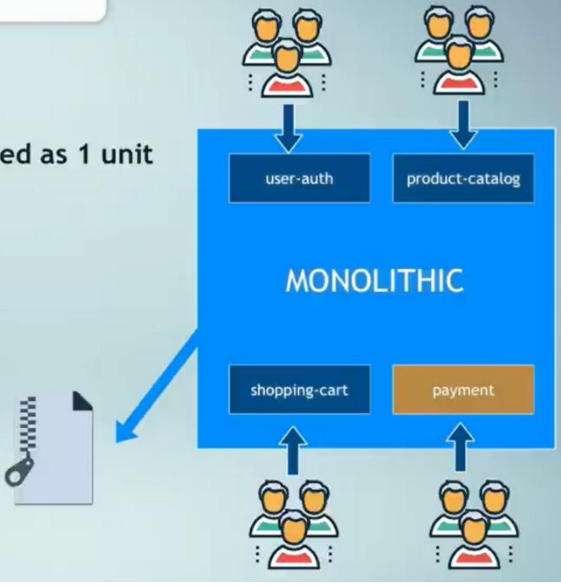
Before microservices, a **monolithic architecture was the standard**

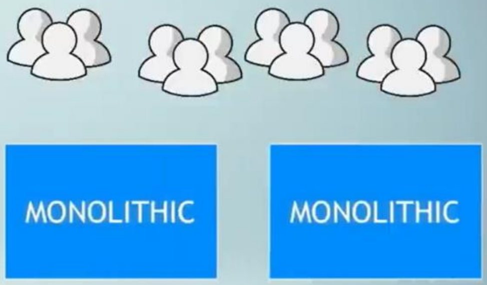
Monolith

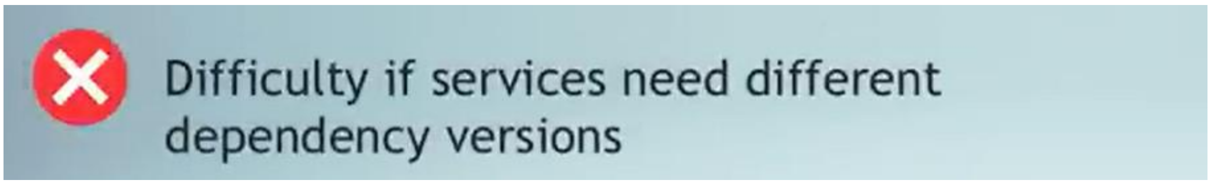
- All components are part of a **single unit**
- Everything is **developed, deployed and scaled as 1 unit**
- App must be written with 1 tech stack
- Teams need to be careful to not affect each other's work
- 1 single artifact, so you must redeploy the entire application on each update



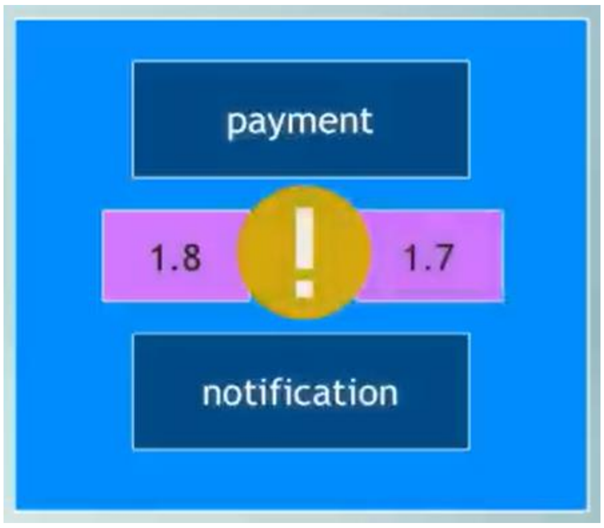
Challenges of monolithic architecture

- ❌ Application is too large and complex
- ❌ Parts are more tangled into each other
- ❌ You can only scale the entire app, instead of a specific service
  - ❌ Higher infrastructure costs

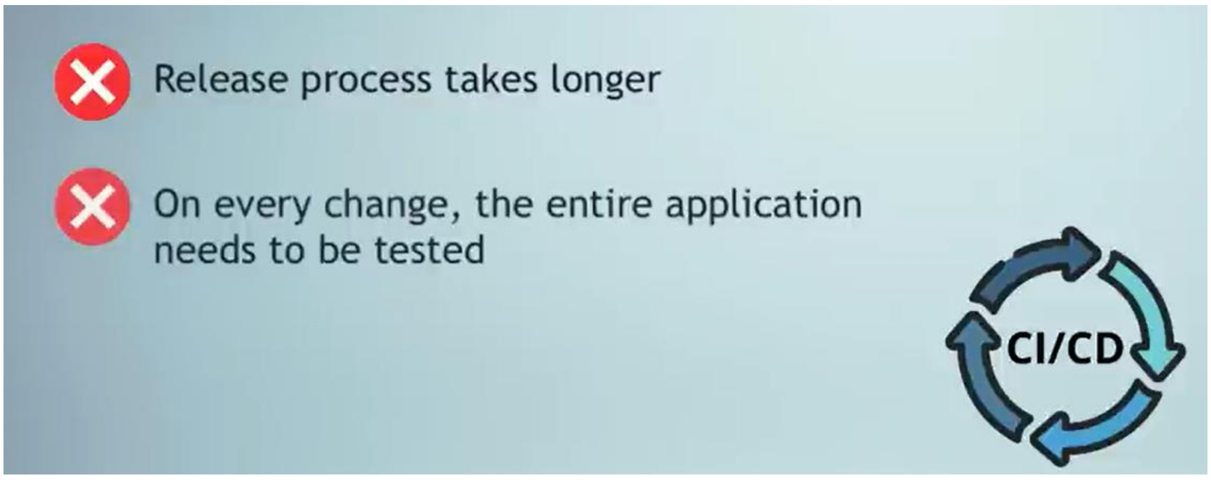




❌ Difficulty if services need different dependency versions

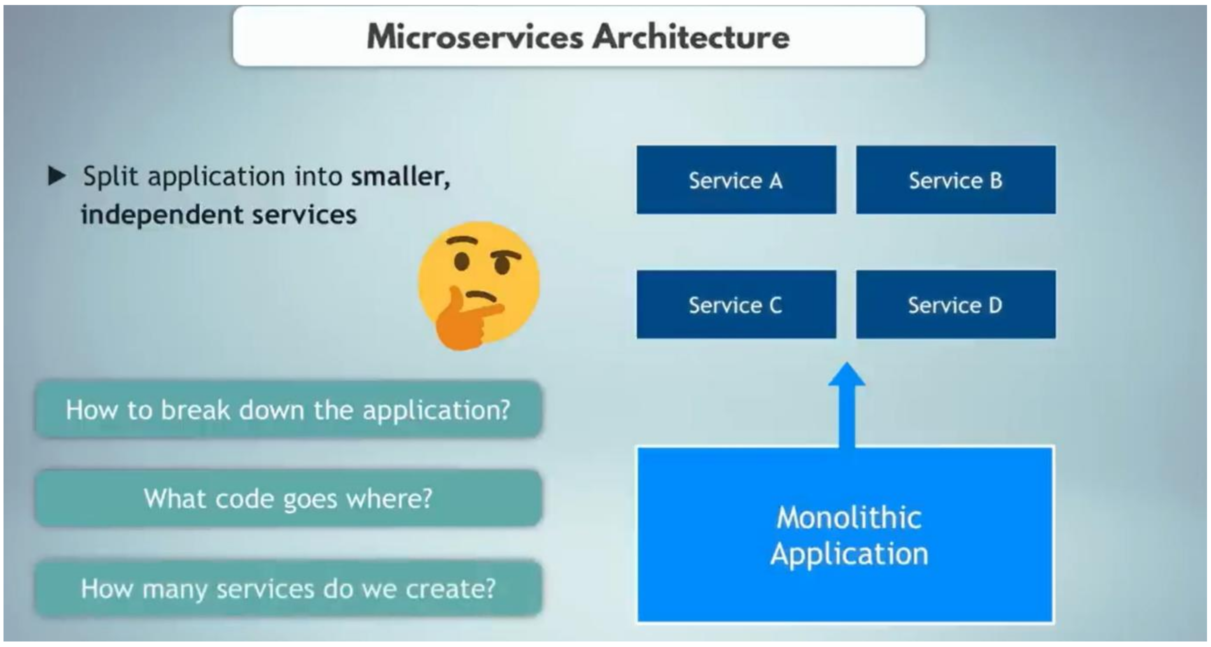


Example: Jenkins and SPC running on same system.



❌ Release process takes longer

❌ On every change, the entire application needs to be tested



Microservices Architecture

- Split application into **smaller, independent services**

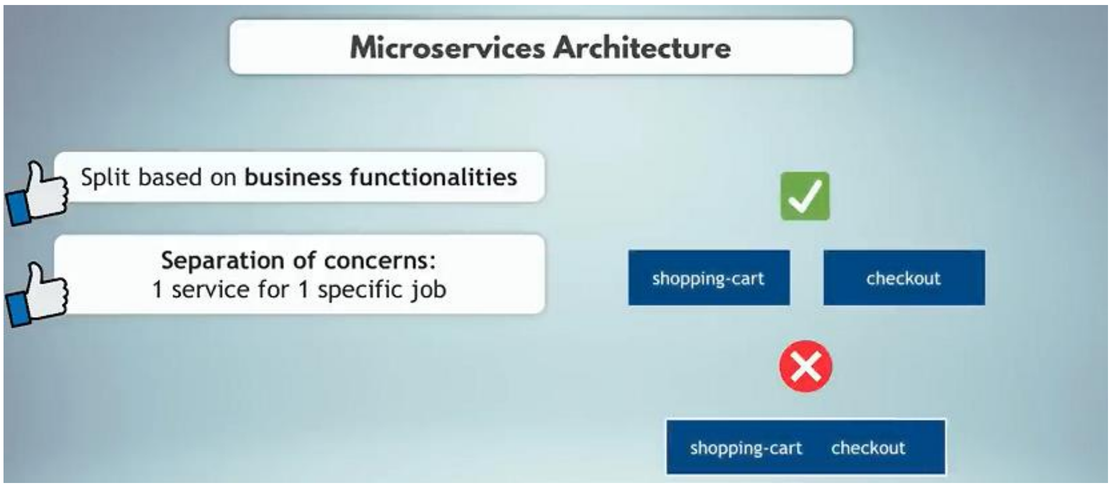
How to break down the application?

What code goes where?

How many services do we create?

1) Based on business functionality break down the application.

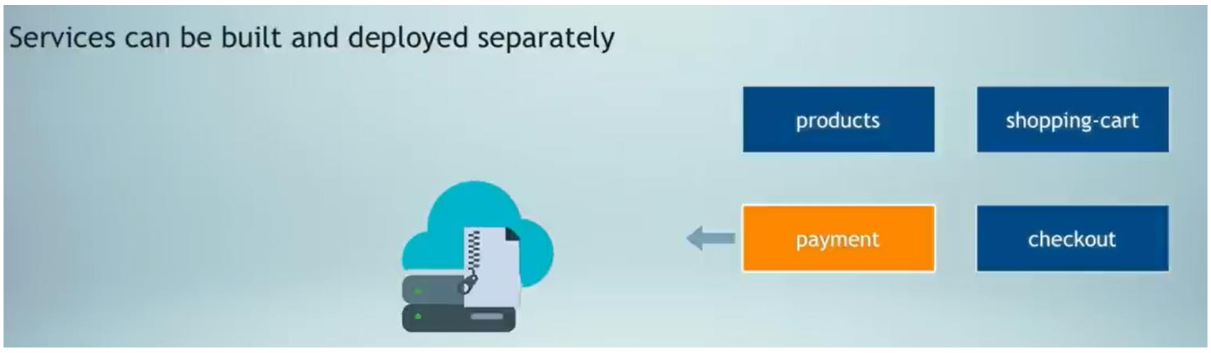
2) Please follow strangler patter while breaking the application.



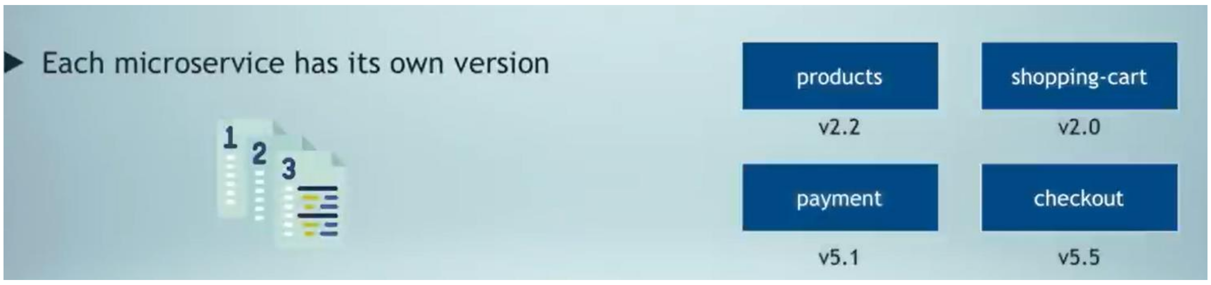
Microservices Architecture

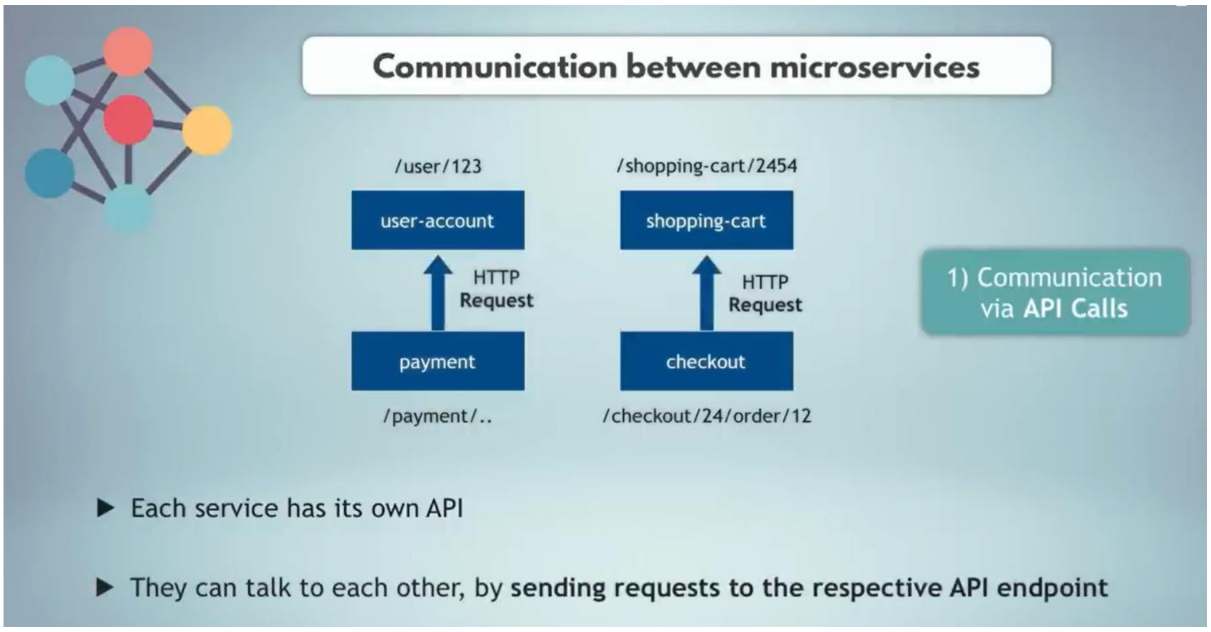
Split based on **business functionalities**

Separation of concerns:
1 service for 1 specific job

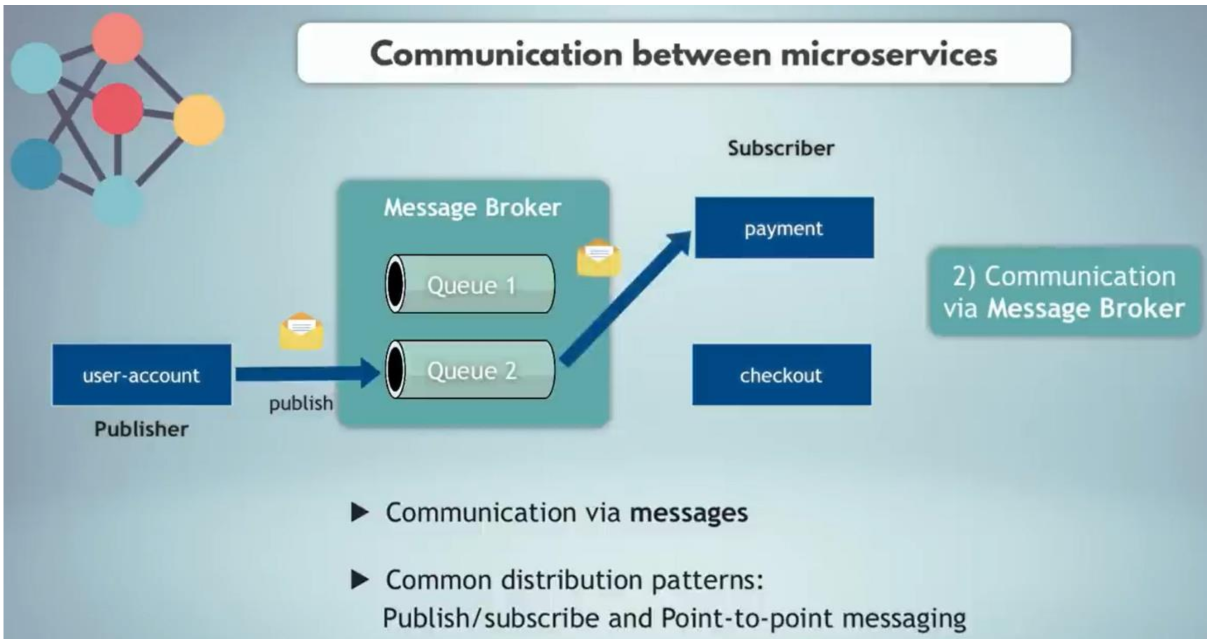


Services can be built and deployed separately

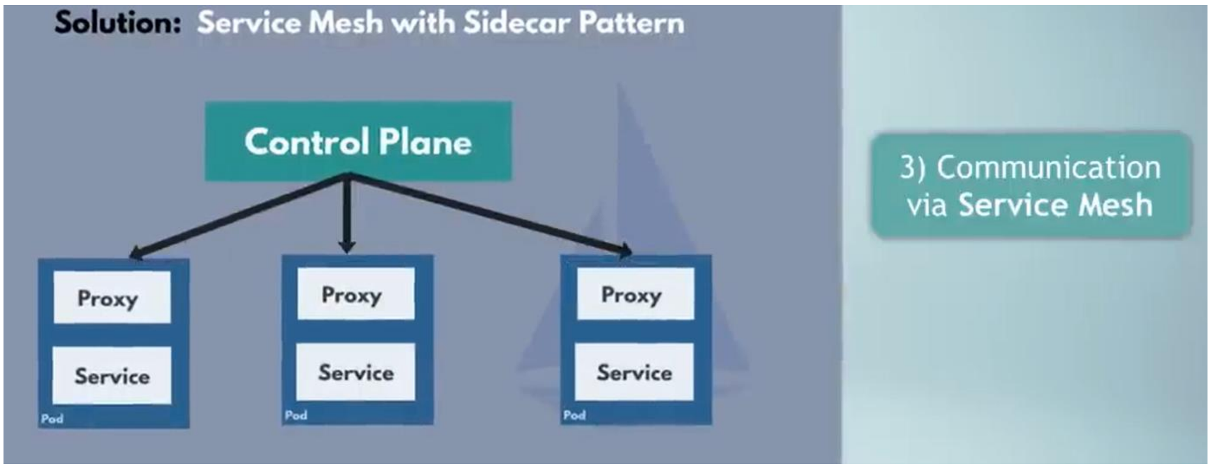

- Each microservice has its own version

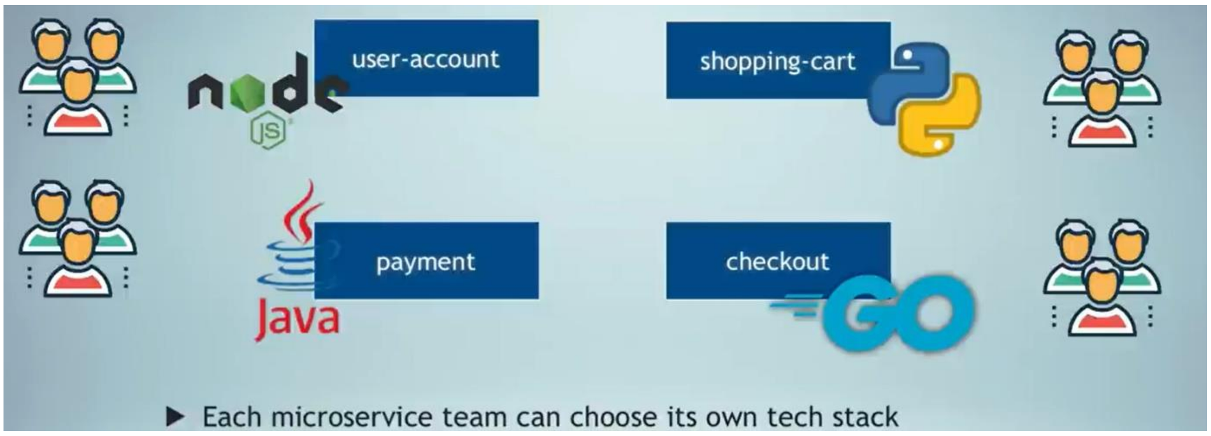

Communication between microservices

- Each service has its own API
- They can talk to each other, by **sending requests to the respective API endpoint**

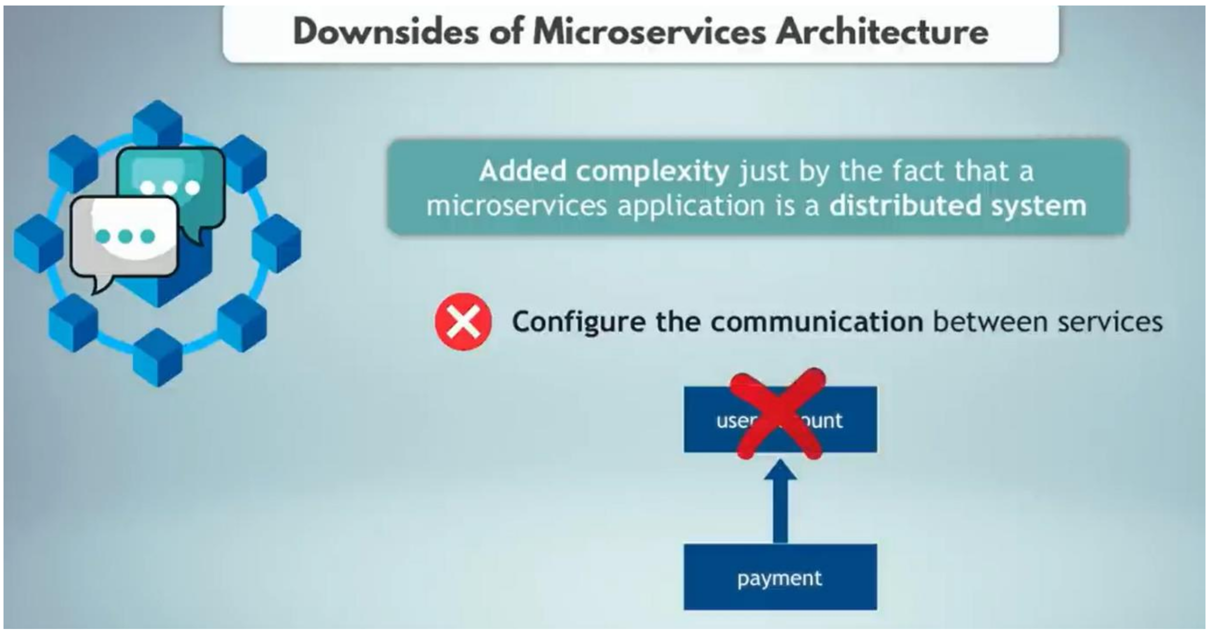

Communication between microservices

- Communication via **messages**
- Common distribution patterns:
  Publish/subscribe and Point-to-point messaging


Solution: Service Mesh with Sidecar Pattern
Control Plane
Proxy
Service
Pod
Proxy
Service
Pod
Proxy
Service
Pod
3) Communication via Service Mesh



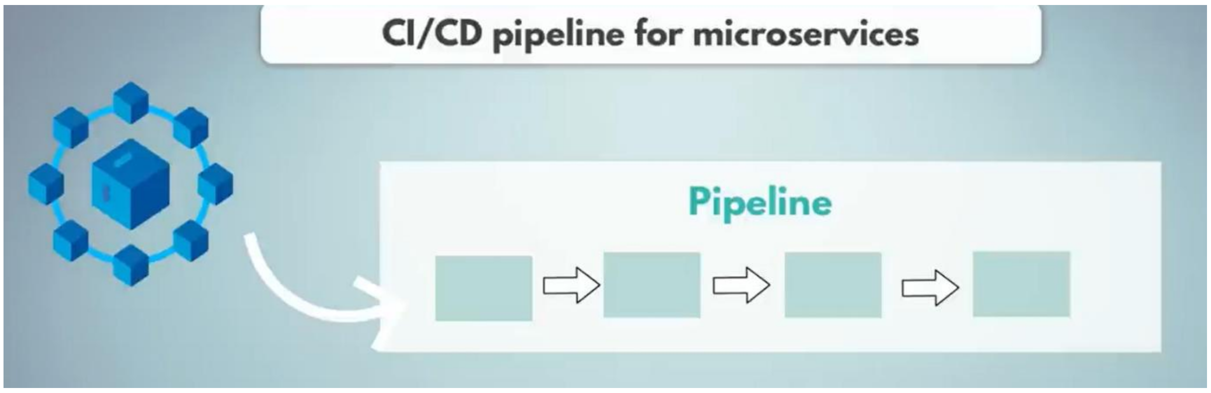
▶ Each microservice team can choose its own tech stack



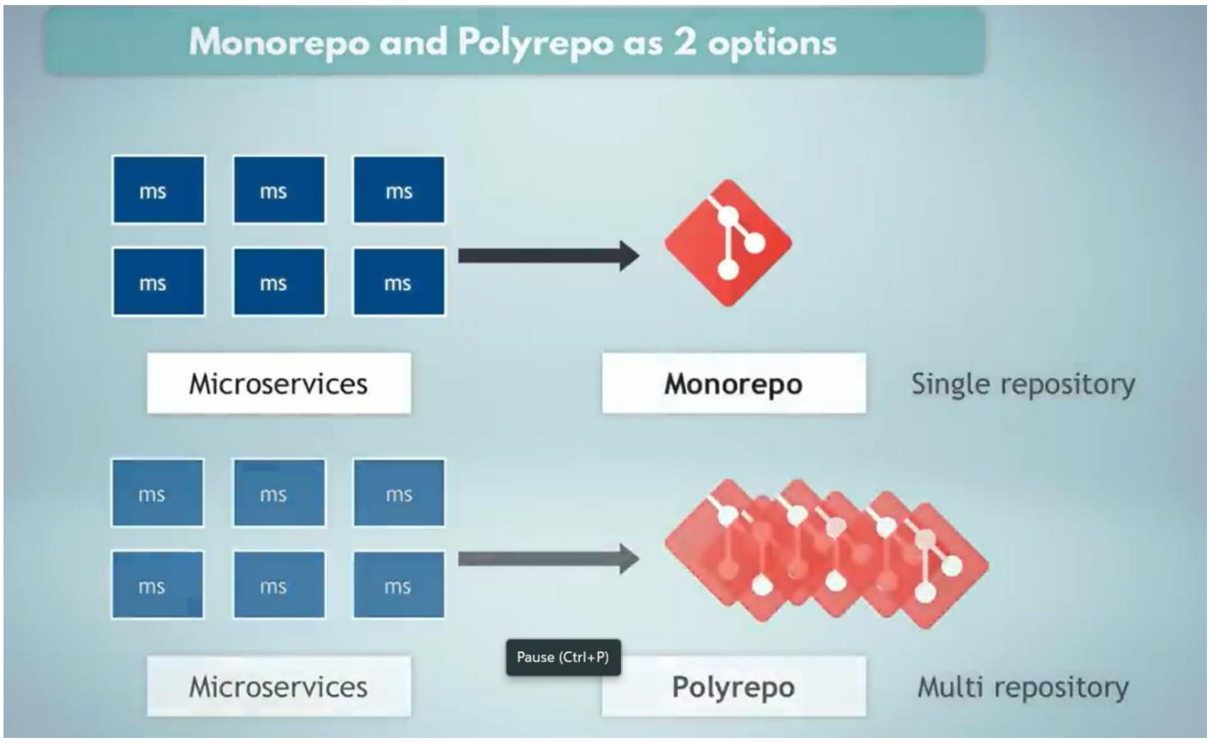
Downsides of Microservices Architecture

Added complexity just by the fact that a microservices application is a **distributed system**

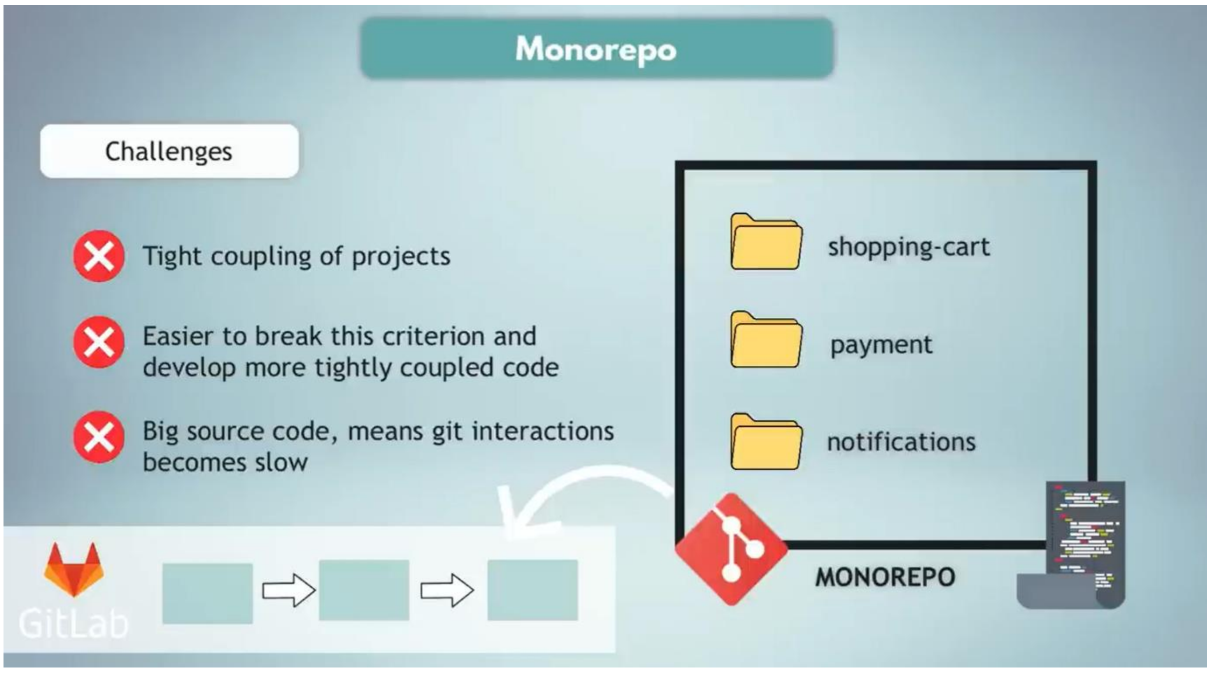
❌ **Configure the communication** between services



CI/CD pipeline for microservices



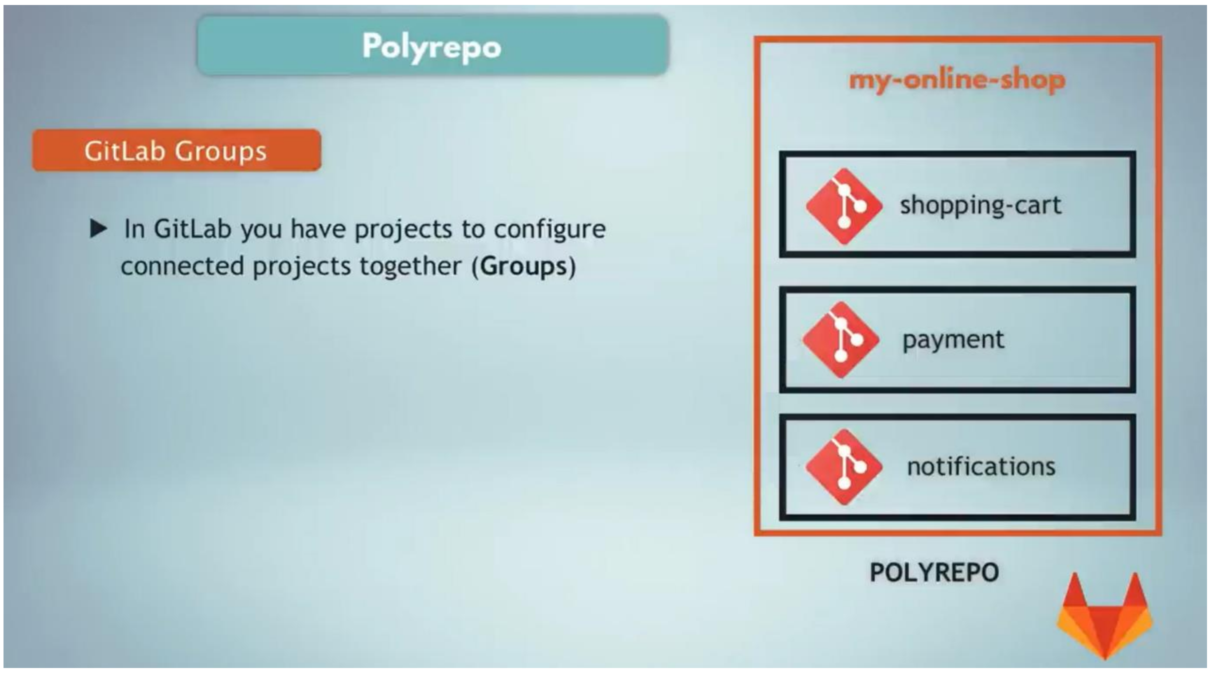
Monorepo and Polyrepo as 2 options



Monorepo

Challenges

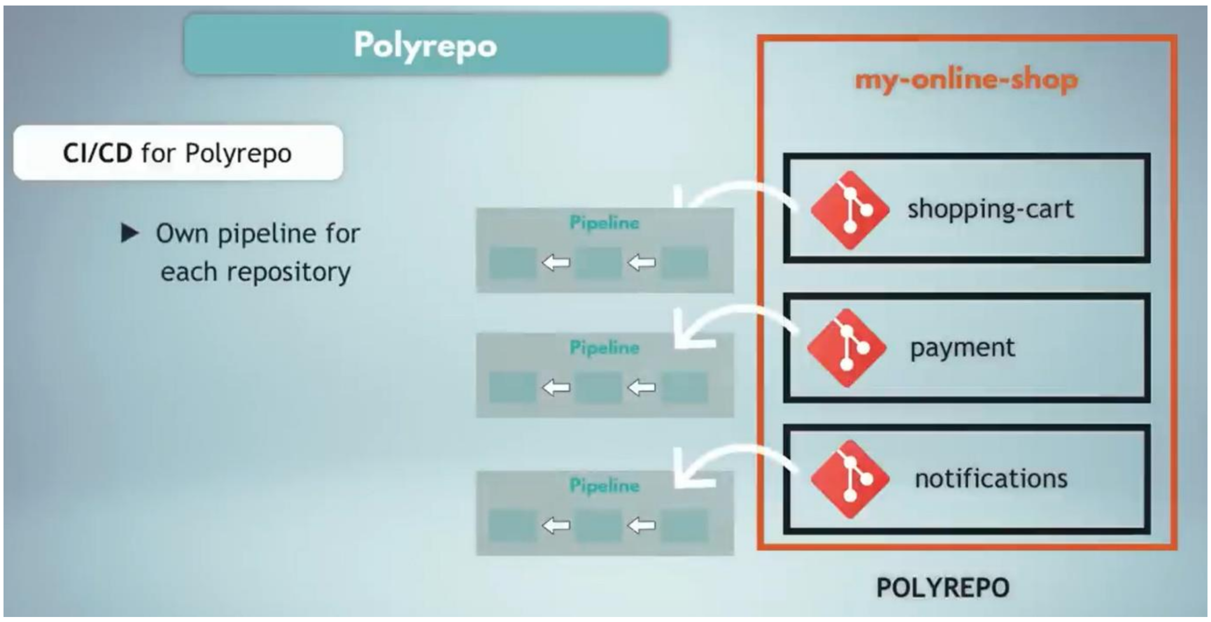
- Tight coupling of projects
- Easier to break this criterion and develop more tightly coupled code
- Big source code, means git interactions becomes slow



Polyrepo

GitLab Groups

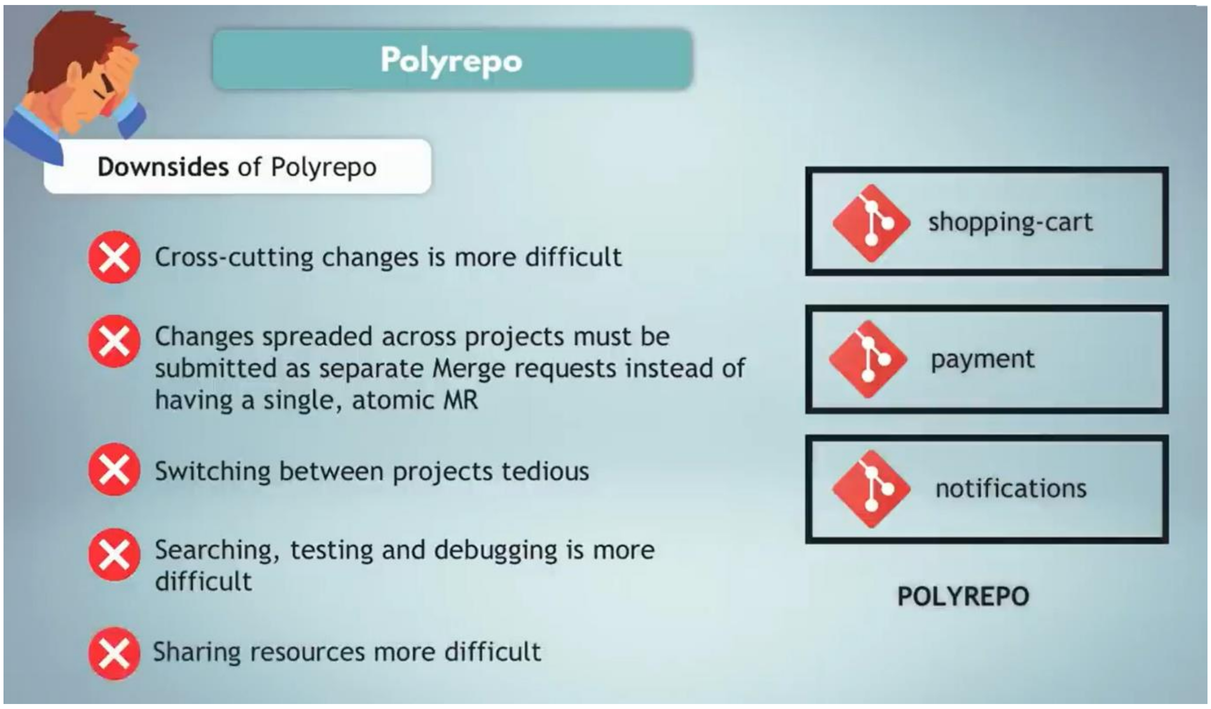
- In GitLab you have projects to configure connected projects together (**Groups**)



Polyrepo

CI/CD for Polyrepo

- Own pipeline for each repository



Polyrepo

Downsides of Polyrepo

- ❌ Cross-cutting changes is more difficult
- ❌ Changes spreaded across projects must be submitted as separate Merge requests instead of having a single, atomic MR
- ❌ Switching between projects tedious
- ❌ Searching, testing and debugging is more difficult
- ❌ Sharing resources more difficult

shopping-cart
payment
notifications
POLYREPO