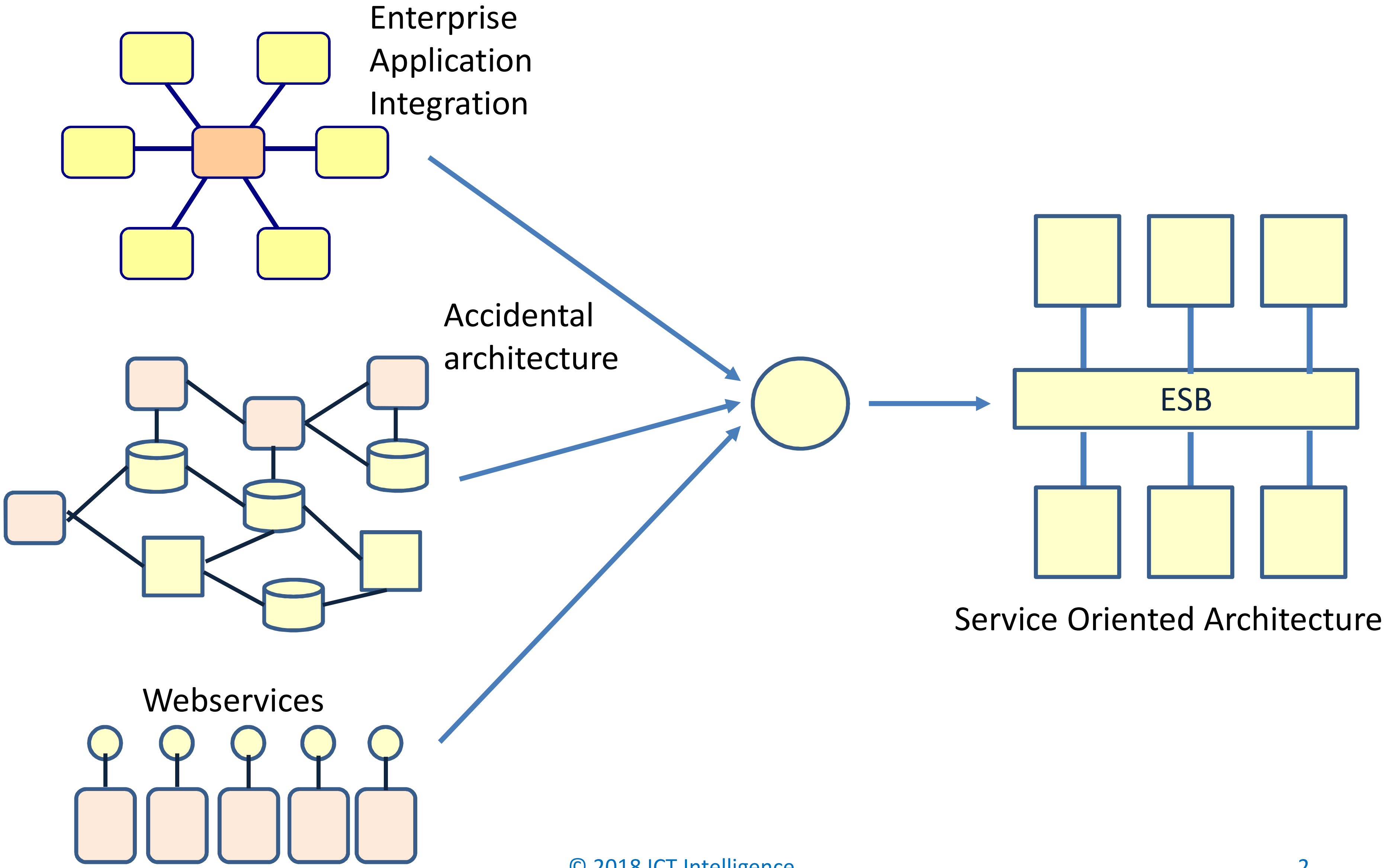
MICROSERVICES

How did we get to SOA?



Enterprise Application Integration

Accidental architecture

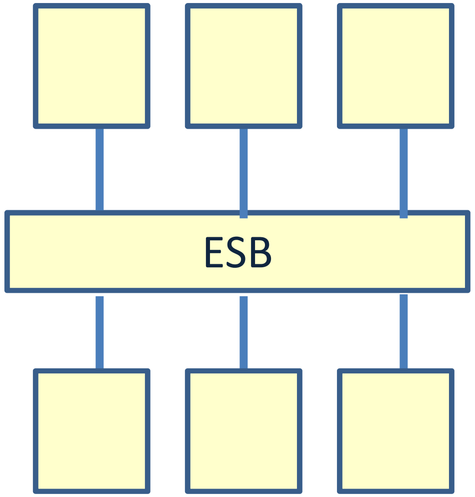
Webservices

ESB

Service Oriented Architecture

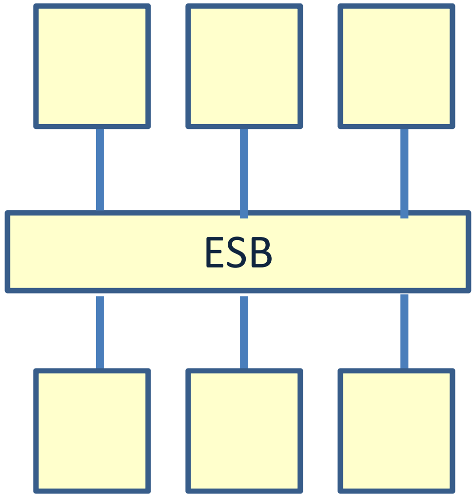
Characteristics of a SOA

- Business processes run on the ESB
- Course grained services
  - To manage performance
  - To manage transactions



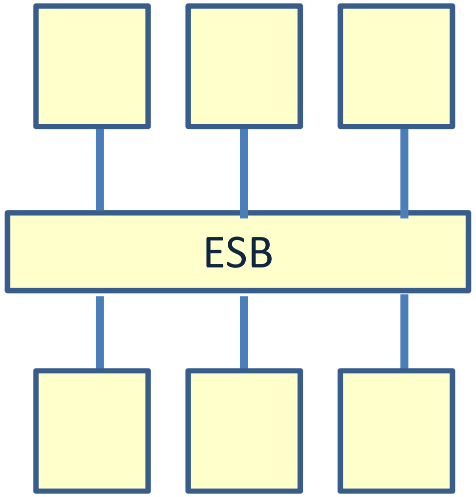
Service Oriented Architecture

- Advantages
  - Independent services
  - Separation of business processes and service logic
  - Architecture is optimized for the business
  - Reuse of services
  - Architecture flexibility



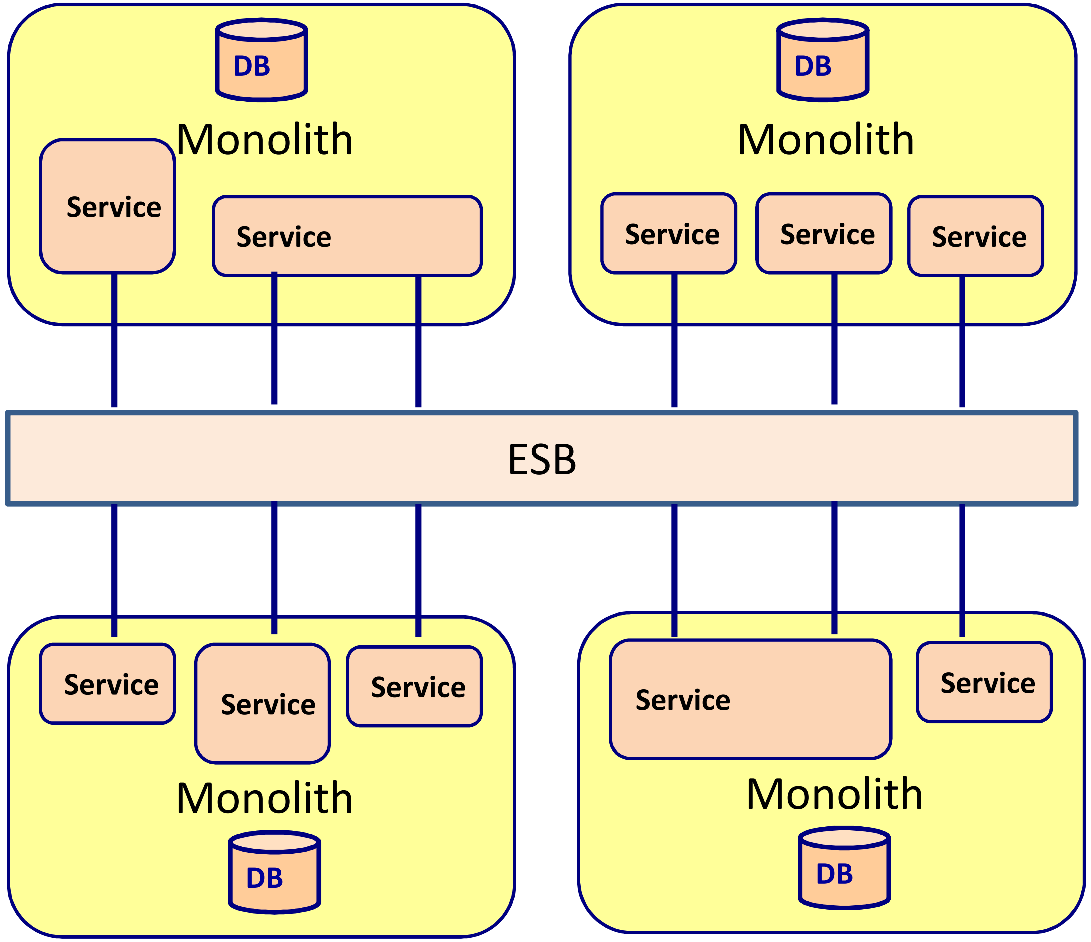
Service Oriented Architecture

- Disadvantages
  - Complex ESB
  - Changing the business process while still business processes are running is very difficult
  - Most SOA's are build on top of monoliths



Problem with SOA

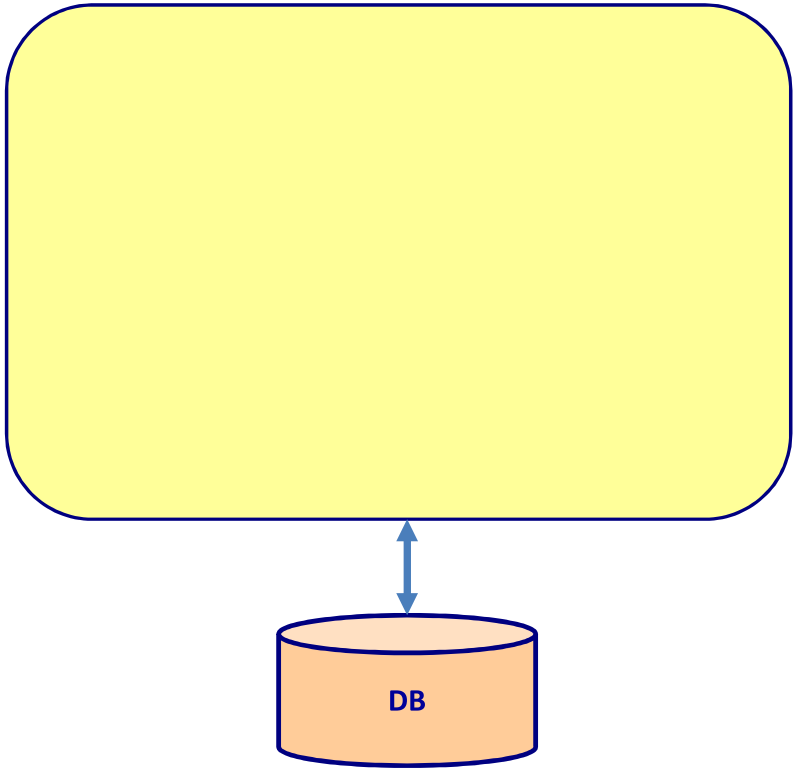
- Most SOA's are build on top of monoliths



MONOLITH ARCHITECTURE

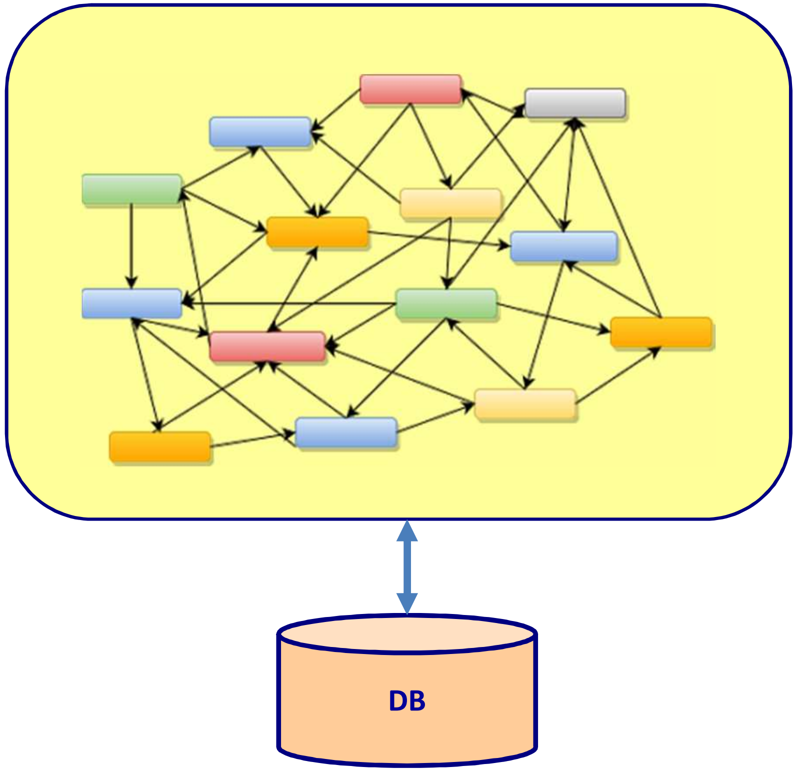
Monolith architecture

- Everything is implemented in one large system



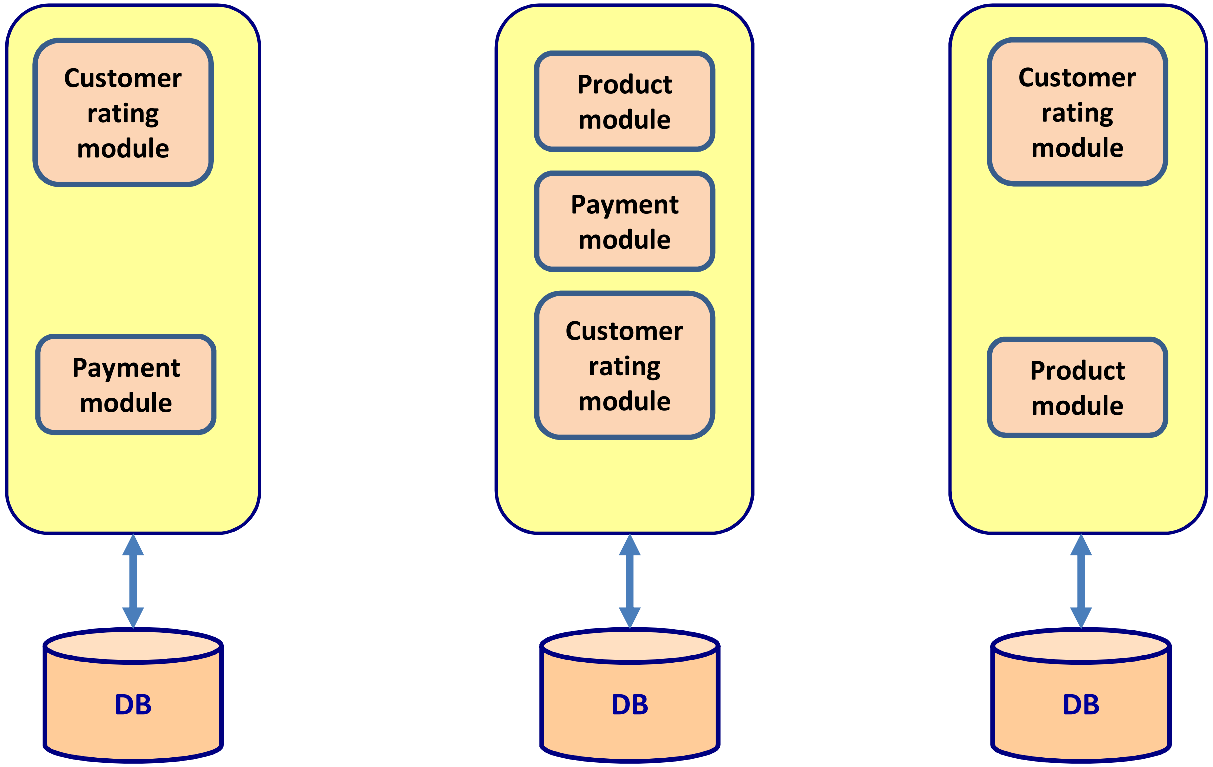
Monolith architecture

- Can evolve in a big ball of mud
  - Large complex system
    - Hard to understand
    - Hard to change



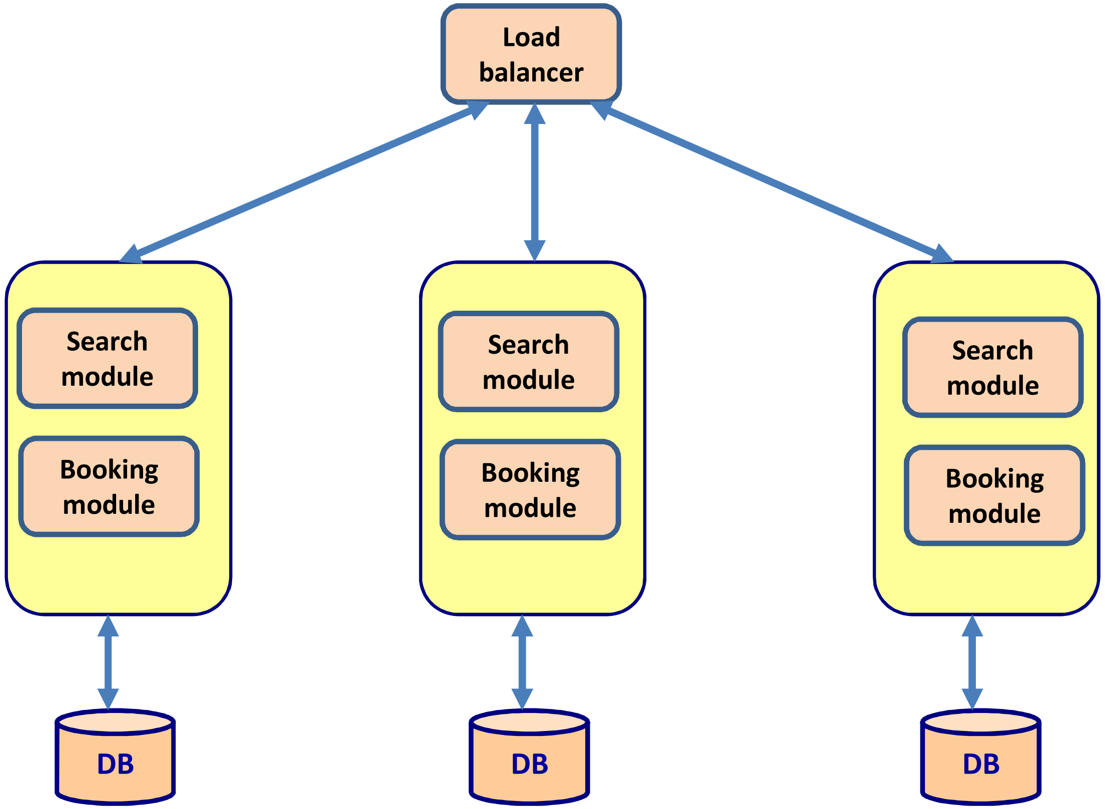
Monolith architecture

- Limited re-use is realized across monolithic applications



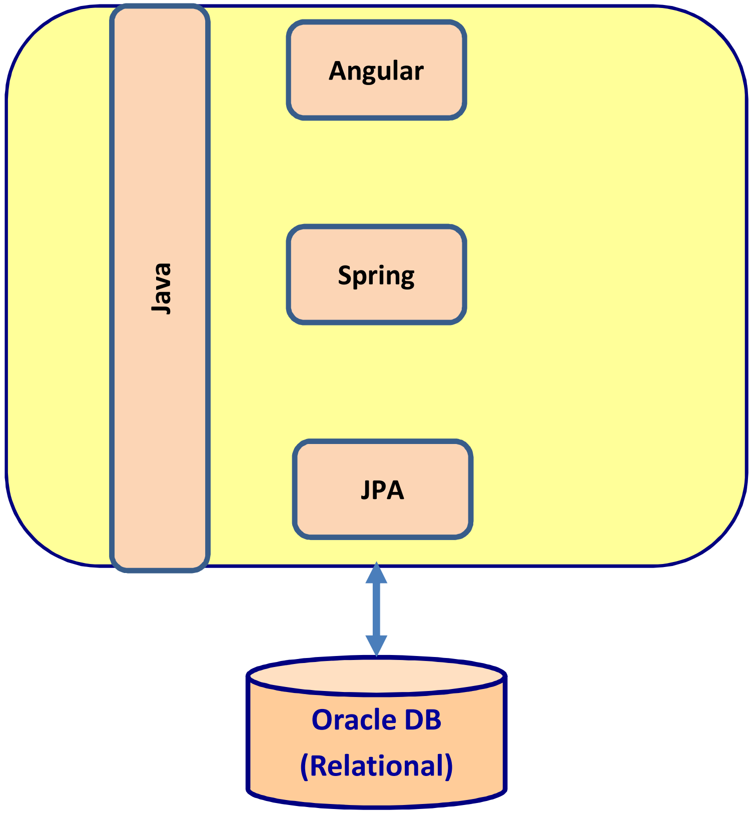
Monolith architecture

- All or nothing scaling
  - Difficult to scale separate parts



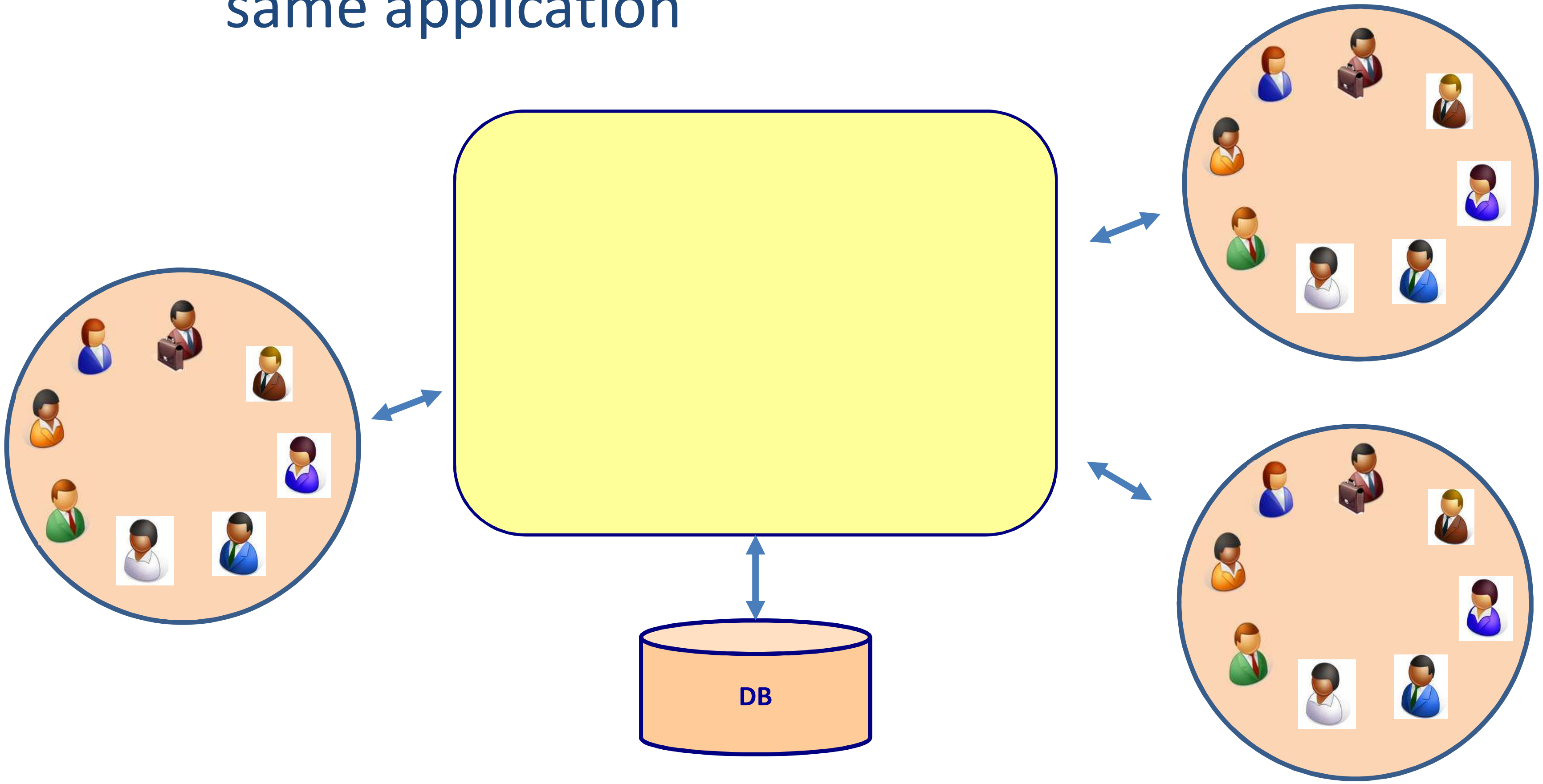
Monolith architecture

- Single development stack
  - Hard to use “the right tool for the job.”



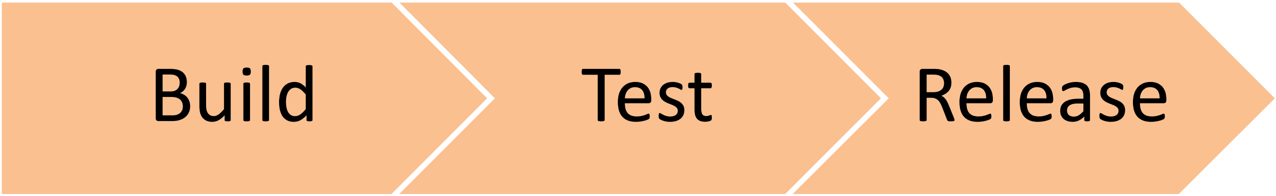
Monolith architecture

- Does no support small agile scrum teams
  - Hard to have different agile teams work on the same application



Monolith architecture

- Deploying a monolith takes a lot of ceremony
  - Every deployment is of high risk
  - I cannot deploy very frequently
  - Long build-test-release cycles

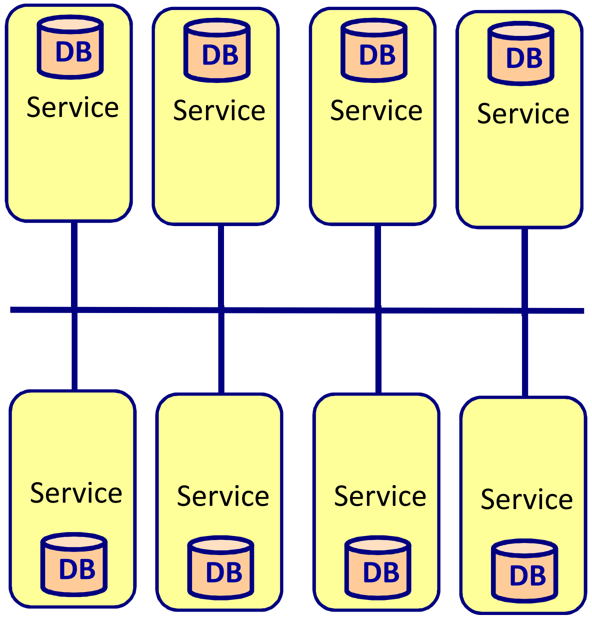


Problems with a monolith architecture

- Can evolve in a big ball of mud
- Limited re-use is realized across monolithic applications
- All or nothing scaling
- Single development stack
- Does no support small agile scrum teams
- Deploying a monolith takes a lot of ceremony

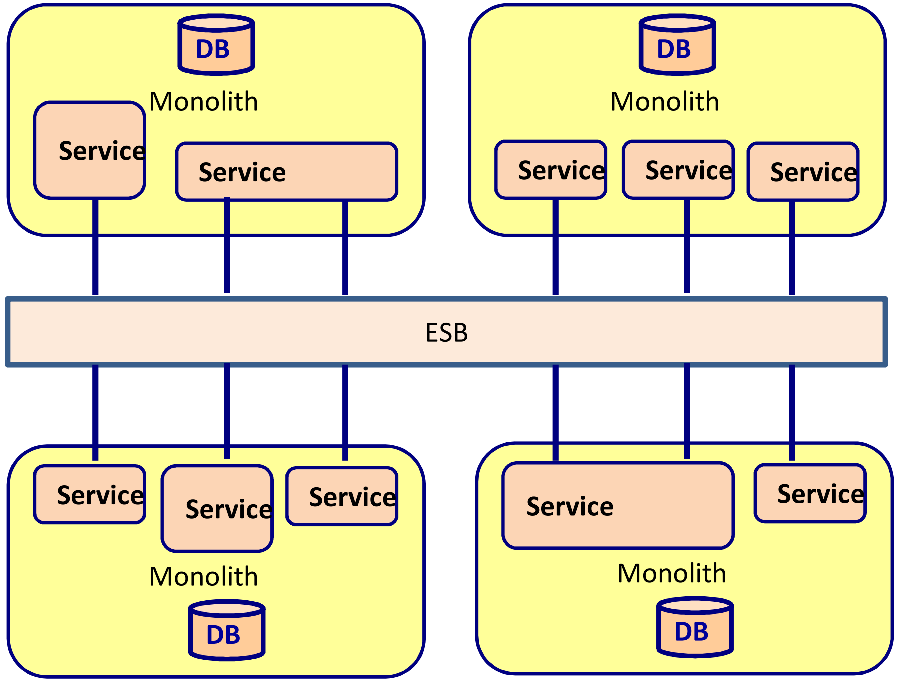
Microservices

- Small independent services
  - Simple and lightweight
  - Runs in an independent process
  - Language agnostic
  - Decoupled

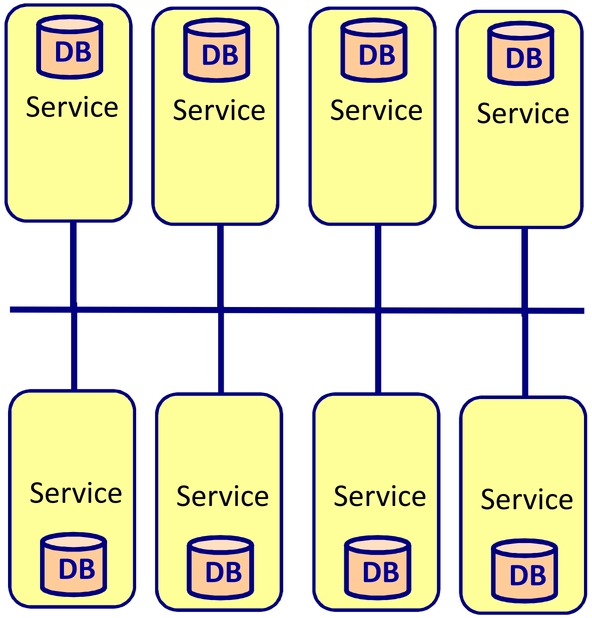


SOA vs Microservice

SOA



Microservice



Microservice early adopters

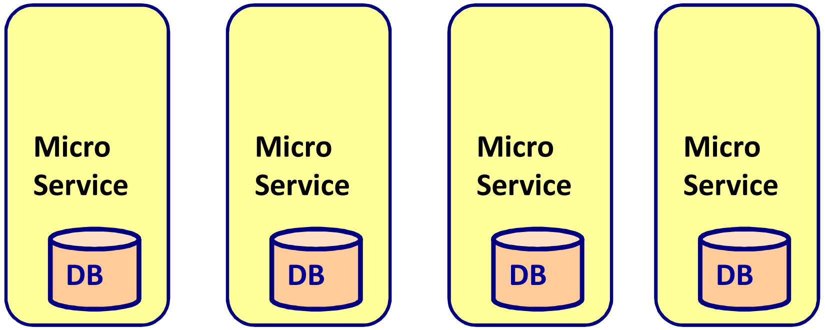
- Netflix
- Uber
- Airbnb
- Orbiz
- eBay
- Amazon
- Twitter
- Nike

Common problem:
How to migrate from a monolith to more scalability, process automation, manageability,…

CHARACTERISTICS OF A MICROSERVICE

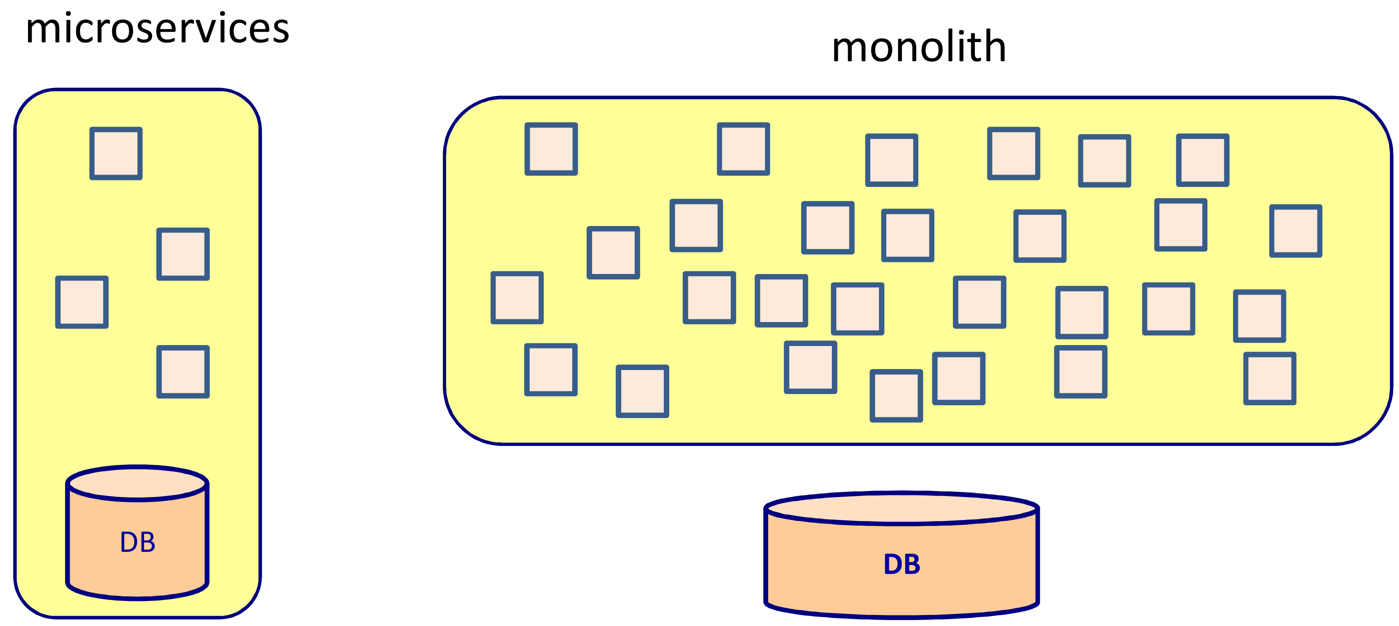
Microservices

- Small independent services
  - Simple and lightweight
  - Runs in an independent process
  - Technology agnostic
  - Decoupled

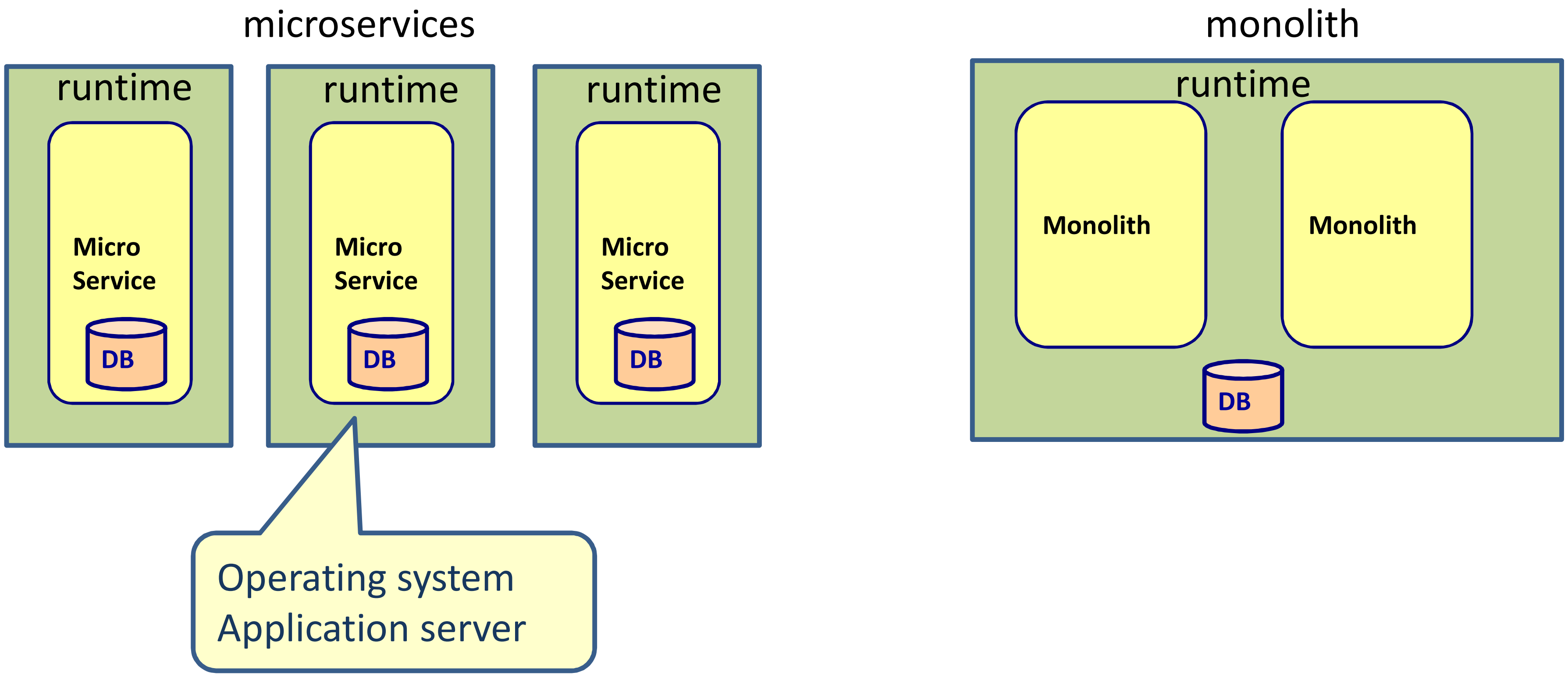


Simple and lightweight

- Small and simple
- Can be build and maintained by 1 agile team



Runs in an independent process



Operating system
Application server

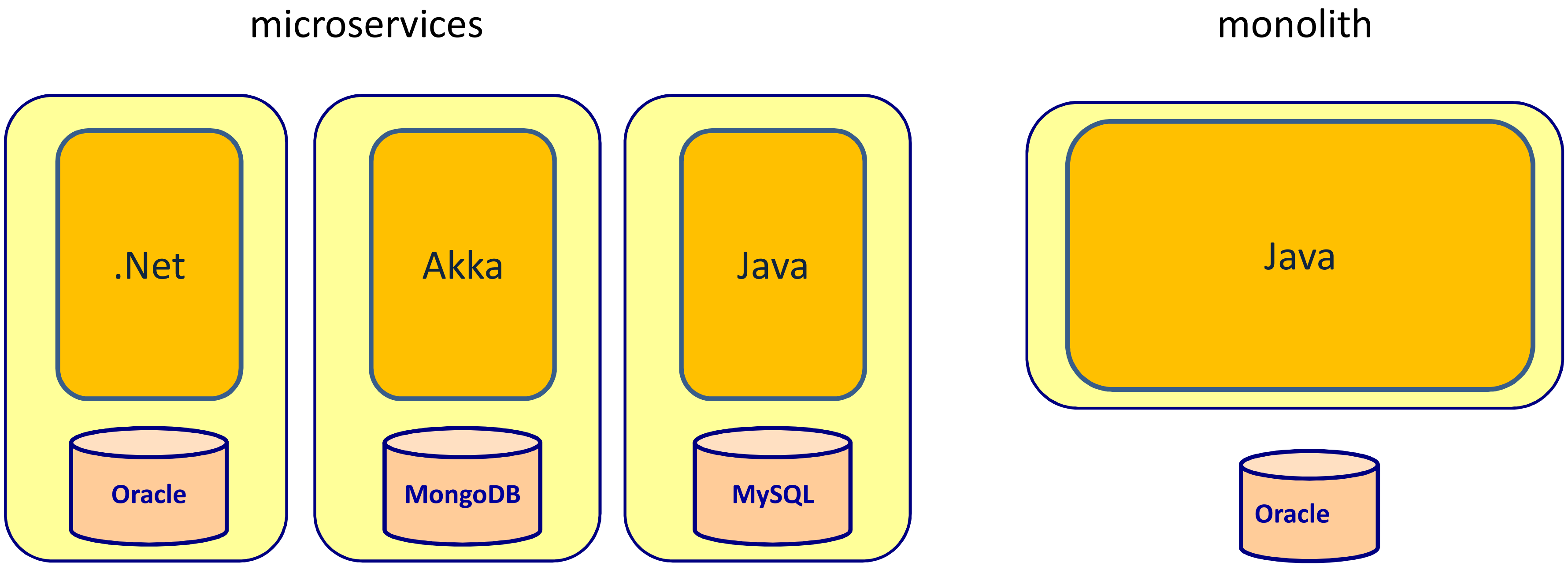
Advantages

- Runtime can be small
  - Only add what you need
- Runtime can be optimized
- Runtime can start and stop fast
- If runtime goes down, other services will still run

Disadvantages

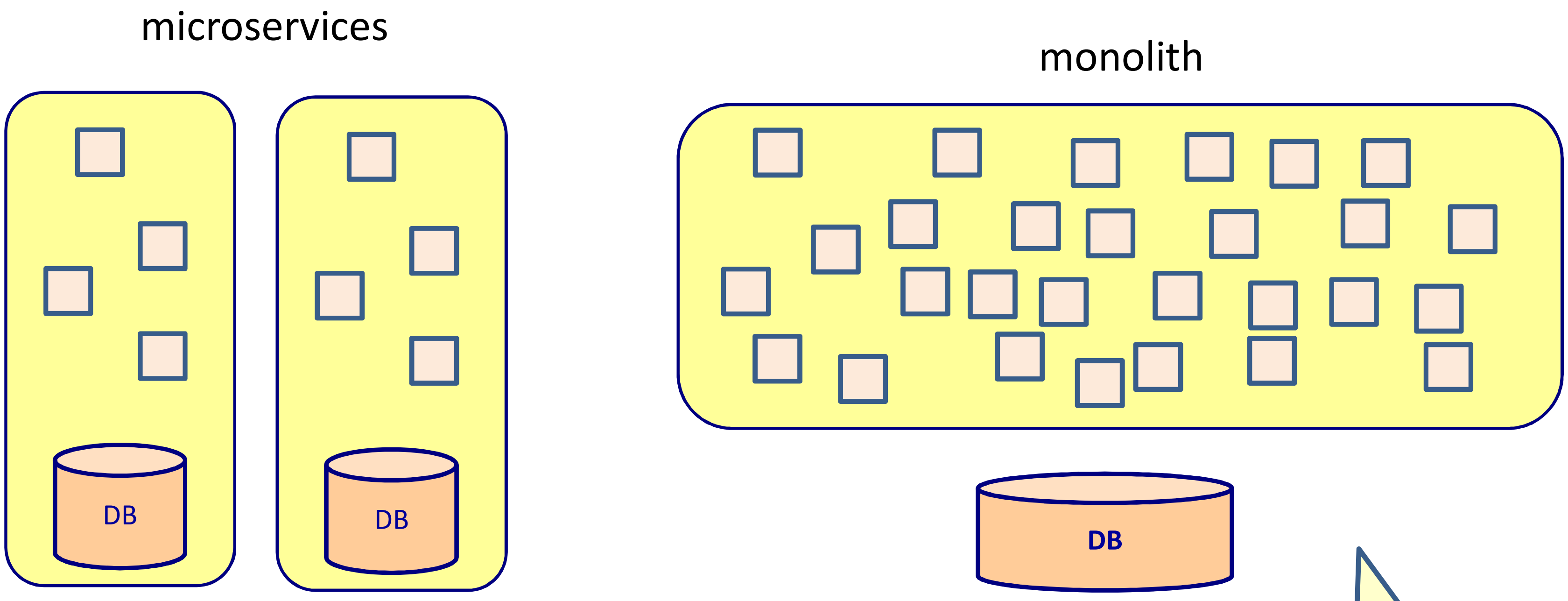
- We need to manage many runtimes

Technology agnostic



- Use the architecture and technologies that fits the best for this particular microservice

Decoupled



One change affects only the service

One change affects the whole monolith

MICROSERVICE USE CASES

Microservice use cases

- Migrate a monolith to improve scalability, manageability, agility or speed of delivery
- Rewrite a monolith to use new technologies
- Utility computing scenarios
  - Optimization service, forecasting service, price calculation service, prediction service, offer service, recommendation service
  - These are independent stateless services
- Reusable services
  - Payment service, login service, flight search service, customer profile service

Microservice use cases

- Backend applications
- Highly agile applications
- Innovation pilots
- Devops projects
- Applications with high speed to delivery
- Large complex applications

MICROSERVICE DESIGN AND IMPLEMENTATION

MICROSERVICE BOUNDARIES

Appropriate boundaries

- DDD bounded context
  - Isolated domains that are closely aligned with business capabilities
- Autonomous functions
  - Accept input, perform its logic and return a result
    - Encryption engine
    - Notification engine
    - Delivery service that accept an order and informs a trucking service

Appropriate boundaries

- Size of deployable unit
  - Manageable size
- Most appropriate function or subdomain
  - What is the most useful component to detach from the monolith?
  - Hotel booking system: 60-70% are search request
    - Move out the search function
- Polyglot architecture
  - Functionality that needs different architecture
    - Booking service needs transactions
    - Search does not need transactions

Appropriate boundaries

- Selective scaling
  - Functionality that needs different scaling
    - Booking service needs low scaling capabilities
    - Search needs high scaling capabilities
- Small agile teams
  - Specialist teams that work on their expertise
- Single responsibility

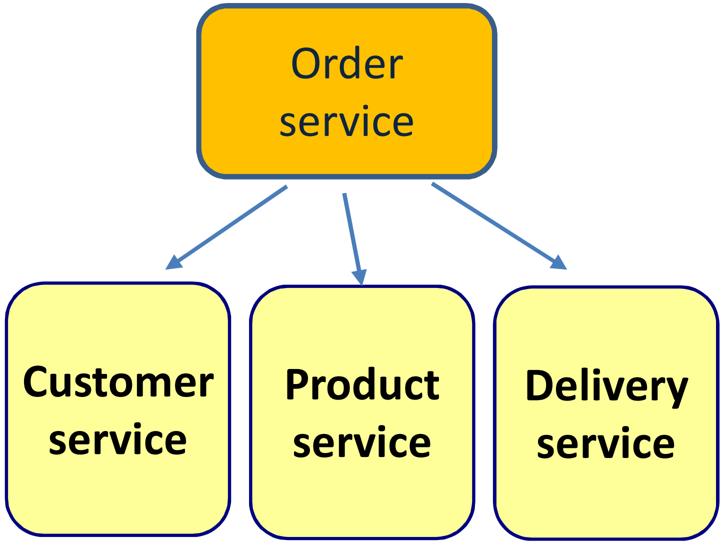
Appropriate boundaries

- Replicability or changeability
  - The microservice is easy detachable from the overall system
  - What functionality might evolve in the future?
- Coupling and cohesion
  - Avoid chatty services
  - Too many synchronous request
  - Transaction boundaries within one service

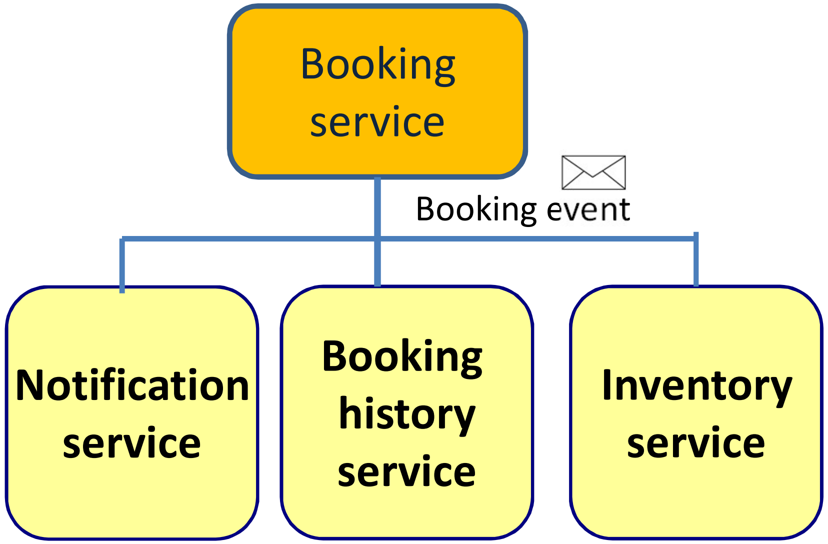
ORCHESTRATION VS. CHOREOGRAPHY

Orchestration vs. choreography

- Orchestration
  - One central brain

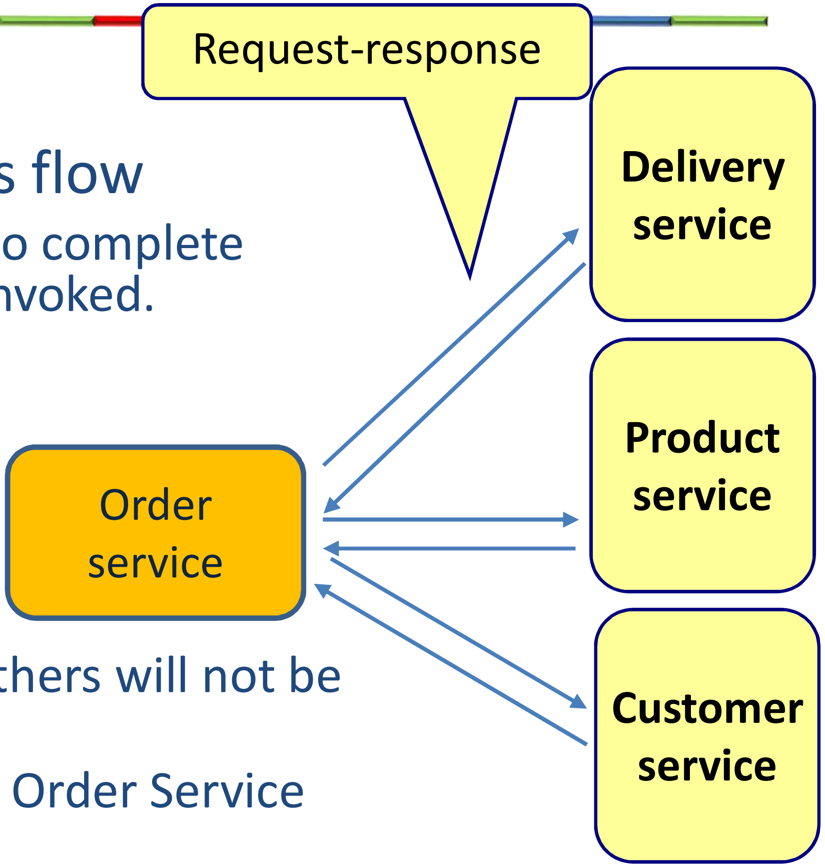


- Choreography
  - No central brain



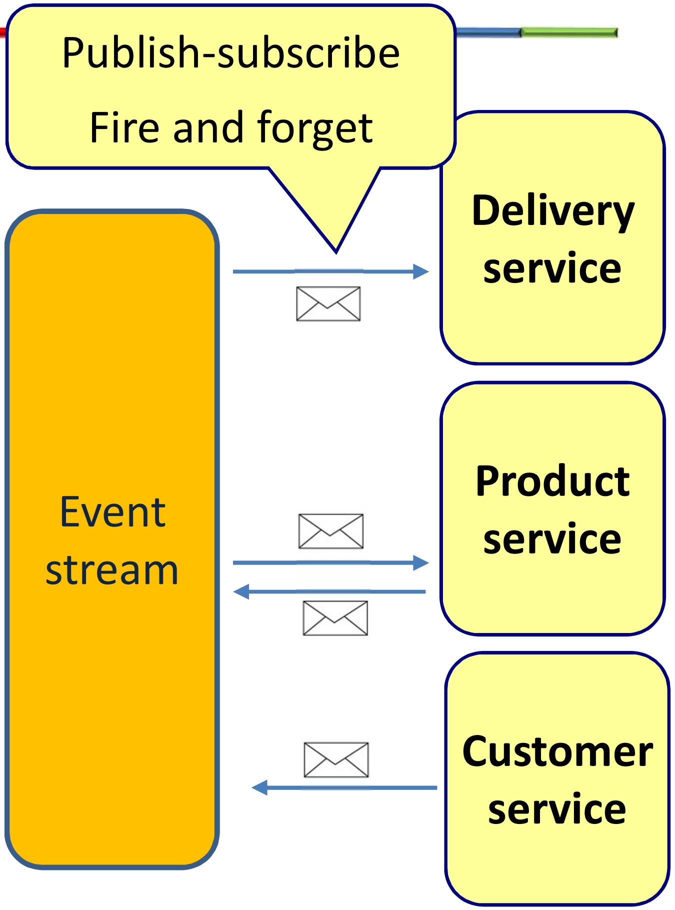
Orchestration

- Advantage
  - Full control of the synchronous flow
    - For example, if Service A needs to complete successfully before Service B is invoked.
  - Easy to monitor the process
- Disadvantages
  - Coupling
    - If the first service is down, the others will not be called
    - If you add/remove a service, the Order Service needs to change
  - Orchestrator is single point of failure
  - No parallel processing



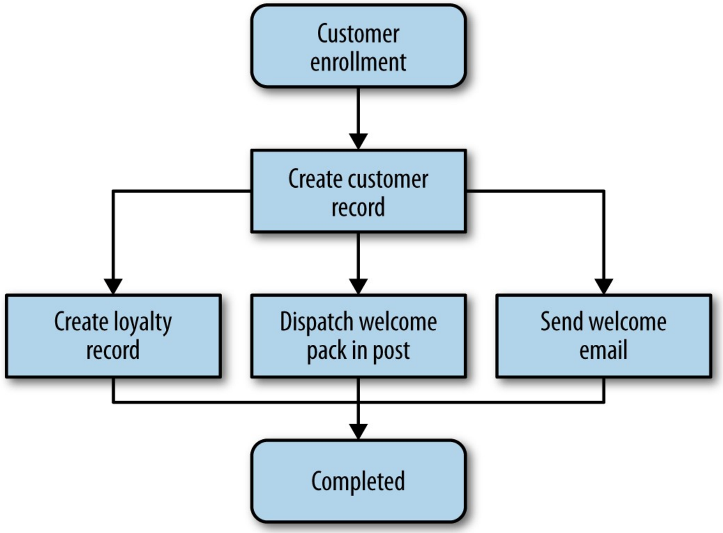
Choreography

- Advantage
  - Less Coupling
    - Easy to add/remove services without impact on other services
  - Fast: parallel processing
  - No single point of failure
- Disadvantages
  - Harder to monitor the process



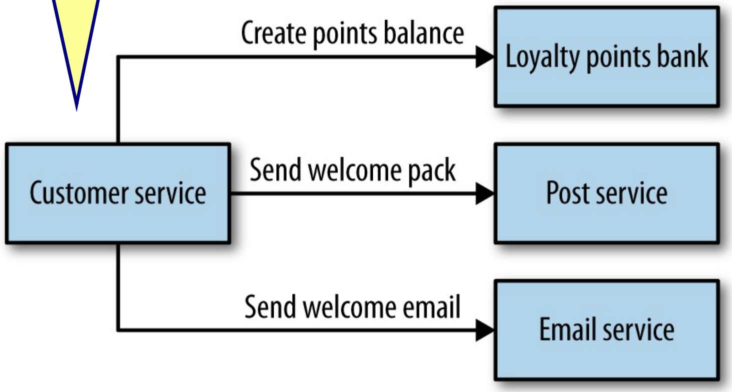
Orchestration versus choreography

Business process



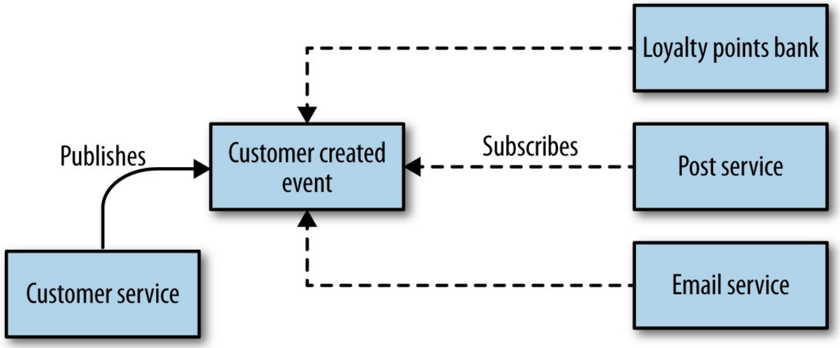
One central brain that orchestrates the process

orchestration



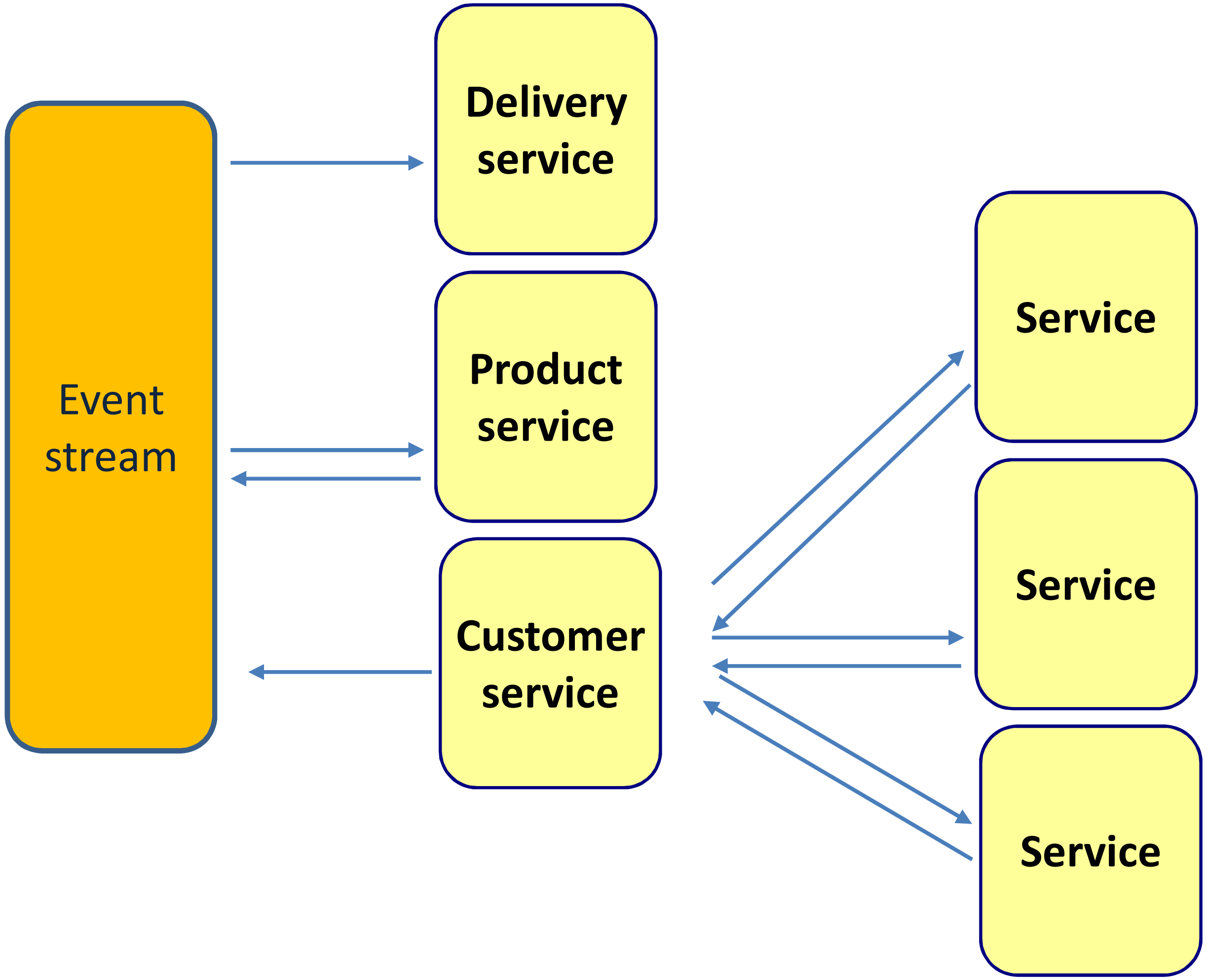
All services know their own responsibility

choreography



Hybrid solution

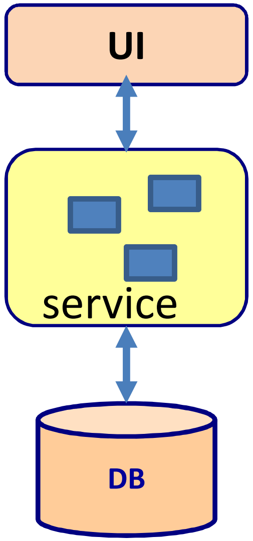
- Prefer choreography over orchestration



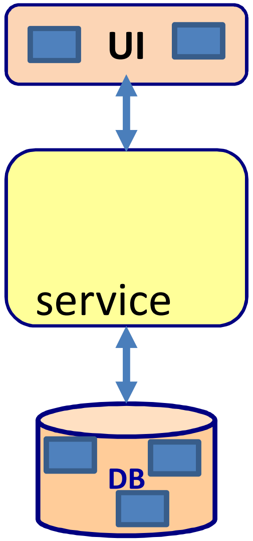
STATEFUL VS. STATELESS

Stateful vs Stateless

- Stateful
  - The service contains in-memory state
  - State is maintained between requests

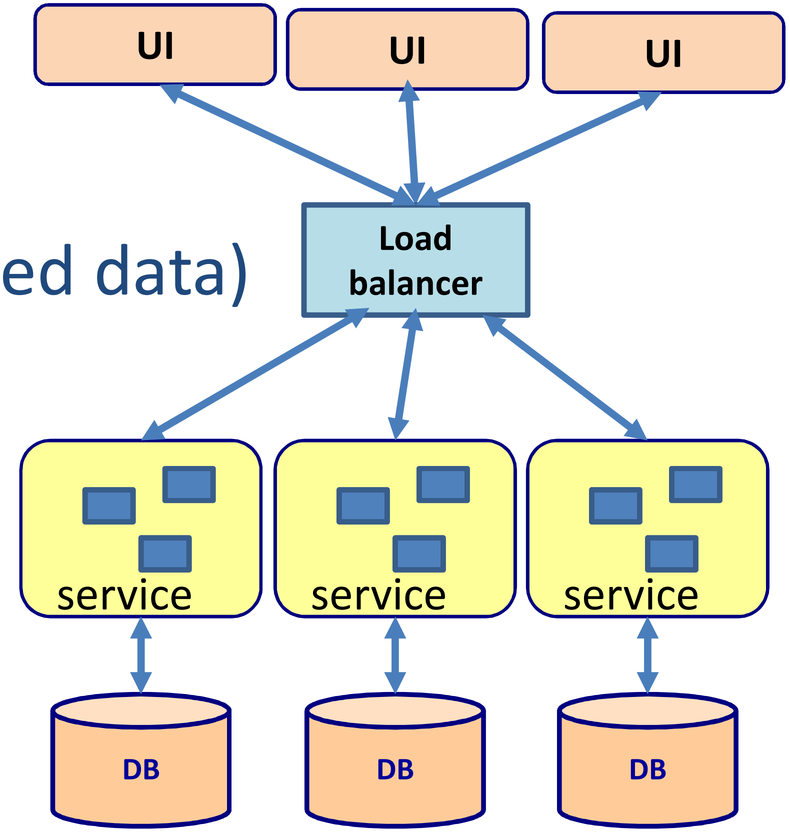


- Stateless
  - No in-memory data
  - All data is stored outside the service



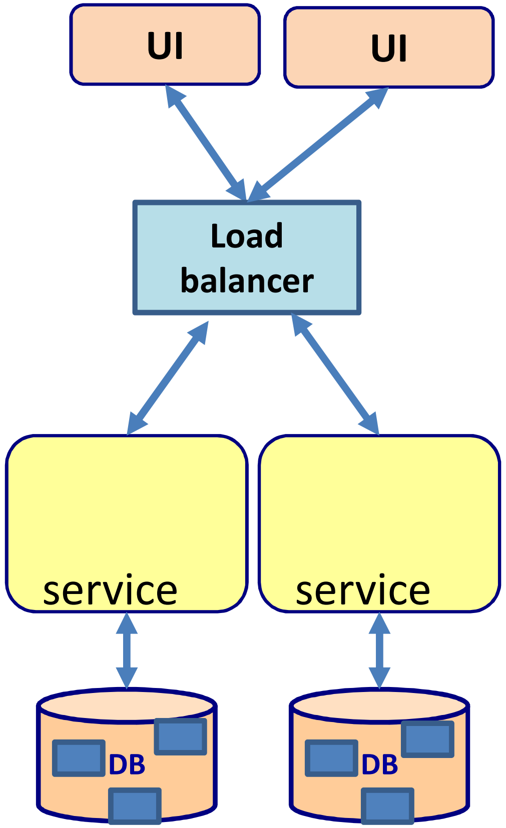
Stateful

- Advantage
  - Fast
- Disadvantages
  - Synchronization issues (shared data)
  - Hard to scale
    - State need to be replicated



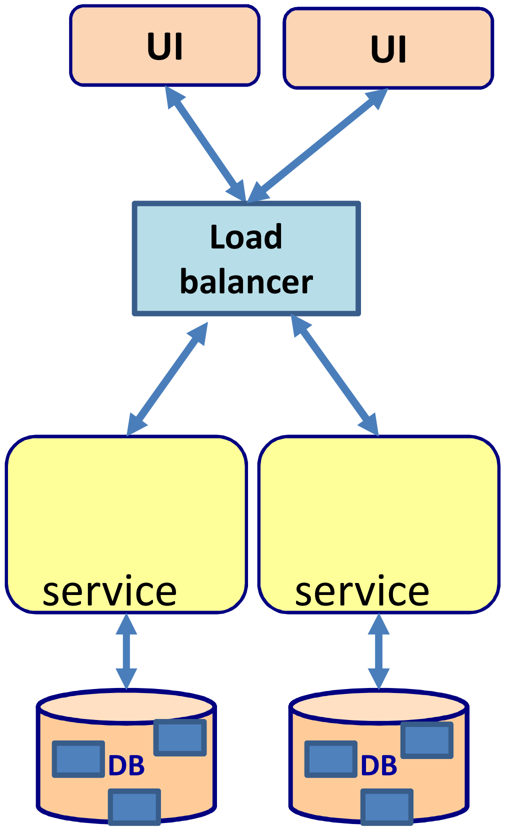
Stateless

- Advantage
  - Database takes care of synchronization
  - Easy to scale
- Disadvantages
  - Performance issue



Stateful vs Stateless

- Always prefer Stateless microservices

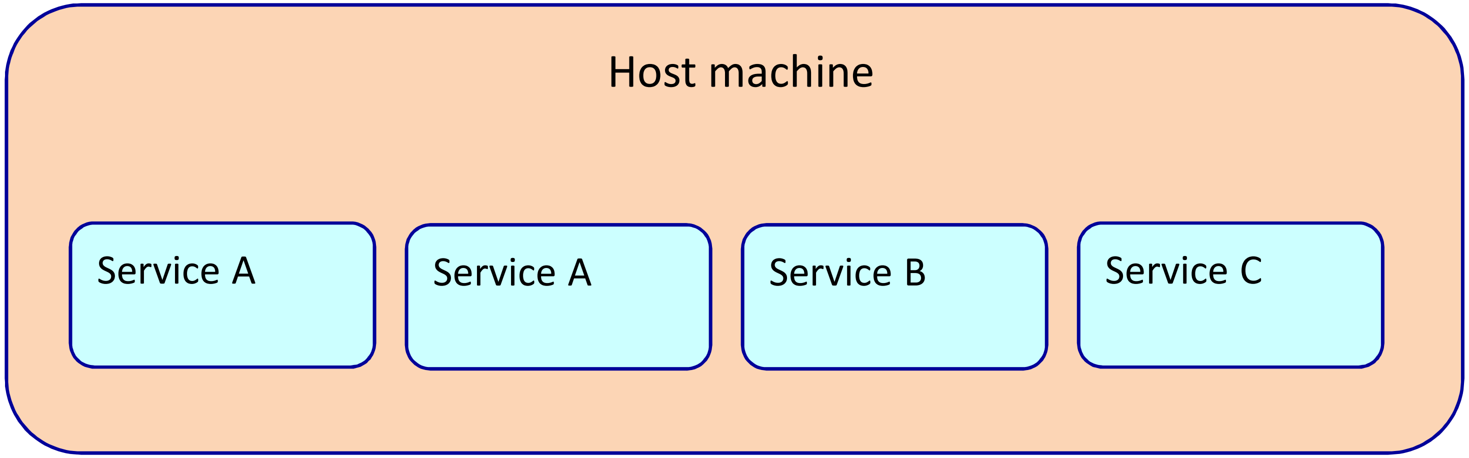


SERVICE DEPLOYMENT

Service deployment

- Service are written using different languages, frameworks, framework versions
- Run multiple service instances of a service for throughput and availability
- Building and deploying should be fast
- Instances need to be isolated
- Constrain the resources a service may consume (CPU, memory, etc.)
- Deployment should be reliable

Multiple service instances per host

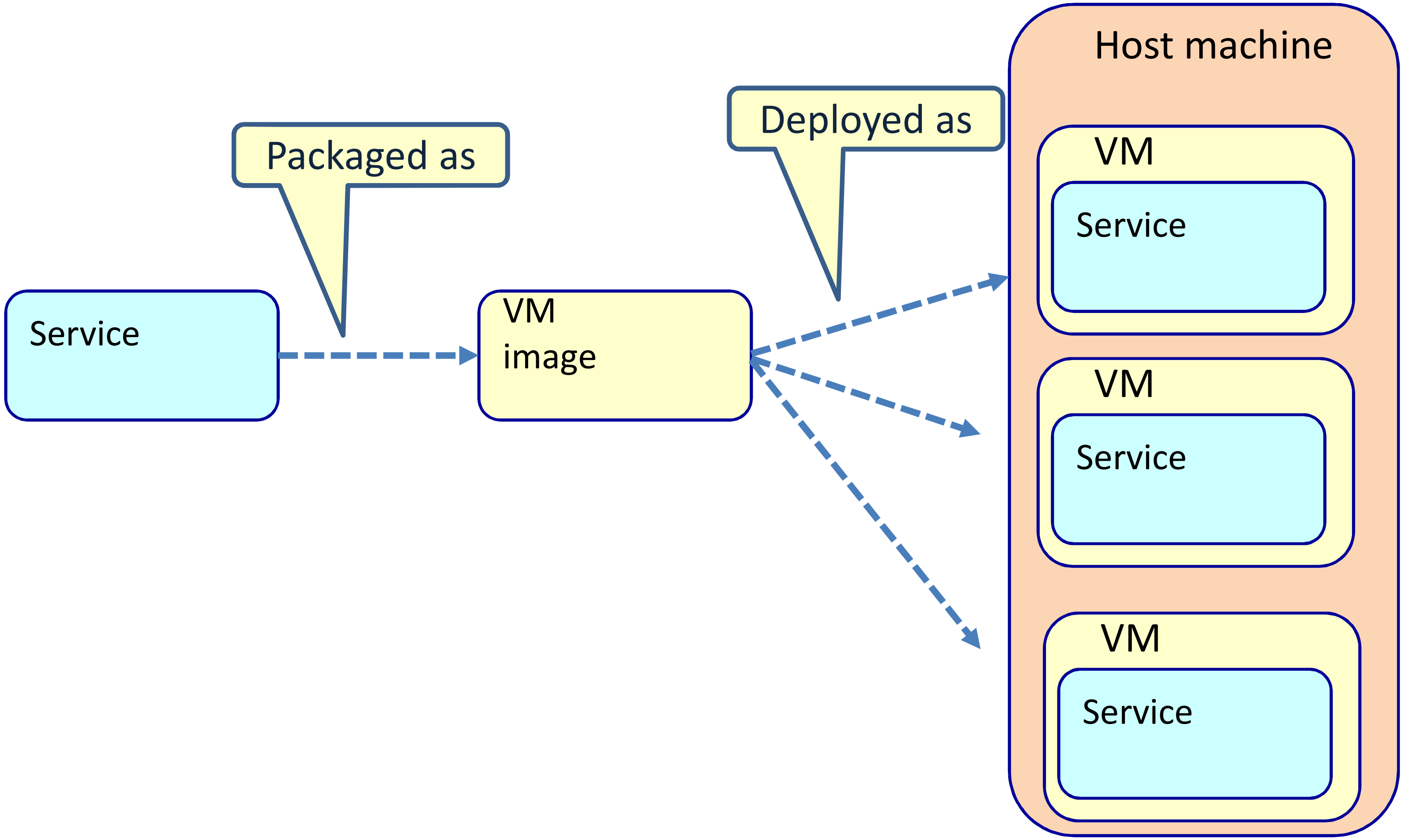


Multiple service instances per host

- Benefits
  - Efficient resource utilization
  - Fast deployment

- Drawbacks
  - Poor isolation
  - Poor visibility of resource utilization
  - Difficult to constrain resource utilization
  - Risk of dependency version conflicts
  - Poor encapsulation of implementation technology

Service per VM

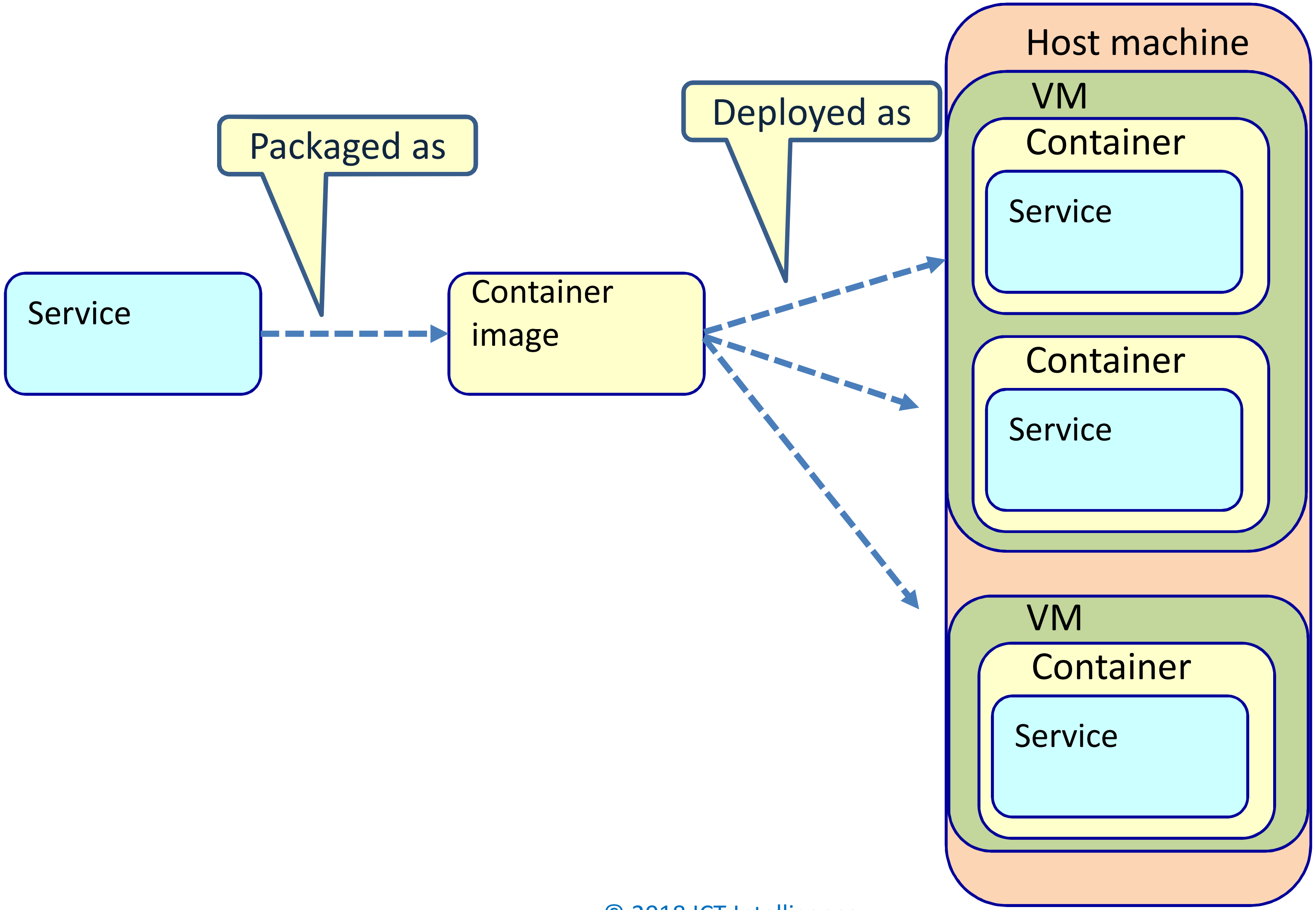


Service per VM

- Benefits
  - Great isolation
  - Great manageability
  - VM encapsulates implementation technology
  - Leverage cloud infrastructure for auto scaling/load balancing

- Drawbacks
  - Less efficient resource utilization
  - Slow deployment

Service per container



© 2018 ICT Intelligence

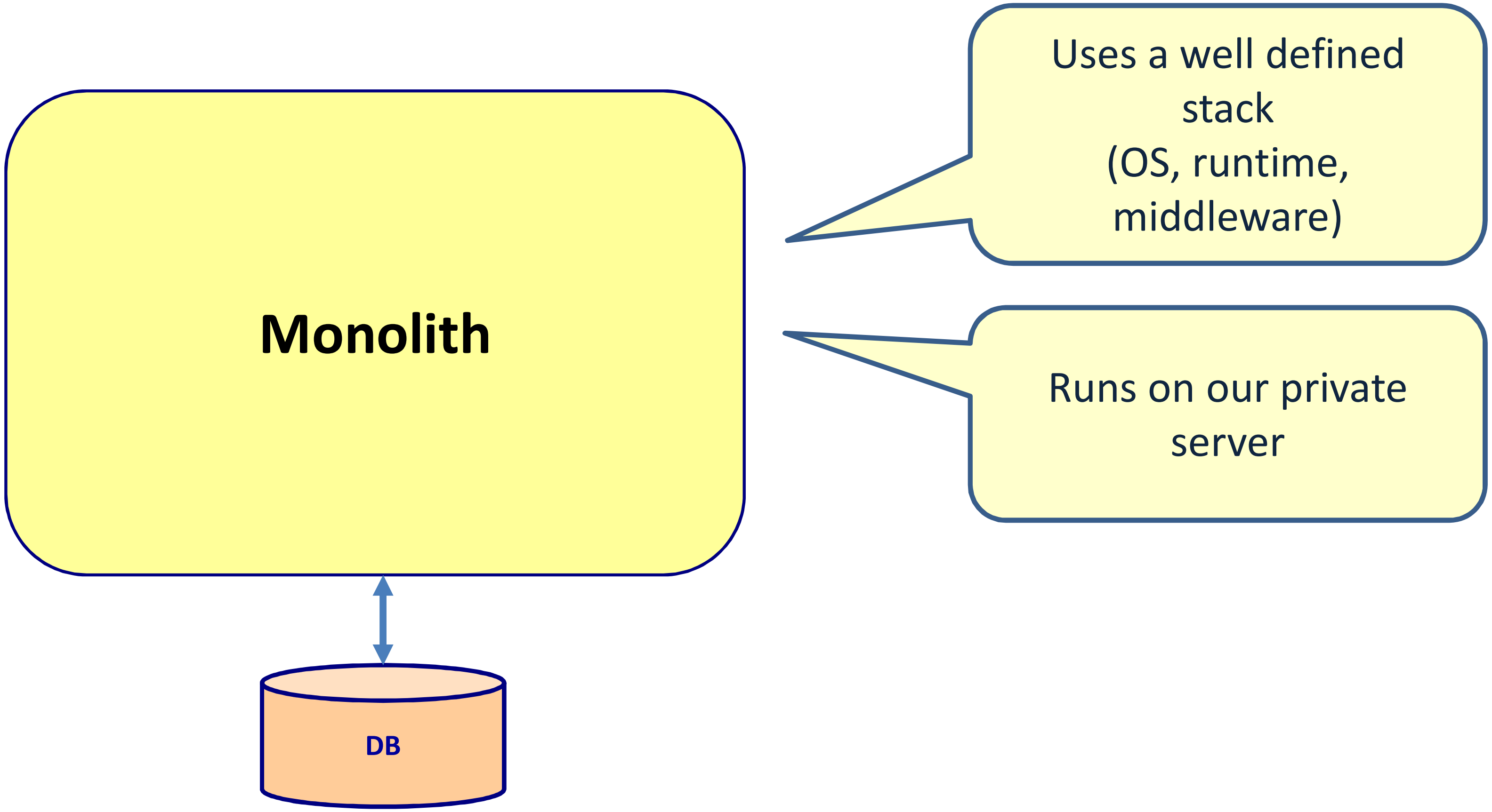
Service per container

- Benefits
  - Great isolation
  - Great manageability
  - Container encapsulates implementation technology
  - Efficient resource utilization
  - Fast deployment

- Drawbacks
  - Technology is not as mature as VM’s
  - Containers are not as secure as VM’s

CONTAINERS

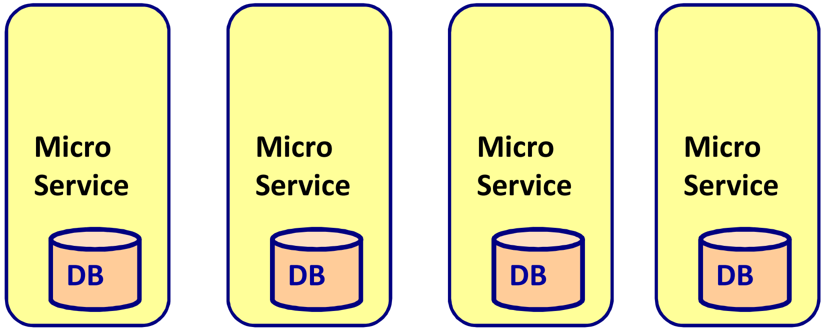
Monolith



Uses a well defined stack (OS, runtime, middleware)

Runs on our private server

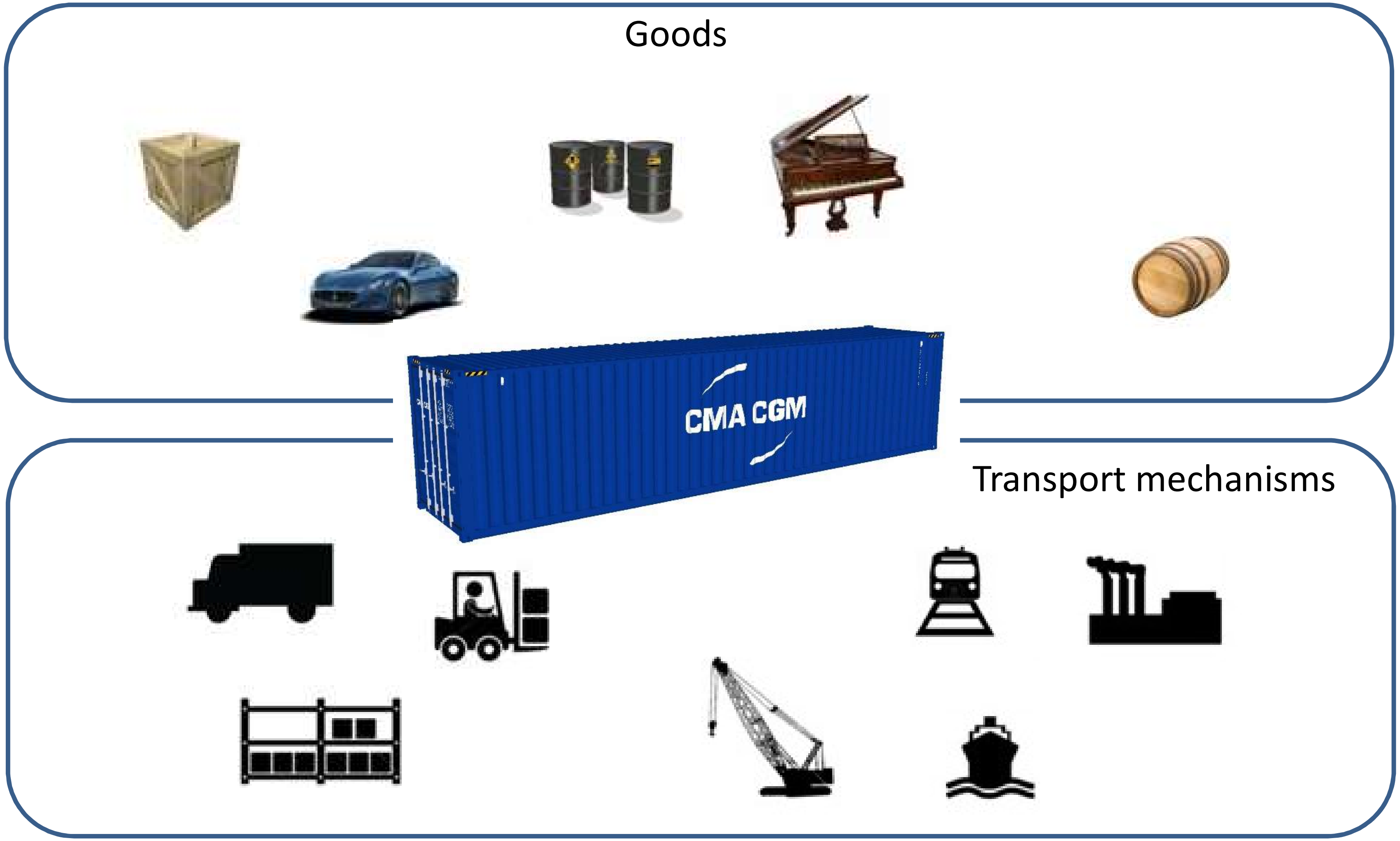
Microservice



Microservices use different stacks. (OS, runtime, middleware)

Runs anywhere (private, cloud)

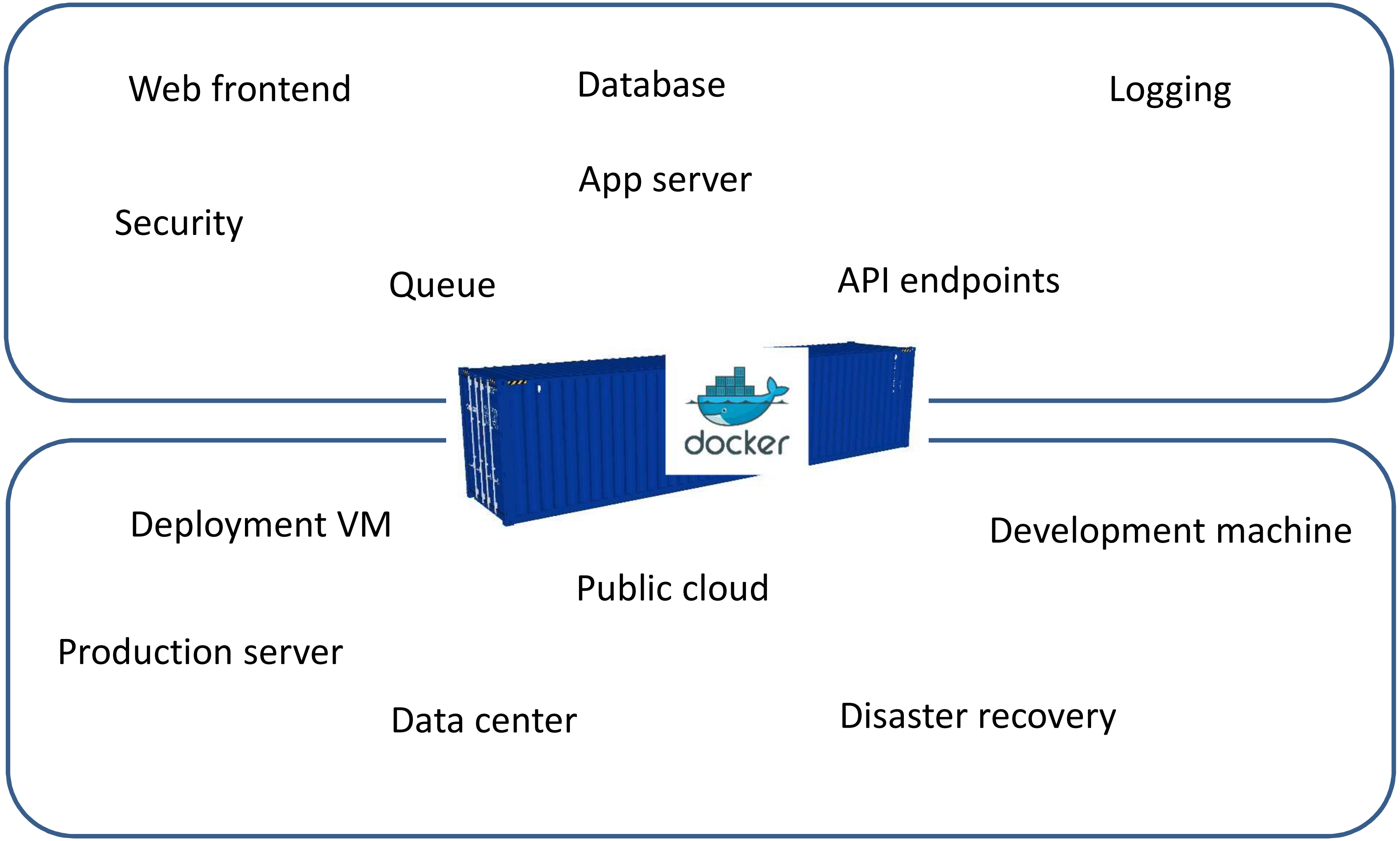
Cargo transport



Goods

Transport mechanisms

Docker: shipping container system for code



Web frontend

Database

Logging

App server

Security

Queue

API endpoints

Deployment VM

Development machine

Public cloud

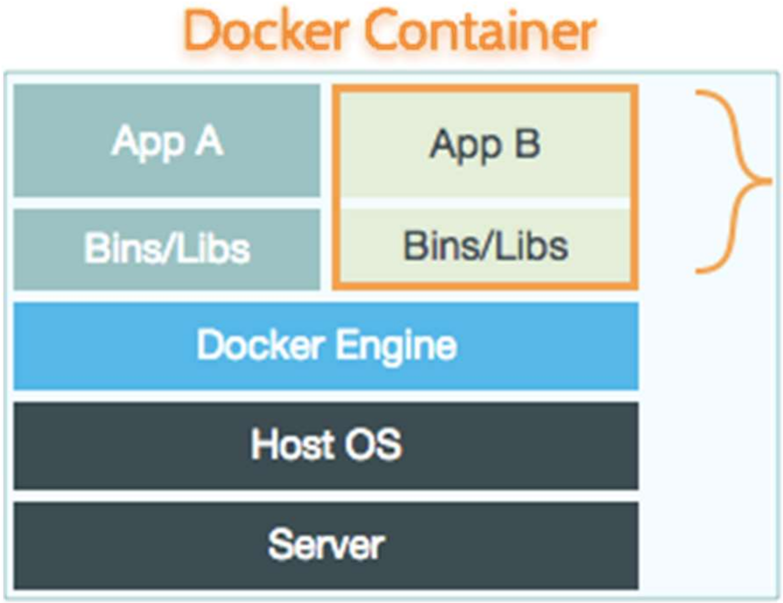
Production server

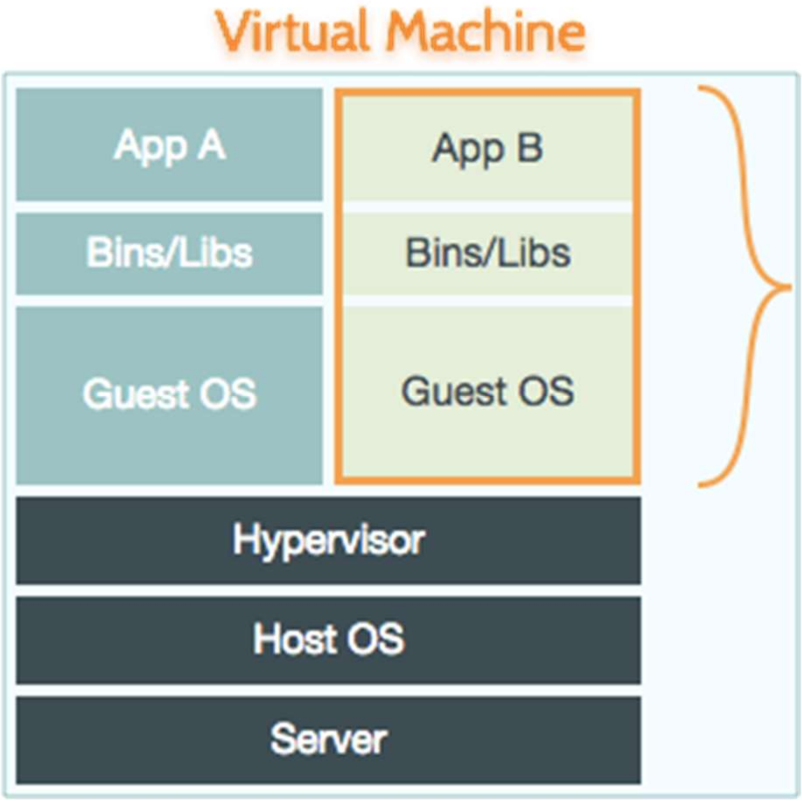
Data center

Disaster recovery

Containers

- Docker



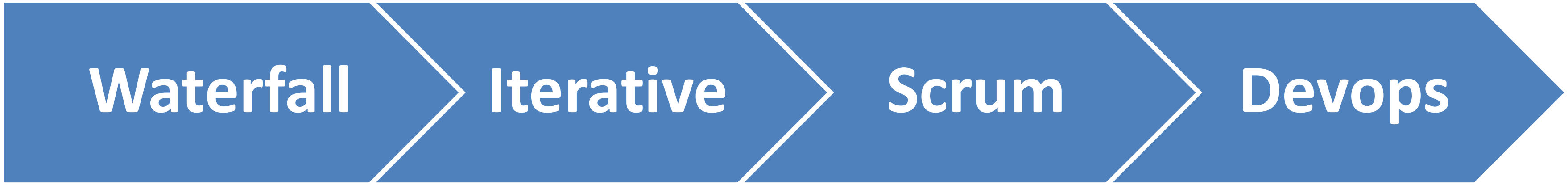


Advantages of Docker

- **Build once...run anywhere**
  - Create a run-time environment once, package it up, then run it again on any other machine.
  - Everything that runs in that environment is isolated from the underlying host.
  - Everything is fast and simple.
- **Configure once...run anything**
  - Make the entire lifecycle more efficient, consistent, and repeatable
  - Eliminate inconsistencies between development, test, production, and customer environments
  - Significantly improves the speed and reliability of continuous deployment and continuous integration systems
  - Much more efficient as VM’s

MICROSERVICES IN THE ORGANIZATION

Software development evolution

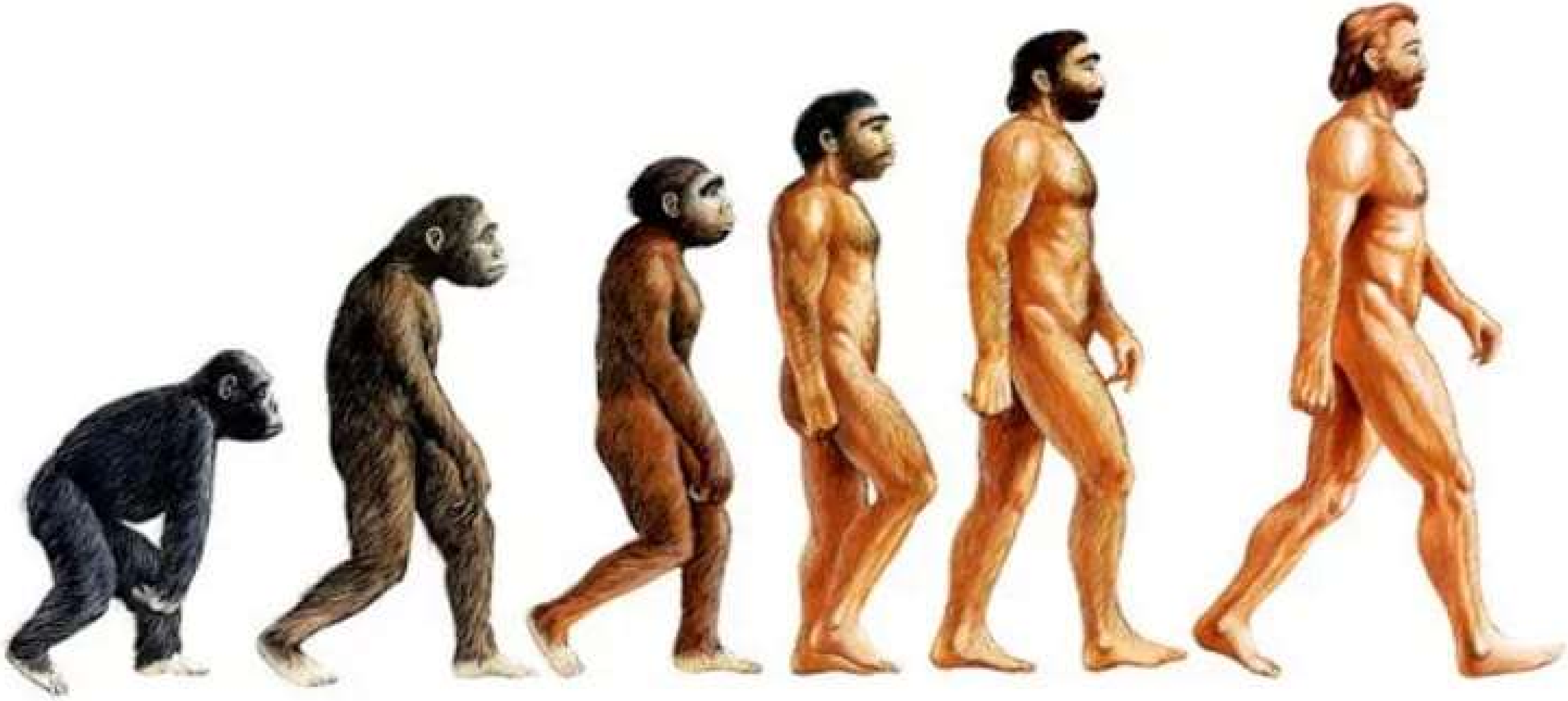


Waterfall

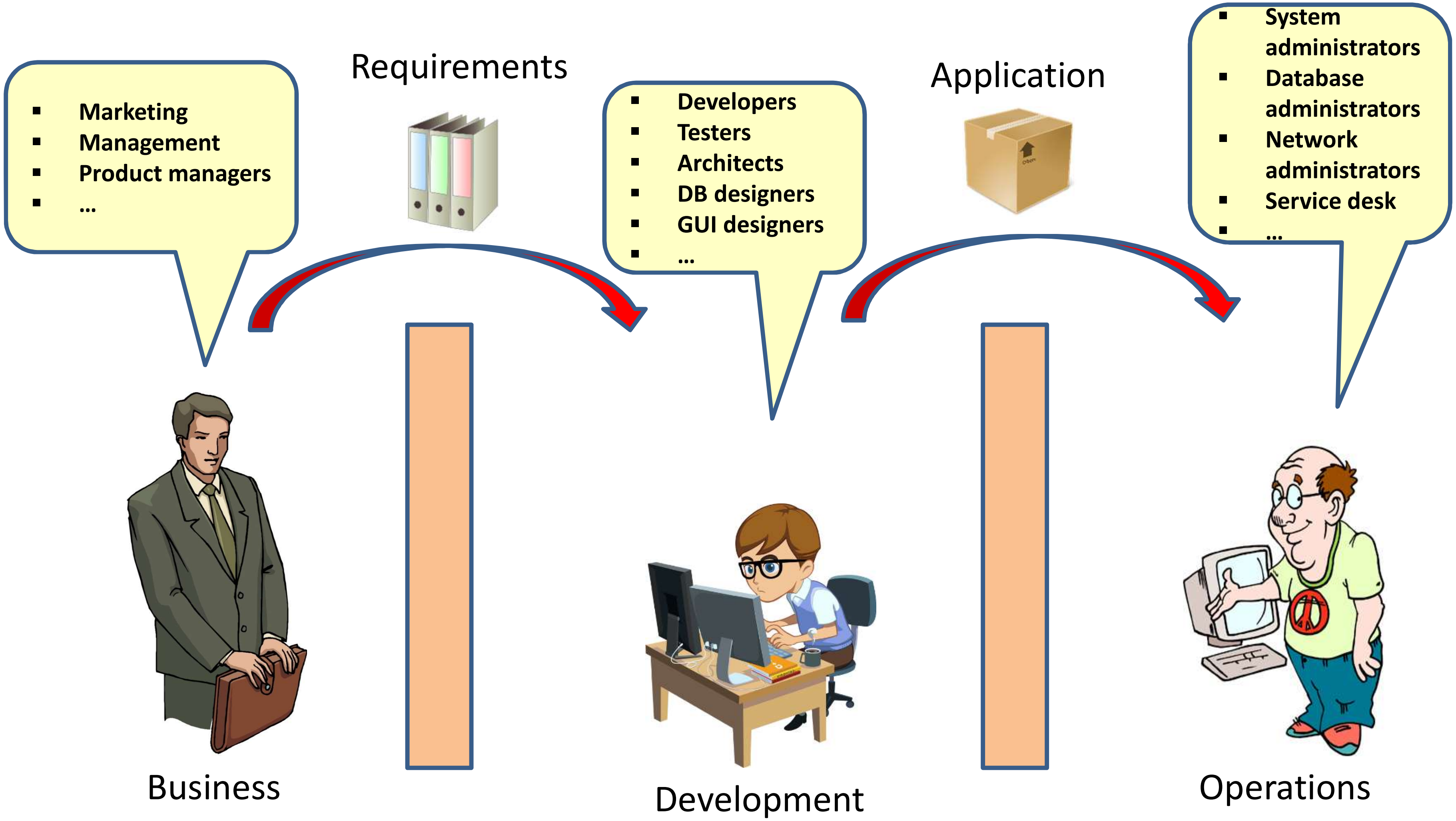
Iterative

Scrum

Devops



Traditional software development



- Marketing
- Management
- Product managers
- …

Requirements

- Developers
- Testers
- Architects
- DB designers
- GUI designers
- …

Application

- System administrators
- Database administrators
- Network administrators
- Service desk
- …

Business

Development

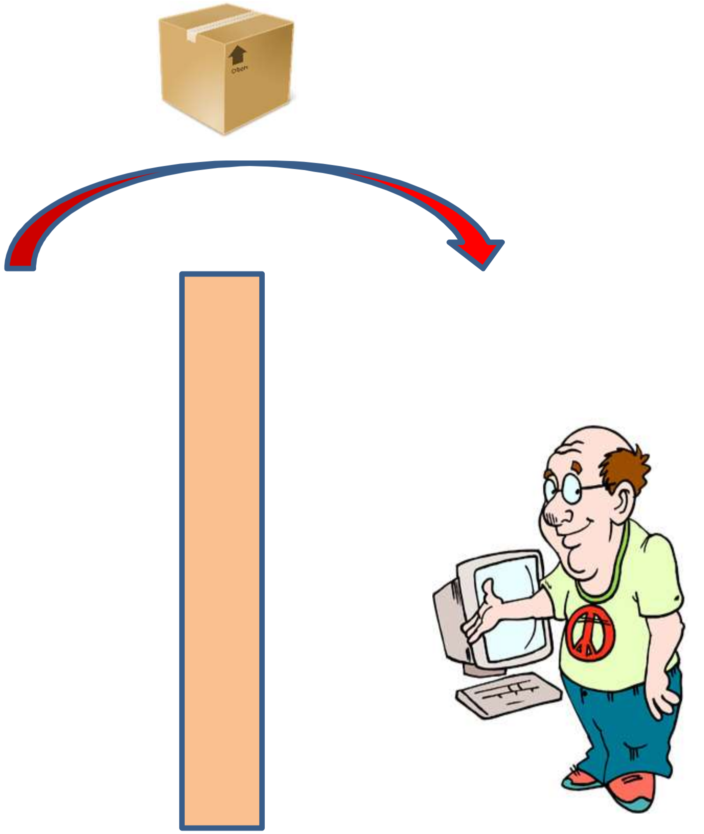
Operations

Agile software development: Scrum

- Close collaboration
- Better communication
- Short delivery cycles
- Short feedback loops

Product owner (business)
and developers in one team

Application



Operations

DevOps

- Close collaboration between developers and operations
- Streamlines the delivery process of software from business requirements to production
- Better communication
- Identical development and production environment
- Shared tools
  - Automate everything
  - Monitor everything

Product owner (business) and developers in one team

Operations

Why DevOps?

Business

I want

1. The software to change to adept to the changing world as fast as possible
2. The existing IT services to remain stable and not disrupted from the changes

- New releases
- New features
- New platforms
- New architecture
- Functional requirements

Development

I provide change

- Stable platforms
- No downtime
- Scalable platform
- Non-functional requirements

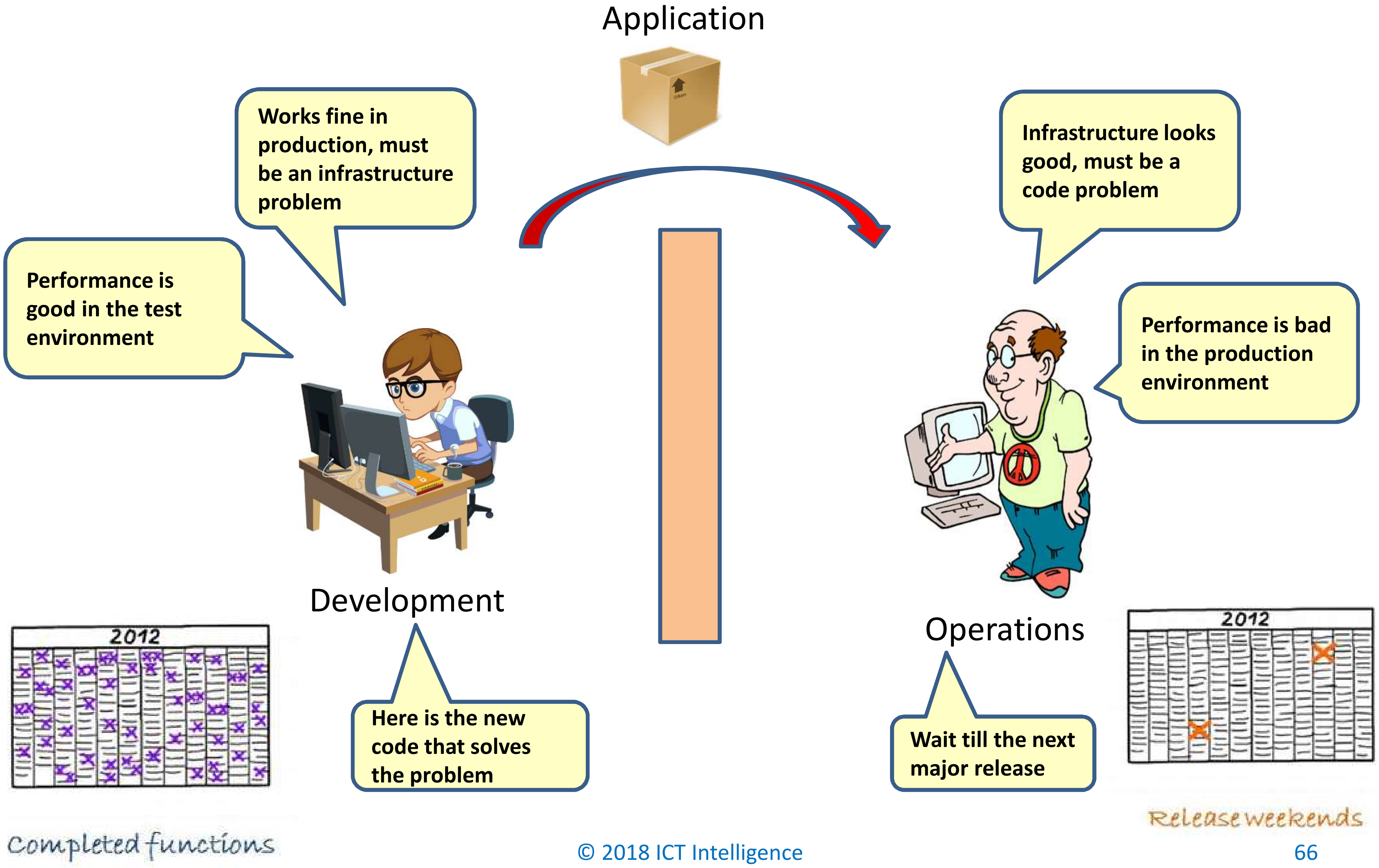
Operations

I provide stability

Both have their own

- Goals
- Tools
- Process
- Schedules

Why DevOps?



Application

Works fine in production, must be an infrastructure problem

Performance is good in the test environment

Development

Here is the new code that solves the problem

Infrastructure looks good, must be a code problem

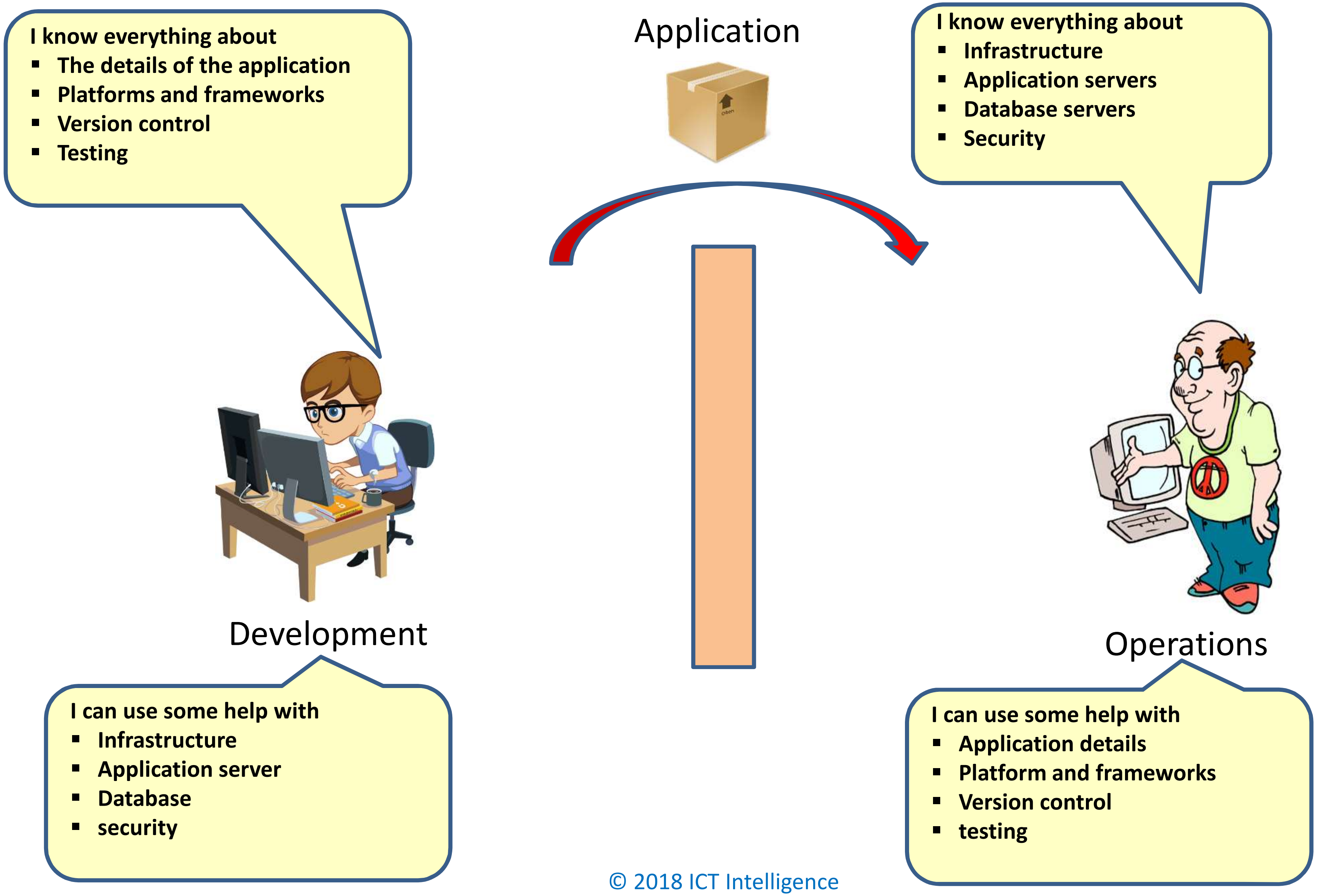
Performance is bad in the production environment

Operations

Wait till the next major release

© 2018 ICT Intelligence

Why DevOps?



I know everything about

- The details of the application
- Platforms and frameworks
- Version control
- Testing

Application

I know everything about

- Infrastructure
- Application servers
- Database servers
- Security

Development

Operations

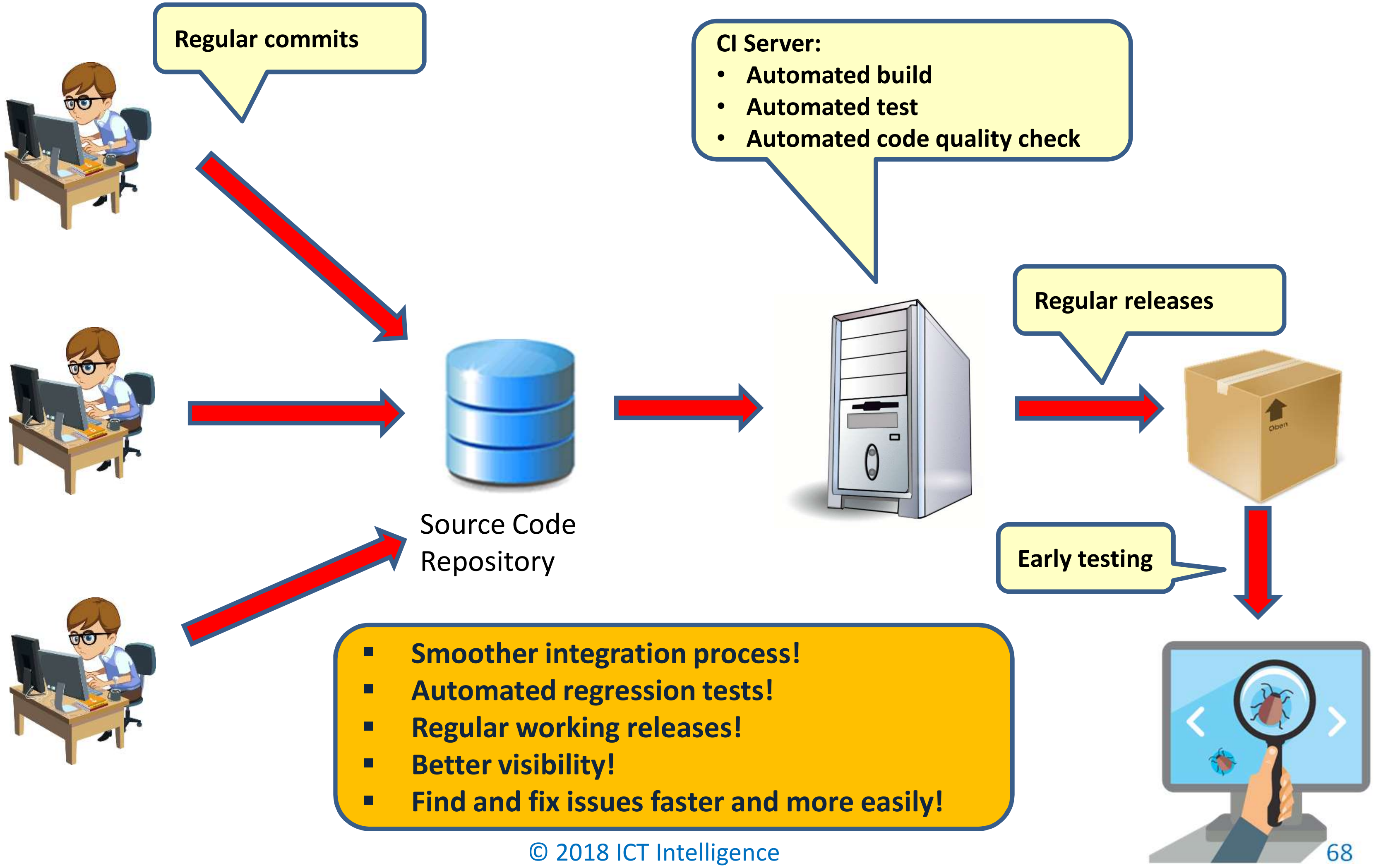
I can use some help with

- Infrastructure
- Application server
- Database
- security

I can use some help with

- Application details
- Platform and frameworks
- Version control
- testing

Continuous Integration (CI)



Regular commits

Source Code Repository

CI Server:

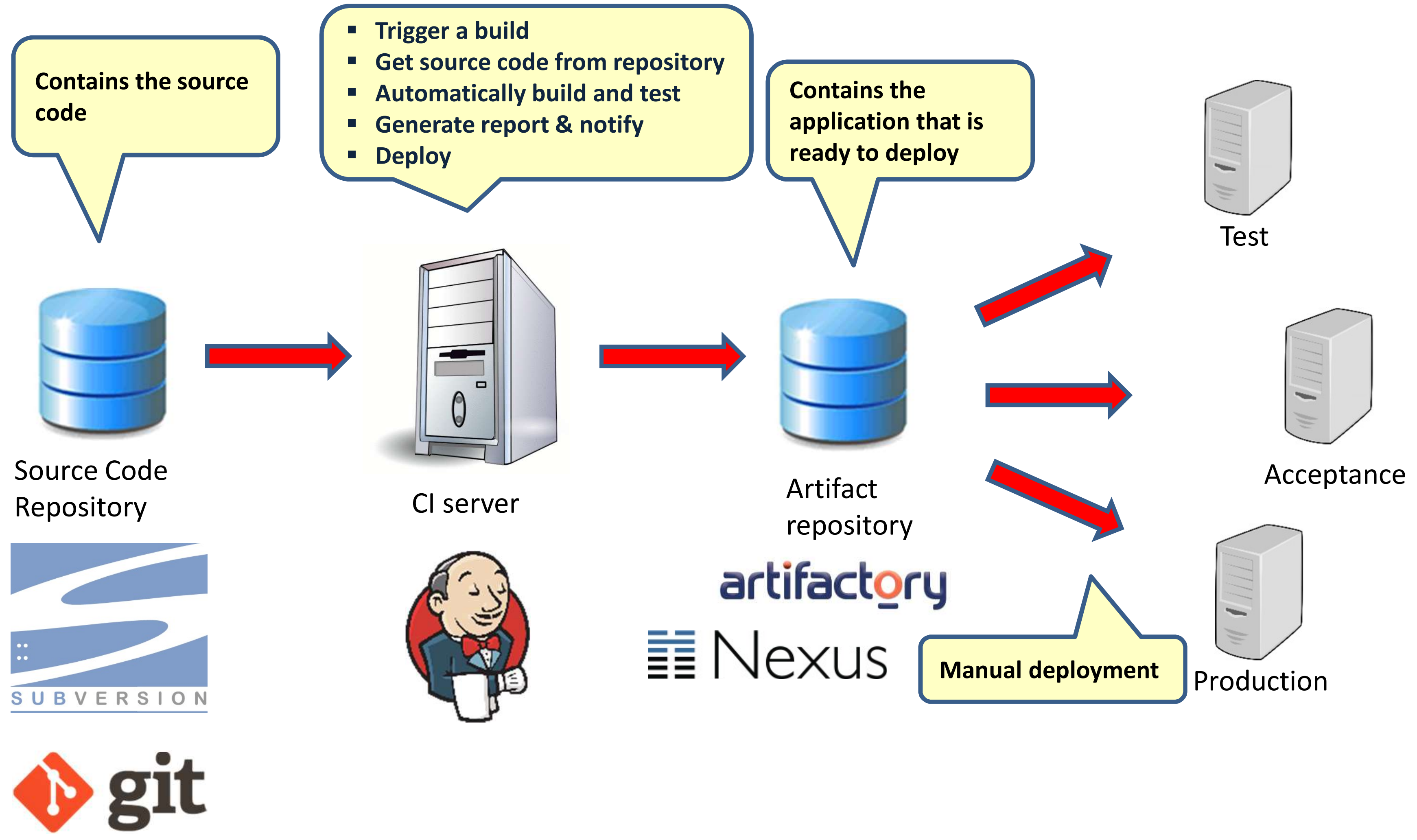
- Automated build
- Automated test
- Automated code quality check

Regular releases

Early testing

- Smoother integration process!
- Automated regression tests!
- Regular working releases!
- Better visibility!
- Find and fix issues faster and more easily!

Scope of CI



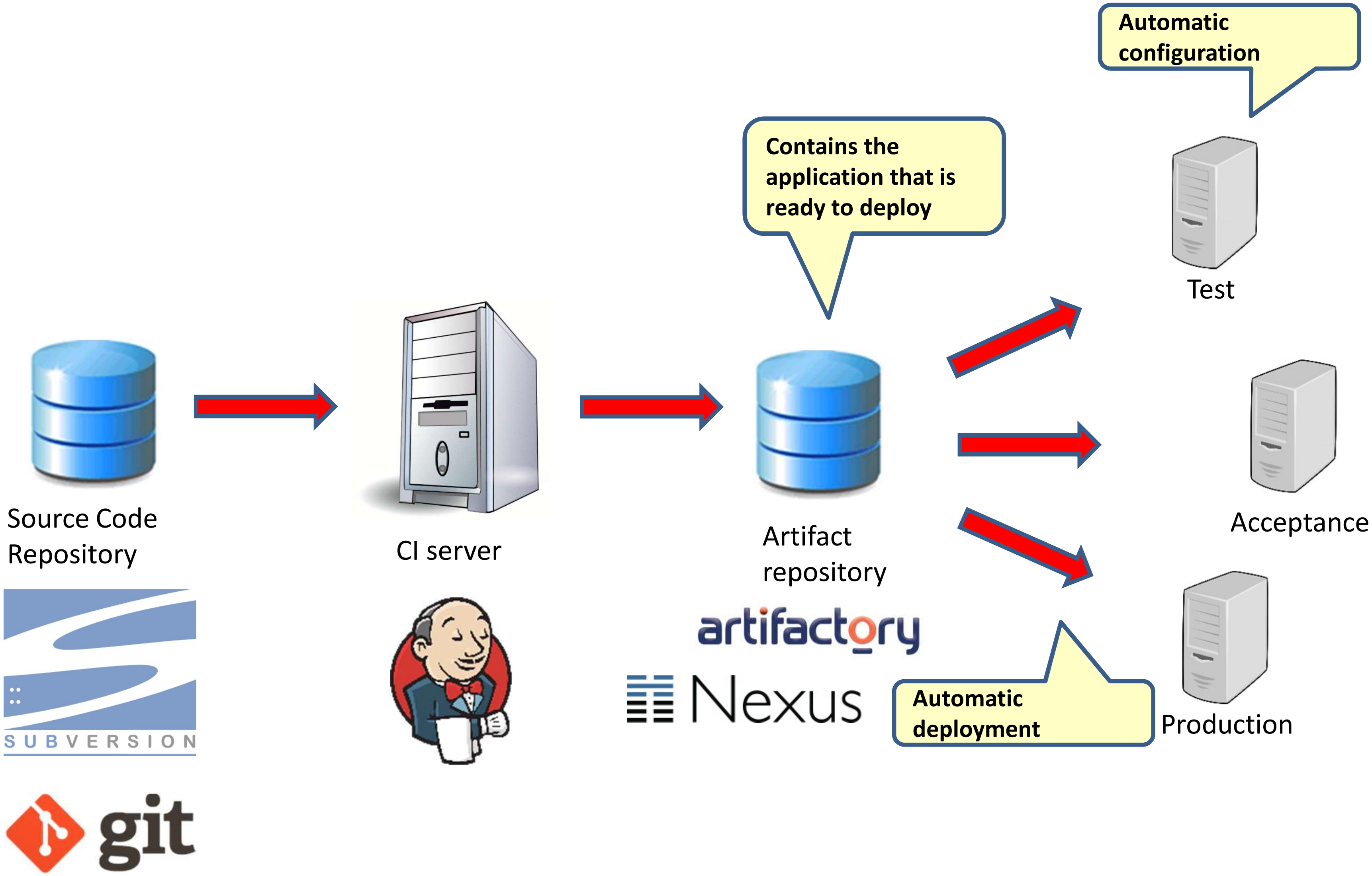
Contains the source code

- Trigger a build
- Get source code from repository
- Automatically build and test
- Generate report & notify
- Deploy

Contains the application that is ready to deploy

Manual deployment

Continuous deployment



Automatic configuration

Contains the application that is ready to deploy

Test

Acceptance

Production

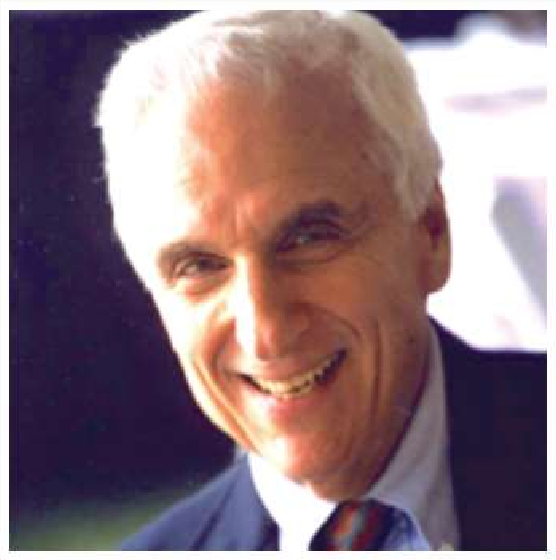
Source Code Repository

CI server

Artifact repository

Automatic deployment

Conways law



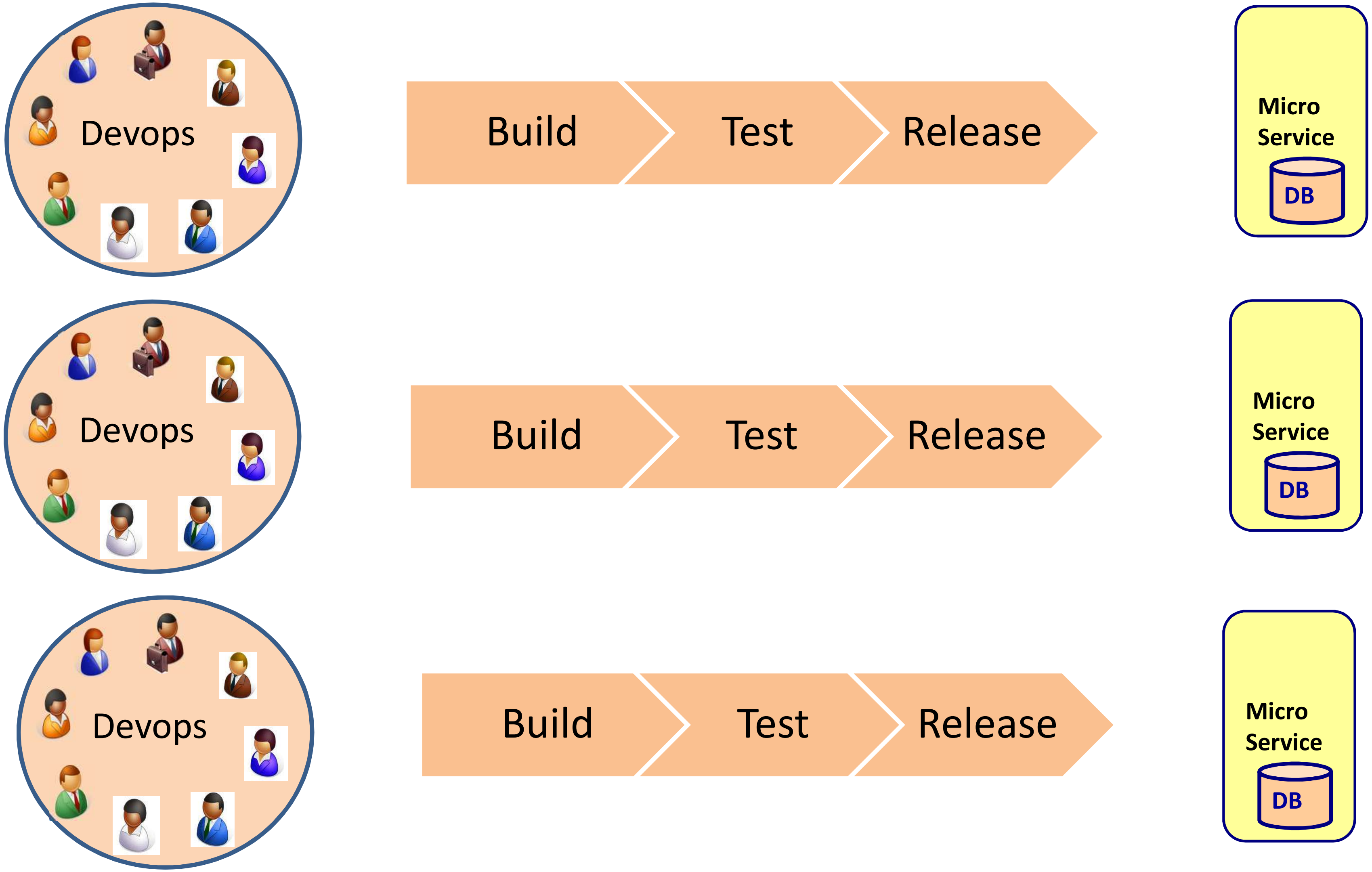
"If you have four groups working on a compiler, you'll get a 4-pass compiler"

—Eric S Raymond

"organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations "

—Melvin Conway

Microservice organization

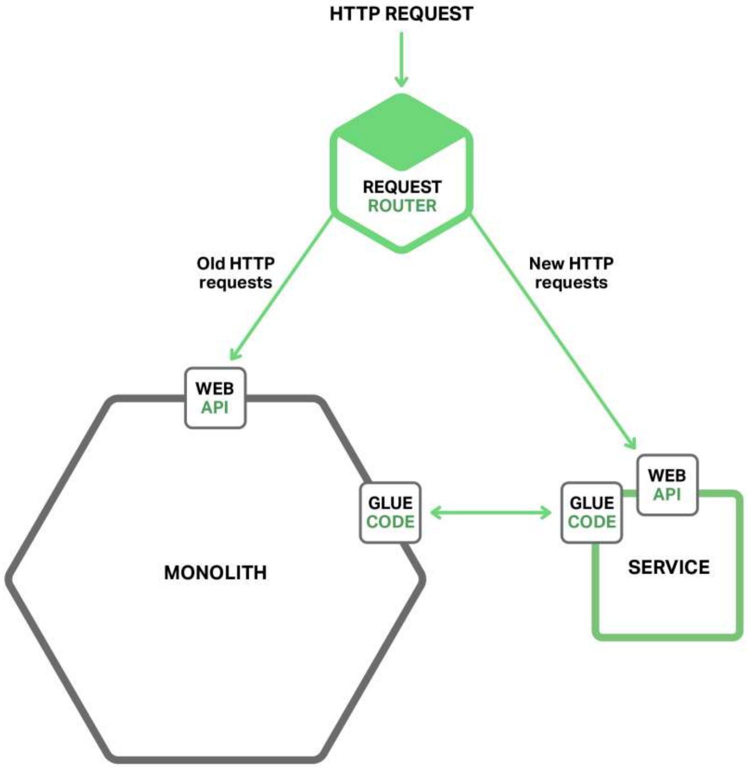


FROM MONOLITH TO MICROSERVICE

From monolith to microservice

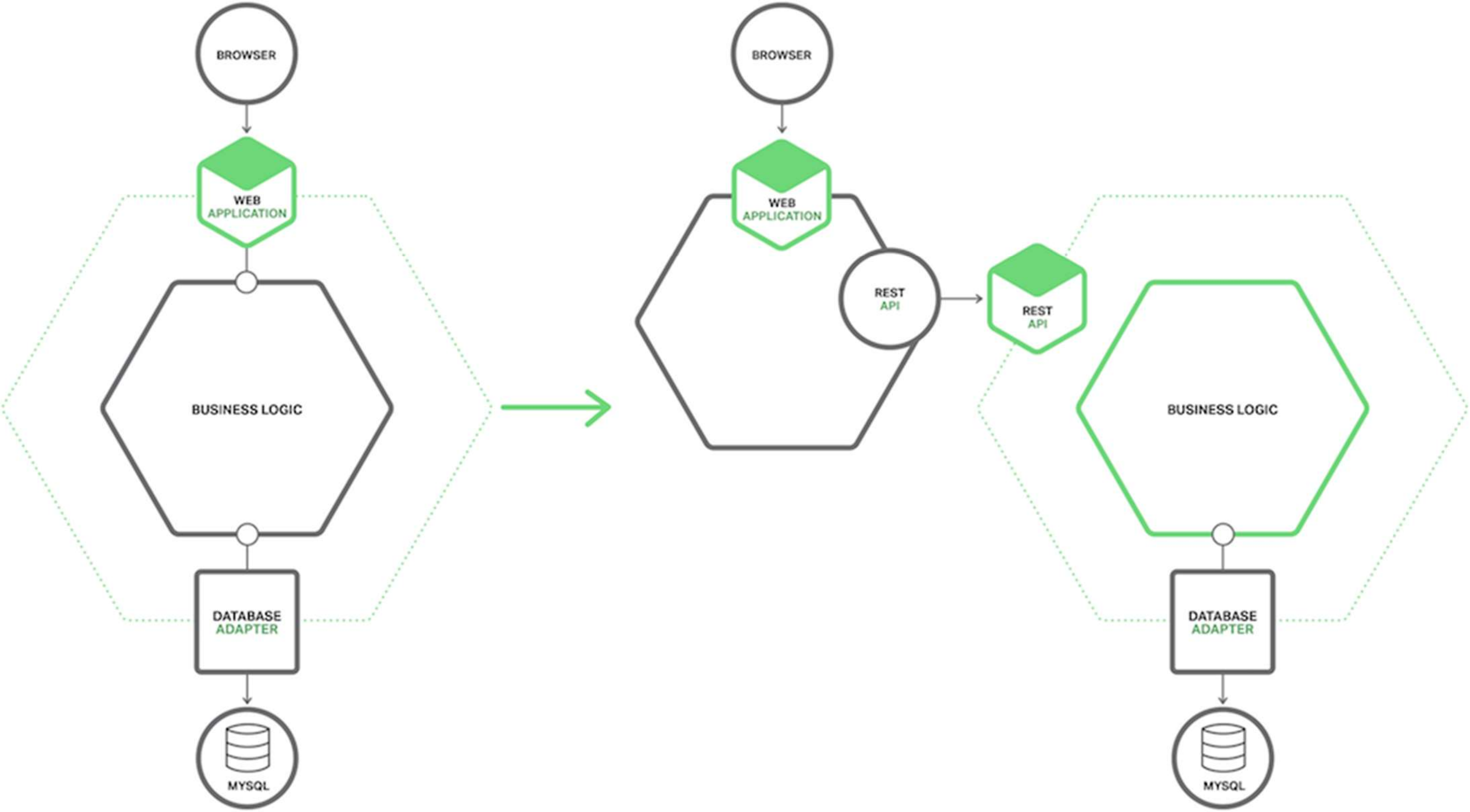
- Not a big bang
- Strangler approach
  - We build new microservices around the monolith

Stop digging strategy

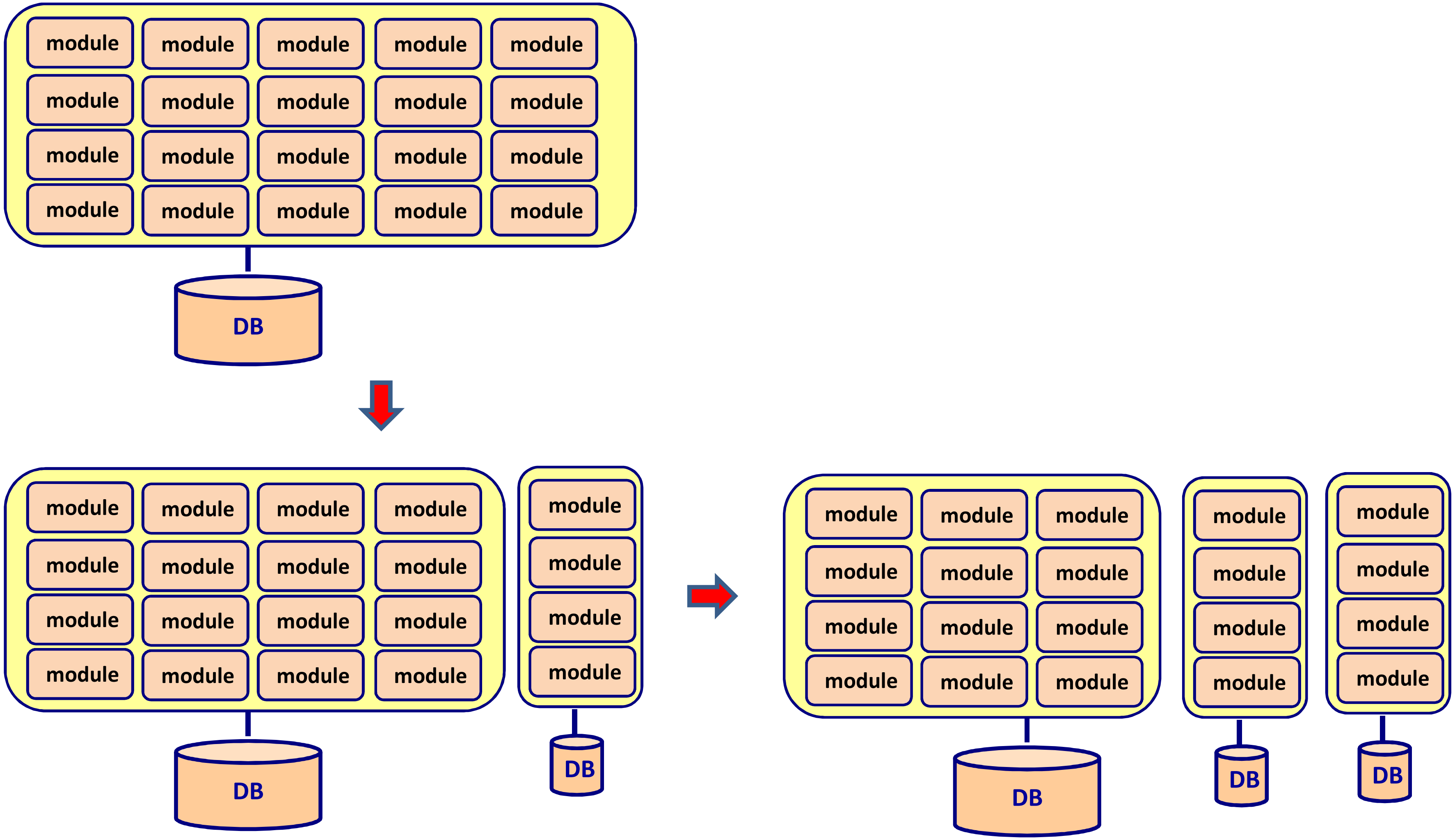


- Add new code in a separate service

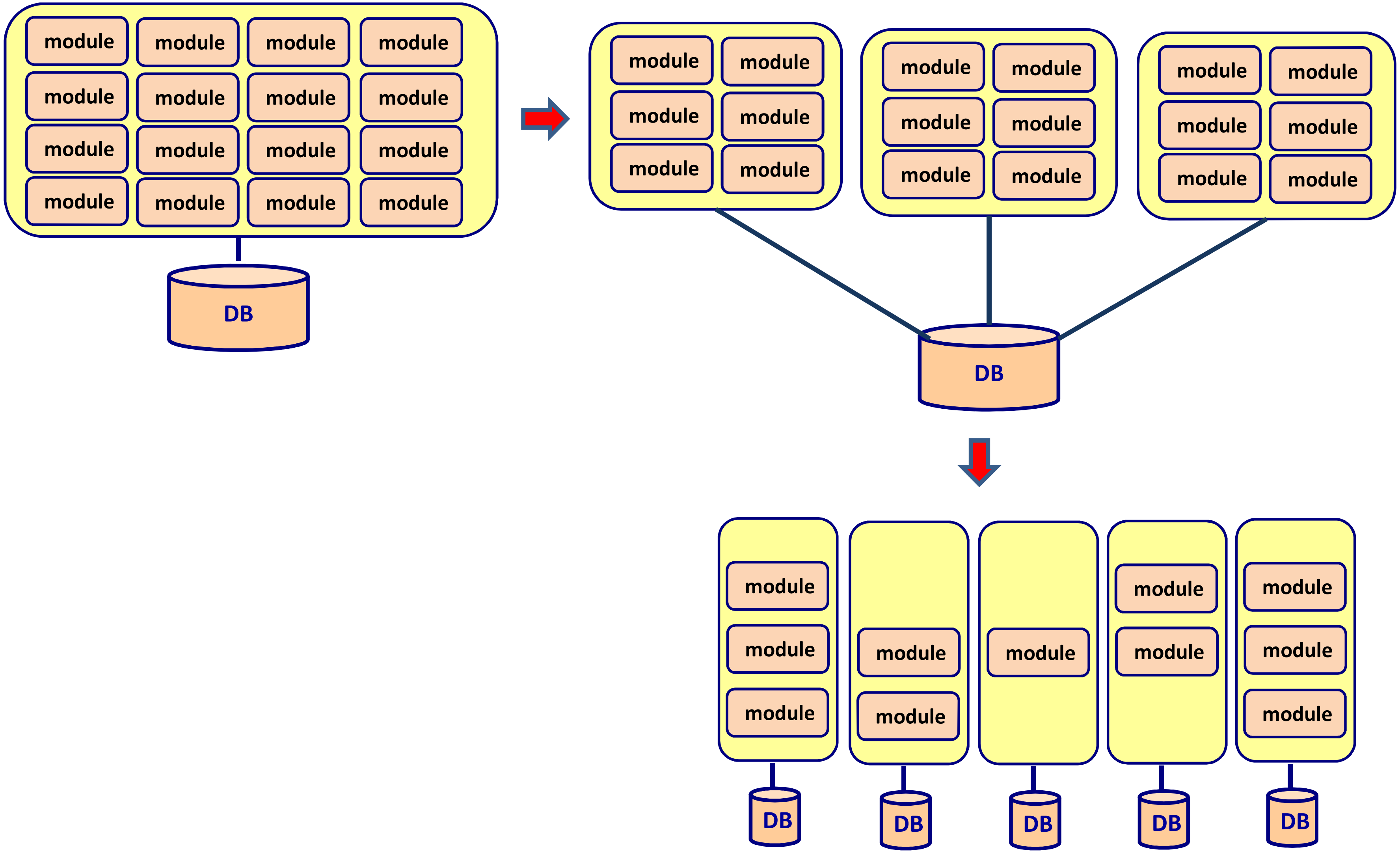
Split frontend and backend strategy



Extract services strategy

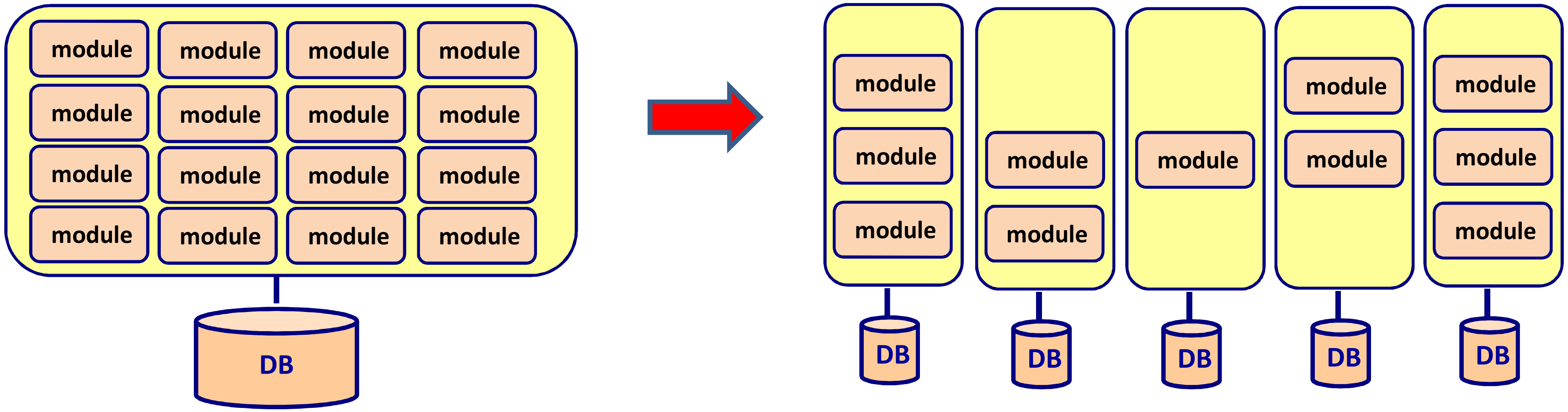

module module module module module
module module module module module
module module module module module
module module module module module
DB
module module module module module
module module module module module
module module module module module
module module module module module
DB
DB
module module module module module module
module module module module module module
module module module module module module
module module module module module module
DB
DB
DB

Monolith -> SOA -> Microservice strategy

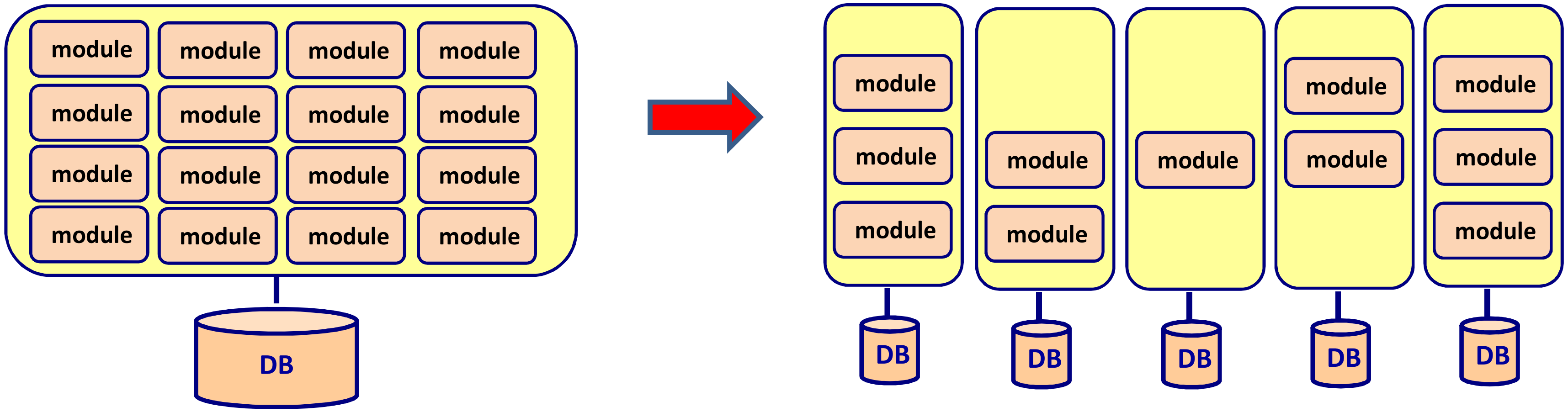


SUMMARY

Why microservices?

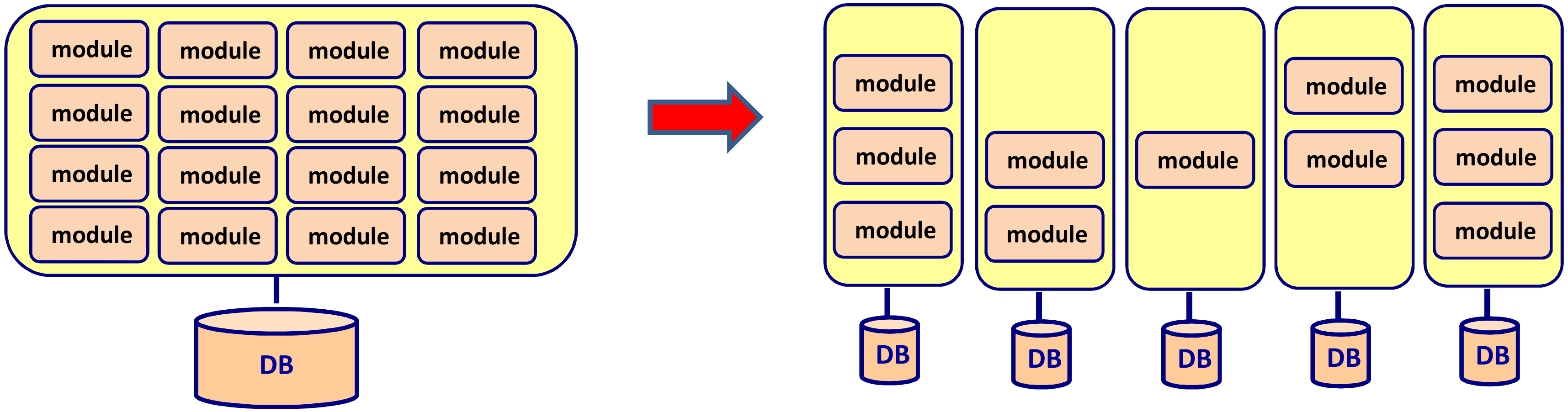


Agility



- Difficult to respond to change
  - One change effects the whole application
- Much easier to respond to change
  - One change effects only one microservice

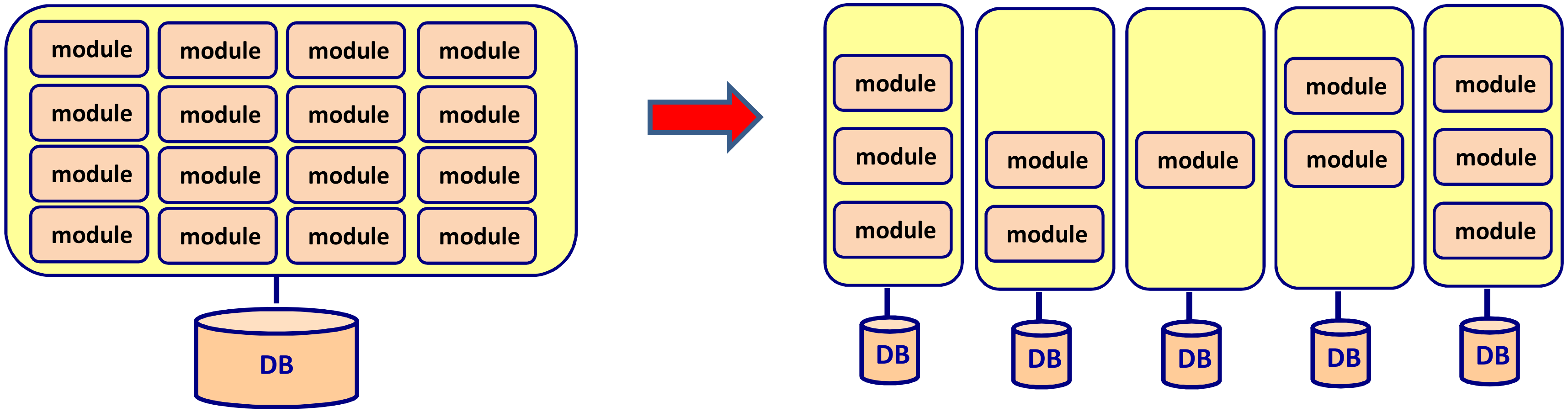
Testability



- Ease of testing
  - Big applications are often harder to test
- Completeness of testing
  - If we make one change, the whole application needs to be tested

- Ease of testing
  - Smaller services are often easier to test
- Completeness of testing
  - If we make one change, only that service needs to be tested
  - Scope is reduced

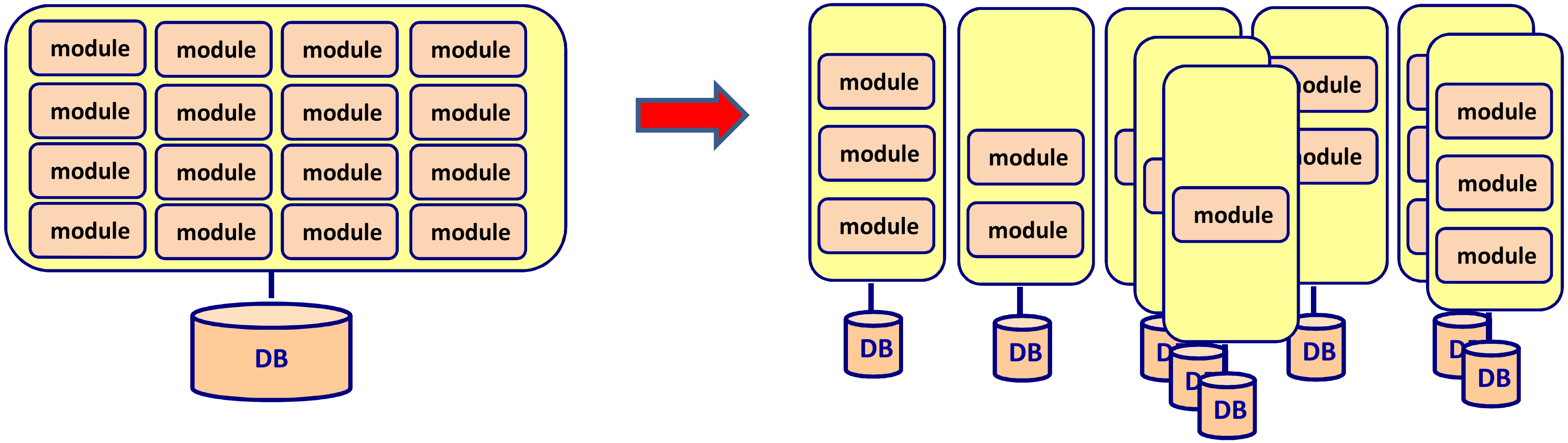
Deployability



- Ease of deployment
  - Requires a lot of ceremony
- Risk of deployment
  - Every deployment is of high risk
  - I cannot deploy very frequently

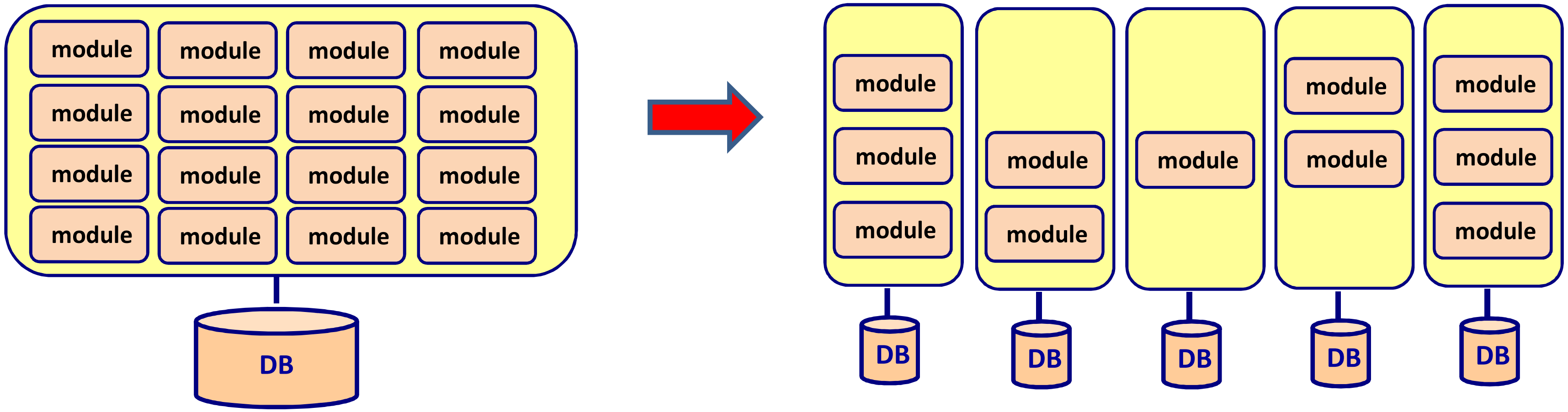
- Ease of deployment
  - Requires less ceremony
- Risk of deployment
  - Every deployment is of much lower risk
  - I can deploy very frequently

Scalability



- You can only scale the whole application
- You can scale up individual portions of the system

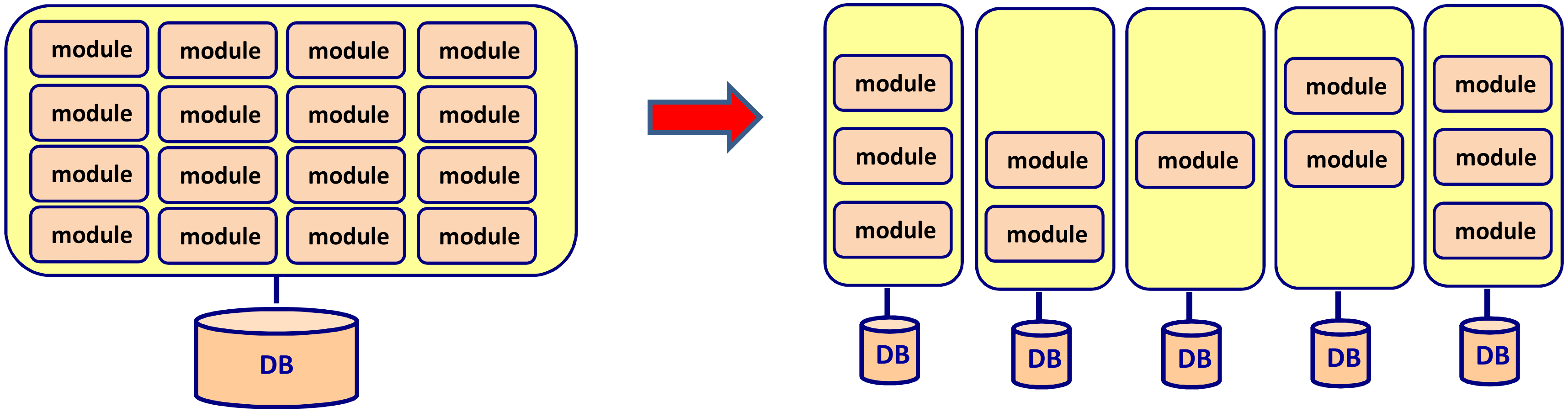
Availability

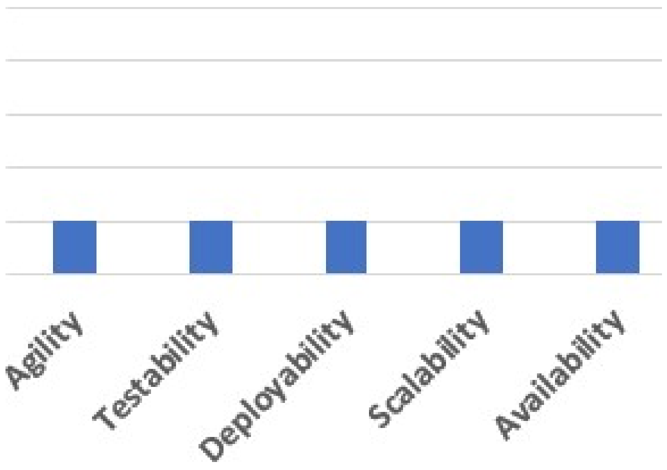


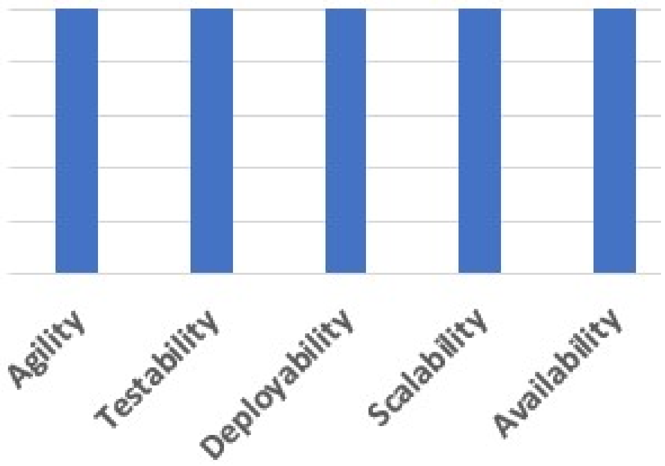
- Fault tolerance
  - A fault impact the whole application
- Application availability
  - Mean time between recovery
    - Usually measured in minutes

- Fault tolerance
  - A fault impact only one service
- Application availability
  - Mean time between failure
    - Usually measured in seconds

Why microservices?







Microservice advantages

- Support for polyglot architecture
- Enabling experimentation and innovation
- Elastically and selectively scalable
- Allowing substitution
- Enabling to build organic systems
- Help reducing technology dept
- Allowing the coexistence of different versions
- Enabling scrum and devops

CONFIG SERVICE

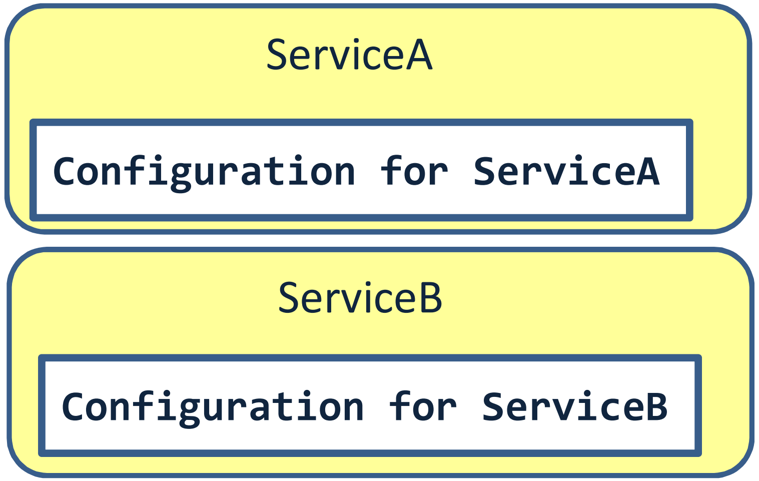
Configuration in microservices

- Remove settings from code
- Change runtime behavior
- Enforce consistency across elastic services

Configuration challenges

- Can fall out of sync
- Changes may enforce a restart
- May contain sensitive information
- Inconsistent usage across teams

Local configuration



ServiceA

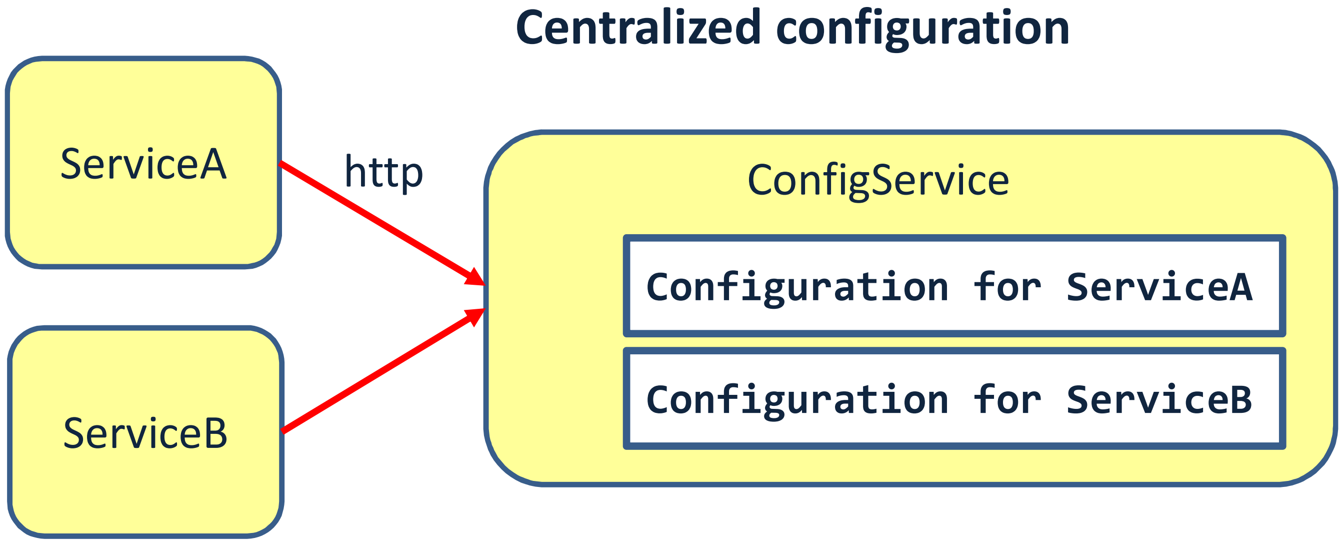
`Configuration for ServiceA`

ServiceB

`Configuration for ServiceB`

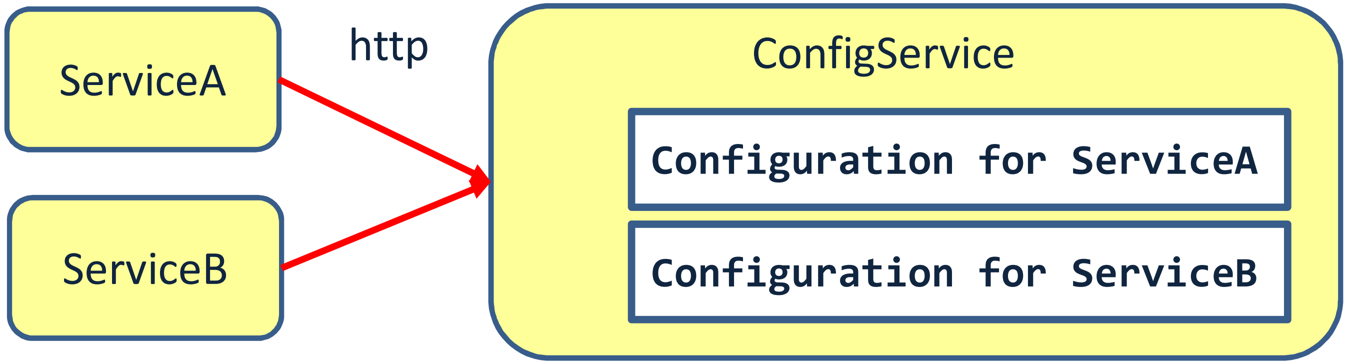
Spring cloud config

- HTTP access to centralized configuration


Centralized configuration
ServiceA
http
ServiceB
ConfigService
Configuration for ServiceA
Configuration for ServiceB

Spring cloud config

- File based configuration



ServiceA

ServiceB

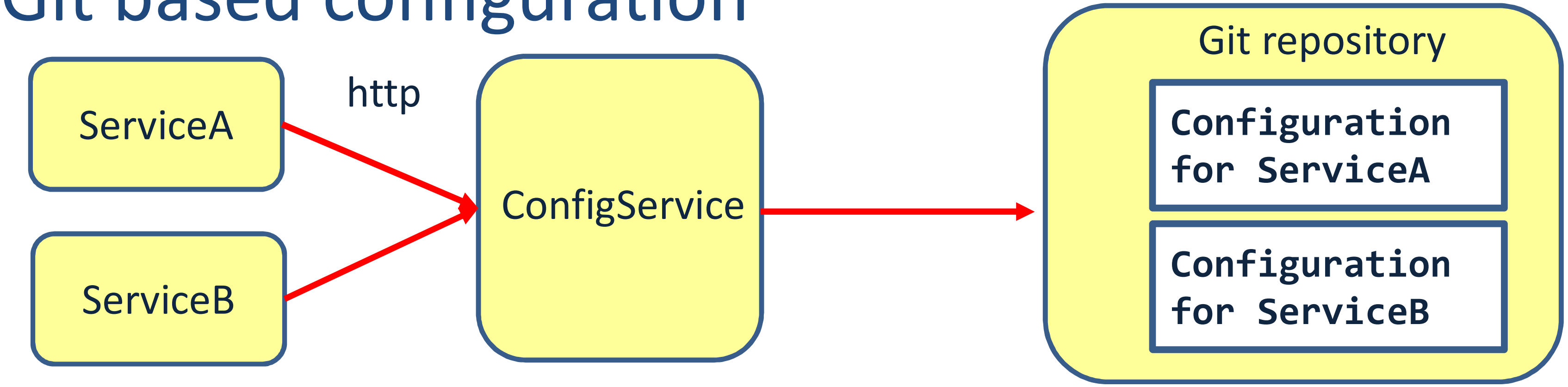
http

ConfigService

Configuration for ServiceA

Configuration for ServiceB

- Git based configuration



ServiceA

ServiceB

http

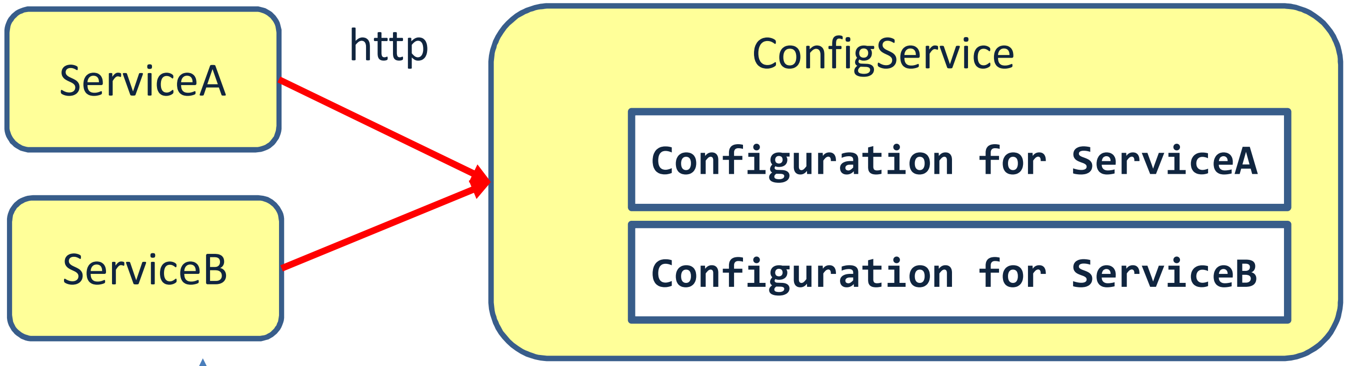
ConfigService

Git repository

Configuration for ServiceA

Configuration for ServiceB

Spring cloud config example



ServiceA

ServiceB

http

ConfigService

Configuration for ServiceA

Configuration for ServiceB

pom.xml

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

pom.xml

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Configuration server

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication
@EnableConfigServer
public class ConfigServiceApplication {

  public static void main(String[] args) {
    SpringApplication.run(ConfigServiceApplication.class, args);
  }
}
```

Do not use GIT, but local files

application.properties

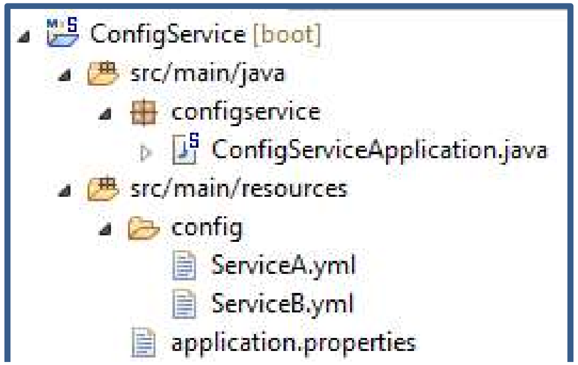
```
spring.profiles.active=native
server.port=8888
```

config/ServiceA.yml

```
greeting: Hello from Service A
```

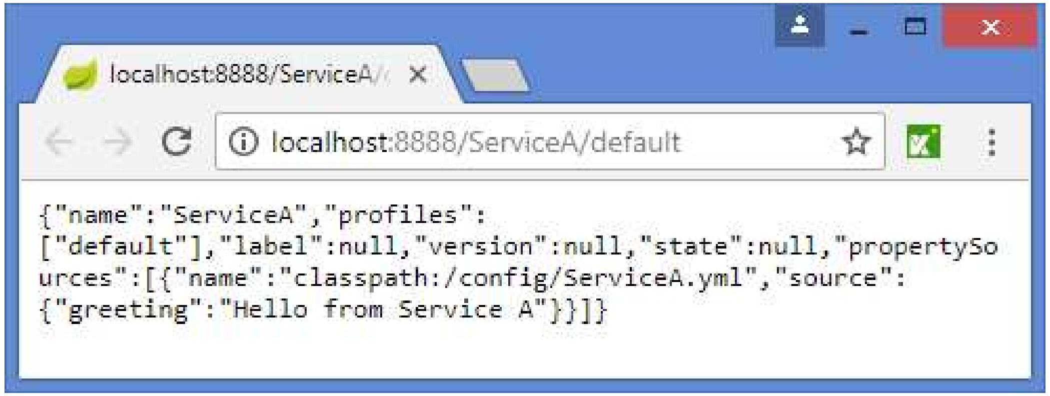
config/ServiceB.yml

```
greeting: Hello from Service B
```

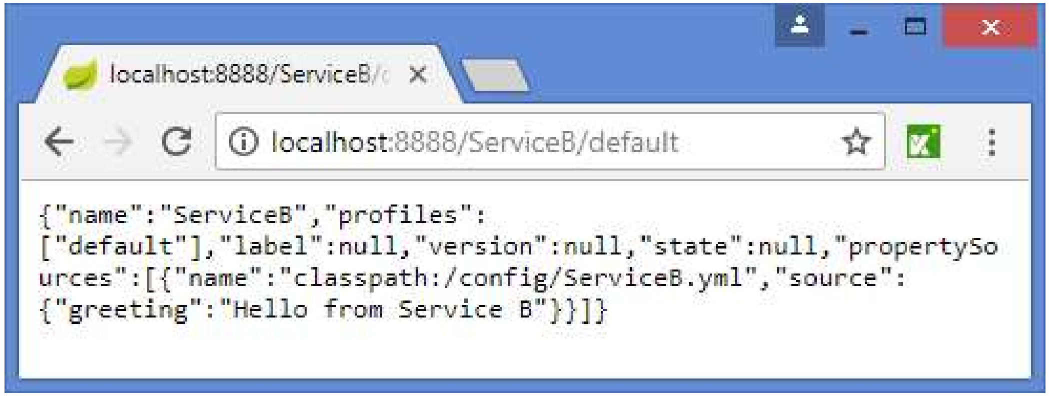


```
ConfigService [boot]
  src/main/java
    configservice
      ConfigServiceApplication.java
  src/main/resources
    config
      ServiceA.yml
      ServiceB.yml
    application.properties
```

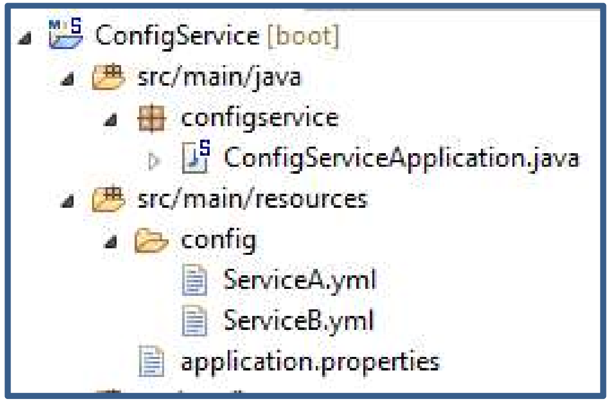
Configuration server



```
{"name":"ServiceA","profiles":
["default"],"label":null,"version":null,"state":null,"propertySo
urces":[{"name":"classpath:/config/ServiceA.yml","source":
{"greeting":"Hello from Service A"}}]}
```



```
{"name":"ServiceB","profiles":
["default"],"label":null,"version":null,"state":null,"propertySo
urces":[{"name":"classpath:/config/ServiceB.yml","source":
{"greeting":"Hello from Service B"}}]}
```



```
ConfigService [boot]
  src/main/java
    configservice
      ConfigServiceApplication.java
  src/main/resources
    config
      ServiceA.yml
      ServiceB.yml
    application.properties
```

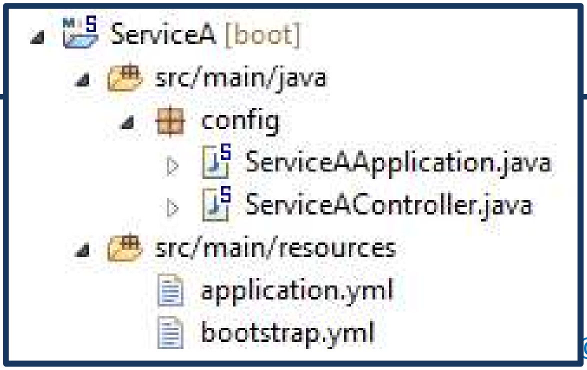
Configuration client: ServiceA

```
@SpringBootApplication
public class ServiceAApplication {

  public static void main(String[] args) {
    SpringApplication.run(ServiceAApplication.class, args);
  }
}
```

```
@RestController
public class ServiceAController {
  @Value("${greeting}")
  private String message;

  @RequestMapping("/")
  public String getName() {
    return message;
  }
}
```



```
ServiceA [boot]
  src/main/java
    config
      ServiceAApplication.java
      ServiceAController.java
  src/main/resources
    application.yml
    bootstrap.yml
```

application.yml

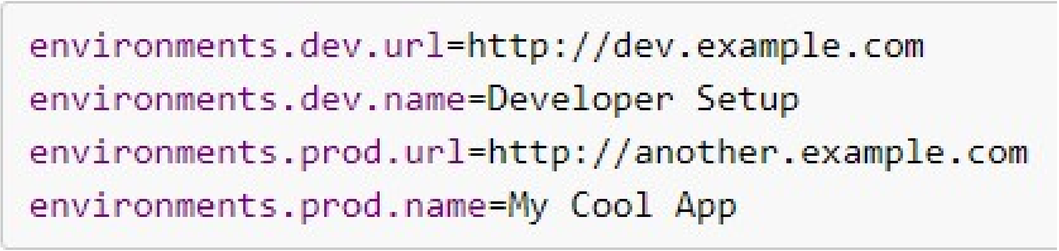
```
server:
  port: 8090
```

bootstrap.yml

```
spring:
  application:
    name: ServiceA
  cloud:
    config:
      url: http://localhost:8888
```

YAML vs. properties file

Properties file

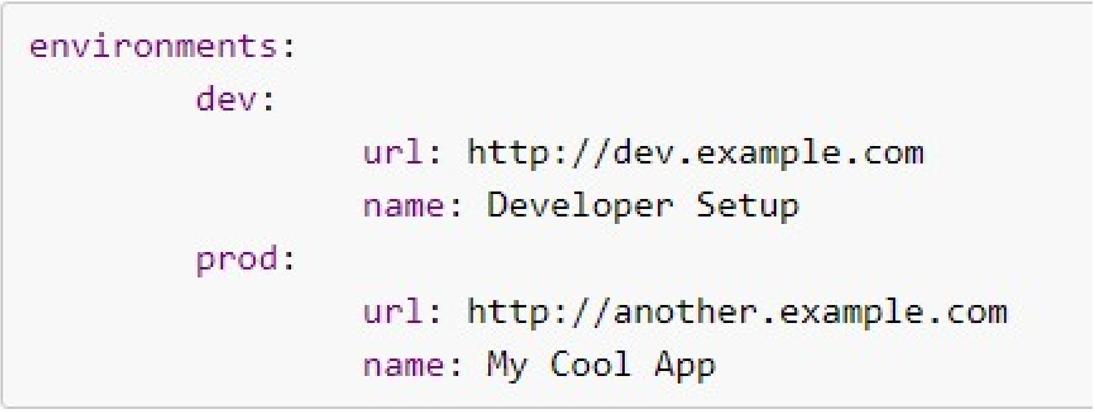


```
environments.dev.url=http://dev.example.com
environments.dev.name=Developer Setup
environments.prod.url=http://another.example.com
environments.prod.name=My Cool App
```

Extension is .properties

Not hierachical

YAML file



```
environments:
        dev:
                url: http://dev.example.com
                name: Developer Setup
        prod:
                url: http://another.example.com
                name: My Cool App
```

Extension is .yml

Hierarchical

Spring cloud applications

- 2 configuration files
  - bootstrap.yml
    - Is loaded before applications.yml
    - Is needed when configuration is stored on a remote config server
    - Contains
      - The name of the application
      - Location of the configuration server
  - applications.yml
    - Contains standard application configuration

bootstrap.yml

```
spring:
  application:
    name: ServiceA
  cloud:
    config:
      url: http://localhost:8888
```

application.yml

```
server:
  port: 8090
```

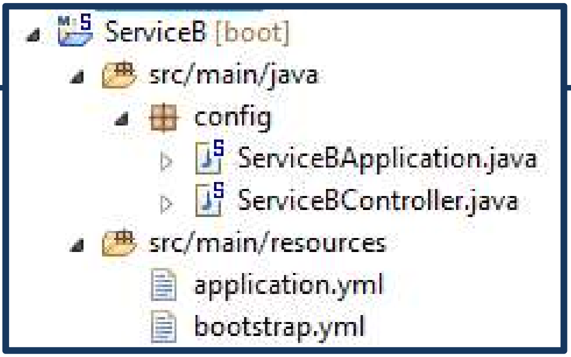
Configuration client: ServiceB

```java
@SpringBootApplication
public class ServiceBApplication {

  public static void main(String[] args) {
    SpringApplication.run(ServiceBApplication.class, args);
  }
}
```

```java
@RestController
public class ServiceBController {
  @Value("${greeting}")
  private String message;

  @RequestMapping("/")
  public String getName() {
    return message;
  }
}
```



```
ServiceB [boot]
  src/main/java
    config
      ServiceBApplication.java
      ServiceBController.java
  src/main/resources
    application.yml
    bootstrap.yml
```

application.yml

```yaml
server:
  port: 8091
```

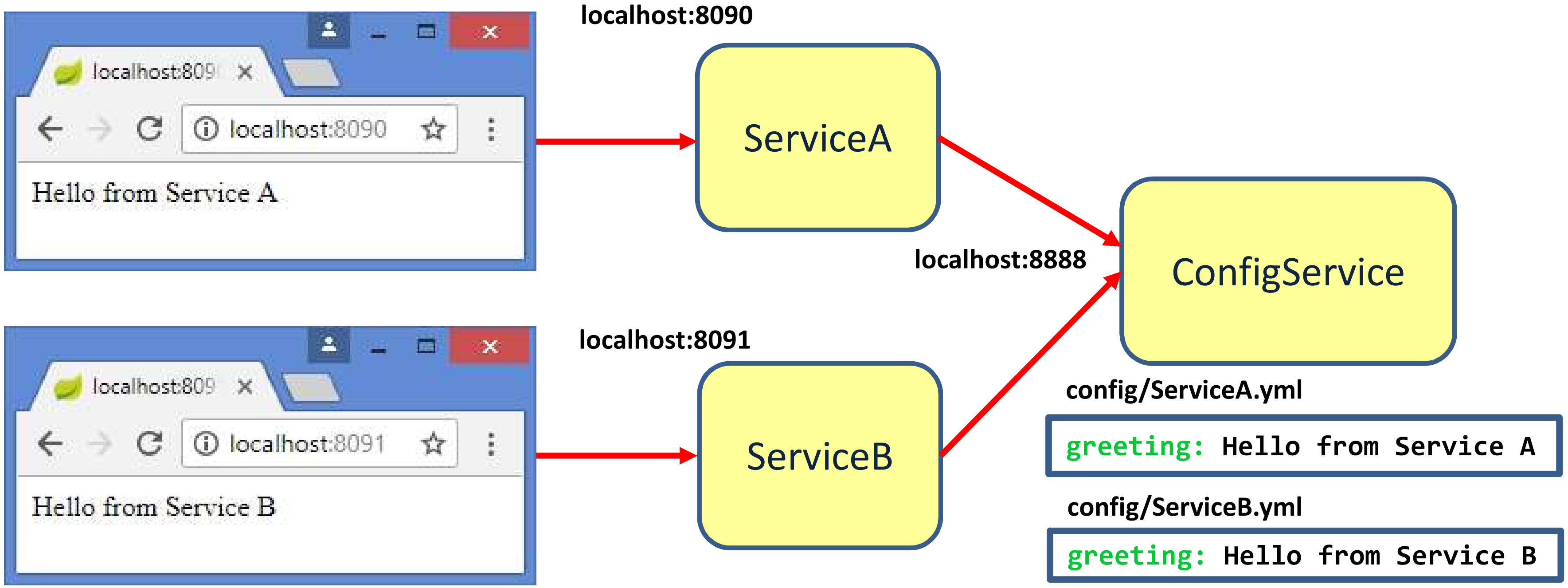
bootstrap.yml

```yaml
spring:
  application:
    name: ServiceB
  cloud:
    config:
      url: http://localhost:8888
```

Use of the Config Server

bootstrap.yml

```
spring:
  application:
    name: ServiceA
  cloud:
    config:
      url: http://localhost:8888
```



Hello from Service A

Hello from Service B

config/ServiceA.yml

```
greeting: Hello from Service A
```

config/ServiceB.yml

```
greeting: Hello from Service B
```

bootstrap.yml

```
spring:
  application:
    name: ServiceB
  cloud:
    config:
      url: http://localhost:8888
```