**Introduction**

In this lesson, we'll walk through what this chapter holds for us.

**We'll cover the following**

* [Motivation](https://www.educative.io/module/lesson/introduction-to-microservices/gxBELwVE02j#Motivation)
* [Definition](https://www.educative.io/module/lesson/introduction-to-microservices/gxBELwVE02j#Definition)
* [Chapter walkthrough](https://www.educative.io/module/lesson/introduction-to-microservices/gxBELwVE02j#Chapter-walkthrough)

**Motivation**[#](https://www.educative.io/module/lesson/introduction-to-microservices/gxBELwVE02j#Motivation)

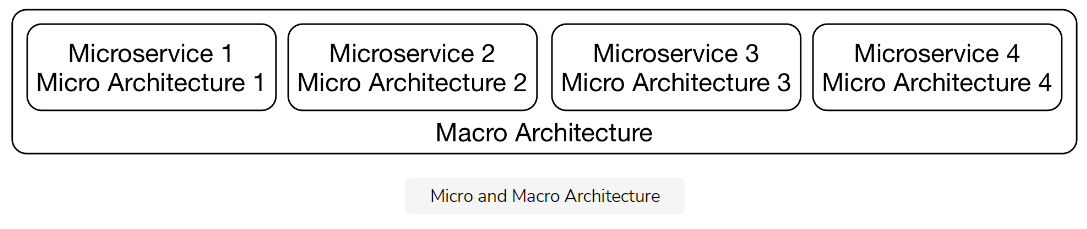
Microservices provide much better decoupling. Therefore, they help to modularize and isolate software modules (see [Advantages](https://www.educative.io/collection/page/10370001/6518081205567488/4998953437233152)). However, microservices are modules of a larger system. Therefore, they must be integrated. This poses a challenge for the architecture:

* On the one hand, the architecture has to **ensure that the microservices can work together** to form the overall system.
* On the other hand, the freedom of the **microservices should not be too restricted since this would compromise their isolation** and independence which are required for most of the benefits of a microservice architecture.

**Definition**[#](https://www.educative.io/module/lesson/introduction-to-microservices/gxBELwVE02j#Definition)

For this reason, it is advisable to divide the architecture into a micro and a macro architecture.

* The **micro architecture** comprises all decisions that can be made individually for each microservice.
* The **macro architecture** consists of all decisions that can be made at a global level and apply to all microservices.



The drawing above illustrates this idea. The overarching **macro** architecture applies to **all microservices**, whereas the **micro** architecture deals with **individual microservices** so that each microservice has its own microarchitecture.

**Chapter walkthrough**[#](https://www.educative.io/module/lesson/introduction-to-microservices/gxBELwVE02j#Chapter-walkthrough)

This chapter illustrates the following:

* The **division of domain logic** into microservices. *Domain-driven design* and *bounded context* are great approaches for such a division.
* The decisions that are part of the *technical micro and macro architecture* and how a **DevOps model** affects these decisions.
* The question of **who** divides the decisions into micro and macro architecture and creates the macro architecture.

1. The e-commerce system discussed in the last chapter, can be divided into microservices like so:

* ordering
* registration
* product search

Suppose the product search team decides to optimize search with a new algorithm. Is this a micro or macro architecture decision?

###### A) Micro architecture

2. It does not matter if the microservices can work with each other.

###### B) False (It matters. The microservices’s ability to work together is what makes the final system.)

The **macro architecture** consists of all decisions which have to be made at an individual level for each microservice.

###### B) False

In the next lesson, we’ll look at domain-driven design and an introduction to bounded contexts and strategic design.

**Domain-Driven Design & Bounded Contexts**

In this lesson, we'll discuss what domain-driven design is and how bounded contexts fit into that definition.

**We'll cover the following**

* [Bounded context and strategic design](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#Bounded-context-and-strategic-design)
  + [An example for a domain architecture](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#An-example-for-a-domain-architecture)
* [Domain-driven design: definition](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#Domain-driven-design:-definition)
* [Bounded context: definition](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#Bounded-context:-definition)
  + [Multiple bounded contexts](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#Multiple-bounded-contexts)
* [Domain events between bounded contexts](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#Domain-events-between-bounded-contexts)
  + [Example](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#Example)
* [Bounded contexts and microservices](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#Bounded-contexts-and-microservices)
* [Evolution](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#Evolution)

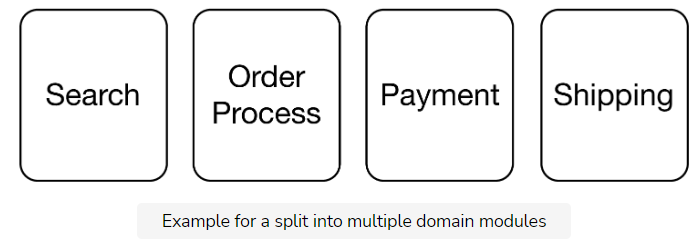
**Bounded context and strategic design**[#](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#Bounded-context-and-strategic-design)

Regarding to the domain architecture, the concept of micro and macro architecture has long been a common practice. A macro architecture divides the domains into coarse-grained modules. These modules are further divided as part of the micro architecture.

For example, an e-commerce system can be divided into modules and sub-modules as follows:

* **Customer registration**
* **Order process**
  + Data validation
  + Freight charge calculation
* **Payment**
* **Shipping**

The internal architecture of the **order process** module is, however, hidden from the outside and can be altered without affecting other modules. This **flexibility to change one module without influencing the other** modules is one of the main advantages of modular software development.



### An example for a domain architecture [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#An-example-for-a-domain-architecture)

The drawing above shows an example of the division of a system into multiple domain modules. In this division, each module has its own domain model. Let’s discuss each.

* To **search** successfully, data, such as descriptions, images or prices, must be stored for the products. Important customer data can include, for example, the recommendations that can be determined based on past orders.
* To process orders in the **order process** module, the contents of the shopping cart have to be tracked. For products, only basic information is required such as name and price. Similarly, not too much data concerning the customer is necessary. The most important component of the domain model of this module is the shopping cart. It is then turned into an order that has to be handed over and processed by the other bounded contexts.
* For **payment**, the payment-associated information like credit card numbers has to be kept for each customer.
* For **shipping**, the delivery address is required information about the customer while the size and the weight are necessary information about the product.

This list reflects that **the modules require different domain models**. Not only does the data concerning customer and product differ but so does the entire model and the logic.

## Domain-driven design: definition [#](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#Domain-driven-design:-definition)

**Domain-driven design (DDD)** offers a collection of **patterns** for the domain model of a system. For microservices, the patterns in the area of strategic design are the most interesting. They describe how a domain can be subdivided.

Here are some books you could look into if you are interested in Domain-Driven Design:

* Domain-driven design offers many more patterns that, for example, facilitate the model of individual modules. The original [DDD book](https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215) provides a lot more information. It introduces the term “domain-driven design” and comprehensively describes DDD.
* The more compact book [Domain-driven Design Distilled](https://www.amazon.com/Domain-Driven-Design-Distilled-Vaughn-Vernon/dp/0134434420) focuses on design, bounded context, and domain events.
* The [Domain-Driven Design Reference](https://domainlanguage.com/ddd/reference/) is also by the author of the original DDD book. It contains all DDD patterns but without any additional explanation or examples.

## Bounded context: definition [#](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#Bounded-context:-definition)

Domain-driven design speaks of a **bounded context**. Each domain model is valid only in a bounded context.

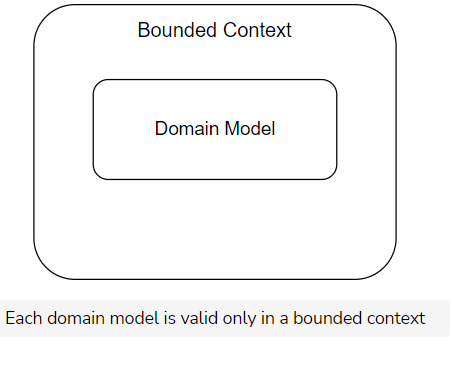
Consequently, search, order process, payment, and shipping **are such bounded contexts** because they each have their own domain model.

### Multiple bounded contexts [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#Multiple-bounded-contexts)

It would be conceivable to implement a domain model that comprises **multiple bounded contexts**. However, such a model would not be the easiest solution.

For example, a price change affects search; however, it must not result in a price change for orders that have already been processed in payment. It is easier to store only the current price of a product in the bounded context search, and to store the price of the product of each order in payment, which can also comprise rebates and other complex logic.

Therefore, the simplest design consists of **multiple specialized domain models that are valid only in a certain context**. Each domain model has its own model for business objects such as customers or products.



## Domain events between bounded contexts [#](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#Domain-events-between-bounded-contexts)

For the communication between bounded contexts, we can use **domain events**.

Events can be useful for integrating bounded contexts. Domain events are a part of the domain model as they represent something that happened in the domain. That means they should also be relevant to domain experts.

### Example [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#Example)

* Ordering a shopping cart can be modeled as such an event.
* This event is triggered by the bounded context order process and is received by the bounded contexts shipping and payment to initiate shipping and invoicing of the order.

## Bounded contexts and microservices [#](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#Bounded-contexts-and-microservices)

Bounded contexts divide a system by domains. They **do not have to be microservices**. They can also be implemented as modules in a deployment monolith.

If the bounded contexts are implemented as microservices, this results in modules that are independent at the domain and technical level. Therefore, it is sensible to combine the concepts of microservices and bounded contexts.

The dependencies of the bounded contexts as part of strategic designs, as we’ll learn in the next lesson, limit this independence. However, since the microservices are part of a larger system, **dependencies between the modules cannot be completely avoided**.

## Evolution [#](https://www.educative.io/module/lesson/introduction-to-microservices/7nDqK5rG691#Evolution)

There are a number of reasons why new bounded context, and therefore new microservices, might be created:

1. Over time, **new functionalities** might justify **new bounded contexts**.
2. It might become apparent that one bounded context should really be split into two. That might be the case because new logic is added to the bounded context, or the team understands the bounded context better.
3. **New microservices** might be created by dividing a current one due to a **technical reason** (recall [division by technicality](https://www.educative.io/collection/page/10370001/6518081205567488/4532272759832576#two-levels-of-microservices-domain-and-technical)!).
   * One reason may be to make scalability easier. A microservice may be split in two since the resulting microservices will be smaller and therefore easier to scale. Such reasons might also lead to a larger number of microservices.

1.Suppose you’re given the following e-commerce system:

* **Customer registration**
* **Order process**
  + Data validation
  + Freight charge calculation
* **Payment**
* **Shipping**

What will happen to the rest of the modules if the internal architecture of **order process** is changed as follows:

* **Customer registration**
* **Order process**
  + Data validation
  + *Input sanitation*
  + Freight charge calculation
* **Payment**
* **Shipping**

###### B) Nothing - the rest of the modules will not have to change anything.

2. Communication between bounded contexts can be triggered from \_\_\_.

###### Domain events

Why are domain models that span multiple bounded contexts difficult to implement?

###### The domain model would be too complex

In the next lesson, we’ll discuss some strategic design and its key patterns.

**Strategic Design & Common Patterns**

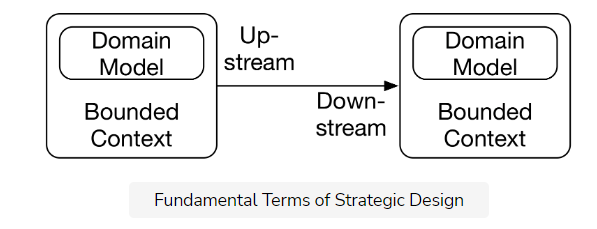
In this lesson, we'll get an introduction to strategic design and look at some important strategic design patterns.

**We'll cover the following**

* [Strategic design](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Strategic-design)
* [The customer/supplier pattern](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#The-customer/supplier-pattern)
  + [Example](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Example)
* [The conformist pattern](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#The-conformist-pattern)
  + [Example](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Example)
* [The anti-corruption layer](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#The-anti-corruption-layer)
  + [Example](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Example)
* [The separate ways pattern](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#The-separate-ways-pattern)
  + [Example](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Example)
* [The shared kernel pattern](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#The-shared-kernel-pattern)
  + [Example](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Example)
* [The open host service pattern](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#The-open-host-service-pattern)
* [The published language model](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#The-published-language-model)
  + [Example](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Example)
* [Selecting patterns](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Selecting-patterns)
  + [Example](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Example)
    - [Tradeoffs to Consider](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Tradeoffs-to-Consider)

**Strategic design**[#](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Strategic-design)

The division of the system into different bounded contexts is part of **strategic design**, which belongs to the practices of domain-driven design (DDD). The strategic design describes the *integration* of bounded contexts.



The drawing above shows the fundamental terms of strategic design.

* The **bounded context** is the context in which a specific **domain model** is valid.
* The bounded contexts depend on each other. Usually, each bounded context is implemented by one team.
* The **upstream** team can influence the success of the **downstream** team. However, the downstream team cannot influence the success of the upstream team.
  + For example, the success of the team responsible for payment depends on the order process team.
  + If data such as prices or credit card numbers are not part of the order, it is impossible to do the payment.
  + However, the order process does not depend on the payment to be successful.
  + Therefore, **order processing is upstream**. It can make payment fail. **Payment is downstream** since it cannot make the order process fail.

DDD describes in several patterns how exactly communication takes place. These patterns not only **describe the architecture**, but also the **cooperation within the organization**.

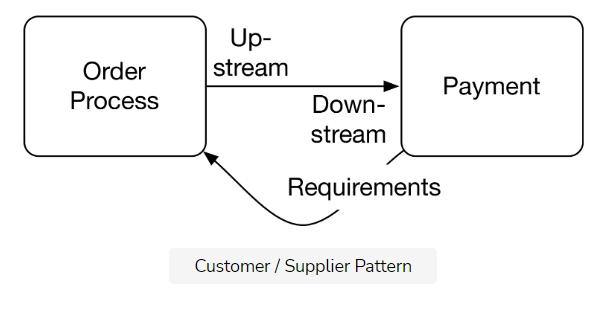
## The customer/supplier pattern [#](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#The-customer/supplier-pattern)

With this customer/supplier pattern, the **supplier is upstream** and the **customer is downstream**. However, the customer can factor their priorities into the planning of the upstream project.

### Example [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Example)

In the drawing below, for example, payment uses the model of the order process. However, payment defines requirements for the order process. Payment can only be done successfully if the order process provides the required data.

So, payment can become a customer of the order process. That way the customer’s requirements can be included in the planning of the order process.



## The conformist pattern [#](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#The-conformist-pattern)

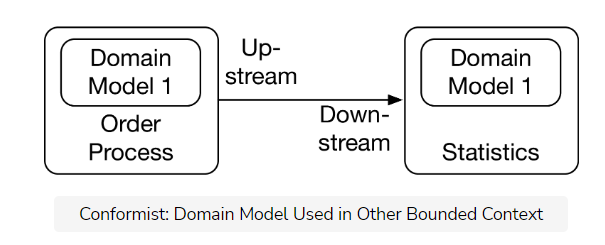
**Conformist** means that **a bounded context simply uses a domain model from another bounded context**.

### Example [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Example)

In the drawing below, the bounded contexts, **statistics**, and **order process**, both **use the same domain model**. The statistics are part of a data warehouse. They use the domain model of the order process bounded context and extract some information relevant to store in the data warehouse.

However, with the conformist pattern, the data warehouse team **does not have a say** in case of changes to the bounded context.

The data warehouse team could not demand additional information from the other bounded context. However, it is still possible that they would receive additional information out of altruism. Essentially, the data warehouse team is not deemed important enough to get a more powerful role.



## The anti-corruption layer [#](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#The-anti-corruption-layer)

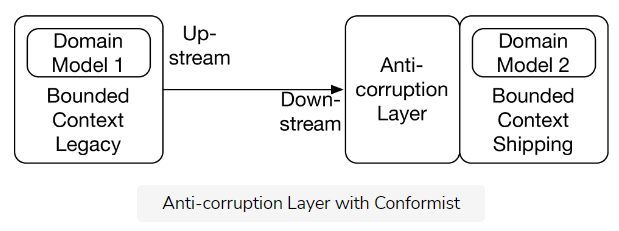
In the case of an **anti-corruption layer (ACL)** pattern, the bounded context does not directly use the domain model of the other bounded context, but it contains a layer for decoupling its own domain model from the model of the bounded context.

This is useful in conjunction with the conformist pattern to generate a separate model decoupled from the other model.

### Example [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Example)

The drawing below shows that the bounded context shipping uses an ACL at the interface to the bounded context legacy so that both bounded contexts have their own independent domain models.

This ensures that the model in the legacy system does not affect the bounded context shipping. Shipping can implement a clean model in its bounded context.



## The separate ways pattern [#](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#The-separate-ways-pattern)

With the **separate ways pattern**, the bounded contexts are **not related** at the software level although a relation would be conceivable.

### Example [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Example)

Let’s assume that in the e-commerce scenario, a new bounded context, **purchasing**, for the purchase department is added. This bounded context could collect the data for listing products, but it is implemented differently.

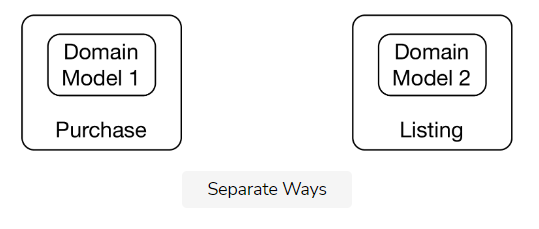
With the separate ways pattern, the purchasing would be separate from the remaining system. When the goods are delivered, a user would use another bounded context like listing to enter the necessary data and list the products.

The purchasing causes the shipping, which in turn triggers the delivery, and thereby triggers the user to list the product with a different bounded context.

***purchasing → shipping → delivery → list product***

The shipping of the products is one event in the real world, however, in the software, the systems are separate.

Consequently, the systems are independent and can be evolved completely independently.



## The shared kernel pattern [#](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#The-shared-kernel-pattern)

**The Shared Kernel Pattern describes a common core that is shared by multiple bounded contexts**.

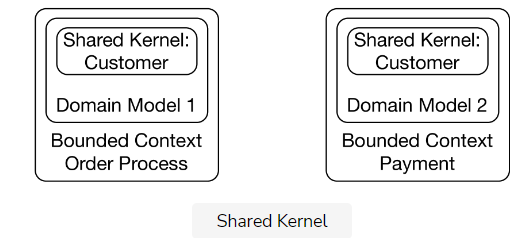
The data of a customer is an example of such a scenario.

However, the shared kernel comprises shared business logic and shared database schema and therefore **should not be used in a microservices environment**.

It is an anti-pattern for microservices systems. But because DDD can also be applied to deployment monoliths, there are still scenarios in which a shared kernel makes sense.

### Example [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Example)

In the drawing below the domain model order process and the payment possess a shared kernel.



Some patterns are primarily useful in cases where more than one bounded context has to be integrated.

## The open host service pattern [#](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#The-open-host-service-pattern)

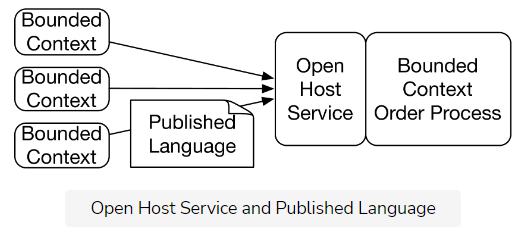
Open host service means that the bounded context offers a generic interface with several services. Other bounded contexts can implement their own integration with these services. This pattern is frequently found at public APIs on the Internet. However, it is also a possible alternative within an enterprise.

## The published language model [#](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#The-published-language-model)

Published language is a domain model accessible by all bounded contexts. For example, this can be a standard format such as EDIFACT for transactions between companies. But it is also possible to define a data structure that is only used inside a company and published, for example, in Wiki.

### Example [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Example)

These models can be used together. The open host service can use published language for communication. For example, the order process might accept orders from external clients. Providing a specific interface for each external client is a lot of effort, so there is a generic open host service and a published language for orders. Each external client can use this interface to submit orders to the bounded context order process.



## Selecting patterns[#](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Selecting-patterns)

The choice of patterns has to be in line with:

1. The domain
2. The power structures
3. The communication relationships between the teams.

### Example[**#**](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Example)

When the bounded context payment does not obtain the necessary data from the bounded context order processing, the products can be ordered but not paid for. Therefore, the **customer/supplier pattern** is an obvious choice.

However, this is not a fact found in the domain, but rather a **consequence of the power structure**, which in turn depends on the company business model.

#### Tradeoffs to Consider[**#**](https://www.educative.io/module/lesson/introduction-to-microservices/RMGONYmMJvO#Tradeoffs-to-Consider)

Of course, the selected patterns influence the effort necessary for coordination and therefore the degree of isolation between the teams. They set the rules by which the teams must work on the integration.

Thus, a pattern like customer/supplier is not desirable as it **requires a lot of coordination**. Still, it might be the right solution depending on domain aspects.

It makes little sense to use a different pattern between payment and the order process just to have less coordination. A different pattern might make it impossible for the business to succeed.

**QUIZ:**

Consider the following two teams as a part of a small social media app.

1. Image processing. They handle operations on images such as compression and storage.
2. Photo Albums. They handle the look of each user’s ‘photo albums’.

Which team is upstream and which is downstream in terms of strategic design?

###### The **image processing** team is **upstream** and the **photo albums** team is **downsteam**. (The success of the photo album team largely depends on images being processed correctly which is the responsibility of the image processing team.)

Consider a pizza delivery app. The two bounded contexts, **delivery** and **purchase** are done by different teams. However, delivery can give requirements to purchase. Which pattern describes this situation?

###### B)Customer / Supplier

Which pattern should NOT be used in a microservices scenario?

###### The shared kernel pattern

How to pass configuration patterns to a microservice MUST be a decision of the individual microservice.

###### False

In the next lesson, we’ll discuss architecture decisions.

# Architecture Decisions

In this lesson, we'll study some key decisions and at what architecture level, micro or macro, they should be taken.

**We'll cover the following**

* [Micro and macro architecture decisions](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Micro-and-macro-architecture-decisions)
  + [Programming languages, frameworks, and infrastructure](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Programming-languages,-frameworks,-and-infrastructure)
  + [Database](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Database)
  + [User interface](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#User-interface)
  + [Documentation](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Documentation)
* [Typical macro architecture decisions](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Typical-macro-architecture-decisions)
  + [Communication protocol](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Communication-protocol)
  + [Authentication](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Authentication)
  + [Integration](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Integration)
* [Typical micro architecture decisions](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Typical-micro-architecture-decisions)
  + [Authorization](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Authorization)
  + [Testing](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Testing)
* [To Summarize](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#To-Summarize)

Microservices provide technological isolation. Therefore, it is possible to extend the concept of micro and macro architecture to **technical decisions**.

For deployment monoliths, these decisions, inevitably, must be implemented globally.

So, only for microservices, technical decisions can be made within the framework of macro or micro architecture. However, some decisions have to be part of the macro architecture. Otherwise, the integration will be compromised.

## Micro and macro architecture decisions [#](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Micro-and-macro-architecture-decisions)

Decisions can be taken in the context of **either micro or macro architecture**. Let’s discuss each.

### Programming languages, frameworks, and infrastructure [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Programming-languages,-frameworks,-and-infrastructure)

Programming languages, frameworks, and infrastructure can be defined for each **microservice** individually at the **micro architecture**.

* Then each microservice can also be implemented with a different language.
* Technology, such as the application server, that best suits the specific problems of each microservice can be used.

Programming languages, frameworks, and infrastructure can be defined uniformly for all microservices in the **macro architecture**. This is useful if:

* a company’s technology strategy allows only certain technologies
* therefore, only developers with knowledge in certain technologies are hired

### Database [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Database)

At first glance, this decision seems to be comparable to the decision concerning the programming languages, frameworks, and infrastructure but databases are different because:

* They store data.
* The loss of data is usually unacceptable.
  + Therefore, there must be a backup strategy and a disaster recovery strategy for a database.
  + Setting these up for many different databases requires considerable effort.

**Micro**: Each microservice can also have its own instance of the database. If databases were defined at the **micro architecture**.

* A crash of one database will cause only one microservice to fail which makes the entire app **more robust**.
* However, the **higher effort involved**, especially concerning operation, is an argument against individual instances.

**Macro**: To avoid needing many different databases, the database can be defined as part of the **macro architecture** for all microservices.

* Even if the database is defined in the macro architecture, **multiple microservices must not share a database schema**. That would contradict the [bounded contexts](https://www.educative.io/collection/page/10370001/6518081205567488/4953894968885248).
* The domain model in the database schema would be used by several microservices. This would couple the microservices too strongly. Even with a unified database, the microservices must have separate schemata in the database.

### User interface [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#User-interface)

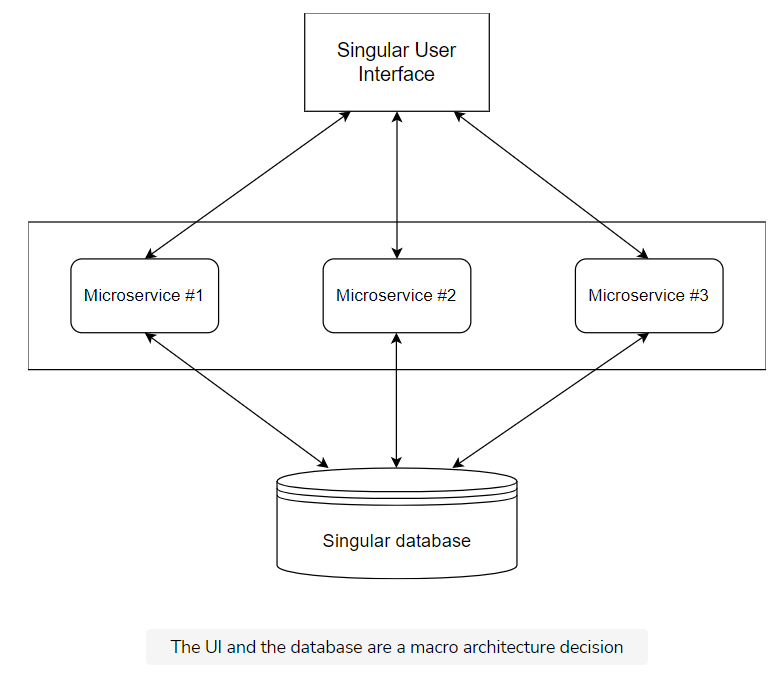
If microservices have their own user interface (UI), the look and feel of microservices can be a micro or macro architecture decision.

**Micro**: sometimes a system has **different types of users** (back office and customers, for example) with **different requirements** for the UI which are often incompatible with a uniform look and feel. A micro architecture decision for the UI is suitable in this case.

* Often there are concerns that a microservice level decision **will cause inconsistencies** in the look and feel; however, the UI can also **diverge in a monolithic system**. Hence, defining appropriate style guides and artifacts is the only way to achieve a consistent look and feel for large systems, regardless of the use of microservices.

**Macro**: Often a system should have a uniform UI; therefore, the look and feel must be a macro architecture decision.

* Shared CSS and JavaScript are often not enough to ensure a common style of the UI of all microservices since uniform technical artifacts can be used to implement very different types of user interfaces. Therefore, a **style guide must become part of the macro architecture**.



### Documentation [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Documentation)

It may be necessary to standardize the documentation.

**Micro**: The documentation should be part of the micro architecture if the same team will build and maintain the microservice.

* A certain level of documentation makes it easier to later hand the microservice over to another team.
* It may also be necessary to document certain aspects of the microservices in a uniform manner.
  + For example, for security reasons, some systems need to keep track of the libraries used in the microservices. In the case of a security vulnerability in a specific library, it is then possible to identify which microservices need to be fixed.

**Macro**: Of course, the decision about the documentation can also be part of the macro architecture.

* Standardized documentation can provide an overview of the system and the dependencies between microservices.

## Typical macro architecture decisions [#](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Typical-macro-architecture-decisions)

There are some decisions that must always be taken at the level of macro architecture. Ultimately, all microservices together should result in a coherent system. This requires some standards.

### Communication protocol [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Communication-protocol)

The communication protocol of the microservices is a typical macro architecture decision.

* Only if all microservices provide a **uniform interface**, for example, a REST interface or a messaging interface, can they communicate with each other coherently.
* In addition, the data format must be **standardized**. It makes a difference whether systems communicate with JSON or XML, for example.

If the communication protocol was a **microservice decision**, i.e., a different communication channel between each microservice, **a coherent system will not exist** and will disintegrate into islands that communicate with each other in different ways.

### Authentication [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Authentication)

With authentication, a **user proves their identity**. This can be done with a password and a username, to name a common example.

Since it is unacceptable for the user to re-authenticate with every microservice, the entire microservice system should use a **single authentication system**. The user then enters a username and password once and can then use any microservice.

### Integration [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Integration)

Integration testing technology is also a typical macro architecture decision. All microservices must be tested together, so they must run together in an integration test. The macro architecture must define the necessary prerequisites for this.

## Typical micro architecture decisions [#](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Typical-micro-architecture-decisions)

Certain decisions should be taken for each microservice individually. Therefore, they are typically part of the micro architecture.

### Authorization [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Authorization)

The authorization of the user determines what a user is **allowed to do**. The authorization should be done in the respective microservice.

Which user is allowed to initiate what action, i.e., authorization, is part of the domain logic, and therefore belongs to the microservice like the other domain logic.

If this was decided at the macro architecture, the domain logic would be implemented in a microservice itself, but the decision about **which part** of the domain logic is available to which user would be made centrally, which is difficult, especially with complex rules.

* For example, if orders up to a certain upper limit can be triggered by certain users, authorization, concrete upper limits, and possible exceptions belong to the microservice **order**.

Authentication assigns the user roles used in authorization.

* For example, a microservice can define which actions a user with the role of customer can trigger and which actions a user with the role of call center agent can trigger.

### Testing [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#Testing)

The testing can be different for each microservice. Even the tests are ultimately part of the domain logic.

In addition, there may be different non-functional requirements for each microservice.

* For example, one microservice can be particularly performance-critical, whereas another is more safety-critical.

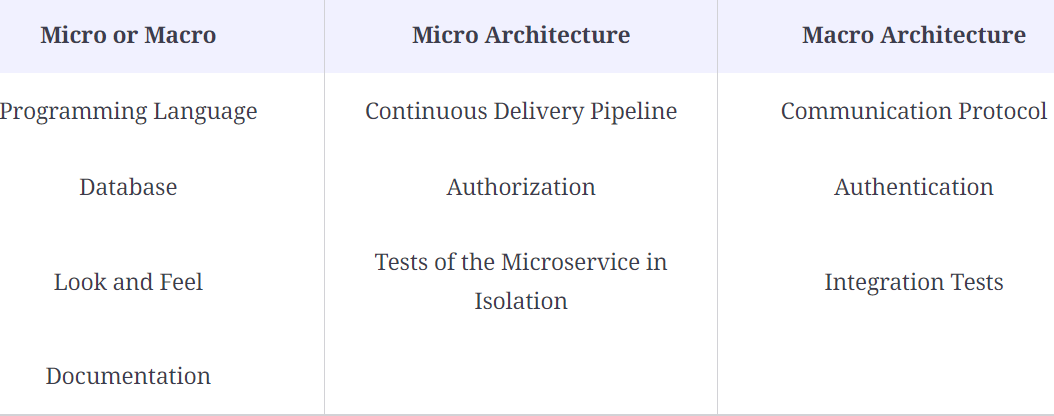
These risks must be covered by an individual focus in the tests.

Since the tests can be different, the **continuous delivery pipeline** is also different for each microservice. It must include the relevant tests. Of course, the technology for the continuous delivery pipeline can be standardized.

* For example, each pipeline can use a tool like Jenkins. What happens in the respective pipelines, however, depends on the respective microservice.

## To Summarize [#](https://www.educative.io/module/lesson/introduction-to-microservices/gk7oPq1gGpl#To-Summarize)

The following table shows the typical micro and macro architecture decisions:



QUIZ:

Suppose that it has been decided to use a REST interface between microservices for communication. What sort of technical decision could this have been?

###### Strictly macro

Suppose you’re part of a team that is building a small social media app. You come across a document that details the dependencies between all the microservices of the app. Was that document written as part of the macro architecture, micro architecture, or both?

###### Macro

Integration tests are a \_\_\_ architectural decision.

###### Macro

In the next lesson, we’ll discuss some factors that influence the operation of applications.

Operation: Micro or Macro Architecture?

In this lesson, we'll discuss some factors that influence the operation of applications.

We'll cover the following

* [Configuration](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#Configuration)
* [Monitoring](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#Monitoring)
* [Log Analysis](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#Log-Analysis)
* [Deployment Technology](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#Deployment-Technology)
* [Macro architecture operation with separate operations teams](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#Macro-architecture-operation-with-separate-operations-teams)
* [Standardize only technologies!](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#Standardize-only-technologies!)
* [Testing the operation macro architecture](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#Testing-the-operation-macro-architecture)
* [“You build it, you run it”: operation as micro architecture](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#%E2%80%9CYou-build-it,-you-run-it%E2%80%9D:-operation-as-micro-architecture)
* [Operation as a whole is micro or macro architecture](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#Operation-as-a-whole-is-micro-or-macro-architecture)

Some decisions in the area of micro and macro architecture mostly influence the operation of the applications. Let’s take a look at a few.

Configuration [#](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#Configuration)

We must define the interface with which a microservice obtains its configuration parameters. For example, a microservice can get these settings via an environment variable or read them from a configuration file. These parameters include both:

* Technical parameters such as thread pool sizes
* Parameters for the domain logic

The decision of how to store and generate the configuration data is independent of these parameters. The data can be stored in a database, for example. Either configuration files or environment variables can be generated from the data in the database.

Note that the information on which computer and under which port a microservice can be reached, does not belong to the configuration, but to the service discovery.

Configuring passwords or certificates is also a challenge that can be solved with other tools. To do this, [Vault](https://www.vaultproject.io/) is a good choice because this information must be stored in a particularly secure way and must be visible to as few employees as possible in order to prevent unauthorized access to production data.



## Monitoring [#](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#Monitoring)

Monitoring is about the **technology that tracks metrics**. Metrics provide information about the state of a system. Examples include the number of requests processed per second or business metrics, such as revenue.

The question of which technology is used to track the metrics is independent of which metrics are captured. Every microservice has different metrics because every microservice has different challenges. For example, if a microservice is under a very high load, then performance metrics are useful.

## Log Analysis [#](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#Log-Analysis)

Log analysis defines a **tool for managing logs**.

Although logs were originally stored in log files, they are now stored on **specialized servers**. This has a few advantages. For example, it makes it **easier to analyze and search the logs**, even with large amounts of data and many microservices.

In addition, new instances of a microservice can be started when the load increases and can be deleted again after the load decreases. In this case, the logs of this microservice instance should still be available, even if the microservice was deleted long ago due to a decreasing load. If the logs are stored only on a local device, the logs would be gone after the microservice has been deleted.

## Deployment Technology [#](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#Deployment-Technology)

Deployment technology determines **how the microservices are rolled out**. For example, this can be done with Docker images (see [chapter 6](https://www.educative.io/collection/page/10370001/6518081205567488/6331908113825792)), Kubernetes Pods, a PaaS, or installation scripts.

These decisions define how a microservice behaves from an operational point of view. Typically, these decisions are either all part of the macro architecture or the micro architecture.

## Macro architecture operation with separate operations teams [#](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#Macro-architecture-operation-with-separate-operations-teams)

Whether decisions in the area of operation belong to micro or macro architecture **depends on the organization**.

For example, a team can develop microservices but bear no responsibility for their operation. The operations team is responsible for the operation of all microservices. In this scenario, decisions for operation must be made at the level of macro architecture.

It is generally unacceptable for the operations team to learn a different approach for the operation of each microservice, especially because the number of microservices is much larger than the number of deployment monoliths for the same project.

Another reason for a macro architecture decision on the operation of microservices is that individual solutions in this area bring few advantages. Although a programming language or framework can be more or less suitable for a particular problem, the same applies to a much lesser extent to technologies in the field of operation.

## Standardize only technologies! [#](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#Standardize-only-technologies!)

When these decisions are made at the level of macro architecture, **they standardize only the technologies**.

Which configuration parameters, monitoring metrics, log messages, and deployment artifacts of a microservice are a decision at the level of the individual microservice.

The independent deployment must also be retained as a core feature of the microservices. This means that it must be possible to independently change the configuration parameters for each microservice in order to adjust the configuration when a new deployment takes place.

## Testing the operation macro architecture [#](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#Testing-the-operation-macro-architecture)

**Adherence to the macro architecture rules can be checked with tests**. The microservices are deployed in an environment. The tests check whether the rules for uniform deployment are adhered to. The test then verifies whether the microservice delivers metrics and log information in the defined way. Something similar is also possible for configuration.

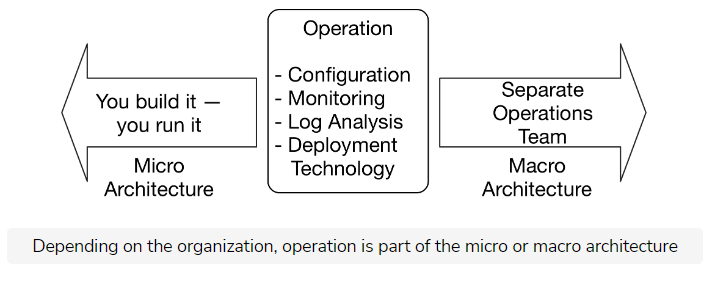
The test environment for these tests should be very minimalistic and should not contain any other microservices or a database. In this manner, the microservice is tested in an environment in which it cannot possibly work. When such a situation occurs in production, it is particularly important that the microservice provides logs and metrics to analyze potential problems. Similarly, the test also checks the resilience of the microservice.

## “You build it, you run it”: operation as micro architecture [#](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#%E2%80%9CYou-build-it,-you-run-it%E2%80%9D:-operation-as-micro-architecture)

There is a form of organization in which operational aspects have to be part of the micro architecture. If the same team is to develop and operate the microservice, they must also be able to choose the technology. This approach can be described as “you build it, you run it”. The teams are each responsible for a microservice, for its operation and development. You can only expect this level of responsibility from the team if you allow them to choose their own technologies.

## Operation as a whole is micro or macro architecture [#](https://www.educative.io/module/lesson/introduction-to-microservices/x1MPnLkxWA9#Operation-as-a-whole-is-micro-or-macro-architecture)

Decisions for operation can be taken either at the level of micro or macro architecture. Making operation decisions part of the macro architecture is useful if there is a separate operations team, while a “you build it, you run it” organization must make these decisions at the level of micro architecture. The drawing below illustrates this point.



**QUIZ:**

Which of the following best describes why storing logs on a specialized log server is adventageous?

###### The logs from all microservices would be available regardless of their current status and searching and analyzing logs is simpler.

Which configuration patterns to include MUST be a decision of the individual microservice.

###### True

In the next lesson, we’ll discuss the reasons for making as many decisions as possible at the micro architecture level rather than the macro architecture level.

**Give a Preference to Micro Architecture!**

There are good reasons for making as many decisions as possible at the micro architecture level rather than at the macro architecture level. Let's discuss each.

**We'll cover the following**

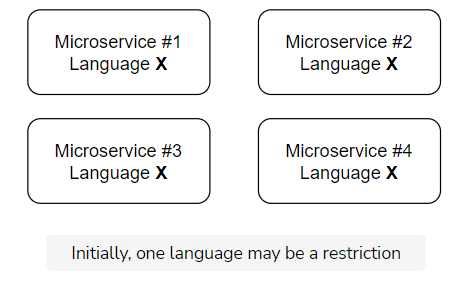
* [Macro architecture decisions: Best practices and advice](https://www.educative.io/module/lesson/introduction-to-microservices/JERLlD3ZJwy#Macro-architecture-decisions:-Best-practices-and-advice)
* [Evolution of macro architecture](https://www.educative.io/module/lesson/introduction-to-microservices/JERLlD3ZJwy#Evolution-of-macro-architecture)

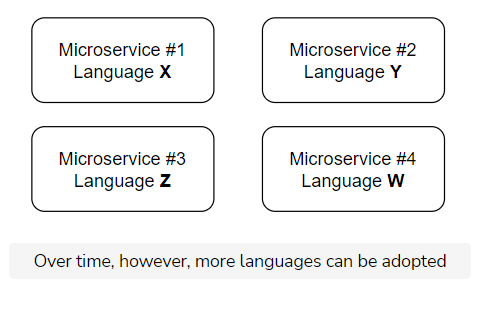
**Macro architecture decisions: Best practices and advice**[#](https://www.educative.io/module/lesson/introduction-to-microservices/JERLlD3ZJwy#Macro-architecture-decisions:-Best-practices-and-advice)

* Specifying only a few points in the macro architecture helps with focusing. Many teams have failed when trying to implement a far-reaching unification in a complex project or IT landscape. If there are few macro architecture rules, the chances increase that the rules are actually successfully implemented.
* The rules should be **minimal**.
  + For example, a macro architecture rule can *define the monitoring technology*. However, it is not necessary to standardize *how* the metrics are measured in the application. After all, it is important only that the metrics are created. How this happens is irrelevant. The macro architecture rule should only define a protocol for transferring metrics but leave the selection of the library for creating and transferring metrics to the micro architecture. In this way, the teams can choose the most appropriate technologies.
* The macro architecture rules have to be **consequently enforced**. For example, when the metrics are not generated, the operations team cannot simply bring the microservice into production. It is important to get rid of all unnecessary macro architecture elements.
* In addition, **independence** is an important goal of microservices. Too many macro architecture rules run counter to this goal as they hinder the independence of the teams through central control.
* Complying with macro architecture should be in the **self-interest** of the teams responsible for the microservices. Violations of macro architecture usually mean that microservices cannot go into production because operations cannot support them.

In addition to mandatory macro architecture rules, recommendations and best practices are advisable. However, they do not have to be enforced but are optional for every microservice.

In the end, the goal of macro architecture is to create freedom. Advice and references to best practices are therefore good additions.





## Evolution of macro architecture [#](https://www.educative.io/module/lesson/introduction-to-microservices/JERLlD3ZJwy#Evolution-of-macro-architecture)

At the beginning of a project, **restrictive rules may initially apply**. For example, a single programming language and a fixed stack of libraries can be defined. This reduces learning effort and operating costs.

Over the duration of the project, **more programming languages and libraries can then be allowed** to introduce modern technologies. This leads to a more heterogeneous system which is certainly preferred over updating all microservices at once because such updates entail a high risk.

**QUIZ:**

Which of the following macro architecture rules are better and why?

1. User data for analytics must be stored in a JSON format
2. The technology xyz should be used to gather user data for analytics and the data should be stored in a JSON format

###### Rule #1 because it is minimal

A microservices system may graduate from one where all microservices use one language to one where each uses a different language.

###### True

What could happen if a macro architecture rule is violated?

###### The microservice may not be able to go into production.

In the next lesson, we’ll discuss organizational aspects!

Organizational Aspects

In this lesson, we'll study some organizational aspects to architectural decisions.

We'll cover the following

* [Uncontrolled growth?](https://www.educative.io/module/lesson/introduction-to-microservices/gx7ZB3lMqjD#Uncontrolled-growth?)
* [Who defines macro architecture?](https://www.educative.io/module/lesson/introduction-to-microservices/gx7ZB3lMqjD#Who-defines-macro-architecture?)
  + [Committee of representative team members](https://www.educative.io/module/lesson/introduction-to-microservices/gx7ZB3lMqjD#Committee-of-representative-team-members)
  + [Independent architecture committee](https://www.educative.io/module/lesson/introduction-to-microservices/gx7ZB3lMqjD#Independent-architecture-committee)
* [How to enforce?](https://www.educative.io/module/lesson/introduction-to-microservices/gx7ZB3lMqjD#How-to-enforce?)
* [Testing conformance](https://www.educative.io/module/lesson/introduction-to-microservices/gx7ZB3lMqjD#Testing-conformance)

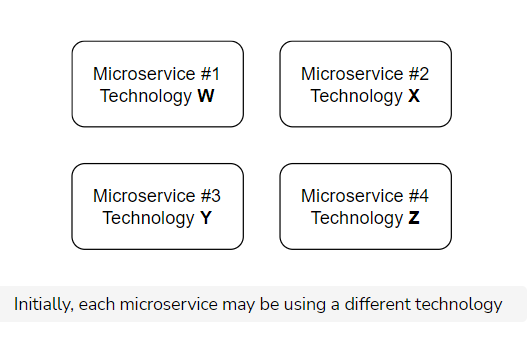
There is a connection between decision and responsibility. Whoever makes the decision takes responsibility. Therefore, if the decision about the technology of metrics is made as part of the macro architecture, then the macro architecture group must take responsibility. For example, they would be responsible if this technology proves unsuitable in the end because it does not cope with the amount of data.

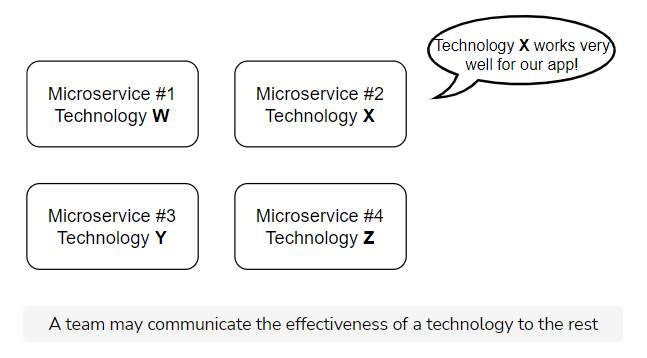
If the responsibility for monitoring microservices is completely transferred to the teams, then the teams must also be allowed to select a technology.

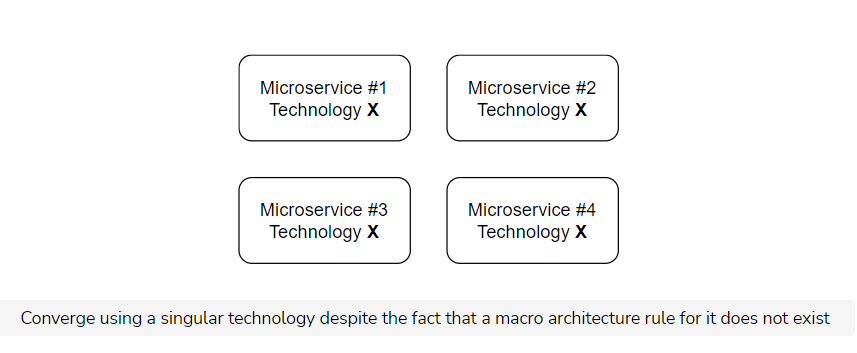
Uncontrolled growth? [#](https://www.educative.io/module/lesson/introduction-to-microservices/gx7ZB3lMqjD#Uncontrolled-growth?)

The freedom regarding micro architecture can lead to a huge number of technologies in use. But this is not necessarily the case. If all teams have had good experiences with a particular monitoring technology, then a new microservice will most likely be monitored with the same tool. Using another tool would require a great deal of effort. Other options are only evaluated and used if the tool used so far is insufficient.

Even without a macro architecture rule, there is standardization if uniform decisions bring advantages for the teams. The prerequisite for this is, of course, an exchange between the teams about best practices and about which technologies work and which do not.







## Who defines macro architecture? [#](https://www.educative.io/module/lesson/introduction-to-microservices/gx7ZB3lMqjD#Who-defines-macro-architecture?)

### Committee of representative team members [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gx7ZB3lMqjD#Committee-of-representative-team-members)

Macro architecture restricts the freedom of the teams when it comes to implementing the microservices. This can be counteracted by having a **committee define the macro architecture**, which consists of one member of each team.

However, it is possible that the committee **may become too large to work effectively**. With ten teams, the team would have ten members and effective work is then hardly possible. You can reduce the number of members by **excluding teams** or sending **individual members as representatives for multiple teams**.

Unfortunately, the team members are often too focused on their own microservices to be interested in the overall picture of macro architecture.

### Independent architecture committee [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gx7ZB3lMqjD#Independent-architecture-committee)

The alternative is to have an **independent architecture committee** decide on macro architecture, which is staffed by architects who do not belong to the teams.

In such a scenario, it is important that:

* This **body’s goal is to support the teams** in their development of microservices and to moderate decisions rather than enforce them. The most important work takes place in the teams. Therefore, the macro architecture should support the teams and not hinder them.
  + Collaboration between the architecture committee and the teams can also be improved by the members of the architecture committee working at least partly in teams.
* The members of the committee are **integrated and interested in the developed system**.
  + The specific domain and business requirements should never be forgotten.
  + An important part of the work on the architecture is to understand stakeholders and ensure that their goals are supported by the architecture.

## How to enforce? [#](https://www.educative.io/module/lesson/introduction-to-microservices/gx7ZB3lMqjD#How-to-enforce?)

The need for macro architecture should be understandable because it ensures that the entire system can be developed and operated. **To enforce** the macro architecture, the team should **document reasons for each rule** to avoid unnecessary discussions behind their reasoning.

For example, certain macro architecture rules may be necessary to allow the operations team to bring the software into production, or to ensure that compliance rules are followed.

It’s not so much about enforcing rules as it is about **promoting macro architecture and conveying the ideas/reasons** for macro architecture. If good reasons for changing the macro architecture exist, improving the architecture might be a better option than enforcing an obsolete one.

## Testing conformance [#](https://www.educative.io/module/lesson/introduction-to-microservices/gx7ZB3lMqjD#Testing-conformance)

In some cases, it is possible to test the conformance to the macro architecture by deploying a microservice and checking its log output and metrics. This ensures that deployment, logging, and monitoring all conform to the defined macro architecture.

Such a test is called a **black box test**, which checks the behavior of the microservice from the outside.

The **benefit** of this approach is that it:

* Does not limit the free choice of technology for implementing microservices,
* Does not enforce unnecessary standards for specific frameworks.

Therefore, testing for the conformance on the code level does not make a lot of sense.

**QUIZ:**

Even if an operational technology is not standardized at the macro architecture level, certain technologies are naturally conformed to by teams that communicate well.

###### True

Which of the following is a good way to enforce a macro architecture decision as described above?

###### Convince the teams of the decisions via well documented and convincing rationale.

A black box conformance test \_\_\_\_\_\_.

###### tests the microservice as a whole

In the next lesson, we’ll look at Independent Systems Architecture principles.

# Independent Systems Architecture Principles

In this lesson, we'll discuss Independent Systems Architecture's nine principles.

**We'll cover the following**

* [Conditions](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Conditions)
* [Principle #1: The system must be divided into modules](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#1:-The-system-must-be-divided-into-modules)
  + [Evaluation](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)
* [Principle #2: Two separate levels of architectural decisions](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#2:-Two-separate-levels-of-architectural-decisions)
  + [Evaluation](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)
* [Principle #3: Modules must be separate processes/containers/VMs](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#3:-Modules-must-be-separate-processes/containers/VMs)
  + [Evaluation](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)
* [Principle #4: Standardized integration & communication](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#4:-Standardized-integration-&-communication)
  + [Evaluation](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)
* [Principle #5: Standardized metadata](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#5:-Standardized-metadata)
  + [Evaluation](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)
* [Principle #6: Independent continuous delivery pipelines](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#6:-Independent-continuous-delivery-pipelines)
  + [Evaluation](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)
* [Principle #7: Operations should be standardized](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#7:-Operations-should-be-standardized)
  + [Evaluation](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)
* [Principle #8: Standardized interface](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#8:-Standardized-interface)
  + [Evaluation](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)
* [Principle #9: Modules have to be resilient](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#9:-Modules-have-to-be-resilient)
  + [Evaluation](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)
* [Summary](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Summary)

Micro and macro architecture are fundamental to the idea of microservices. However, it is hard to understand why there should be two levels of architecture.

[**ISA**](http://isa-principles.org/)**(Independent Systems Architecture)** is the term for a collection of fundamental principles for microservices. It is based on experiences with microservices gained from many different projects.

The name already suggests that these principles aim to build software out of **independent systems**. Macro and micro architecture are very important for this goal.

A minimal macro architecture leaves a lot of freedom to the level of the micro architecture, making the systems independent. Technical decisions can be made for each system without influencing the other systems.

ISA defines the term micro and macro architecture. Also, the principles explain what the minimum requirements for macro and micro architecture are.

## Conditions [#](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Conditions)

**Must** is used for principles when they **absolutely have to be adhered to**.

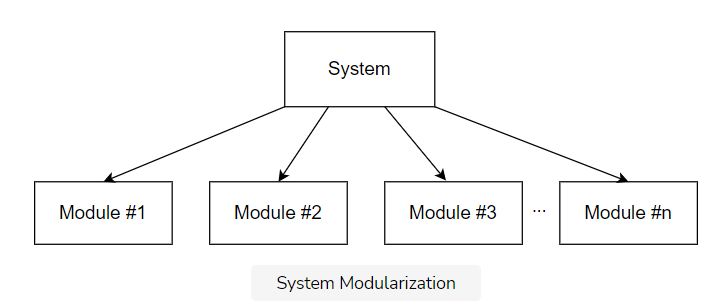
**Should** describes principles which have many advantages but **do not have to be strictly followed**. We’ll now discuss each principle. The **ISA principles** are not only a great guideline for building microservices, but they also explain why macro and micro architecture are so important.

## Principle #1: The system must be divided into modules [#](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#1:-The-system-must-be-divided-into-modules)

The system **must be divided into modules** that offer interfaces. Accessing modules is only possible via these interfaces. Therefore, modules may not depend directly on the implementation details of another module, such as the data model in a database.

### Evaluation [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)

The **first** ISA principle states that a system must be built from modules. This is common knowledge.



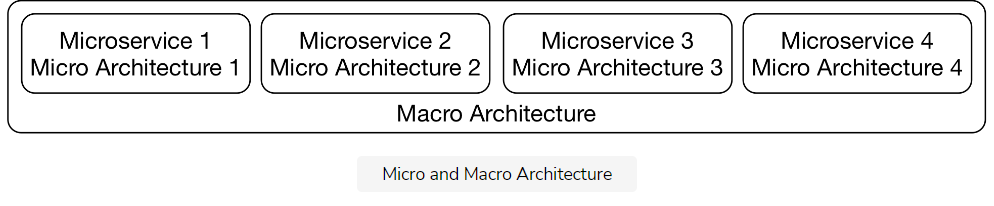
## Principle #2: Two separate levels of architectural decisions [#](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#2:-Two-separate-levels-of-architectural-decisions)

The system **must have two clearly separated levels** of architectural decisions:

* **Macro architecture** comprises decisions which concern all modules. All further principles are part of the micro architecture.
* **Micro architecture** comprises those decisions which can be made differently for each individual module.

### Evaluation [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)

The **second principle** defines two levels of architecture: macro and micro architecture.



## Principle #3: Modules must be separate processes/containers/VMs [#](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#3:-Modules-must-be-separate-processes/containers/VMs)

Modules **must be separate processes, containers, or virtual machines** to maximize independence.

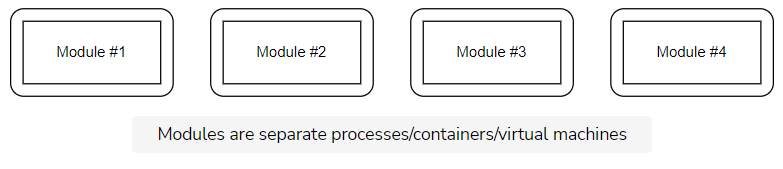
### Evaluation [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)

In a deployment monolith, **most of the decisions will be on the macro architecture level**. For example, a deployment will be written in one programming language, so the programming language has to be a decision on the macro architecture level. The same is true for frameworks and most of the other technologies.

To make more decisions on the micro architecture level, each module must be implemented in a separate container as this principle states. ISA says that **the reason** why microservices run in containers **is the additional technological freedom** that cannot be achieved in a deployment monolith. Therefore, microservices add more independence and decoupling to the architecture.

An approach where each microservice is a [WAR](https://en.wikipedia.org/wiki/WAR_(file_format)) and all run together in one Java application server does not fit this principle. Actually, the compromise concerning the free choice of technology and the robustness is so high that this approach usually does not make a lot of sense.

Because decoupling is so important, ISA and microservices actually provide fundamental improvements to modularization.



## Principle #4: Standardized integration & communication [#](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#4:-Standardized-integration-&-communication)

The **choice of integration and communication options must be limited and standardized** for the system.

* The integration might be done with synchronous or asynchronous communication, and/or on the UI level.
* **Communication must use a limited set of protocols** like RESTful HTTP or messaging. It might make sense to use just one protocol for each integration option.

### Evaluation [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)

Although the goal of ISA is to create a minimal macro architecture, **some decisions still need to be made on the macro level**. This is what the rest of the principles explain. As a start, principle four states that integration and communication must be standardized. The last three chapters of this course discuss a few technology stacks for integration and communication.

The decision to use a specific technology for integration and communication influences all modules and must, therefore, be done on the macro architecture level. It is therefore also a very important decision in microservices systems.

Without a common integration approach and communication technology, it is hard to consider the system as a system and not just a few services that can barely communicate with each other.

## Principle #5: Standardized metadata [#](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#5:-Standardized-metadata)

**Metadata**, for example, for authentication, **must be standardized**. Otherwise, the user would need to log in to each microservice.

* This might be done using a token that is transferred with each call/request.
* Other examples might include a trace ID to track a call and its dependent calls through the microservices.

### Evaluation [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)

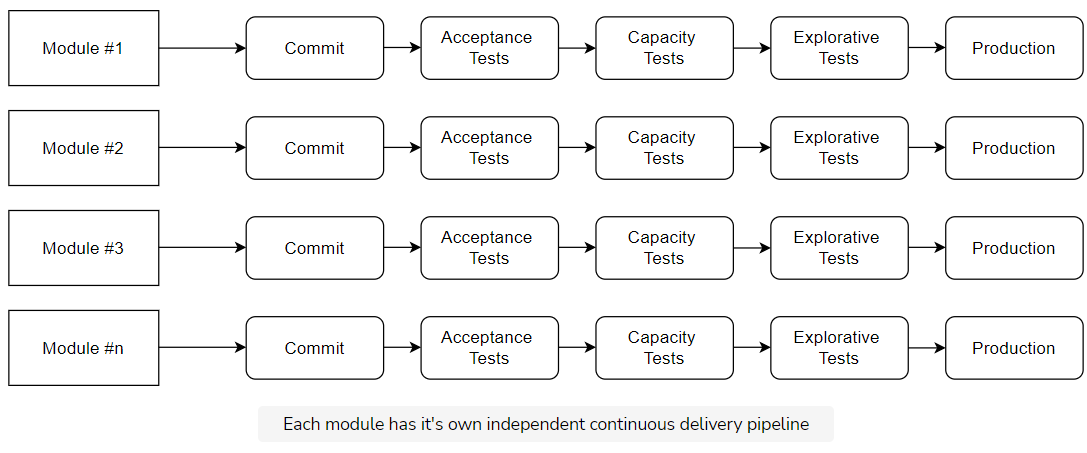
This principle states that metadata for tracing and authentication must be standardized. Such **metadata must be transferred between the microservices** and must, therefore, **also be a part of the macro architecture**. This course does not discuss security aspects of microservices, including metadata for authentication.

## Principle #6: Independent continuous delivery pipelines[#](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#6:-Independent-continuous-delivery-pipelines)

Each module **must have its own independent continuous delivery pipeline**. Tests are part of the continuous delivery pipeline; therefore, the tests of the modules have to be independent, too.

### Evaluation[**#**](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)

This principle extends the idea of independent deployment as the definition of microservices from the [last chapter](https://www.educative.io/collection/page/10370001/6518081205567488/6272204058656768#microservices-definition-section-microservices-definition).



## Principle #7: Operations should be standardized [#](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#7:-Operations-should-be-standardized)

**Operations should be standardized**. There can be exceptions from the standard when a module has very specific requirements. These operations comprise:

* configuration
* deployment
* log analysis
* tracing
* monitoring
* alarms

### Evaluation [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)

This principle says that the operations of microservices should be standardized. It is not in all cases necessary to standardize operations. With a separate operations department, standardization is the only way to handle a large number of microservices.

However, with a “you build it – you run it” organization, standards are not necessary as each team operates their microservices. Actually, a standardized operations approach might not fit all microservices. In that case, the teams need to come up with their own operations technologies. A standard may not be useful then.

## Principle #8: Standardized interface [#](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#8:-Standardized-interface)

**Standards** for operations, integration, or communication **should be enforced on the interface level**.

* For example, the communication protocol and data structures could be standardized to a specific JSON payload format exchanged using HTTP, but every module should be free to use a different REST library/implementation.

### Evaluation [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)

This principle states that standards should only be defined on the interface level. The technologies discussed throughout the last three chapters can be used in this way. They provide interfaces and client libraries for all commonly used programming languages.

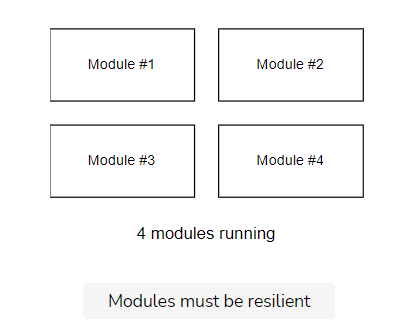
## Principle #9: Modules have to be resilient [#](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Principle-#9:-Modules-have-to-be-resilient)

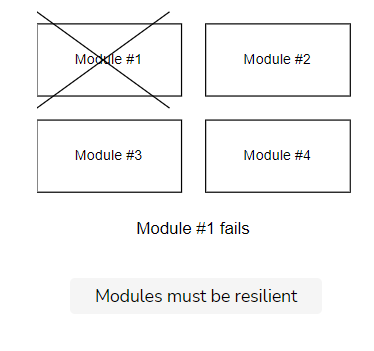
**Modules have to be resilient**. This means that:

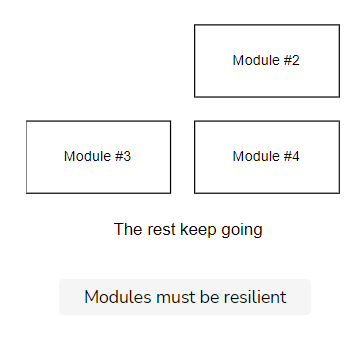
* They may not fail when other modules are unavailable or when communication problems occur.
* They must be able to shut down without losing data or state.
* It must be possible to move them to other environments (server, networks, configurations and so on) without the module failing.

### Evaluation [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Evaluation)

This principle addresses resilience. Asynchronous communication makes resilience easier. If a microservice fails, a message will be transferred later but the failed microservice will not cause another microservice to fail.







## Summary [#](https://www.educative.io/module/lesson/introduction-to-microservices/xVD50PG9q4n#Summary)

The ISA principles represent a good summary of the ideas introduced in this chapter – that is, a division between micro and macro architecture as the main benefit of microservices. ISA also explains why the rest of the course focuses on integration and communication technologies and technologies for operations. These are the fields that a macro architecture has to cover. Therefore, these decisions are very important and also hard to change because they influence all microservices in the system.

QUIZ:

The communication method between microservices **has** to be standardized.

###### True

Does a system where two microservices use different technologies for log analysis **violate** ISA rules?

###### No, because the rules says that operations “should” be standardized. It does not say “must”.

Once a user logs in, they must be able to access all functionalities of all microservices.

###### False (They are not necessarily granted full access to the entire system. That depends on the type of the user.)

In the next lesson, we’ll discuss some variations on what we’ve discussed so far in this chapter.

**Variations**

In this lesson, we'll discuss some variations to the approaches discussed in this chapter.

**We'll cover the following**

* [More complex rules](https://www.educative.io/module/lesson/introduction-to-microservices/gxJjMx83BkY#More-complex-rules)
* [Experiments](https://www.educative.io/module/lesson/introduction-to-microservices/gxJjMx83BkY#Experiments)

In the domain macro architecture, strategic design and domain-driven design are ultimately unrivaled as approaches.

However, the bounded contexts depend on the specific project. **Identifying the right bounded contexts is a central challenge** when designing the architecture of a microservices system.

The **technical micro and macro architecture also have to be devised** for each project. This depends on many **factors**:

* Organizational aspects such as *DevOps organization* or having a separate operations team has an influence.
* In addition, *strategic technology decisions* can play a role.
* Even the *hiring policy* can be a factor. Eventually, there have to be experts available who can work in the teams to manage the technologies.

**More complex rules**[#](https://www.educative.io/module/lesson/introduction-to-microservices/gxJjMx83BkY#More-complex-rules)

In reality, the rules of micro and macro architecture are **often more complex**.

For example, **a**[**whitelist**](https://en.wikipedia.org/wiki/Whitelisting)**can exist for the programming language**. In addition, there can be a procedure for adding more programming languages to the whitelist, for example, via a committee. And finally, there can be a general limitation to programming languages that run on the JVM (Java Virtual Machine).

Such a rule **has elements of a macro architecture decision**. There is a whitelist and a restriction to JVM languages. **At the same time, it also has micro architecture elements**. After all, a team can select one of the programming languages from the whitelist and even extend the whitelist.

Therefore, rules are often in place for every point in practice that allow the teams and microservices a certain amount of leeway. These rules are **not purely micro or macro architecture rules** but lie somewhere in between.

**Experiments**[#](https://www.educative.io/module/lesson/introduction-to-microservices/gxJjMx83BkY#Experiments)

The approach for defining micro and macro architecture can look like this:

* Consider a project you are familiar with. Look at its domain model and **consider the following questions**:
  + Would a division into multiple domain models and bounded contexts make the system easier?
  + In how many bounded contexts would you split the system? Typical projects consist of about ten bounded contexts. However, the exact number will vary for each individual project.
  + Determine the use cases which the system implements. Group use cases and analyze whether these use cases can be addressed by a domain model. By doing so, these use cases form a bounded context in which the domain model is valid.
  + Is a further division for technical reasons sensible? The technical reasons can comprise independent scalability or security (see also “Two Levels of Microservices” in [this lesson](https://www.educative.io/collection/page/10370001/6518081205567488/4532272759832576#two-levels-of-microservices-domain-and-technical)).
* **Define for your project whether the individual decision should be part of micro or macro architecture**.
* **Work out at least one of the decisions in more detail**. For example, there could be a whitelist of programming languages, or only one programming language might be allowed that can be used by all microservices, or even a procedure for extending the whitelist.

QUIZ:

Suppose a whitelist for the databases is defined at the macro level. Which of the following is true concerning the decision to use a specific database?

###### This is a complex decision that is neither purely macro nor purely micro (While a whitelist of acceptable databases exists, each team can select one out of them. So it is not a purely micro or macro decison.)

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