**. In this case, each microservice can be replaced individually.

**The reasons are already known**

The reasons for a migration are in the end identical to the reasons for using microservices. These have already been discussed in detail in the [last chapter](https://www.educative.io/collection/page/10370001/6518081205567488/4998953437233152) and may include:

* Increased security
* Robustness
* Independent scaling of individual microservices

**A typical reason: speed of development**

A typical reason for introducing microservices is the lack of speed in developing with a deployment monolith. When many developers are working on a deployment monolith, they need to **closely coordinate their work**. This costs time and therefore **slows down development**.

But even with a small team, a deployment monolith can be problematic because the **deployment is quite huge**. The size makes continuous delivery difficult to implement, and each release requires a lot of testing.

**Typical Migration Strategies**

In this lesson, we'll discuss a few typical migration strategies.

**We'll cover the following**

* [A typical scenario](https://www.educative.io/module/lesson/introduction-to-microservices/B8AzyYgPNpx#A-typical-scenario)
* [Give a preference to asynchronous communication](https://www.educative.io/module/lesson/introduction-to-microservices/B8AzyYgPNpx#Give-a-preference-to-asynchronous-communication)
* [Give preference to UI integration](https://www.educative.io/module/lesson/introduction-to-microservices/B8AzyYgPNpx#Give-preference-to-UI-integration)
* [Avoid synchronous communication](https://www.educative.io/module/lesson/introduction-to-microservices/B8AzyYgPNpx#Avoid-synchronous-communication)
* [Reuse old interfaces?](https://www.educative.io/module/lesson/introduction-to-microservices/B8AzyYgPNpx#Reuse-old-interfaces?)
* [Integrating authentication](https://www.educative.io/module/lesson/introduction-to-microservices/B8AzyYgPNpx#Integrating-authentication)
* [Replicating data](https://www.educative.io/module/lesson/introduction-to-microservices/B8AzyYgPNpx#Replicating-data)
  + [Replication should be done in one direction](https://www.educative.io/module/lesson/introduction-to-microservices/B8AzyYgPNpx#Replication-should-be-done-in-one-direction)
* [Black box migration](https://www.educative.io/module/lesson/introduction-to-microservices/B8AzyYgPNpx#Black-box-migration)
  + [Choosing the first microservice for the migration](https://www.educative.io/module/lesson/introduction-to-microservices/B8AzyYgPNpx#Choosing-the-first-microservice-for-the-migration)
* [Extreme migration strategy: all changes in microservices](https://www.educative.io/module/lesson/introduction-to-microservices/B8AzyYgPNpx#Extreme-migration-strategy:-all-changes-in-microservices)
* [Further procedure: step-by-step migration](https://www.educative.io/module/lesson/introduction-to-microservices/B8AzyYgPNpx#Further-procedure:-step-by-step-migration)

Often **there is a concept** for the final target architecture that the migration should achieve, **but no concrete plan** for the first steps to be taken or for the first microservices to be implemented.

In particular, the small steps into which development can be broken down are a major advantage of microservices. A simple microservice is written quickly. Because of its small size, it is also easy to deploy. And if the microservice should not prove itself, not much effort has gone into the microservice and it can easily be removed again.

In the further course of the project, the new architecture can be implemented step by step, microservice by microservice. In this way, the team will avoid large risks.

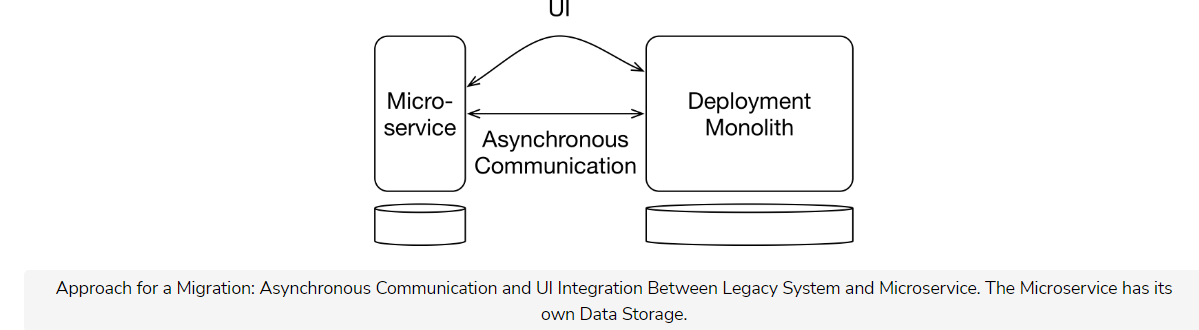
Because the migration process depends on the goals and on the structure of the legacy system, there is no universal approach. Therefore, the strategy presented here must not simply be used as is but must be adapted to the respective situation.

**A typical scenario**

A typical scenario for a migration to microservices is:

* The aim of the migration is to **increase the development speed**.
  + Microservices need fewer tests for a release and provide easier continuous delivery because they are smaller.
  + Also, the development of individual microservices is quite independent, so less time is spent on coordination. All of this can make development faster.
* The migration to microservices **must provide an advantage in development as quickly as possible**. It makes no sense to invest in an architectural change that only leads to improvement much later down the line.

The migration **strategy proposed** here is **based on extracting individual microservices** in order to achieve an improvement of the situation as quickly as possible. Have a look at the drawing below for a visual.



## Give a preference to asynchronous communication

Integration with the legacy system should take place via asynchronous communication. This **decouples the domain logic of the microservice from the legacy system**.

The legacy system sends events. To do this, the legacy system must be adapted to create events. Since the legacy system is usually poorly maintained, this can be a challenge. Microservices then can decide how to react to these events.

One of the possible benefits is availability: **A failure of the legacy system does not lead to the failure of the microservice**, and a failure of the microservice does not lead to the failure of the legacy system.

## Give preference to UI integration

Further integration is conceivable at the UI level. If the legacy system and the microservice are **integrated with each other via links**, then only the URLs are known. What hides behind the URLs can be decided by the linked system and can change without much impact on other systems.

Via links, additional resources can be available. This is the basis of [HATEOAS](https://en.wikipedia.org/wiki/HATEOAS) (Hypermedia as the Engine of Application State). The client can interact with the system through the links and does not need to know about possible interaction possibilities.

For example, a link to cancel an order would be sent along with the order. New interaction possibilities can be easily supplemented by new links.

The UI integration also **offers an easy way to operate the microservice and the legacy system in parallel**. Individual requests can be redirected to the microservices, while the remaining requests are still processed by the legacy system.

Often, there is a web server anyway which processes every request and carries out TLS/SSL termination, for example, parallel operation of microservices and the legacy system is then quite simple. The web server only has to forward each request either to a microservice or to the legacy system.

UI integration is particularly **easy if the legacy system is a web application**. But it is also possible, for example, to integrate web views in a mobile application to thereby integrate parts of the UI as web pages. In such cases, UI integration should really be considered as an option because of its many advantages.

## Avoid synchronous communication

Synchronous communication should be used sparingly. It **leads to a close dependence with regard to availability**. If the called system fails, the calling system must be able to deal with this. The degree of coupling in the domain logic is also quite high.

A synchronous call usually describes exactly what needs to be done. Synchronous communication may be necessary if you want the last changes to be visible in the other systems as soon as possible. In a synchronous call, the state at the time of the call is always used, whereas asynchronous communication and replication can lead to a delay until the current state is known everywhere.

## Reuse old interfaces?

If there is already an interface, it may be useful to use this interface to **save the effort** of introducing a new interface.

However, the interface **may not be well adapted to the needs of the microservice**. In addition, it **cannot be changed easily** because there are already other systems that also use this interface and are influenced by changes.

The technology the interface uses is not too important for the decision of whether it should be used by a microservice. Microservices can use almost any type of interface. For a migration, it might be a lot easier to use an existing interface even with an awkward technology than to create a new one.

More important are the dependencies the microservices establish: As discussed in [this lesson](https://www.educative.io/collection/page/10370001/6518081205567488/4953894968885248#bounded-context-and-strategic-design) in the previous chapter, the selected pattern for the integration influences the coordination effort and the degree of independence. Just reusing an existing interface **might compromise a goal like independent development**.

## Integrating authentication

For a system that consists of a legacy system and microservices, the user should have to log in only once. Legacy systems and microservices do not necessarily have the same authentication technologies, but the systems must be integrated in such a way that **a single sign-on is possible**, and the user does not have to log on to the legacy system and microservices separately.

Authentication may also need to provide roles and permissions for authorization in the microservices. Adjustments may also be necessary here.

## Replicating data

Even in a migration scenario, **each microservice should have its own database or at least its own database schema**. The goal of the migration is to achieve independent development and simple continuous delivery of the microservices. This is not possible if the microservices and the legacy system use the same database.

A change to the database schema then might have hard-to-predict effects. As a result, the microservice is hardly changeable and difficult to put into production.

Together with asynchronous communication, a separate database means **data replication**. This is the only way for the microservices to implement their own data model. **Changes to the data can be communicated via events**.

### Replication should be done in one direction

Replication for a specific part of the data should take place in one direction only. Usually, replication can be done using business events – that is, events that have a meaning for a business expert. One system (a microservice or the legacy system) should trigger the events, and the other system reacts to the events.

So, for example, one system could generate an event like “customer registered" and the other system could store the customer data relevant to them. However, there should be only one source of each type of event. Otherwise, it can be very complicated to bring together the changes of the different systems into a consistent state.

## Black box migration

Often the code of the deployment monolith is hard to understand and modify. This might even be a reason for migration.

Therefore, it does not make a lot of sense to reverse-engineer the existing code or even restructure it. That way, migrating an existing system requires little or a minimum of knowledge of the system.

### Choosing the first microservice for the migration

A legacy system comprises numerous domain functionalities. For deciding about the migration strategy, it can be useful to analyze the domain logic of the legacy system. The result should be a complete and split of the legacy system into [bounded contexts](https://www.educative.io/collection/page/10370001/6518081205567488/4953894968885248).

It is not implemented in the legacy system but can be the goal for the migration to microservices. This analysis can be done without understanding the code. **It is about what the system does, meaning it is enough to treat it as a black box**.

Extracting one of these bounded contexts as a microservice has this advantage: a bounded context is largely independent of other bounded contexts from a domain perspective since it has its own domain model.

However, the question is, **which bounded context should we extract first** from the legacy system? There are a few different approaches.

* To keep the risk as low as possible, **an unimportant bounded context with little load** can be the right choice. This makes it possible to gain experience with the challenges of microservices, for example, concerning operation, without taking too great a risk.
* Microservices are meant to simplify development. In order to exploit the advantages of this approach as quickly as possible, you can **migrate a bounded context to a microservice that will have to be changed a lot** in the foreseeable future. The changes should become easier to introduce after migrating to a microservice, so that the cost of migration quickly pays for itself.

## Extreme migration strategy: all changes in microservices

An extreme migration strategy serves to **prevent any changes to the legacy system, and allow only for changes to microservices**.

When a change would have to be made to the legacy system, a new microservice must be created first. The change is then implemented in this microservice instead.

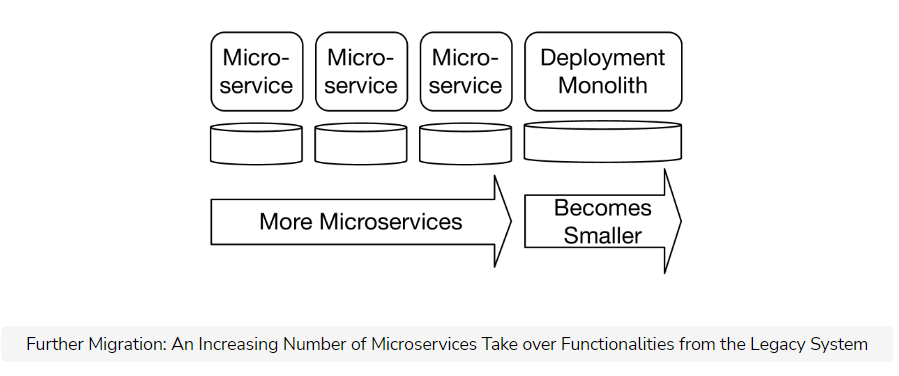
This automatically results in a migration to microservices as more and more logic is implemented in microservices over time. It is very easy to follow this rule.

One problem with this approach is that the microservices are created in random places, namely where the system is currently being changed.

This can result in microservices that implement only parts of a bounded context, whereas other parts of the bounded context are still implemented in the legacy system. Therefore, microservices and legacy systems have numerous dependencies, **making independent development difficult**.

## Further procedure: step-by-step migration

The **legacy system can be gradually replaced by microservices**. In the course of the migration, the focus should be on converting parts of the system into microservices that are currently undergoing major changes so that the migration to microservices is worthwhile. This is called the [**strangler pattern**](https://www.martinfowler.com/bliki/StranglerApplication.html). The microservices increasingly strangle the legacy system until nothing is left of the legacy system anymore. Have a look at the drawing below for a visual.



The full migration to microservices **can take a very long time**. However, this is not a problem: **only those parts of the system are migrated for which migration brings an advantage**.

For example, if a part of the system must be changed, it is migrated to a microservice. That makes the changes much easier. Parts that are never or very seldom changed will be migrated later or even never.

The total time the full migration takes is a result of the flexibility to migrate only what is actually needed. The \*migration stops if there is no longer anything worth migrating. In the end, it makes no sense to invest in the optimization of system parts that are rarely or never changed.

It might even happen that the legacy system could, in theory, be completely migrated, but still retained. Retaining the legacy system can be the best solution when hardly any changes to the legacy system are necessary because all parts that require changes have been migrated to microservices.

In a black box migration scenario, which bounded context is a good choice to extract first?

###### C)An unimportant bounded context that does not have too much load.

Changes to the data can be communicated via?

###### A)Events

In what circumstances can synchronous communication be preferable?

###### B)When the changes need to take effect as soon as possible

# Alternative Strategies

In this lesson, we'll study a few alternative migration strategies.

**We'll cover the following**

* [Goal: reliability](https://www.educative.io/module/lesson/introduction-to-microservices/xoW2JxrVlAE#Goal:-reliability)
* [Migration based on layers](https://www.educative.io/module/lesson/introduction-to-microservices/xoW2JxrVlAE#Migration-based-on-layers)
* [Copy/change](https://www.educative.io/module/lesson/introduction-to-microservices/xoW2JxrVlAE#Copy/change)
  + [Caveats](https://www.educative.io/module/lesson/introduction-to-microservices/xoW2JxrVlAE#Caveats)

There are many more migration strategies. There is a [presentation](https://speakerdeck.com/ewolff/monolith-to-microservices-a-comparison-of-strategies) that gives a good overview of the different approaches to the migration to microservices. Let’s go over some of the most common approaches.

## Goal: reliability

As mentioned previously, there can be very divergent approaches for the migration to microservices. The strategy depends mainly on the objectives to be achieved.

When the main objective for switching to microservices is **an increase in robustness**, at first, **reliability can be improved at the interfaces to external systems** or databases with libraries like [Hystrix](https://github.com/Netflix/Hystrix) and [Resilience4j](https://github.com/resilience4j/resilience4j).

Then, **the system can be split step by step into individual microservices that run independently** of each other so that a failure of one microservice no longer affects the other microservices. There is an interesting talk about this approach to which [slides](https://www.innoq.com/de/talks/2015/11/javaday-kyiv-modernization-legacy-systems-microservices-hystrix/) are available.

## Migration based on layers

Another alternative is a **migration based on layers**.

For example, the UI can be migrated first. This can make sense when changes to the UI are imminent, and therefore the migration can be combined with necessary changes.

Of course, this migration strategy is in contrast to the idea of combining UI, logic, and data in one self-contained system.

However, it can still be the first step towards this goal. In that case, **the remaining layers would have to be migrated into the same microservice afterward**. Alternatively, one stays with a division of microservices in layers, although it is not optimal. An ideal architecture, however, into which it is impossible to migrate, is a lot less helpful than a less optimal architecture that can actually be implemented.

## Copy/change

Another possibility is copy/change. Here, the code of the **legacy system is copied**.

* In one copy, **one part of the system is developed further, while the other part is removed**.
* In a second copy, **it is the other way around**.

In this manner, the legacy system is converted into **two microservices**.

This approach has the **advantage** that the **old code is still used**, and therefore the functionalities of the microservices very likely correspond accurately to the functionalities of the legacy system.

### Caveats

* However, **at the same time it is a great disadvantage to continue using the old code**. In most cases, the code of a legacy system is hard to maintain, and it is problematic to keep using this code.
* In addition, the database schema remains unaltered. The **shared use of the database schema by the legacy system and the microservices results in a tight coupling between the two**, which really should be avoided to be able to profit from the advantages microservices offer. That is why a black box migration might be better.
* Moreover, the structure of the legacy system and the technology stack largely remain the same. Thus, the project has a lot of **technical debt from the very beginning** and does **not represent a new start**.
* This approach does not take advantage of the benefits of microservices such as freedom of technology.

Therefore, **it should only be used in exceptional cases**.

###### Suppose as part of a migration, the UI, logic and data are all migrated separately. What migration strategy is likely being followed here? C)Layer-based migration

###### When should the copy/change strategy be used? B)When old code and functionality of the legacy system needs to be kept

# Build, Operation, and Organization

In this lesson, we'll discuss the build, operation, and organization pertaining to moving a legacy system to a microservices architecture.

**We'll cover the following**

* [Co-existence between microservices and legacy systems](https://www.educative.io/module/lesson/introduction-to-microservices/NEWPqLw4zk2#Co-existence-between-microservices-and-legacy-systems)
* [Integration test of microservices and legacy systems](https://www.educative.io/module/lesson/introduction-to-microservices/NEWPqLw4zk2#Integration-test-of-microservices-and-legacy-systems)
* [Coordinated deployment between legacy systems and microservices](https://www.educative.io/module/lesson/introduction-to-microservices/NEWPqLw4zk2#Coordinated-deployment-between-legacy-systems-and-microservices)
* [Organizational aspects](https://www.educative.io/module/lesson/introduction-to-microservices/NEWPqLw4zk2#Organizational-aspects)
* [Recommendation: do not implement all aspects at once](https://www.educative.io/module/lesson/introduction-to-microservices/NEWPqLw4zk2#Recommendation:-do-not-implement-all-aspects-at-once)

Code migration alone is not enough to turn a legacy system into a microservices system.

* **The microservices must also be built**. A suitable tool must be selected for this purpose. In addition, the continuous integration server has to cope with the multitude of microservices.
* Similarly, technologies and approaches must be introduced to **enable the deployment and operation** of microservices.
* Finally, **a suitable test strategy must be established**. This also requires the automated setup of test environments and the assurance that the tests are independent.
  + For example, stubs that simulate microservices or the legacy system are useful for this purpose, as are [consumer-driven contract tests](https://martinfowler.com/articles/consumerDrivenContracts.html). They safeguard the requirements for the interfaces of microservices or legacy systems with the help of tests.
  + However, legacy systems are often very complicated, so these techniques are difficult to implement.

Therefore, dealing with the first microservice can require extra effort because the infrastructure for build and deployment needs to be set up. It is conceivable to build the infrastructure later, but it is recommended to start building the infrastructure as early as possible in order to reduce the risk of migration.

One or a few microservices can still be operated with an inadequate solution for build and deployment. However, once the number of microservices increases, without appropriate infrastructure, the necessary effort will become so high that it can lead to project failure.

## Co-existence between microservices and legacy systems

During migration, the legacy system must be deployed and further developed in addition to the microservices. It is unrealistic to deploy the legacy system as often as the microservices because the effort of deploying the legacy system is usually far too high.

Therefore, **changes affecting both the legacy system and the microservices are difficult to implement**. They require at least one deployment of the microservices and one deployment of the legacy system. Solutions can be found at the architectural level.

If a new feature is implemented only in a microservice, then the deployment of only this microservice is necessary. This speaks for a division of the microservices according to bounded context.

**Another option** would be to **integrate the monolith with patterns** such as open host service or published language, as described in the [previous chapter](https://www.educative.io/collection/page/10370001/6518081205567488/4851164619210752), to provide a generic interface that rarely needs to be changed.

## Integration test of microservices and legacy systems

There must also be integration tests that test microservices with the version of the legacy system currently in production and with the one currently being developed. This ensures that the microservices continue to work when the legacy system is deployed.

The legacy system can support two different versions of the interfaces, so that microservices can switch to a new version of an interface when it is provided. However, no microservice is forced to use a new interface that has not yet been tested together with the microservice. In this way, the version of the microservice that uses a new interface of the legacy system can be deployed at any time.

## Coordinated deployment between legacy systems and microservices

Coordinated deployment of microservices together with the legacy system is an alternative. When a change is made, the new version of **the microservices and the legacy system are rolled out at the same time**. However, this approach has a few disadvantages:

* This increases the risk because more changes occur at the same time
* It is harder to roll back the deployment.
* It is also difficult to implement this approach without downtime.
* With a complex microservices environment, this option is hardly possible anymore because too many microservices would have to be deployed at once.

Therefore, the deployment of microservices and legacy systems should be decoupled from the outset.

## Organizational aspects

An essential advantage of microservices is the possibility to [scale the development process](https://www.educative.io/collection/page/10370001/6518081205567488/4998953437233152).

If the goal of migration to microservices is to have independent teams, the migration of the architecture must be accompanied by a reorganization. [This lesson](https://www.educative.io/collection/page/10370001/6518081205567488/5143635785940992) in the previous chapter discusses the essential aspects of the target organization.

The organizational change must be coordinated with the technical migration. For example, a microservice can be detached from the legacy system and then developed autonomously by a team. At the same time, the other organizational structures can be set up – for example, the ones required for defining the macro architecture.

## Recommendation: do not implement all aspects at once

Microservices require changes in architecture and organization as well as the introduction of new technologies.

Implementing all these changes at once is risky and complicated. Unfortunately, many of the changes are connected. Without new technologies, the architecture is difficult to implement. Without the architecture, organizational changes are difficult to introduce.

However, making all these changes at once should still be avoided. For each change, the question as to whether the change is actually necessary should be asked, in order to implement it at a later point in time.

Why are changes that affect both the legacy system and the microservices difficult to implement?

That entails deploying the legacy system which takes considerable effort

That entails deploying the microservices which takes considerable effort

# Variations

The ideas for migration can easily combine with many other approaches. Let's study a few.

**We'll cover the following**

* [Combining approaches to migration](https://www.educative.io/module/lesson/introduction-to-microservices/JYo5Z6j4BlD#Combining-approaches-to-migration)
* [Experiments](https://www.educative.io/module/lesson/introduction-to-microservices/JYo5Z6j4BlD#Experiments)

## Combining approaches to migration

* The ideas concerning typical migration strategies from the lesson on [typical migration strategies](https://www.educative.io/collection/page/10370001/6518081205567488/4984247502766080) fit very well to the concept of **self-contained systems (SCS)**. The migration can therefore simply separate a part of the legacy system into an SCS.
* Rules for authentication or communication between microservices as well as between microservices and the legacy system can be the starting point of a **macro architecture** (see [chapter 3](https://www.educative.io/collection/page/10370001/6518081205567488/6218432796164096)). A domain macro architecture is very useful, which can also include the legacy system in addition to microservices.
* **Frontend integration** can make sense for the integration between the legacy system and microservices.
* **Asynchronous microservices** fit very well with migration because they allow for a loose coupling. Especially for a migration, it can be sensible to continue to use an existing messaging technology for asynchronous communication to minimize the effort.
* **Synchronous microservices** should be used cautiously because this creates a tight coupling and resilience becomes difficult.
* **Kubernetes, PaaS, or Docker** are also interesting in a migration scenario. However, they represent a new environment that needs to be operated. It may, therefore, make sense to use a classical deployment and operation environment at least at the beginning to reduce the initial migration effort. In the long term, however, such environments have many advantages. In addition, of course, the old system can be operated in such an environment.

## Experiments

The migration strategy must match the respective scenario. The following questions are important in order to design your own strategy.

* What are the **goals of the migration** to microservices?
  + Which are especially important?
  + What impact does this have on the migration strategy?

In principle, **migration should take place gradually**. The selection of the parts to be migrated to microservices can be made according to technical or domain criteria. However, domain criteria are better suited, at least in the long term.

The **following approach is suitable for a migration based on domain criteria**:

* Split the system into bounded contexts.
* Which of the bounded contexts will you migrate first? Why? Reasons can be the simple migration of the bounded context or many planned changes in the bounded context. Consider the different scenarios.

In the next lesson, we’ll look at a quick chapter conclusion.