軟體工程
(Software Engineering)

微服務架構：
RESTful服務、服務部署
(Microservices Architecture: RESTful services, Service deployment)

1091SE08
MBA, IM, NTPU (M5118) (Fall 2020)
Tue 2, 3, 4 (9:10-12:00) (B8F40)

Min-Yuh Day
戴敏育
Associate Professor
副教授
Institute of Information Management, National Taipei University
國立臺北大學 資訊管理研究所
https://web.ntpu.edu.tw/~myday

2020-11-17

課程大綱 (Syllabus)

週次 (Week)	日期 (Date)	內容 (Subject/Topics)
1	2020/09/15	軟體工程概論 (Introduction to Software Engineering)
2	2020/09/22	軟體產品與專案管理：軟體產品管理，原型設計 (Software Products and Project Management: Software product management and prototyping)
3	2020/09/29	敏捷軟體工程：敏捷方法、Scrum、極限程式設計 (Agile Software Engineering: Agile methods, Scrum, and Extreme Programming)
4	2020/10/06	功能、場景和故事 (Features, Scenarios, and Stories)
5	2020/10/13	軟體架構：架構設計、系統分解、分散式架構 (Software Architecture: Architectural design, System decomposition, and Distribution architecture)
6	2020/10/20	軟體工程個案研究 I (Case Study on Software Engineering I)

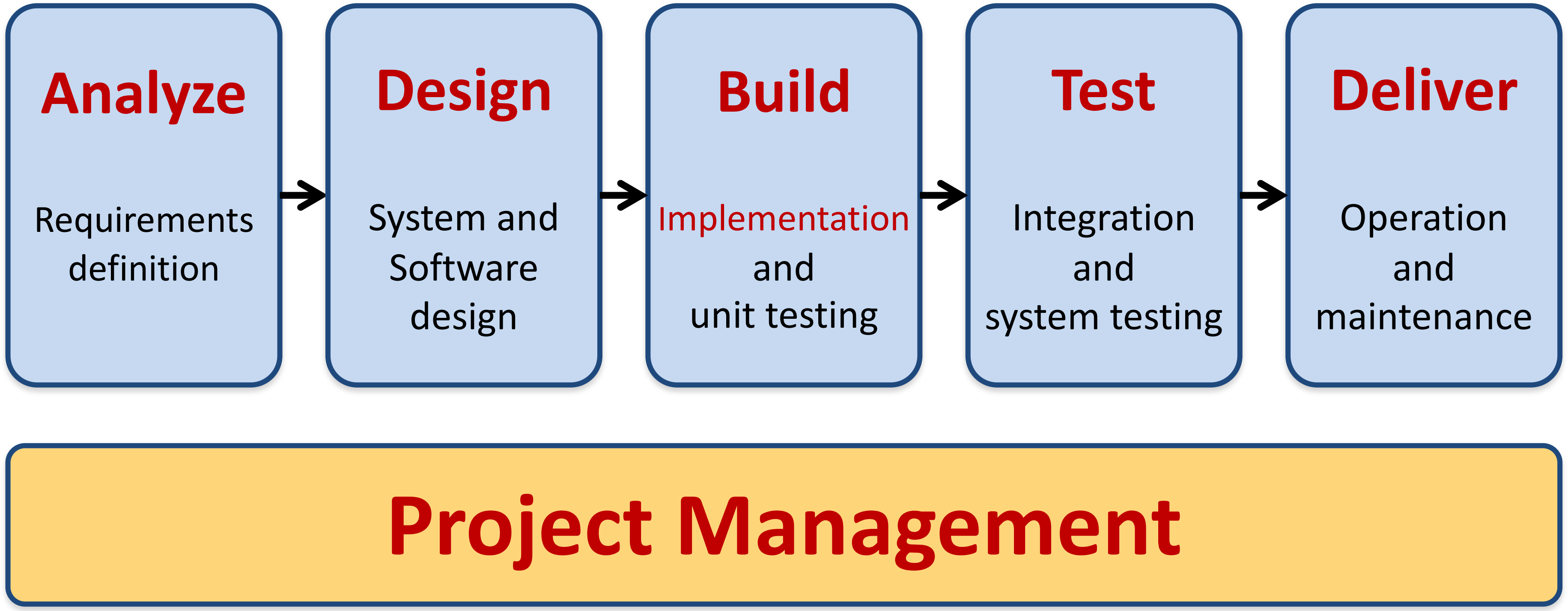
課程大綱 (Syllabus)

週次 (Week)	日期 (Date)	內容 (Subject/Topics)
7	2020/10/27	基於雲的軟體：虛擬化和容器、軟體即服務 (Cloud-Based Software: Virtualization and containers, Everything as a service, Software as a service)
8	2020/11/03	雲端運算與雲軟體架構 (Cloud Computing and Cloud Software Architecture)
9	2020/11/10	期中報告 (Midterm Project Report)
10	2020/11/17	微服務架構：RESTful服務、服務部署 (Microservices Architecture: RESTful services, Service deployment)
11	2020/11/24	軟體工程產業實務 (Industry Practices of Software Engineering)
12	2020/12/01	安全和隱私 (Security and Privacy)

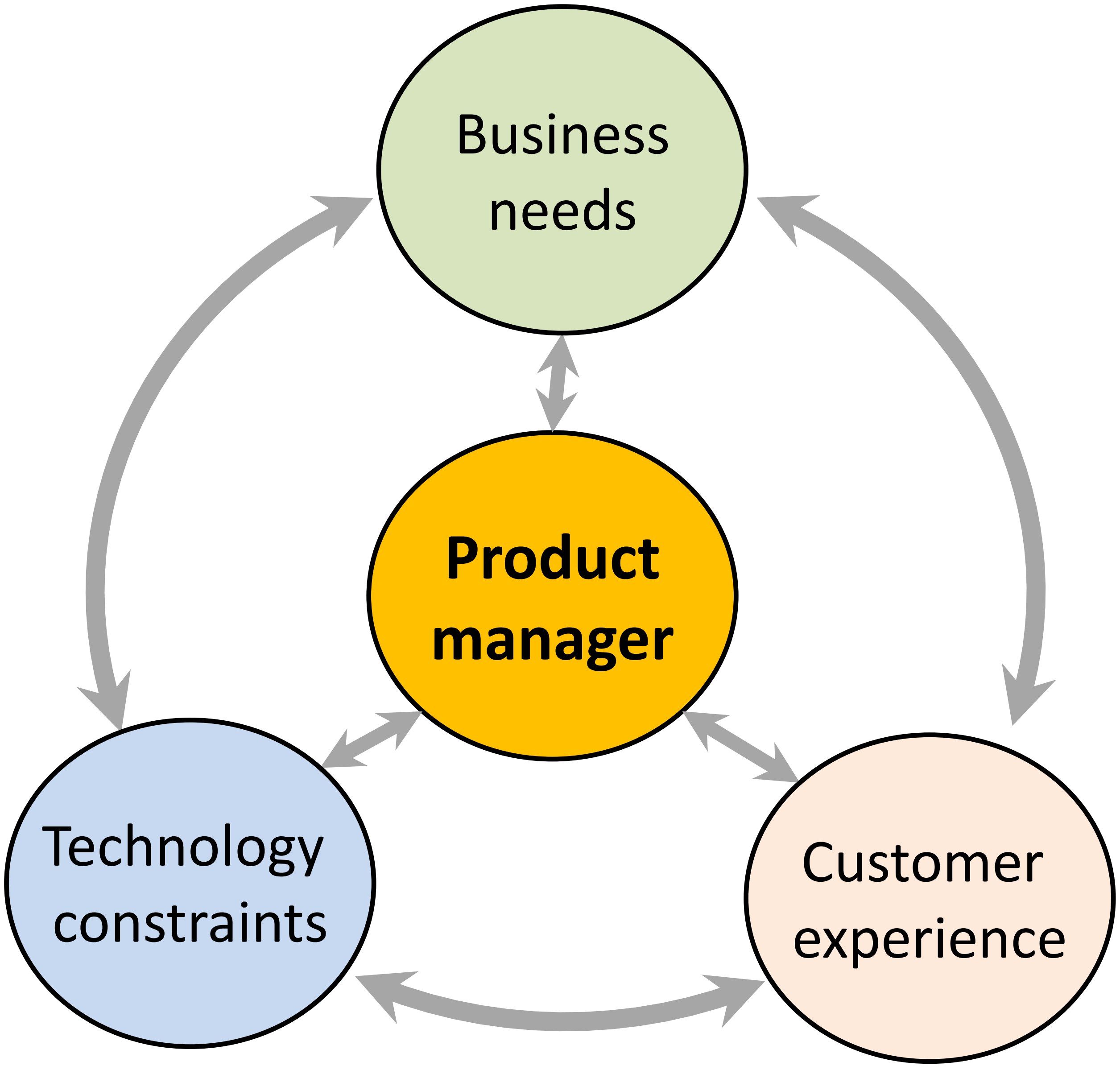
課程大綱 (Syllabus)

週次 (Week)	日期 (Date)	內容 (Subject/Topics)
13	2020/12/08	軟體工程個案研究 II (Case Study on Software Engineering II)
14	2020/12/15	可靠的程式設計 (Reliable Programming)
15	2020/12/22	測試：功能測試、測試自動化、測試驅動的開發、程式碼審查 (Testing: Functional testing, Test automation, Test-driven development, and Code reviews)
16	2020/12/29	DevOps和程式碼管理：程式碼管理和DevOps自動化 (DevOps and Code Management: Code management and DevOps automation)
17	2021/01/05	期末報告 I (Final Project Report I)
18	2021/01/12	期末報告 II (Final Project Report I)

Software Engineering and Project Management

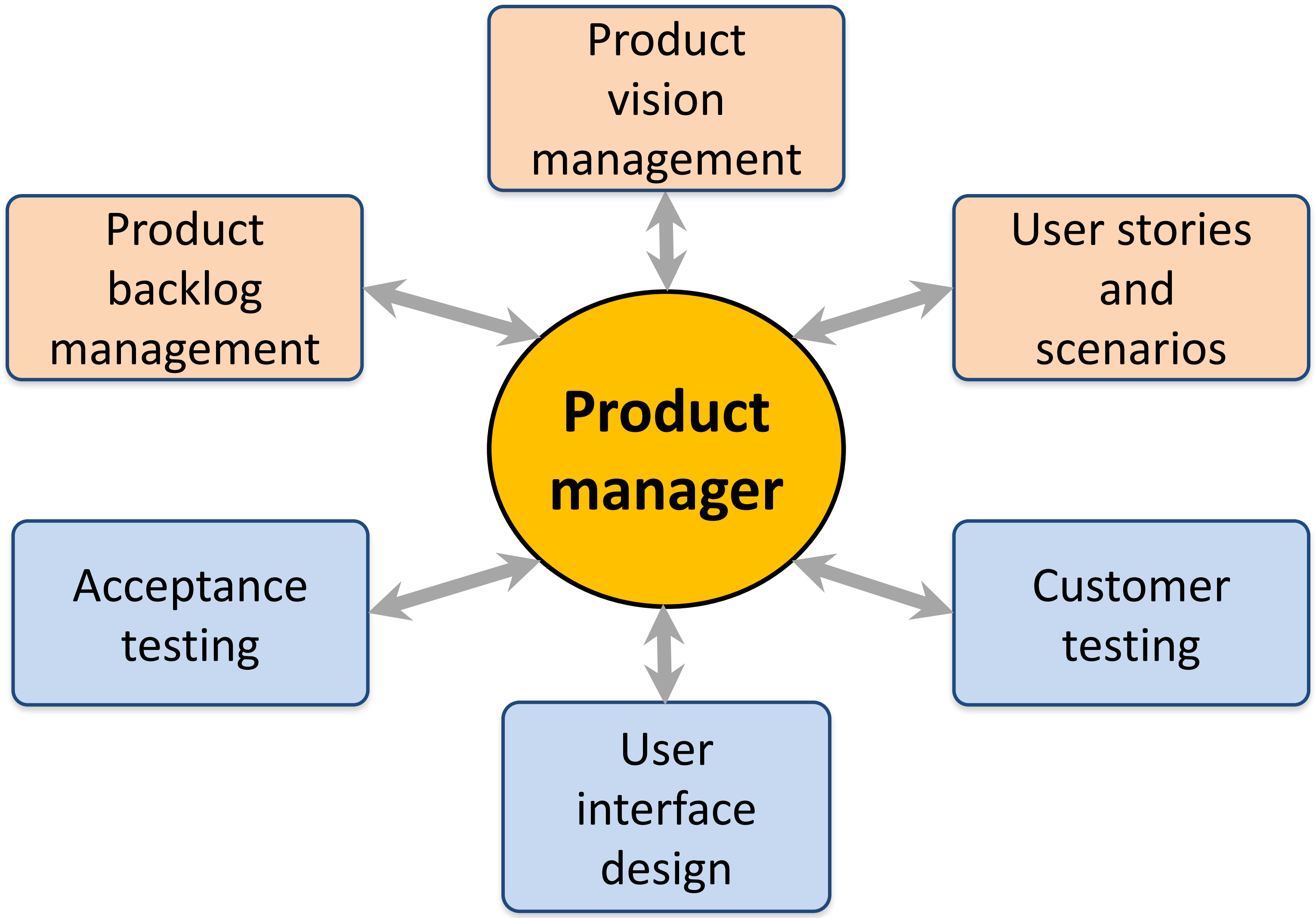

Analyze
Requirements definition
Design
System and Software design
Build
Implementation and unit testing
Test
Integration and system testing
Deliver
Operation and maintenance
Project Management

Product management concerns

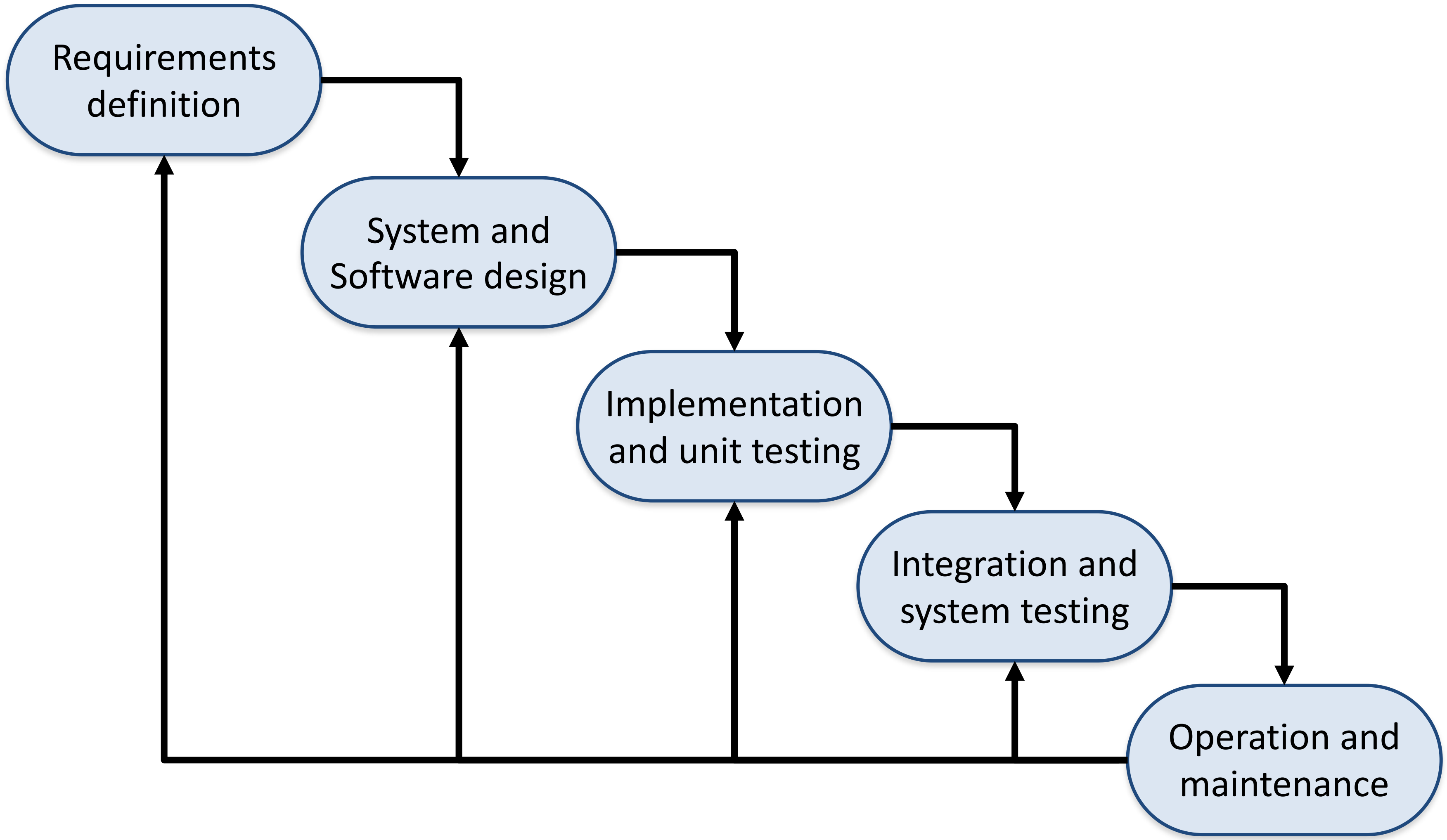


Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

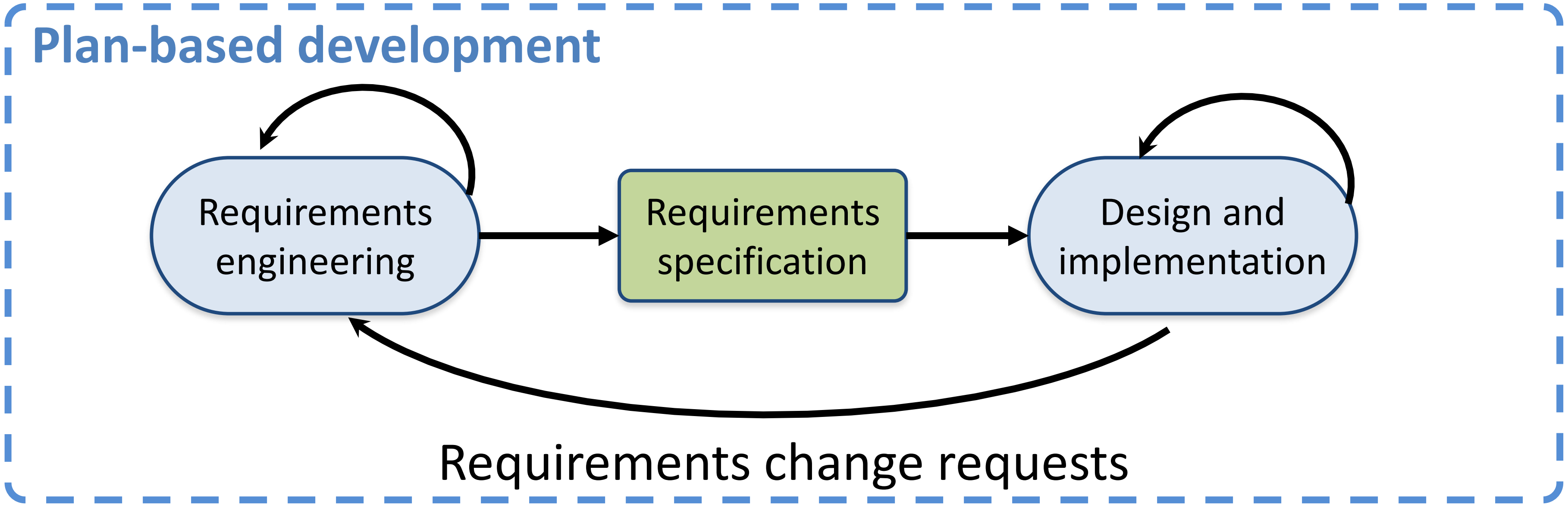
Technical interactions of product managers

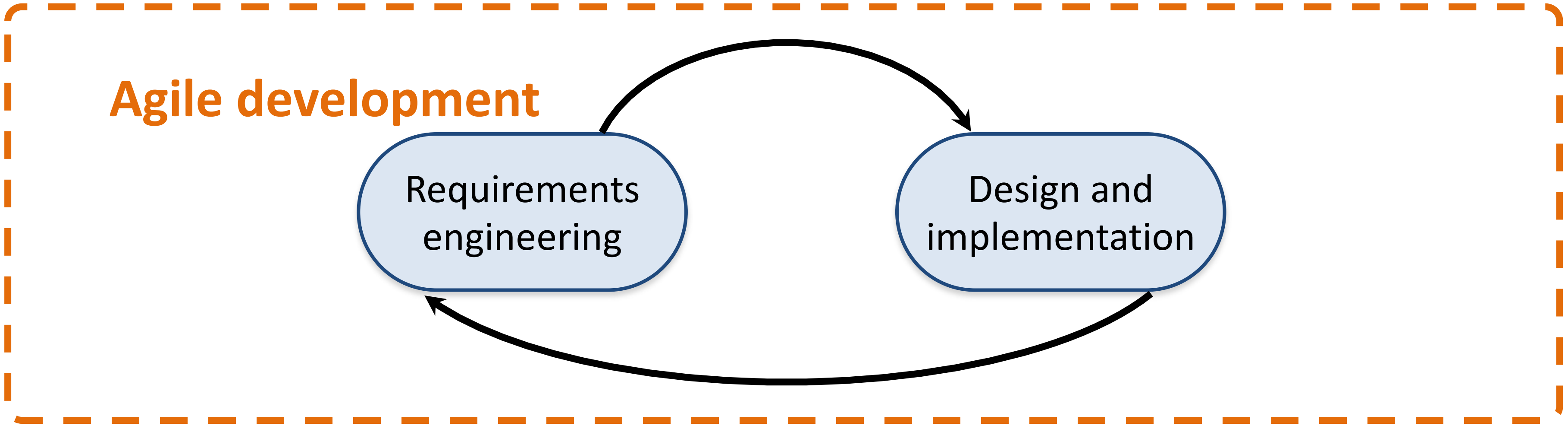

Product vision management
Product backlog management
User stories and scenarios
Product manager
Acceptance testing
Customer testing
User interface design

Software Development Life Cycle (SDLC)
The waterfall model

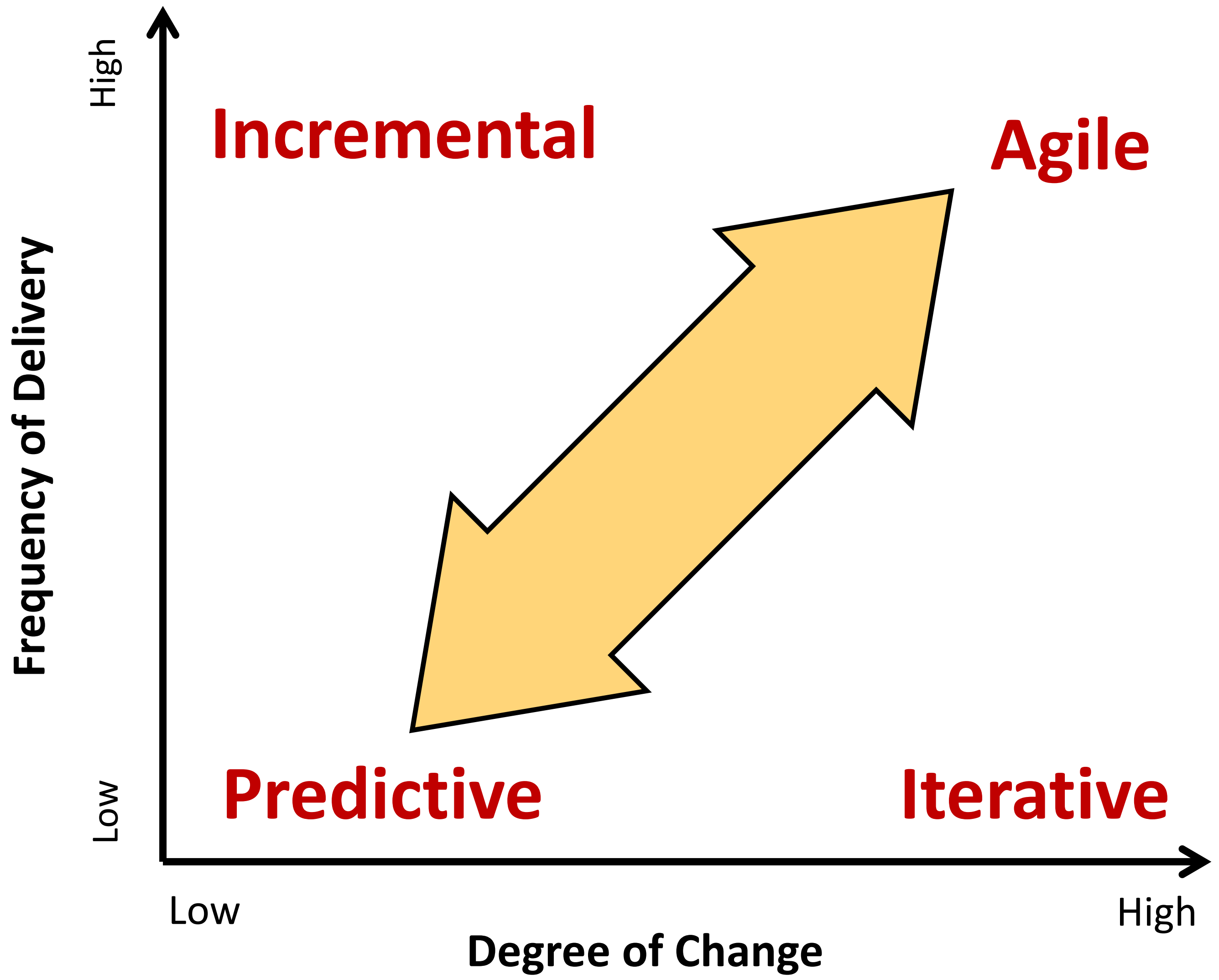

Requirements definition
System and Software design
Implementation and unit testing
Integration and system testing
Operation and maintenance

Plan-based and Agile development


Plan-based development
Requirements engineering
Requirements specification
Design and implementation
Requirements change requests

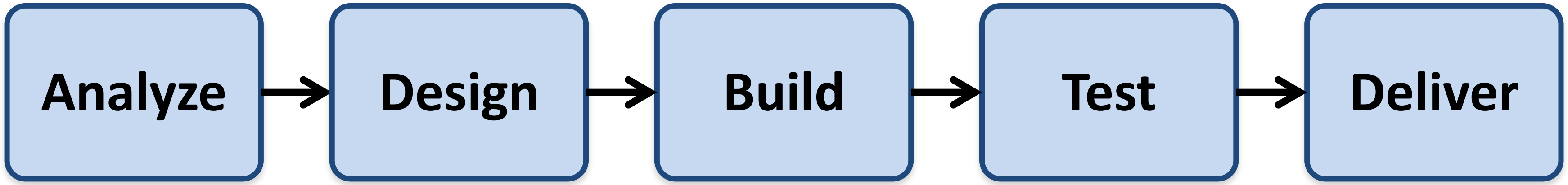


The Continuum of Life Cycles



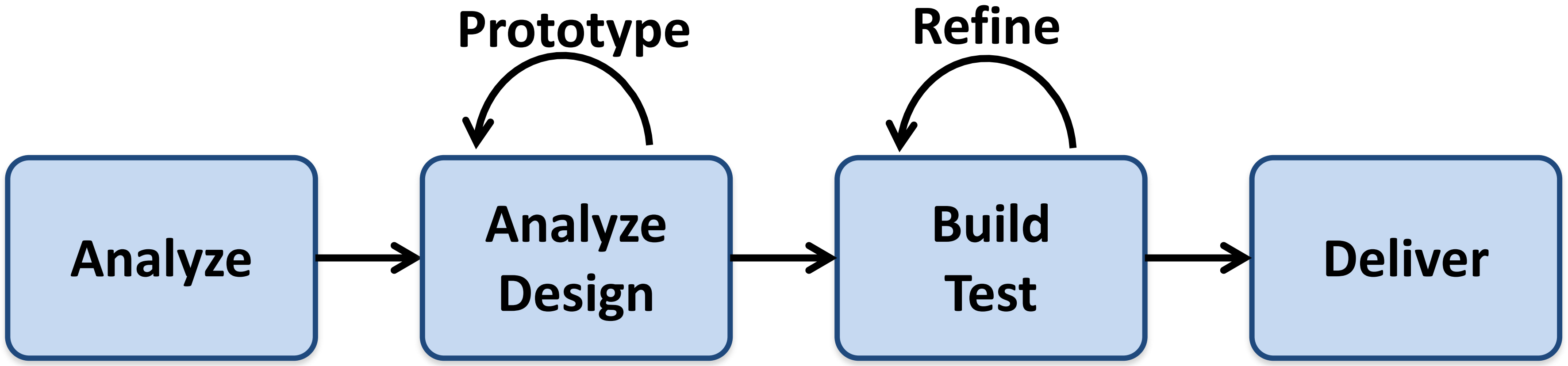
Source: Project Management Institute (2017), Agile Practice Guide, Project Management Institute

Predictive Life Cycle

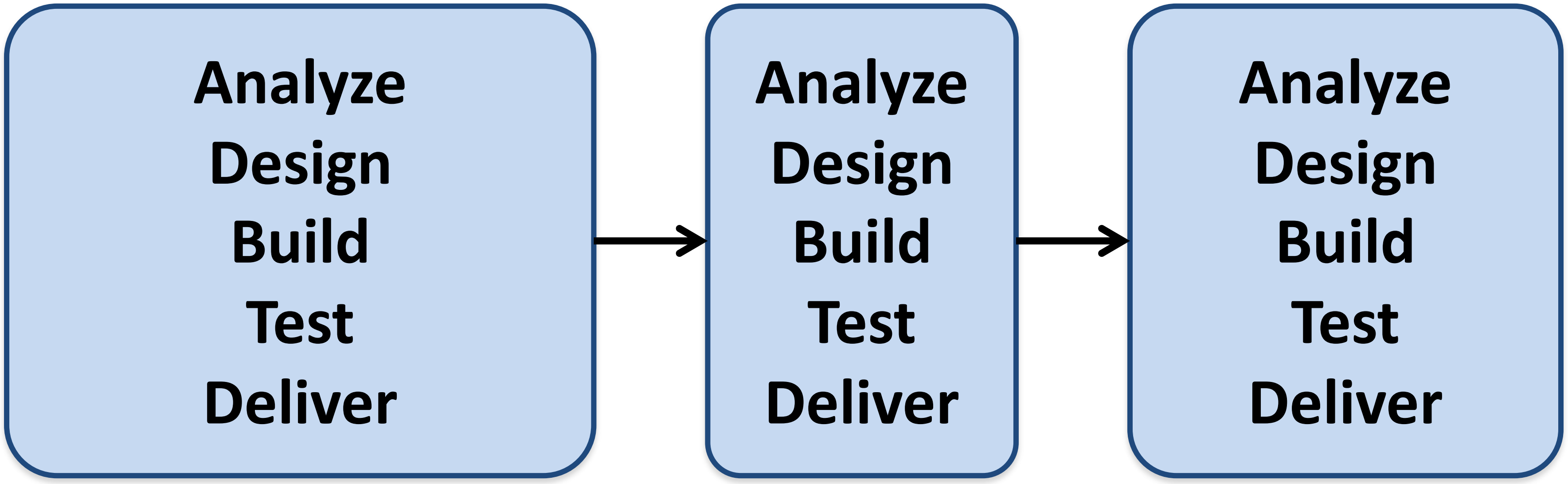

$$\begin{tikzcd}\text{Analyze} \arrow[r] & \text{Design} \arrow[r] & \text{Build} \arrow[r] & \text{Test} \arrow[r] & \text{Deliver}\end{tikzcd}$$

Source: Project Management Institute (2017), Agile Practice Guide, Project Management Institute

Iterative Life Cycle



A Life Cycle of Varying-Sized Increments



Iteration-Based and Flow-Based Agile Life Cycles

Iteration-Based Agile

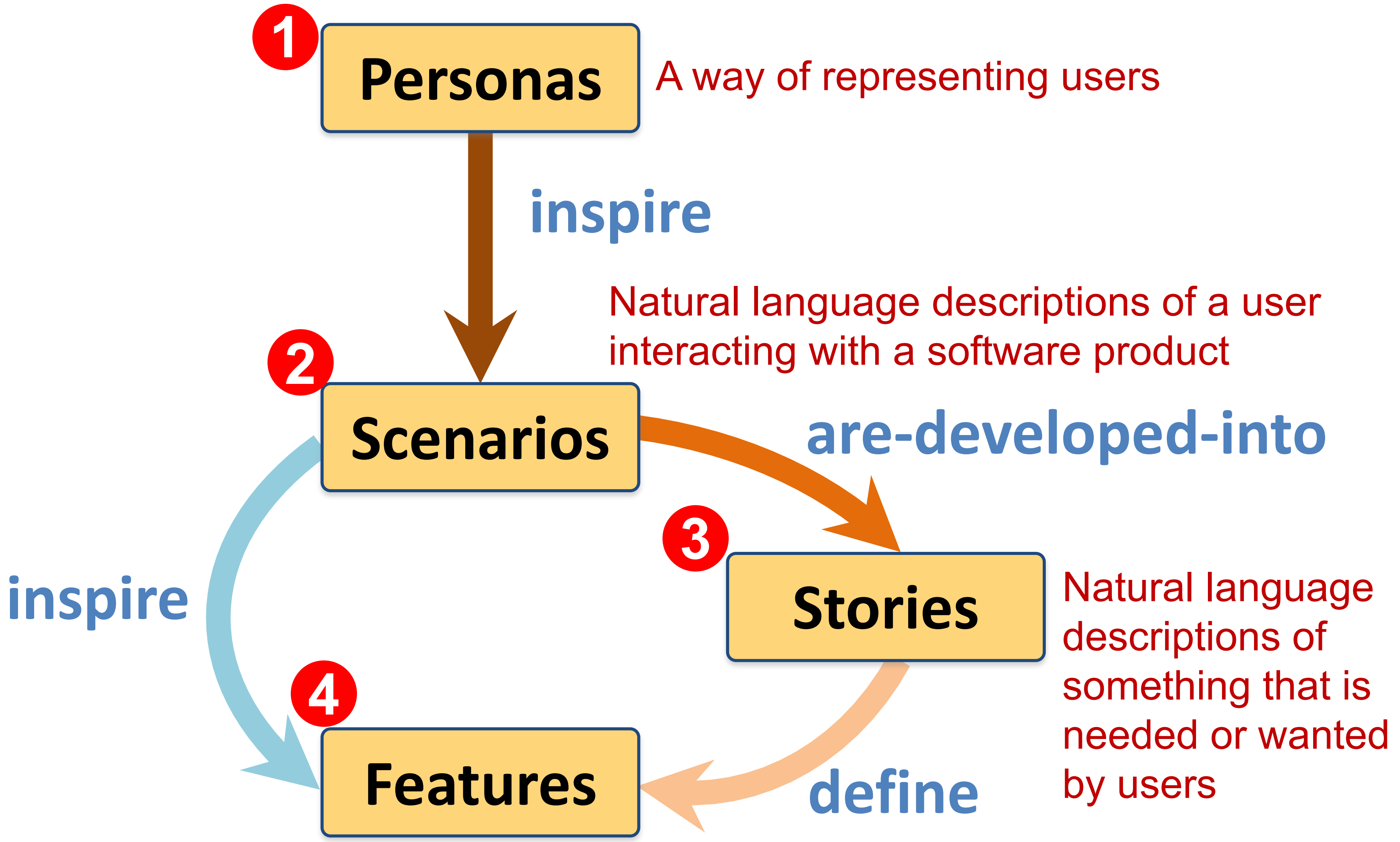
Requirements Analysis Design Build Test	Requirements Analysis Design Build Test	Requirements Analysis Design Build Test	Requirements Analysis Design Build Test	Repeat as needed …	Requirements Analysis Design Build Test	Requirements Analysis Design Build Test

Flow-Based Agile

Requirements Analysis Design Build Test the number of features in the WIP limit	Requirements Analysis Design Build Test the number of features in the WIP limit	Requirements Analysis Design Build Test the number of features in the WIP limit	Repeat as needed …	Requirements Analysis Design Build Test the number of features in the WIP limit	Requirements Analysis Design Build Test the number of features in the WIP limit

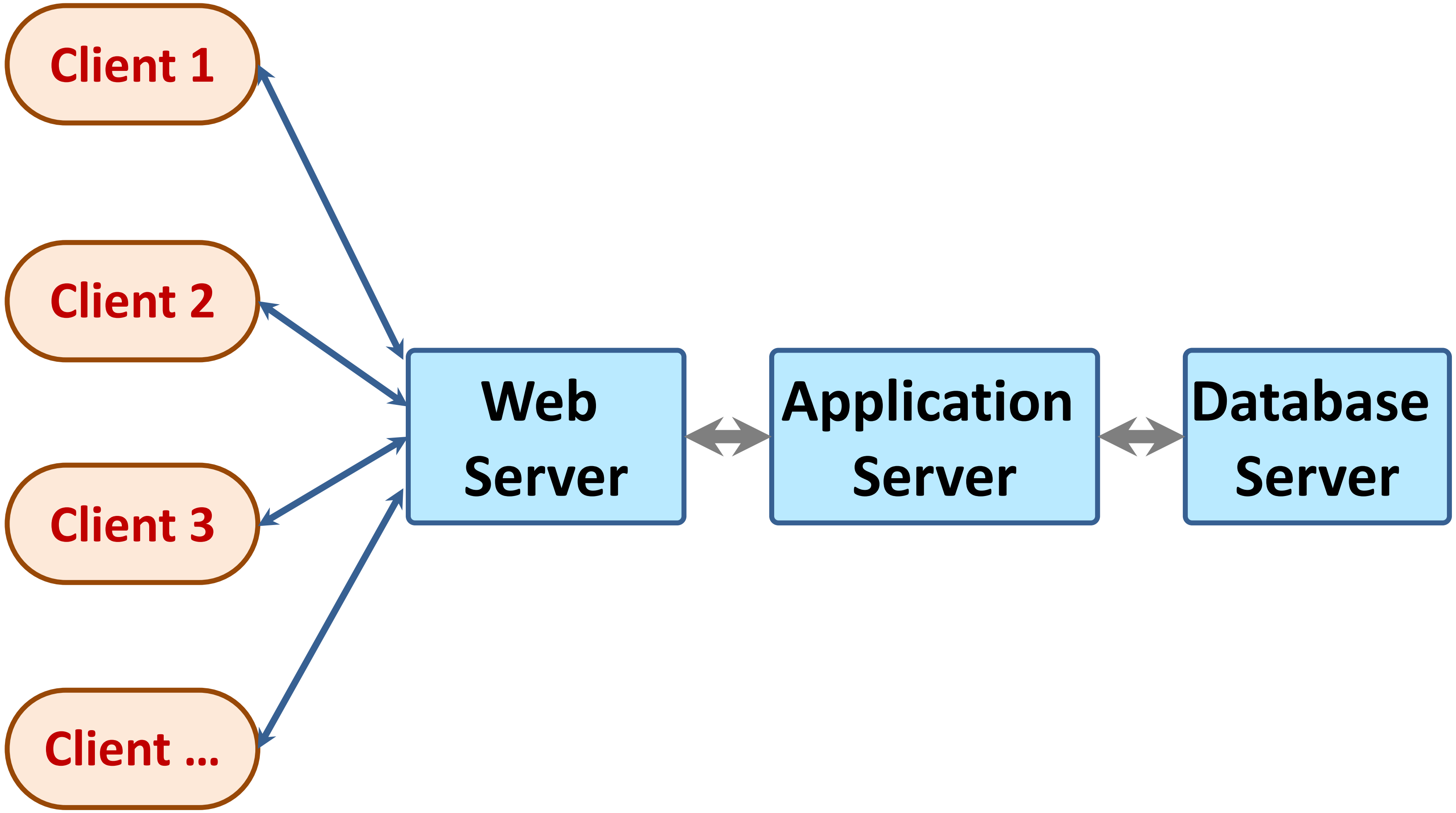
Source: Project Management Institute (2017), Agile Practice Guide, Project Management Institute

From personas to features

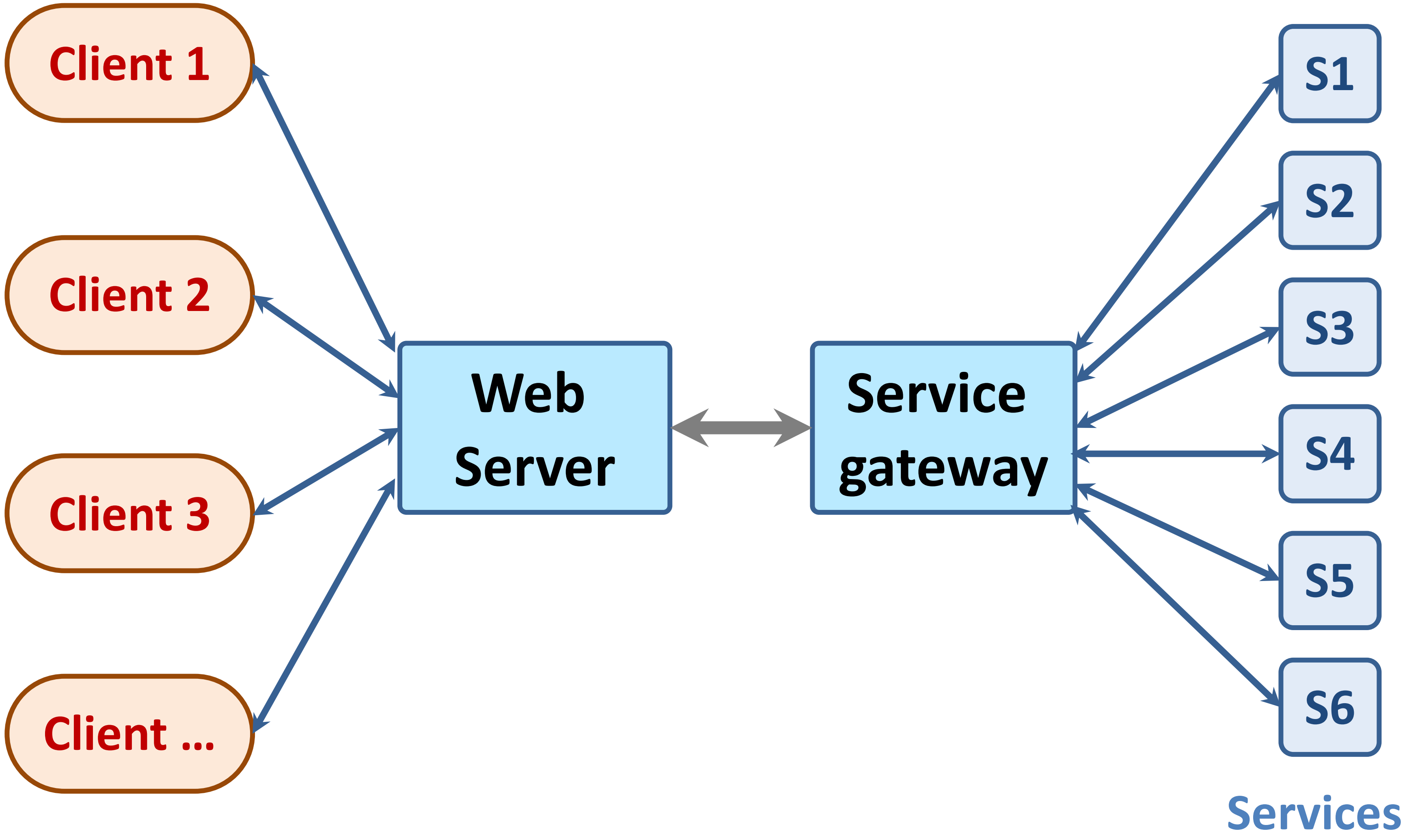

1
Personas
A way of representing users
inspire
Natural language descriptions of a user interacting with a software product
2
Scenarios
are-developed-into
3
Stories
Natural language descriptions of something that is needed or wanted by users
inspire
4
Features
define

Fragments of product functionality

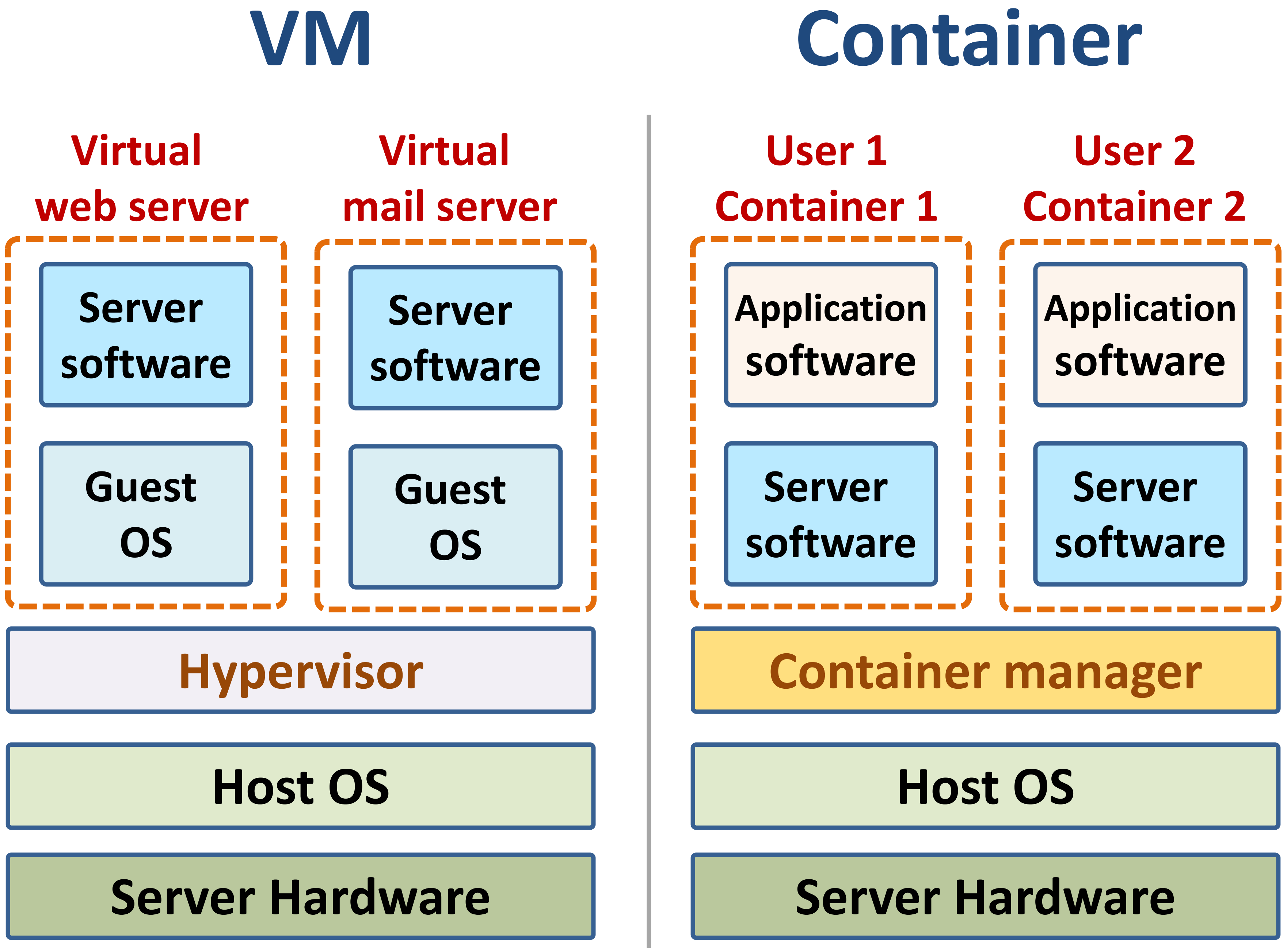
Multi-tier client-server architecture



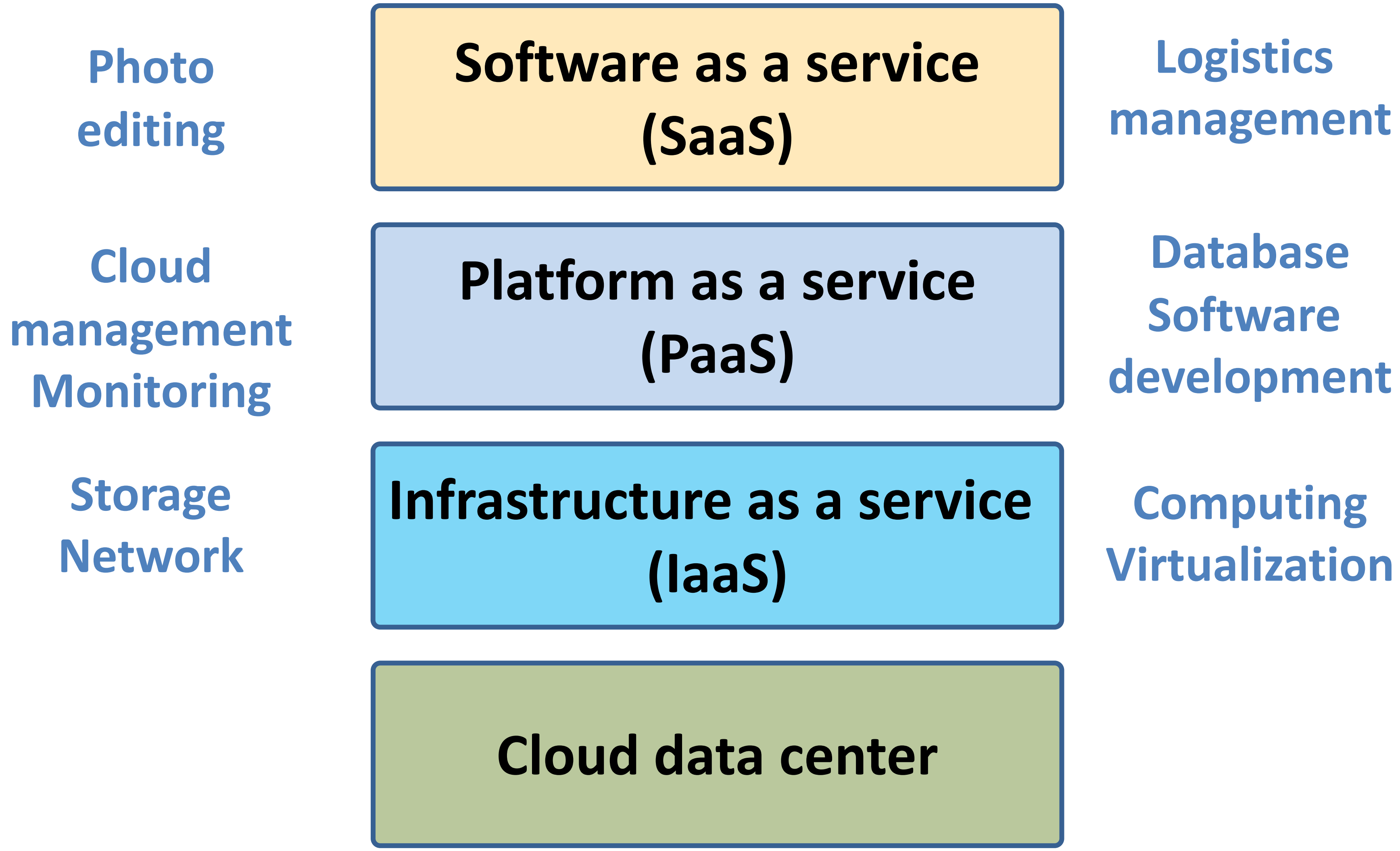
Service-oriented Architecture



Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

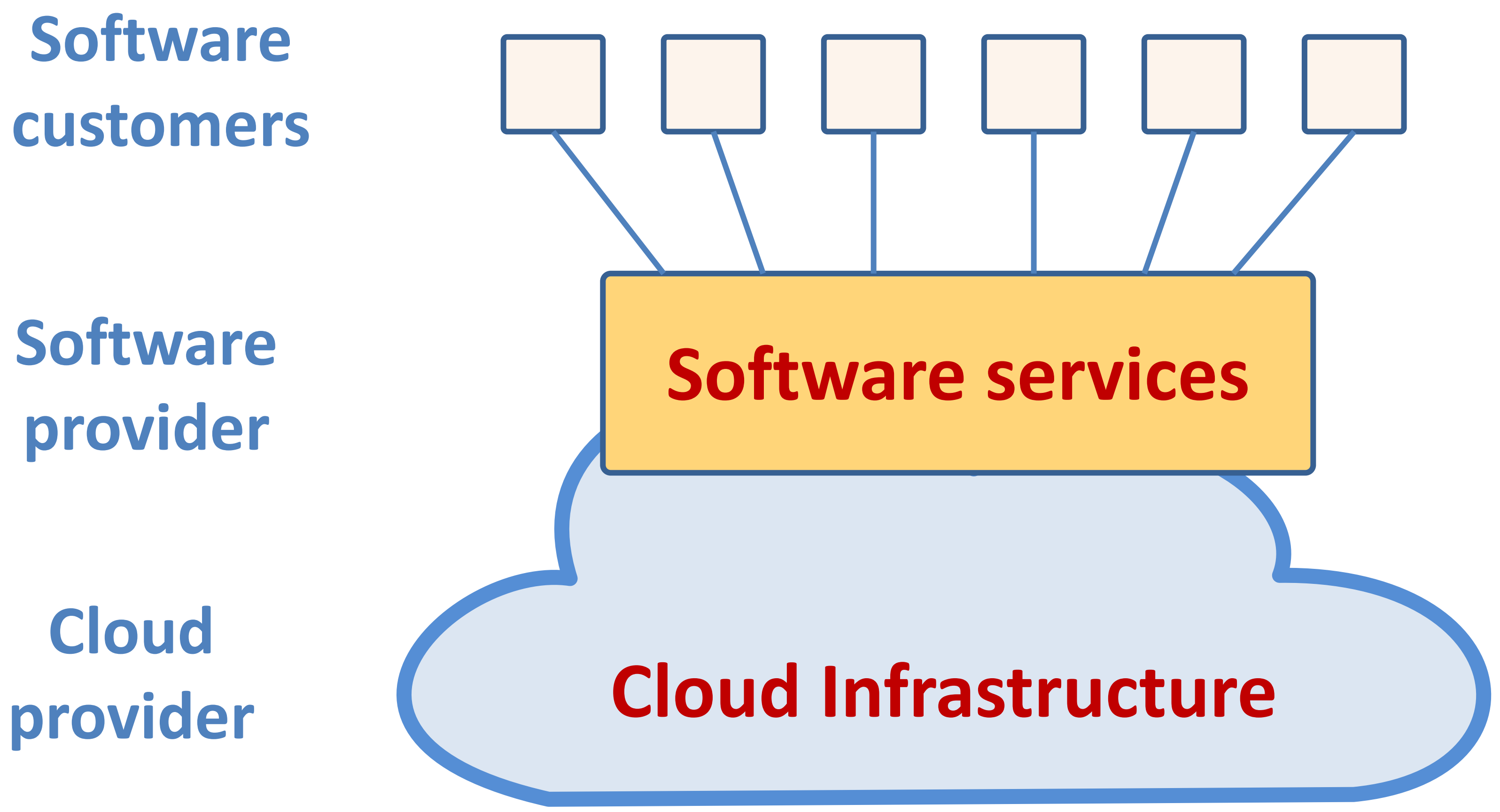

VM
Container
Virtual web server
Virtual mail server
Server software
Guest OS
Server software
Guest OS
Hypervisor
Host OS
Server Hardware
User 1 Container 1
User 2 Container 2
Application software
Server software
Application software
Server software
Container manager
Host OS
Server Hardware

Everything as a service



Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Software as a service



Microservices Architecture: RESTful services, Service deployment

Outline

- Microservices Architecture
- RESTful services
- Service deployment

Microservices architecture

Software services

- A **software service** is a software component that can be accessed from remote computers over the Internet. Given an input, a service produces a corresponding output, without side effects.
  - The service is accessed through its published interface and all details of the service implementation are hidden.
  - Services do not maintain any internal state. State information is either stored in a database or is maintained by the service requestor.

Software services

- When a service request is made, the state information may be included as part of the request and the updated state information is returned as part of the service result.
- As there is no local state, services can be dynamically reallocated from one virtual server to another and replicated across several servers.

Modern web services

- After various experiments in the 1990s with service-oriented computing, the idea of 'big' Web Services emerged in the early 2000s.
  - These were based on XML-based protocols and standards such as Simple Object Access Protocol (SOAP) for service interaction and Web Service Definition Language (WSDL) for interface description.
  - Most software services don't need the generality that's inherent in the design of web service protocols.
- Consequently, modern service-oriented systems, use simpler, 'lighter weight' service-interaction protocols that have lower overheads and, consequently, faster execution.

Microservices

- Microservices are small-scale, stateless, services that have a single responsibility. They are combined to create applications.
- They are completely independent with their own database and UI management code.
- Software products that use microservices have a microservices architecture.
  - Create cloud-based software products that are adaptable, scaleable and resilient.

A microservice example

- **System authentication**
  - User registration, where users provide information about their identity, security information, mobile (cell) phone number and email address.
  - Authentication using UID/password.
  - Two-factor authentication using code sent to mobile phone.
  - User information management e.g. change password or mobile phone number.
  - Reset forgotten password.

A microservice example

- **System authentication**
- Each of these features could be implemented as a separate service that uses a central shared database to hold authentication information.
- However, these features are too large to be microservices. To identify the microservices that might be used in the authentication system, you need to break down the coarse-grain features into more detailed functions.

Functional breakdown of authentication features

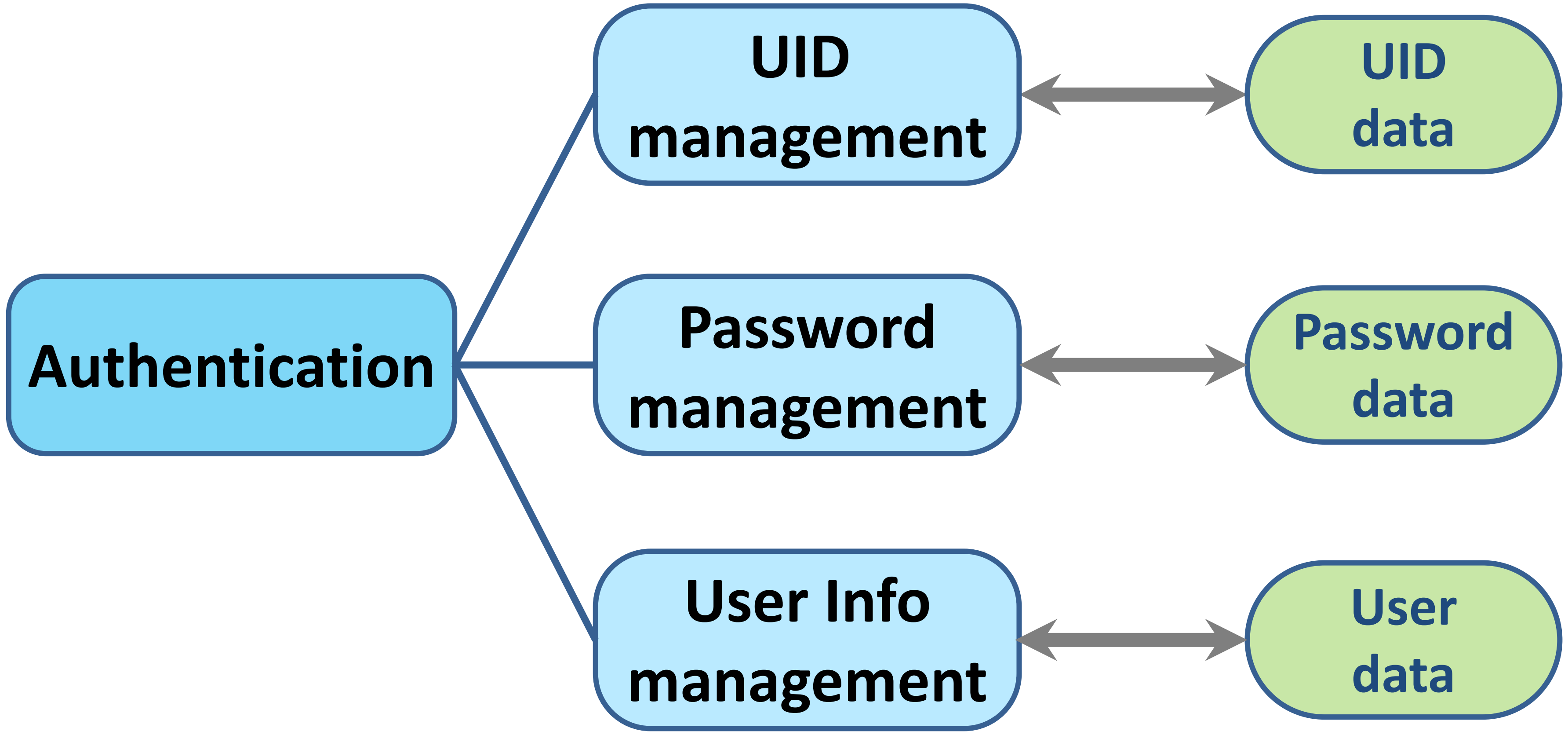
User registration

- Setup new login ID
- Setup new Password
- Setup Password recovery information
- Setup Two-factor authentication
- Confirm registration

Authentication using UID/Password

- Get login ID
- Get Password
- Check Credentials
- Confirm authentication

Authentication microservices



Characteristics of microservices

- **Self-contained**
  - Microservices do not have external dependencies. They manage their own data and implement their own user interface.
- **Lightweight**
  - Microservices communicate using lightweight protocols.
- **Implementation-independent**
  - Microservices may be implemented using different programming languages and may use different technologies in their implementation.
- **Independently deployable**
  - Each microservice runs in its own process and is independently deployable, using automated systems.
- **Business-oriented**
  - Microservices should implement business capabilities and needs, rather than simply provide a technical service.

Source: Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.

Microservice communication

- Microservices communicate by exchanging messages.
- A message that is sent between services includes some administrative information, a service request and the data required to deliver the requested service.
- Services return a response to service request messages.
  - An authentication service may send a message to a login service that includes the name input by the user.
  - The response may be a token associated with a valid user name or might be an error saying that there is no registered user.

Microservice characteristics

- A **well-designed microservice** should have high cohesion and low coupling.
  - **Cohesion** is a measure of the number of relationships that parts of a component have with each other.
    - **High cohesion** means that all of the parts that are needed to deliver the component’s functionality are included in the component.
  - **Coupling** is a measure of the number of relationships that one component has with other components in the system.
    - **Low coupling** means that components do not have many relationships with other components.

Microservice characteristics

- Each microservice should have a single responsibility i.e. it should do one thing only and it should do it well.
  - However, ‘one thing only’ is difficult to define in a way that’s applicable to all services.
  - Responsibility does not always mean a single, functional activity.

Password management functionality

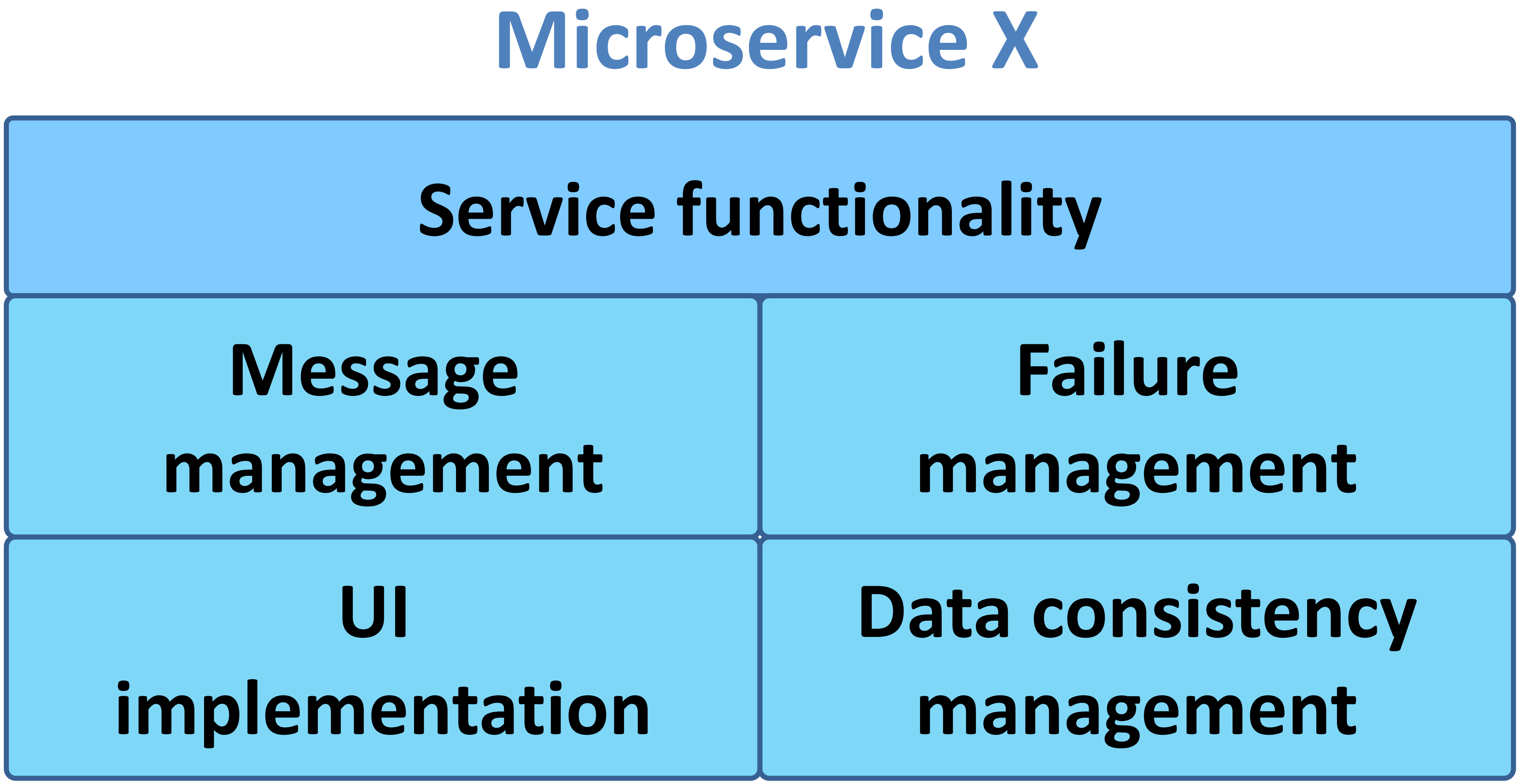
User functions

- Create password
- Change password
- Check password
- Recover password

Supporting functions

- Check password validity
- Delete password
- Backup password database
- Recover password database
- Check database integrity
- Repair database DB

Microservice support code



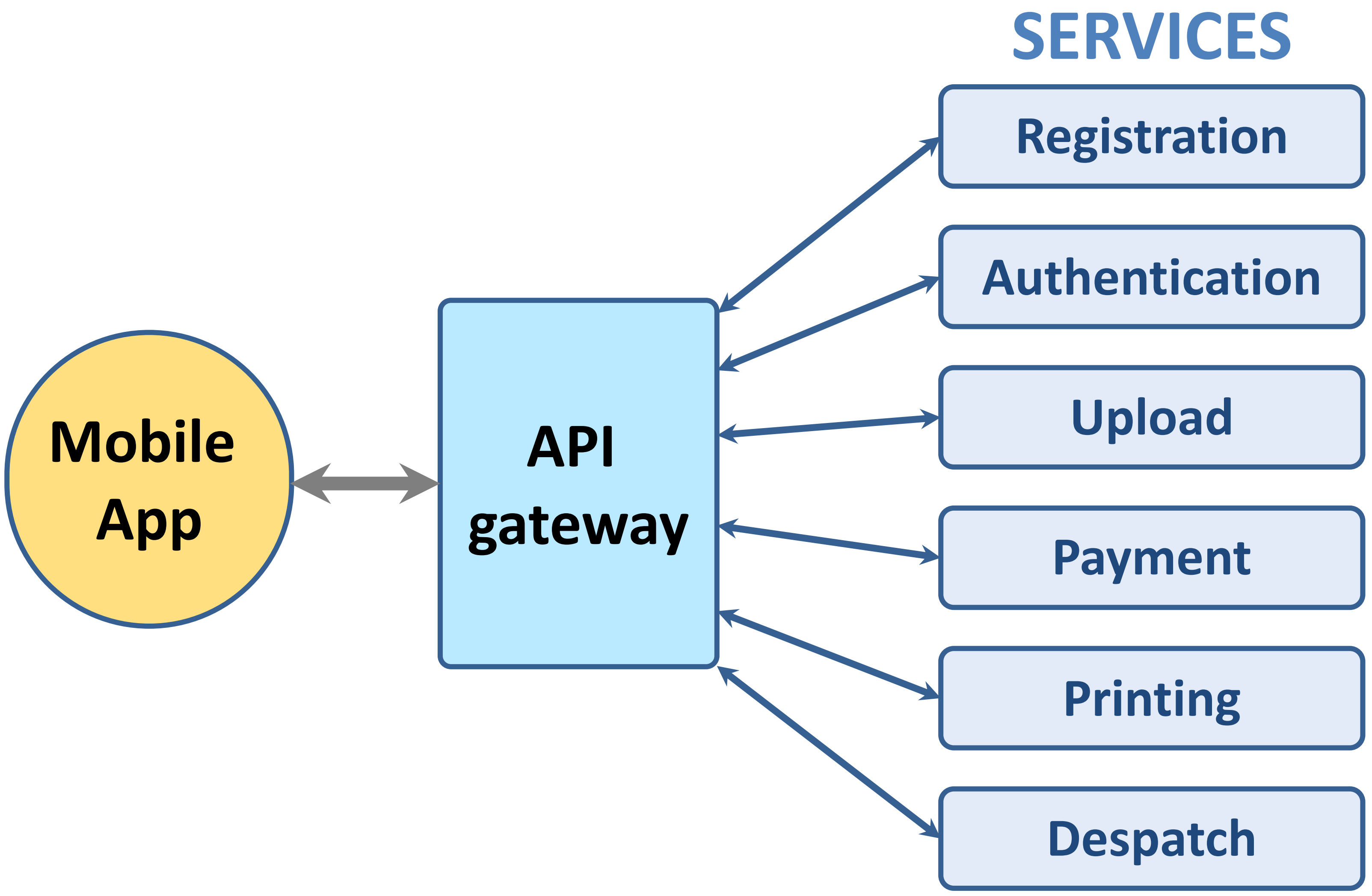
Microservices architecture

- A **microservices architecture** is an architectural style – a tried and tested way of implementing a logical software architecture.
- This architectural style addresses two problems with **monolithic applications**
  - The whole system has to be rebuilt, re-tested and re-deployed when any change is made. This can be a slow process as changes to one part of the system can adversely affect other components.
  - As the demand on the system increases, the whole system has to be scaled, even if the demand is localized to a small number of system components that implement the most popular system functions.

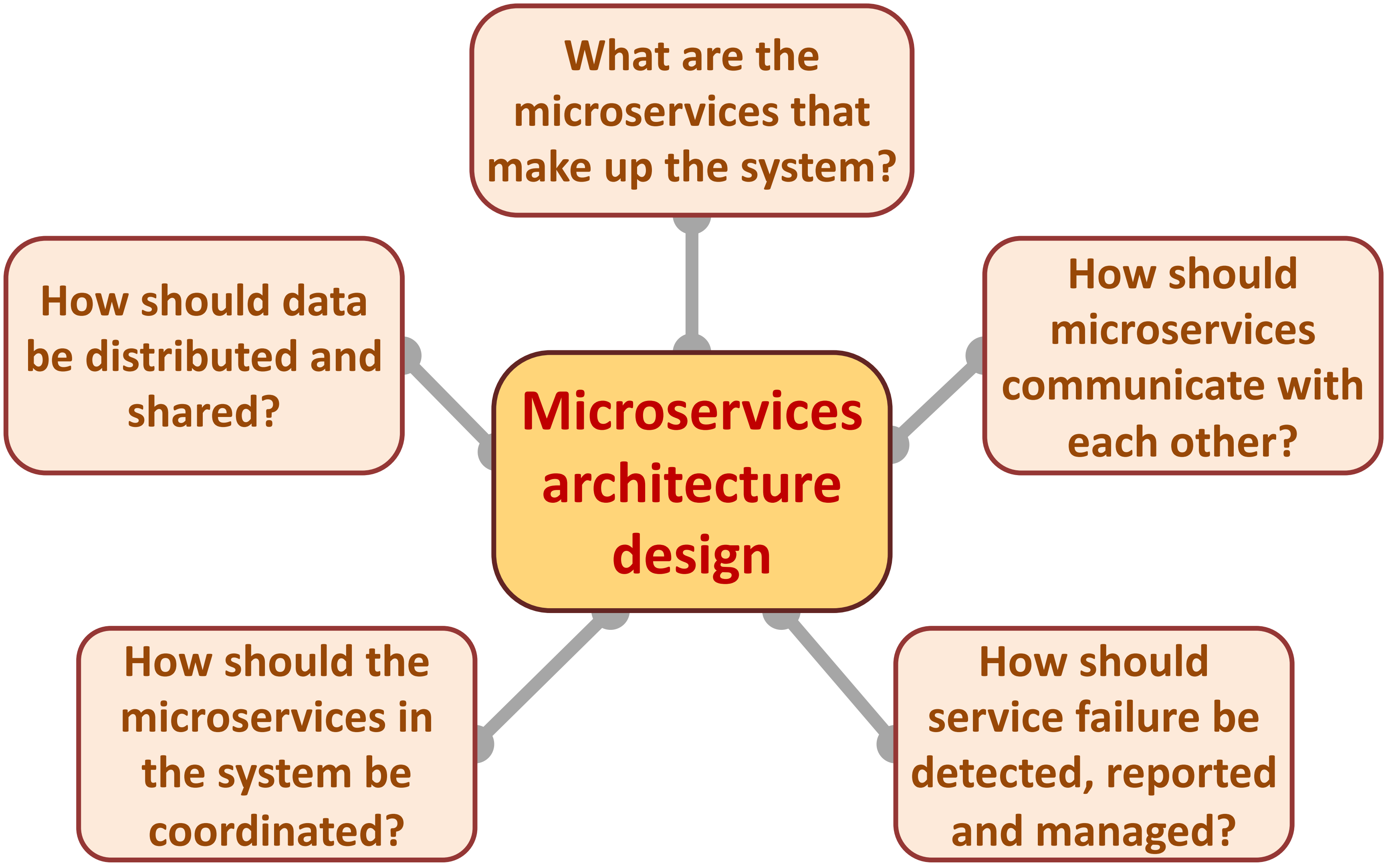
Benefits of microservices architecture

- Microservices are self-contained and run in separate processes.
- In cloud-based systems, each microservice may be deployed in its own container. This means a microservice can be stopped and restarted without affecting other parts of the system.
- If the demand on a service increases, service replicas can be quickly created and deployed. These do not require a more powerful server so ‘scaling-out’ is, typically, much cheaper than ’scaling up’.

A microservices architecture for a photo printing system

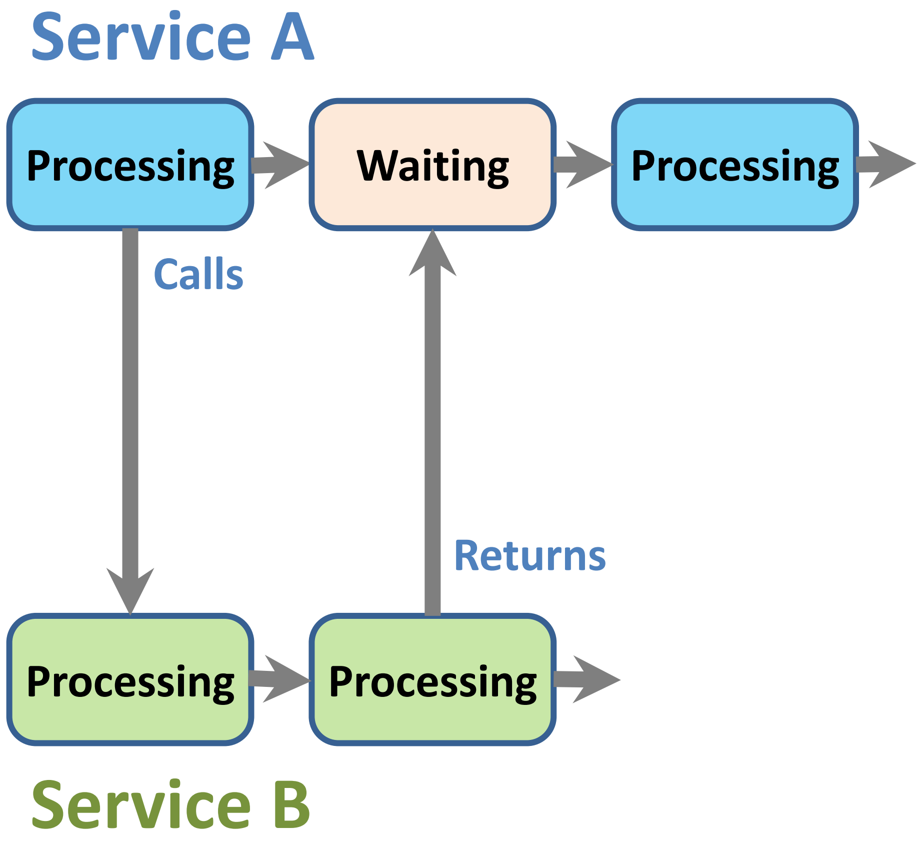


Microservices architecture – key design questions

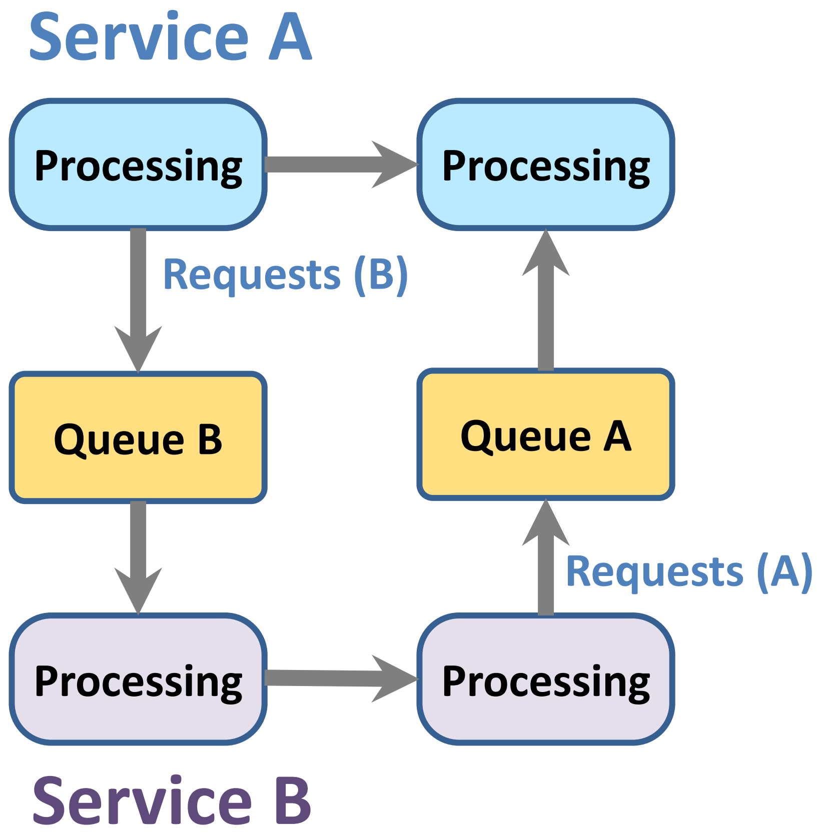

What are the microservices that make up the system?
How should data be distributed and shared?
Microservices architecture design
How should microservices communicate with each other?
How should the microservices in the system be coordinated?
How should service failure be detected, reported and managed?

Synchronous and asynchronous microservice interaction

Synchronous – A waits for B

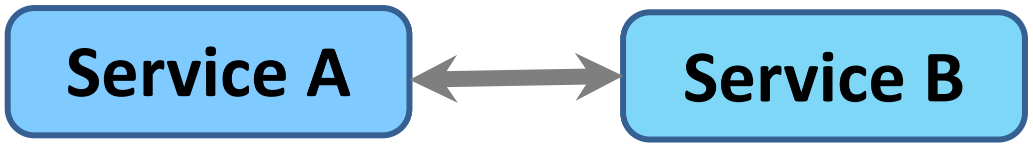


Asynchronous – A and B execute concurrently

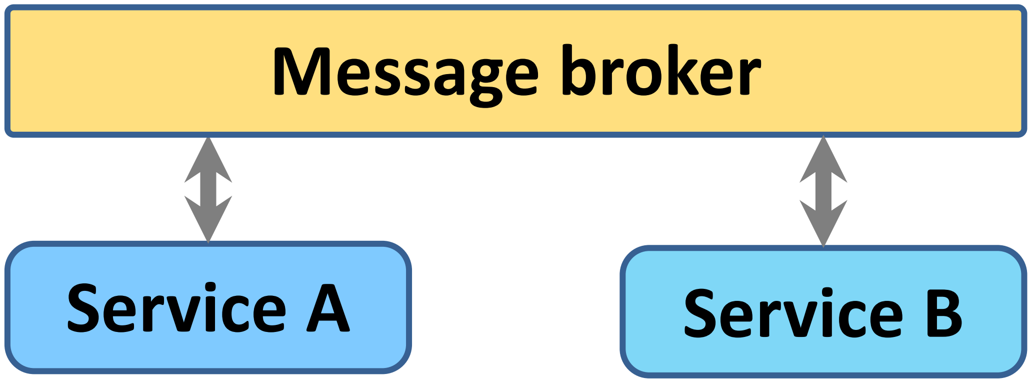


Direct and indirect service communication

Direct communication –
A and B send message to each other



Indirect communication –
A and B communicate through a message broker



Microservice data design

- You should isolate data within each system service with as little data sharing as possible.
- If data sharing is unavoidable, you should design microservices so that most sharing is ‘read-only’, with a minimal number of services responsible for data updates.
- If services are replicated in your system, you must include a mechanism that can keep the database copies used by replica services consistent.

Inconsistency management

- An ACID (atomicity, consistency, isolation, durability) transaction bundles a set of data updates into a single unit so that either all updates are completed or none of them are. ACID transactions are impractical in a microservices architecture.
- The databases used by different microservices or microservice replicas need not be completely consistent all of the time.

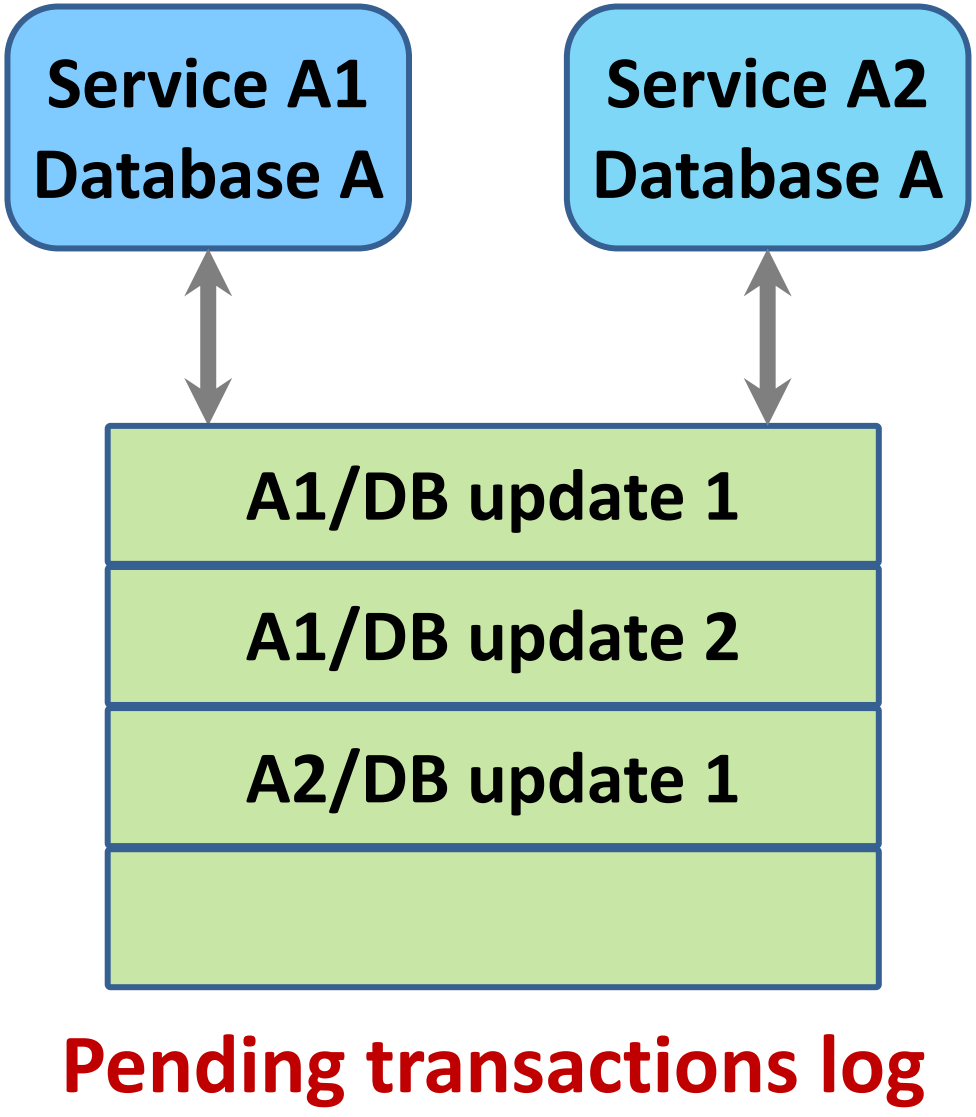
Inconsistency management

- Dependent data inconsistency
  - The actions or failures of one service can cause the data managed by another service to become inconsistent.
- Replica inconsistency
  - There are several replicas of the same service that are executing concurrently. These all have their own database copy and each updates its own copy of the service data. You need a way of making these databases ‘eventually consistent’ so that all replicas are working on the same data.

Eventual consistency

- Eventual consistency is a situation where the system guarantees that the databases will eventually become consistent.
- You can implement eventual consistency by maintaining a transaction log.
- When a database change is made, this is recorded on a ‘pending updates’ log.
- Other service instances look at this log, update their own database and indicate that they have made the change

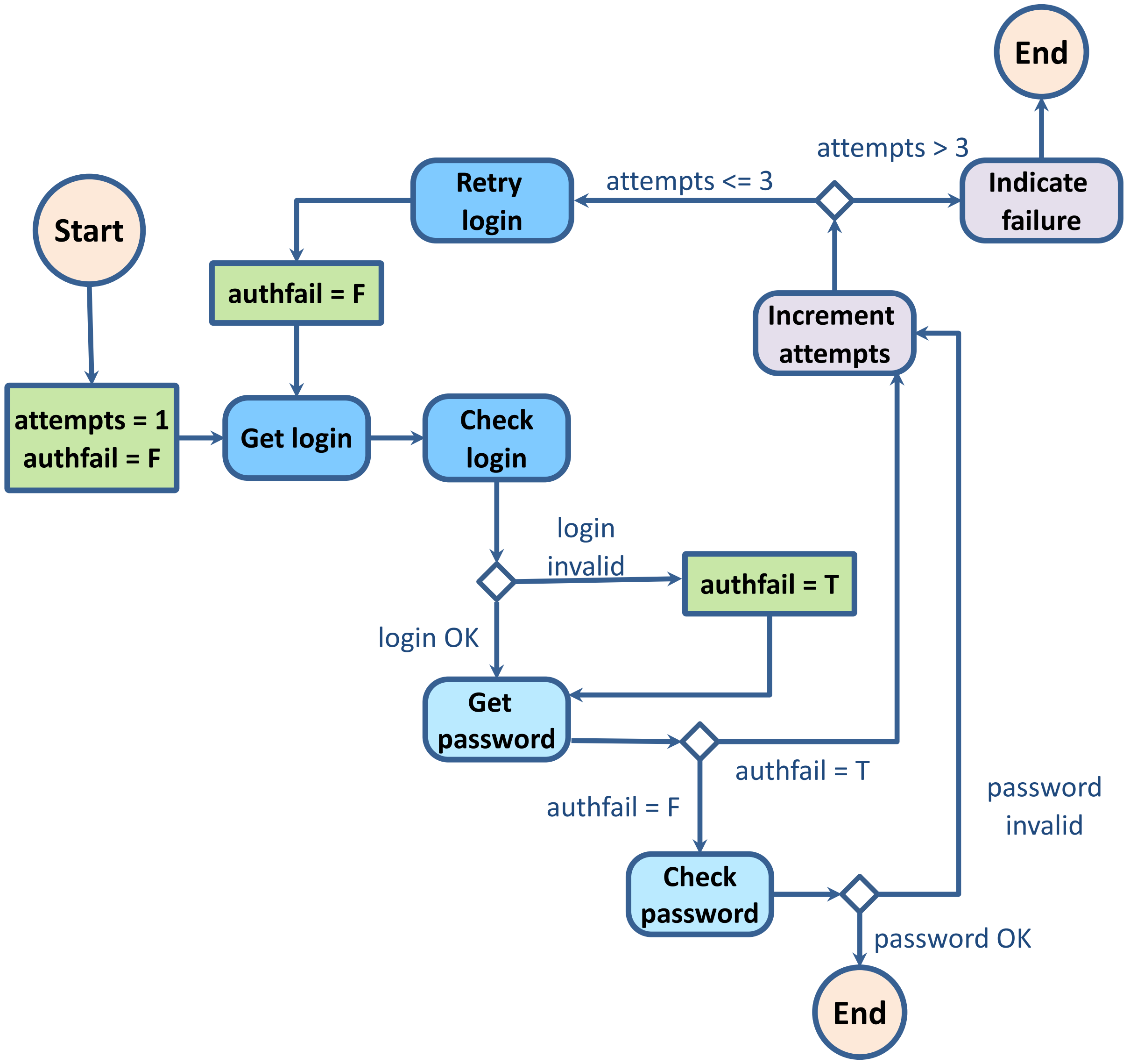
Using a pending transaction log



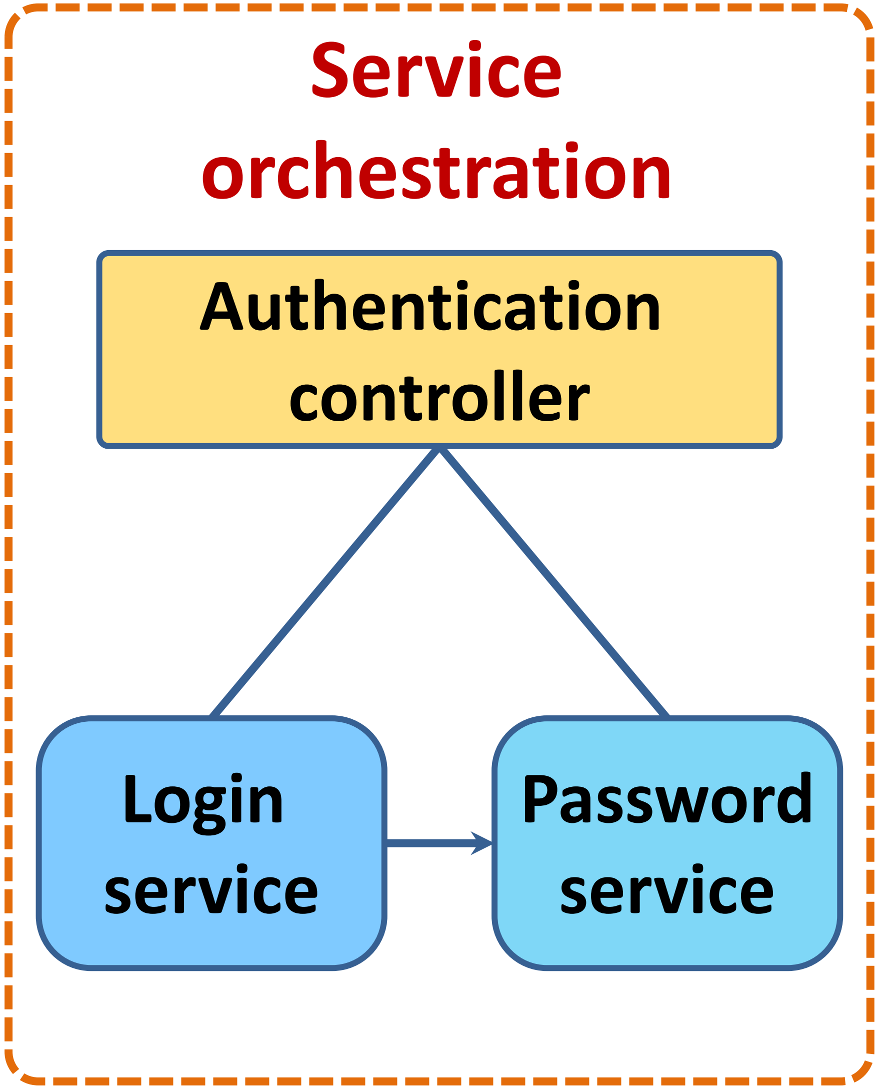
Service coordination

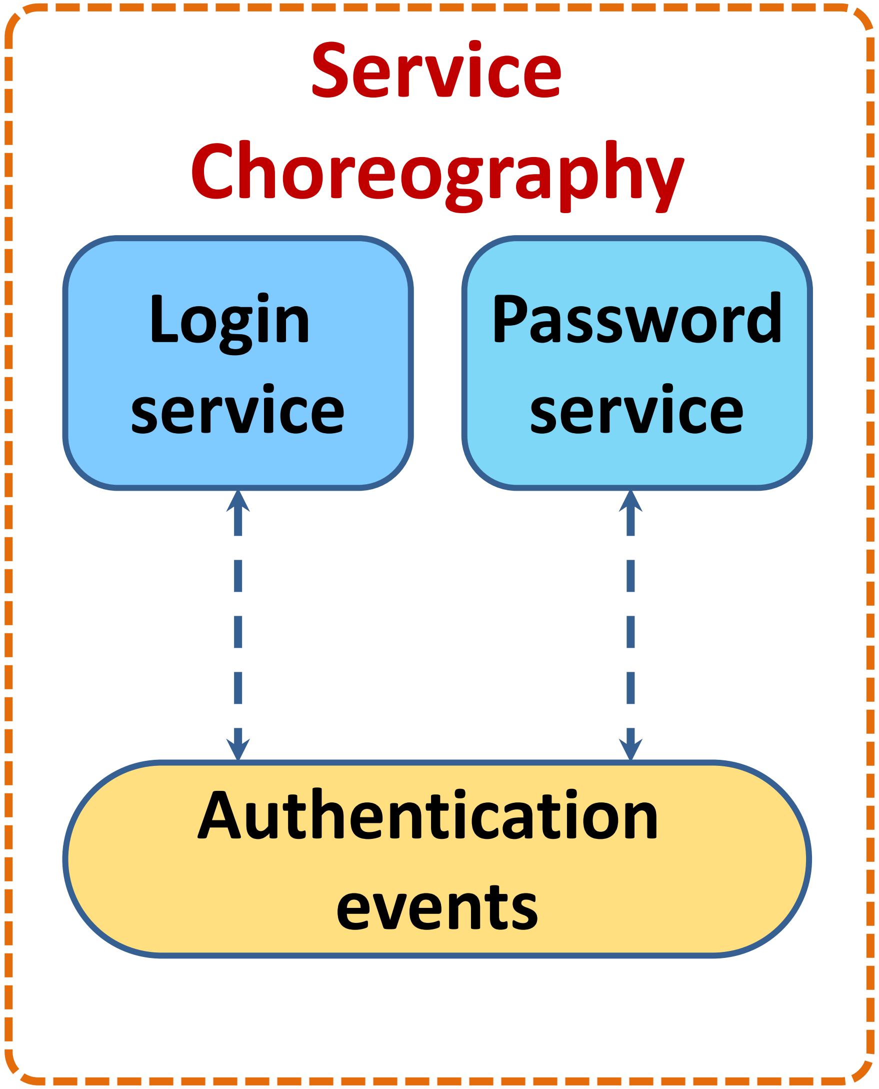
- Most user sessions involve a series of interactions in which operations have to be carried out in a specific order.
- This is called a workflow.
  - An authentication workflow for UID/password authentication shows the steps involved in authenticating a user.
  - In this example, the user is allowed 3 login attempts before the system indicates that the login has failed.

Authentication workflow



Orchestration and choreography





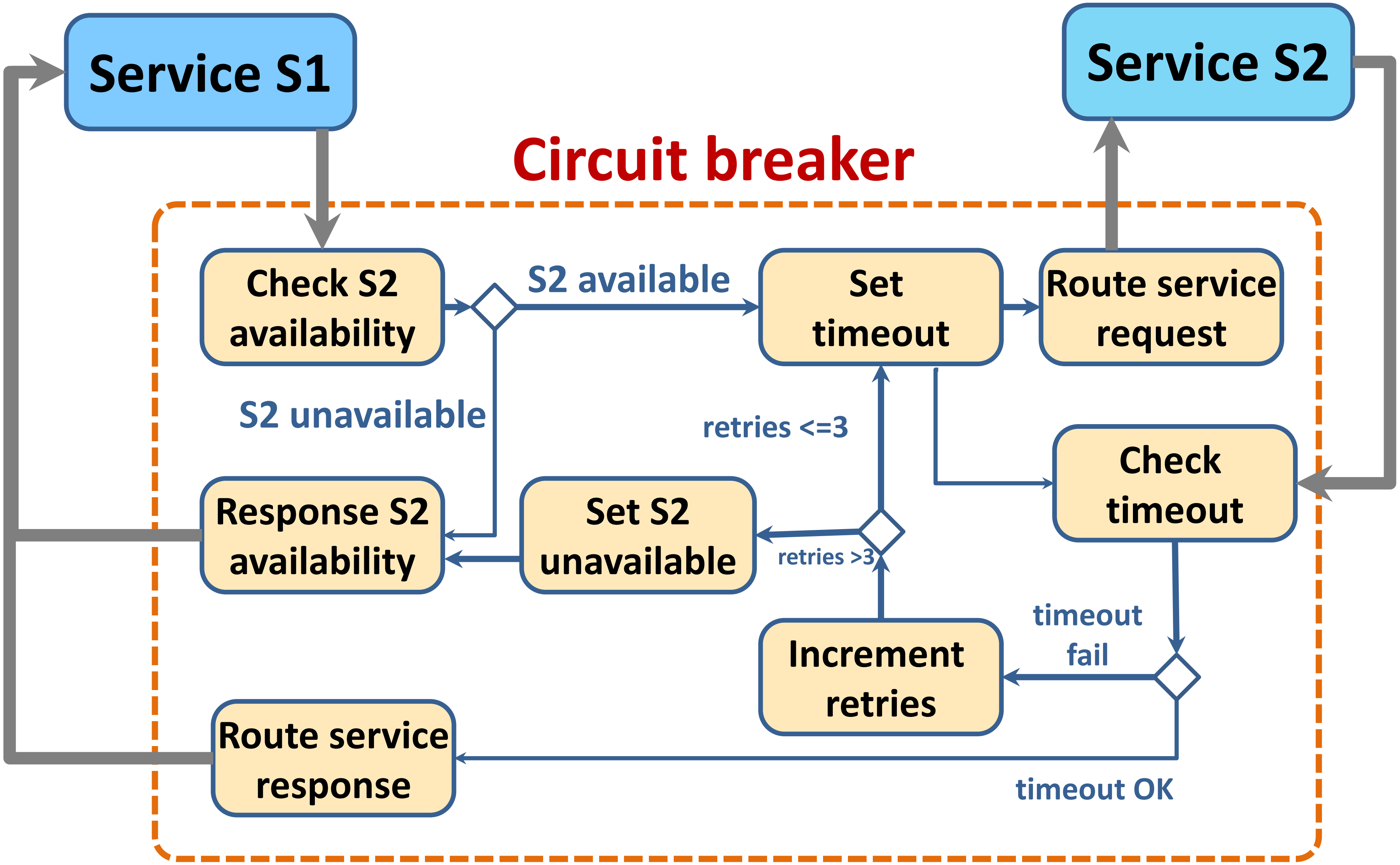
Failure types in a microservices system

- **Internal service failure**
  - These are conditions that are detected by the service and can be reported to the service client in an error message. An example of this type of failure is a service that takes a URL as an input and discovers that this is an invalid link.
- **External service failure**
  - These failures have an external cause, which affects the availability of a service. Failure may cause the service to become unresponsive and actions have to be taken to restart the service.
- **Service performance failure**
  - The performance of the service degrades to an unacceptable level. This may be due to a heavy load or an internal problem with the service. External service monitoring can be used to detect performance failures and unresponsive services.

Timeouts and circuit breakers

- **Timeout**
  - A timeout is a counter that this associated with the service requests and starts running when the request is made.
  - Once the counter reaches some predefined value, such as 10 seconds, the calling service assumes that the service request has failed and acts accordingly.
  - The problem with the timeout approach is that every service call to a 'failed service' is delayed by the timeout value so the whole system slows down.
- **Circuit breaker**
  - Instead of using timeouts explicitly when a service call is made
  - Like an electrical circuit breaker, this immediately denies access to a failed service without the delays associated with timeouts.

Using a circuit breaker to cope with service failure



RESTful services

- The REST (REpresentational State Transfer) architectural style is based on the idea of transferring representations of digital resources from a server to a client.
  - You can think of a resource as any chunk of data such as credit card details, an individual’s medical record, a magazine or newspaper, a library catalogue, and so on.
  - Resources are accessed via their unique URI and RESTful services operate on these resources.

RESTful services

- This is the fundamental approach used in the web where the resource is a page to be displayed in the user's browser.
  - An HTML representation is generated by the server in response to an HTTP GET request and is transferred to the client for display by a browser or a special-purpose app.

RESTful service principles

- **Use HTTP verbs**
  - The basic methods defined in the HTTP protocol (GET, PUT, POST, DELETE) must be used to access the operations made available by the service.
- **Stateless services**
  - Services must never maintain internal state. As I have already explained, microservices are stateless so fit with this principle.
- **URI addressable**
  - All resources must have a URI, with a hierarchical structure, that is used to access sub-resources.
- **Use XML or JSON**
  - Resources should normally be represented in JSON or XML or both. Other representations, such as audio and video representations, may be used if appropriate.

RESTful service operations

- **Create**
  - Implemented using HTTP POST, which creates the resource with the given URI. If the resource has already been created, an error is returned.
- **Read**
  - Implemented using HTTP GET, which reads the resource and returns its value. GET operations should never update a resource so that successive GET operations with no intervening PUT operations always return the same value.
- **Update**
  - Implemented using HTTP PUT, which modifies an existing resource. PUT should not be used for resource creation.
- **Delete**
  - Implemented using HTTP DELETE, which makes the resource inaccessible using the specified URI. The resource may or may not be physically deleted.

Service operations

- **Retrieve**
  - Returns information about a reported incident or incidents. Accessed using the GET verb.
- **Add**
  - Adds information about a new incident. Accessed using the POST verb.
- **Update**
  - Updates the information about a reported incident. Accessed using the PUT verb.
- **Delete**
  - Deletes an incident. The DELETE verb is used when an incident has been cleared.

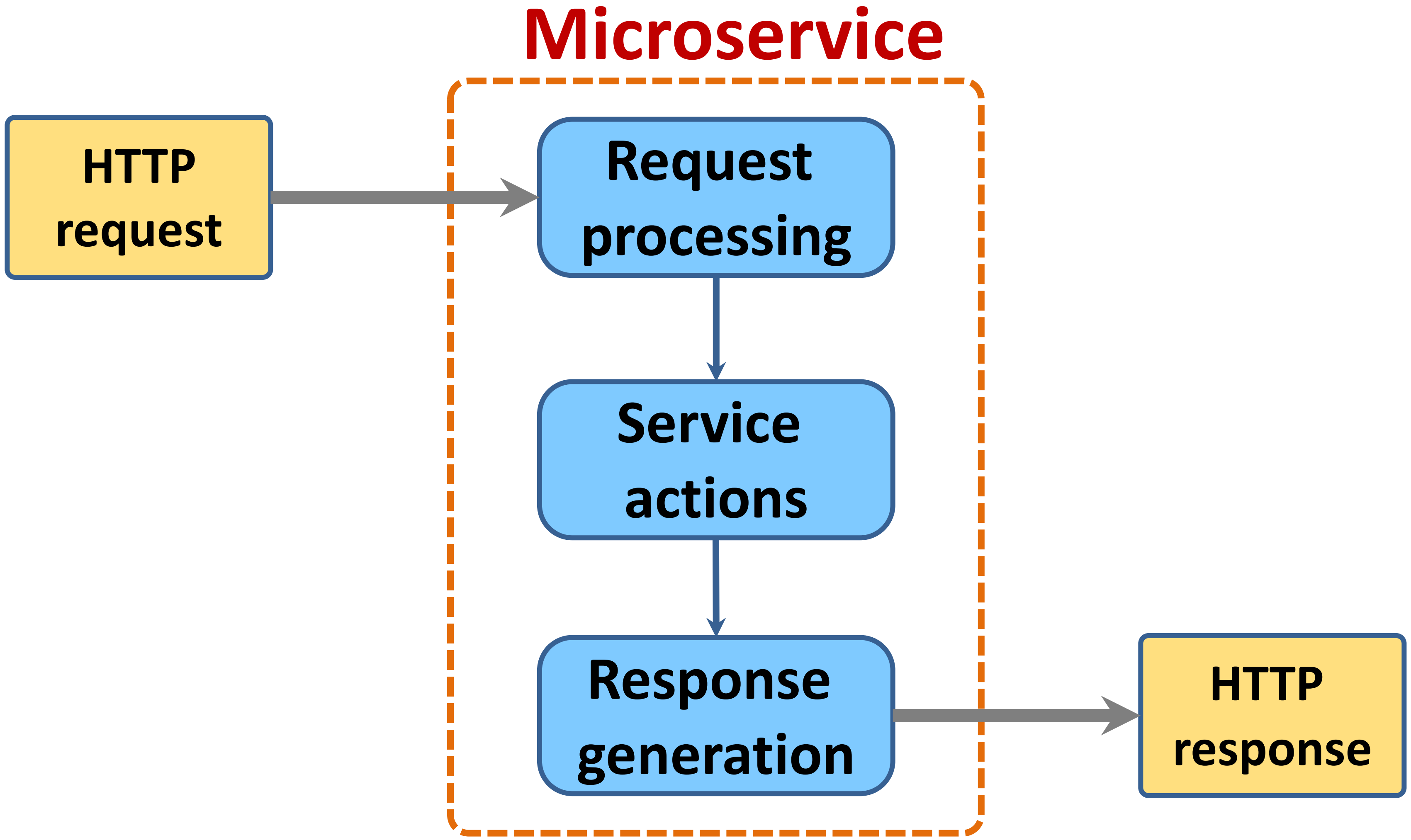
Road information system

- Imagine a system that maintains information about incidents, such as traffic delays, roadworks and accidents on a national road network. This system can be accessed via a browser using the URL:
  - https://trafficinfo.net/incidents/
- Users can query the system to discover incidents on the roads on which they are planning to travel.

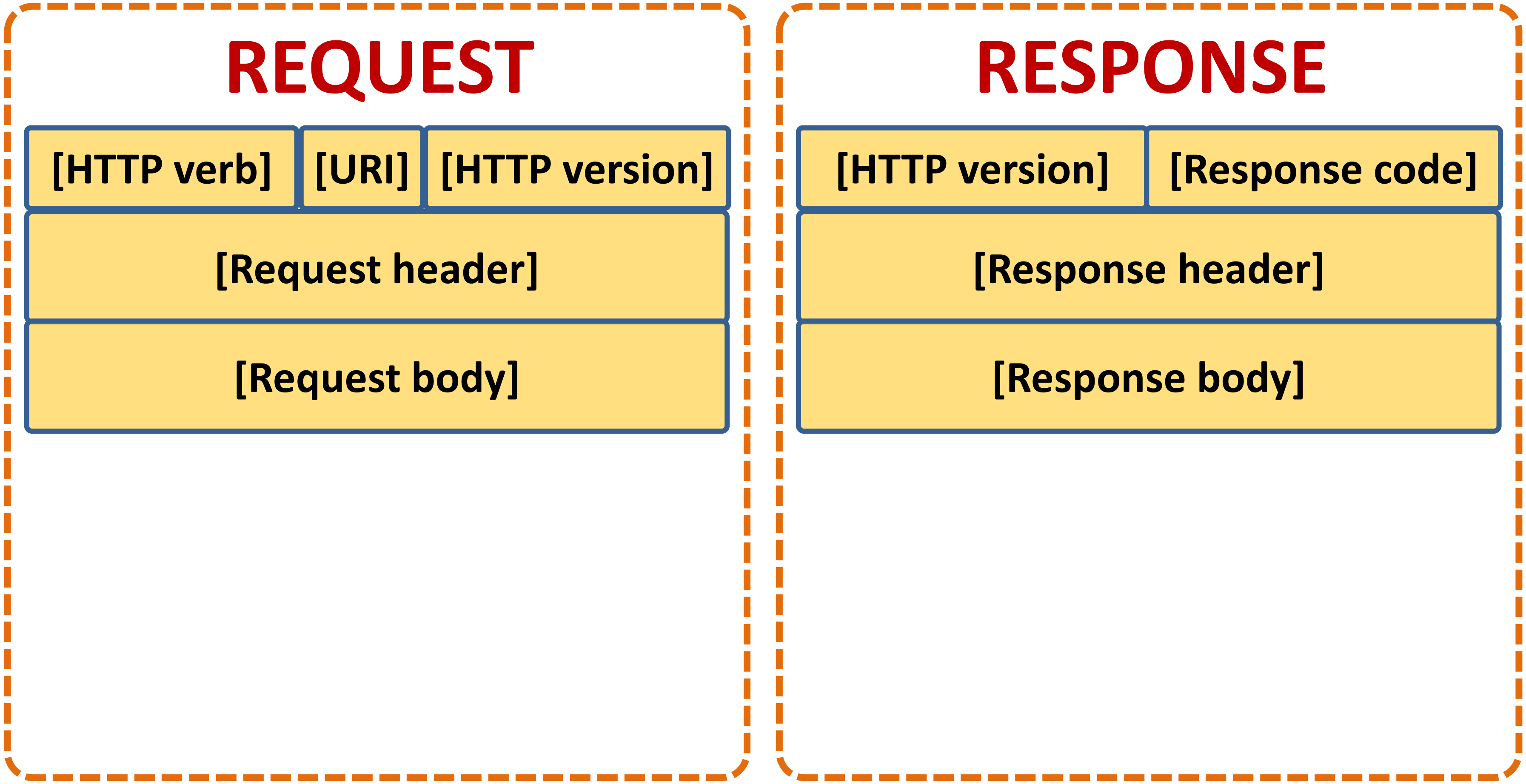
Road information system

- When implemented as a RESTful web service, you need to design the resource structure so that incidents are organized hierarchically.
- For example, incidents may be recorded according to the road identifier (e.g. A90), the location (e.g. stonehaven), the carriageway direction (e.g. north) and an incident number (e.g. 1). Therefore, each incident can be accessed using its URI:
  - https://trafficinfo.net/incidents/A90/stonehaven/north/1

HTTP request and response processing



HTTP request and response processing



XML and JSON descriptions

XML

```
<id>
A90N17061714391
</id>
<date>
20170617
</date>
<time>
1437
</time>
…
<description>Broken-down bus on north carriageway. One lane
closed. Expect delays of up to 30 minutes.
</description>
```

XML and JSON descriptions

JSON

```
{
id: “A90N17061714391”,
“date”: “20170617”,
“time”: “1437”,
“road_id”: “A90”,
“place”: “Stonehaven”,
“direction”: “north”,
“severity”: “significant”,
“description”: “Broken-down bus on north carriageway. One lane
closed. Expect delays of up to 30 minutes.”
}
```

Service deployment

- After a system has been developed and delivered, it has to be deployed on servers, monitored for problems and updated as new versions become available.
- When a system is composed of tens or even hundreds of microservices, deployment of the system is more complex than for monolithic systems.
- The service development teams decide which programming language, database, libraries and other support software should be used to implement their service. Consequently, there is no ‘standard’ deployment configuration for all services.

Service deployment

- It is now normal practice for microservice development teams to be responsible for deployment and service management as well as software development and to use continuous deployment.
- **Continuous deployment** means that as soon as a change to a service has been made and validated, the modified service is redeployed.

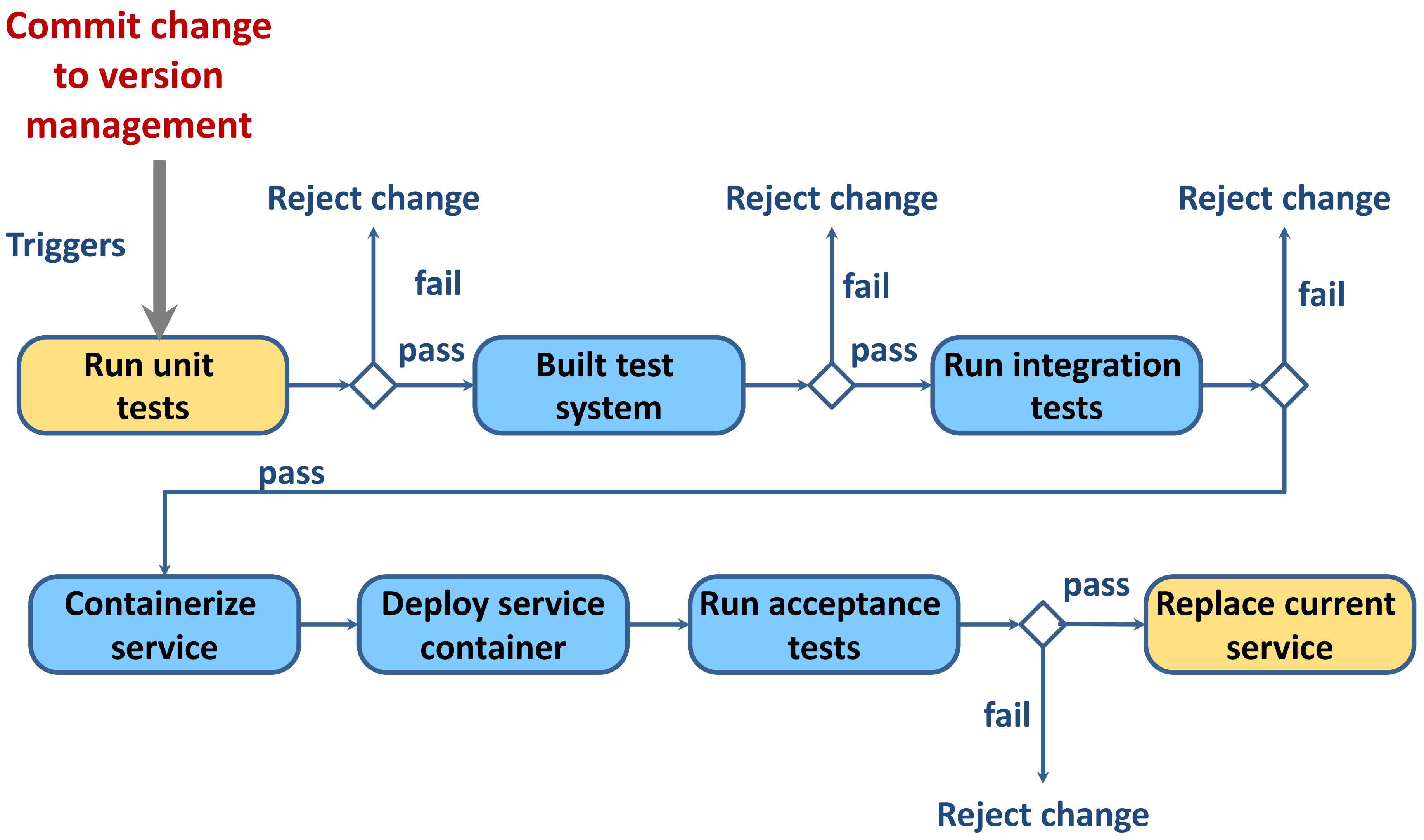
Deployment automation

- Continuous deployment depends on automation so that as soon as a change is committed, a series of automated activities is triggered to test the software.
- If the software 'passes' these tests, it then enters another automation pipeline that packages and deploys the software.
- The deployment of a new service version starts with the programmer committing the code changes to a code management system such as Git.

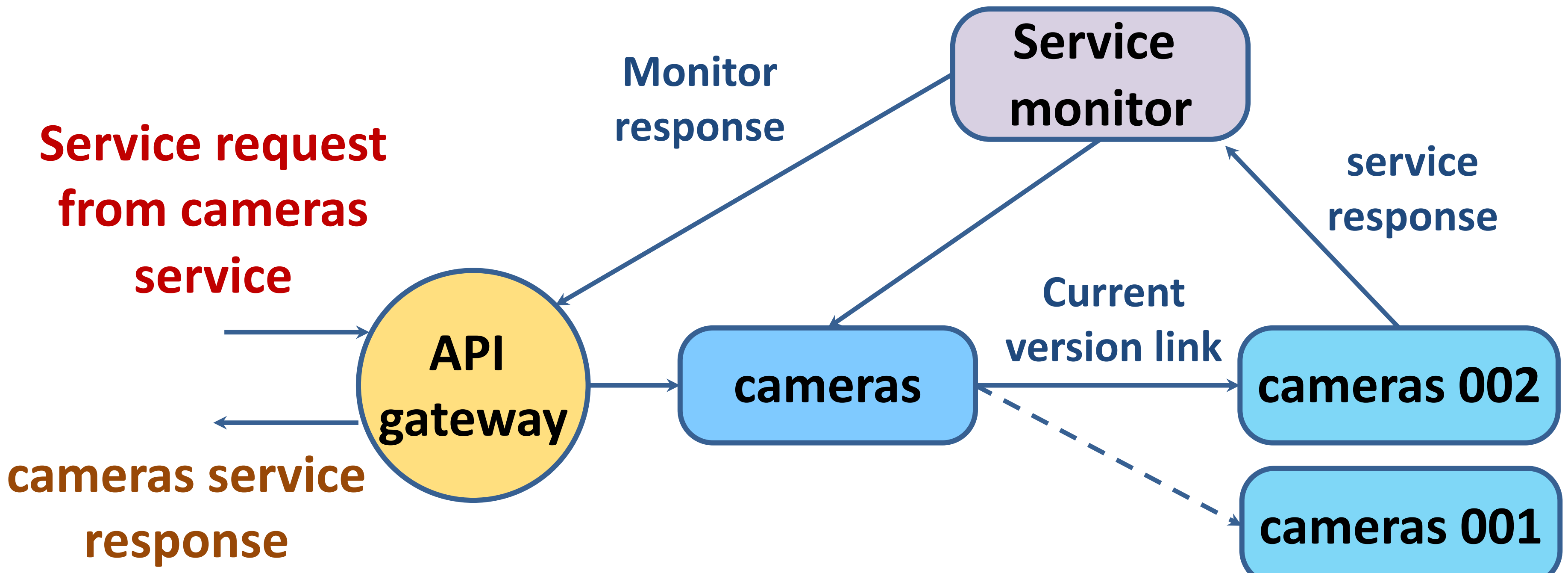
Deployment automation

- This triggers a set of automated tests that run using the modified service. If all service tests run successfully, a new version of the system that incorporates the changed service is created.
- Another set of automated system tests are then executed. If these run successfully, the service is ready for deployment.

A continuous deployment pipeline



Versioned services



Summary

- A **microservice** is an independent and self-contained software component that runs in its own process and communicates with other microservices using lightweight protocols.
- Microservices in a system can be implemented using different programming languages and database technologies.
- Microservices have a single responsibility and should be designed so that they can be easily changed without having to change other microservices in the system.

Summary

- **Microservices architecture** is an architectural style in which the system is constructed from communicating microservices. It is well-suited to cloud based systems where each microservice can run in its own container.
- The **two most important responsibilities of architects** of a microservices system are to decide how to structure the system into microservices and to decide how microservices should communicate and be coordinated.

Summary

- **Communication and coordination decisions** include deciding on microservice communication protocols, data sharing, whether services should be centrally coordinated, and failure management.
- The **RESTful** architectural style is widely used in microservice-based systems. Services are designed so that the HTTP verbs, GET, POST, PUT and DELETE, map onto the service operations.
- The **RESTful style** is based on digital resources that, in a microservices architecture, may be represented using XML or, more commonly, JSON.

Summary

- **Continuous deployment** is a process where new versions of a service are put into production as soon as a service change has been made. It is a completely automated process that relies on automated testing to check that the new version is of 'production quality'.
- If continuous deployment is used, you may need to maintain multiple versions of deployed services so that you can switch to an older version if problems are discovered in a newly-deployed service.

References

- Ian Sommerville (2019), Engineering Software Products: An Introduction to Modern Software Engineering, Pearson.
- Ian Sommerville (2015), Software Engineering, 10th Edition, Pearson.
- Titus Winters, Tom Manshreck, and Hyrum Wright (2020), Software Engineering at Google: Lessons Learned from Programming Over Time, O'Reilly Media.
- Project Management Institute (2017), A Guide to the Project Management Body of Knowledge (PMBOK Guide), Sixth Edition, Project Management Institute
- Project Management Institute (2017), Agile Practice Guide, Project Management Institute