Coupling:

Coupling is the degree of interdependence between software [modules](https://en.wikipedia.org/wiki/Modular_programming); a measure of how closely connected two routines or modules are the strength of the relationships between modules.

Tight Coupling:

if changes to the design, implementation, or behavior in one system cause changes in another system, then the systems are Tightly coupled

When it comes to microservices, coupling can happen if a change to one microservice enforces an almost immediate change to all other microservices that collaborate with it directly or indirectly.

Disadvantages of Tight Coupling:

1. A change in one module usually forces a [ripple effect](https://en.wikipedia.org/wiki/Ripple_effect) of changes in other modules.
2. Assembly of modules might require more effort and/or time due to the increased inter-module dependency.
3. A particular module might be harder to [reuse](https://en.wikipedia.org/wiki/Code_reuse) and/or test because dependent modules must be included.

Loosely Coupling:

if changes to the design, implementation, or behavior in one system won’t cause changes in another system, then the systems are loosely coupled.

In short, loose coupling in microservice architecture means microservices should know little about each other, and any change to one service should not affect the others.

Chart

Description automatically generated

There are two main types of coupling.

1. Temporal or Runtime Coupling:

Runtime coupling is the degree to which the availability of one service is impacted by the availability of another service.

Example: The order service handles a create order request by making a PUT request to the customer service to reserve credit. While this seems simple, it's actually an example of tight runtime coupling. The order service cannot respond to the POST request until it receives a response from the customer service.

The availability of the create order endpoint is the product of the availability of both order and customer services, which is less than the availability of a single service.

Diagram

Description automatically generated

1. Design Time Coupling:

Design-time coupling is the degree to which one service is forced to change because of a change to another service. Coupling occurs because one service directly or indirectly depends upon concepts that are owned by another service.

Example: The order service consumes the API of the customer service. It either invokes the services operations, or it subscribes to its events. The reason design-time coupling is a potential problem is because concepts can change, that a change to the customer service will force this API to change in a way that requires the order service to also change.

The degree of coupling is a function of the stability of the customer domain, the design of the customer service API, and how much of that API is consumed by the order service.

Consequently, loose coupling is essential. It's important to remember that loose coupling is not guaranteed. You must carefully design your services to be loosely coupled. Ideally, we should design services in a way that avoids any design-time coupling.

Microservices Design:

The Idea behind microservices architecture is simple, to develop and build a large system, you must decompose its function into relatively small, single-purpose and loosely coupled services. An essential principle of the microservice architecture is loose coupling.

If you ignore this principle and develop tightly coupled services, the result will mostly likely be yet another “microservices failure story”. Your application will be brittle and have all of disadvantages of both the monolithic and complexities of microservice architectures.

## Why is loose coupling so important in microservice architecture?

An architecture that accomplishes loose coupling has several advantages, for instance:

* Loosely coupled services increase the evolvability, encourage multiple changes, and new solutions, especially in situations in which the system should be able to adjust to environmental changes. As we all know; in software development, everything changes all the time!
* Loosely coupled services increase the optimum efficiency of the architecture. It enables us to break or reconfigure the link between services. Therefore it also reduces the coordination cost.
* Having loose coupling in services increases the agility, which allows individual teams work on a small, focused piece of functionality quickly, yielding equally quick results.
* Loose coupling allows changes to be deployed independently, which makes quick releases possible.
* The service independence removes impediments when waiting for the other service implementation(s). This way, we will have the frequency and stability of deployments, increasing our productivity.

How to minimize Run-Time and Design-Time coupling?

1. Avoid Synchronous Communication wherever possible:

Run-Time coupling happens when a service - the caller - expects an instantaneous response from another - the callee - before it can resume processing. This situation usually happens when services use synchronous communication.

If you have no choice but to keep synchronous calls in place, you could decrease your dependency by caching the responses to your requests or using the [circuit breaker pattern](https://martinfowler.com/bliki/CircuitBreaker.html) as a mechanism to control cascading failures.

Diagram

Description automatically generated

A better alternative is to switch to asynchronous communication through polling or by relying on a message broker like Kafka to transfer messages to their destinations. When adopting asynchronous communication, services should consider how a lag in reaching a state of eventual consistency with their downstream services might affect their response time and make necessary adjustments to avoid breaking their contracts. Service-level agreements are important parts of a contract.

1. Avoid database sharing:

Another key principle that promotes loose coupling is a database per service.

Diagram

Description automatically generated

Example: let's imagine that you refracted your monolith to services but left the database unchanged. In this partially refracted architecture, the order service reserves credit by directly accessing the customer table. It seems simple, but this results in tight design-time coupling. If the team that owns the customer service changes the customer table, the order service would need to be changed. In order to ensure loose design-time coupling, services must not share tables. Instead, they must only communicate via APIs.

1. Avoid Code Sharing:

Despite having independent codebases, microservices can still fall into the trap of implementation coupling by sharing dependency libraries. Apart from the danger of creating coupling, bloated shared libraries with so many dependencies might end up needing frequent updates to fulfill the changing needs of their clients. As a result, shared code must be as lightweight as possible with limited dependencies and should exclude domain specific logic.

Example: Customers defines the customer object in a library which is shared by Orders. Customers uses this object to model its response to requests for customer data. Orders uses the same object for reading the response body of those requests. If Customers decides to make a change to the internal structure of the customer object, such as breaking the address field into multiple address lines, Orders will break.

Diagram

Description automatically generated

Note that this anti-pattern could also affect Customers’ choice of programming language. If Customers decides to switch to a different programming language, it must be considerate of all other services which are using its object model implementation.

Customers and Orders should each have their own copy of the customers object in separate dependency libraries. If Customers stays committed to its contract everyone will be happy

1. Avoid Sharing of the Domain Data:

Domain driven design is a recommended technique for breaking a monolith into microservices. The general rule is to start with one microservice per business subdomain. Each microservice would operate within the boundaries of its subdomain without having to deal with anything outside of it.

If microservices start to share domain specific data they will create a distributed monolith through domain coupling, which defeats the initial purpose of separating their boundaries. A service will not have control over what its clients do with that shared data. A client can inadvertently turn into the source of truth for data that it doesn’t own, or might misuse it due to its lack of domain specific knowledge.

Also, if a service shares too widely it might introduce a security threat by sharing sensitive data. You might have perfect protection around what you consider as sensitive, but there is no guarantee that your clients would do the same because that responsibility or knowledge falls outside of their domain anyways.

Example: Orders asks for customer data from Customers and receives the customer's credit card number along with their address. It then makes a call to Billing, passing all that data in addition to fee and item id. After Billing charges, the customer successfully, Orders sends the same exact set of data fields to Delivery. The right hand of this figure, however, shows the desirable amount of data exchange between these microservices.

Diagram

Description automatically generated

If designed correctly, Billing should be the only microservice which owns and stores the billing information. It won’t need to receive it from other services.

Only share the data that your clients absolutely need. If they need something that is outside of their domain boundary, it’s time to rethink your service boundaries.

Conclusion:

When migrating your monolith to a microservices architecture, there are many ways that your design could go wrong, and lack of loose coupling is an important one to be mindful of. Coupling can appear in different forms; *implementation, temporal, deployment,*and*domain coupling*.