

# Software Engineering

# Software Architecture: Architectural design, System decomposition, and Distribution architecture

1132SE05

MBA, IM, NTPU (M5010) (Spring 2025)

Wed 2, 3, 4 (9:10-12:00) (B3F17)

Min-Yuh Day, Ph.D,  
Professor

Institute of Information Management, National Taipei University

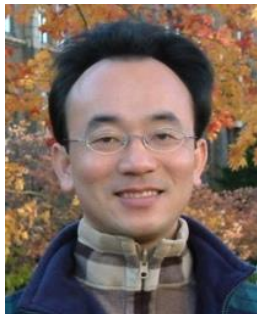
<https://web.ntpu.edu.tw/~myday>



<https://meet.google.com/ish-gzmy-pmo>



aws  
educate | Cloud  
Ambassador  
2020 Cohort



# Syllabus

Week	Date	Subject/Topics
1	2025/02/19	Introduction to Software Engineering
2	2025/02/26	Software Products and Project Management: Software product management and prototyping with Generative AI
3	2025/03/05	Agile Software Engineering: Agile methods, Scrum, and Extreme Programming
4	2025/03/12	Case Study on Software Engineering I
5	2025/03/19	Features, Scenarios, and Stories
6	2025/03/26	Software Architecture: Architectural design, System decomposition, and Distribution architecture

# Syllabus

**Week Date Subject/Topics**

7 2025/04/02 Make-up holiday for NTPU Sports Day (No Classes)

**8 2025/04/09 Midterm Project Report**

**9 2025/04/16 Cloud-Based Software: Virtualization and containers,  
Everything as a service, Software as a service**

**10 2025/04/23 Cloud Computing and Cloud Software Architecture**

**11 2025/04/30 Case Study on Software Engineering II**

**12 2025/05/07 Microservices Architecture, RESTful services,  
Service deployment**

# Syllabus

**Week Date Subject/Topics**

**13 2025/05/14 Industry Practices of Software Engineering**

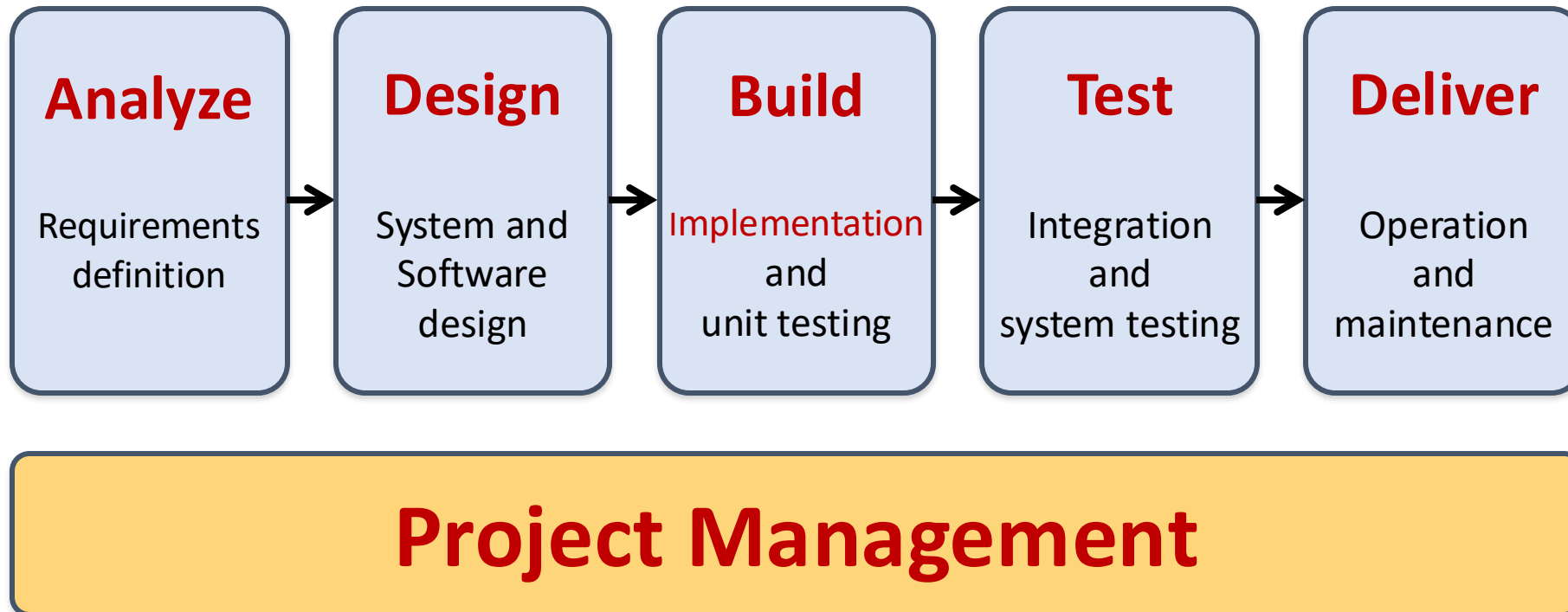
**14 2025/05/21 Security and Privacy; Reliable Programming;  
Testing: Functional testing, Test automation,  
Test-driven development, and Code reviews;  
DevOps and Code Management:  
Code management and DevOps automation**

**15 2025/05/28 Final Project Report I**

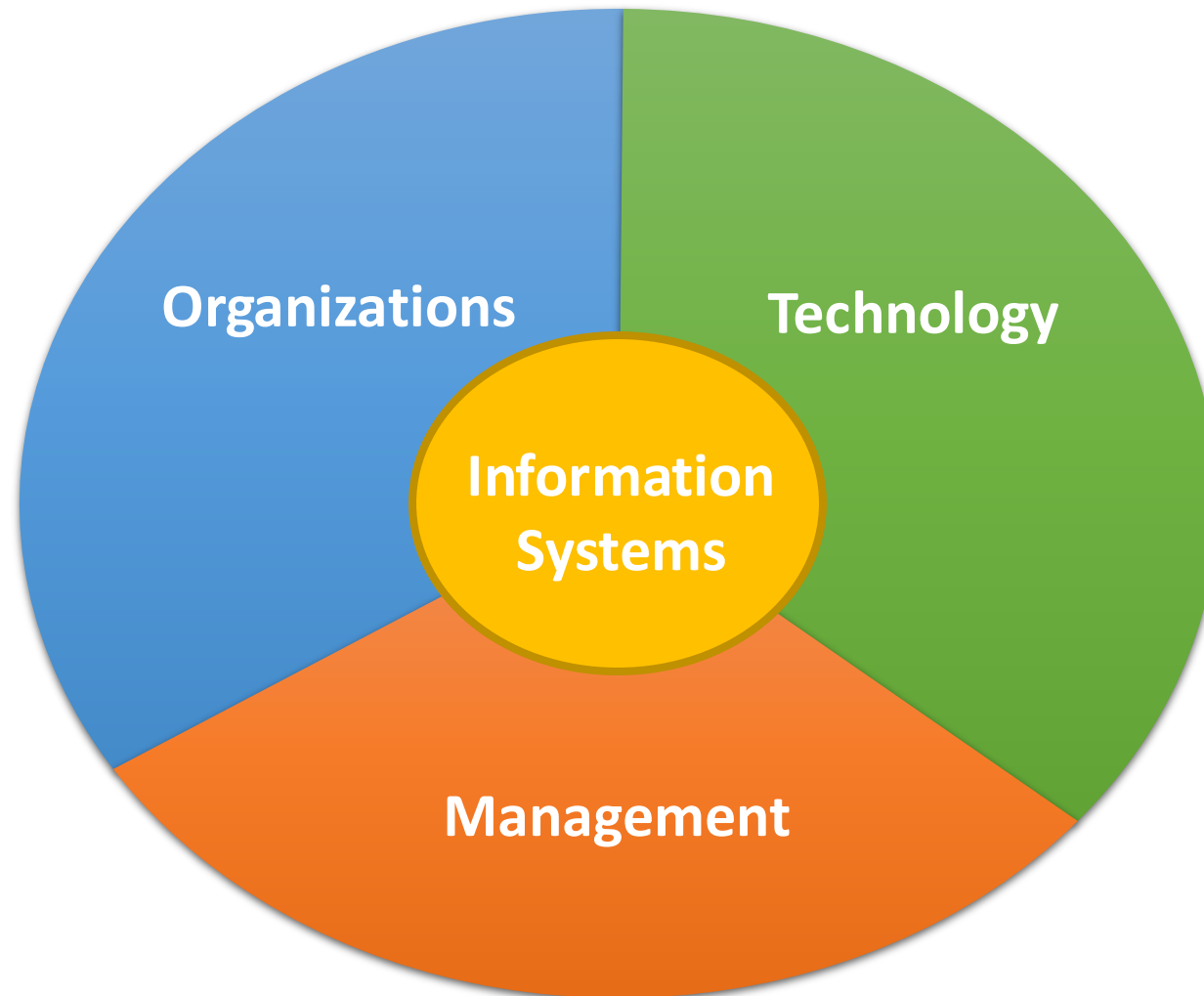
**16 2025/06/04 Final Project Report II**

# **Software Architecture: Architectural design, System decomposition, and Distribution architecture**

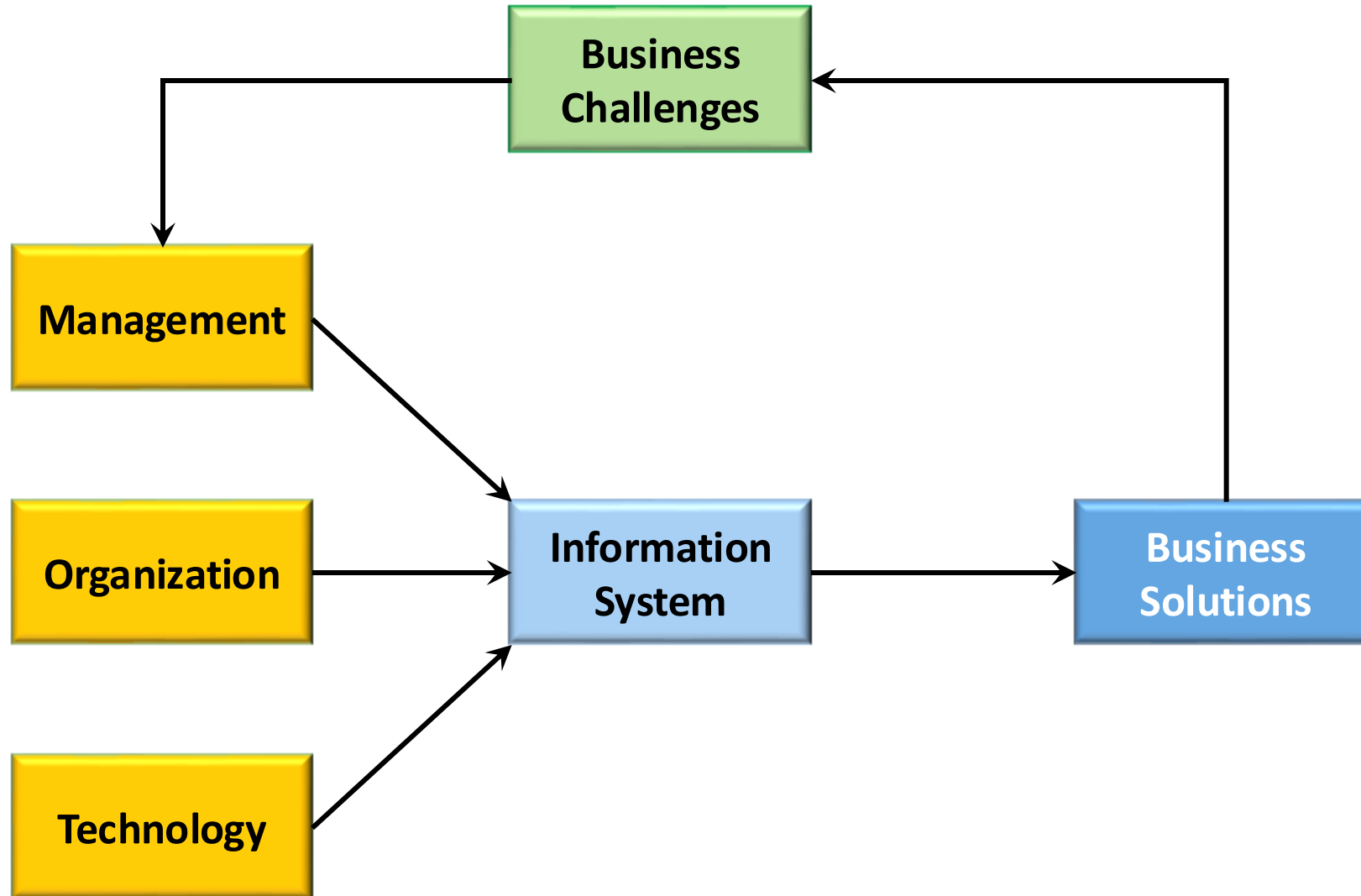
# Software Engineering and Project Management



# Information Management (MIS) Information Systems

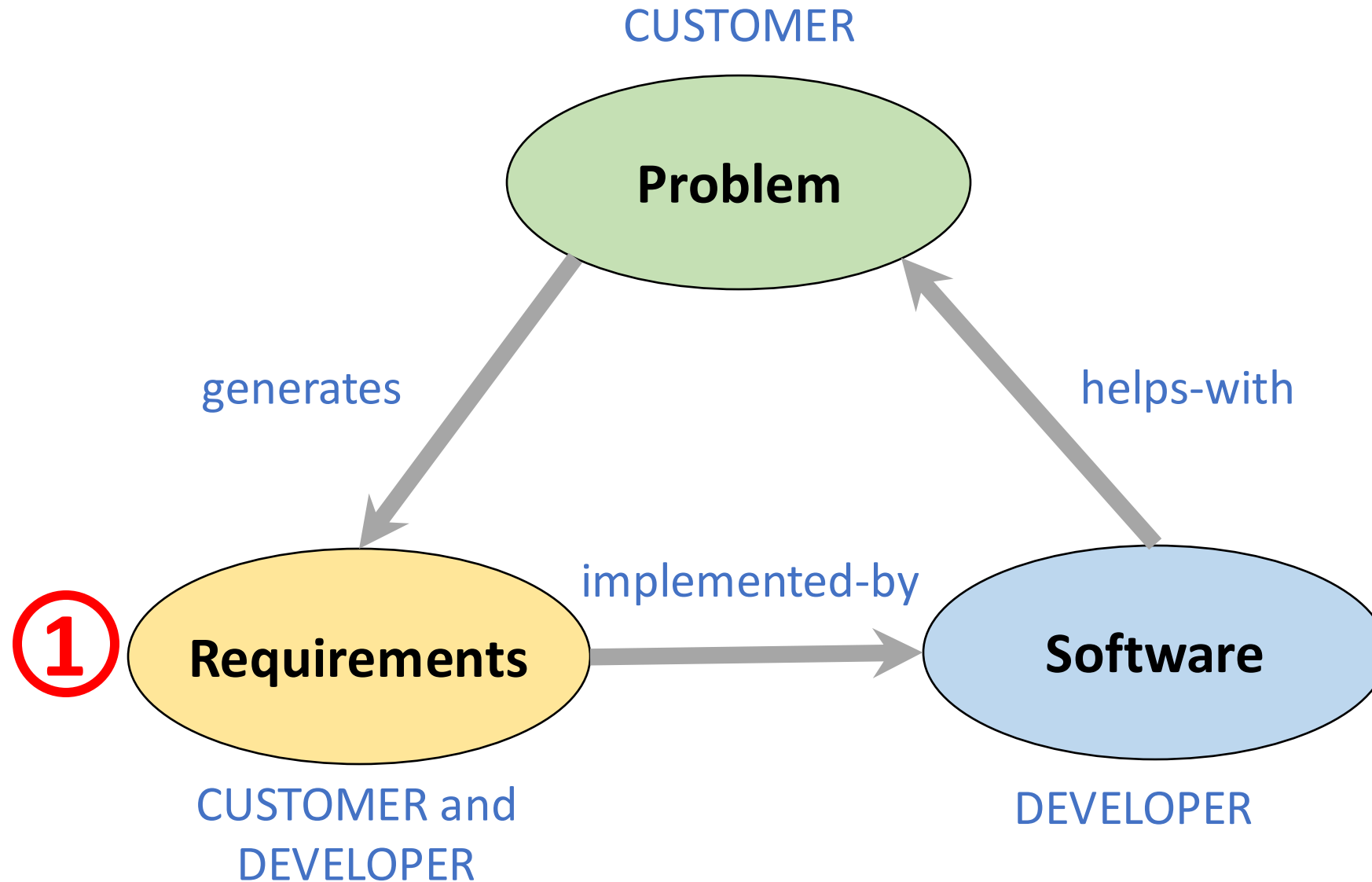


# Fundamental MIS Concepts

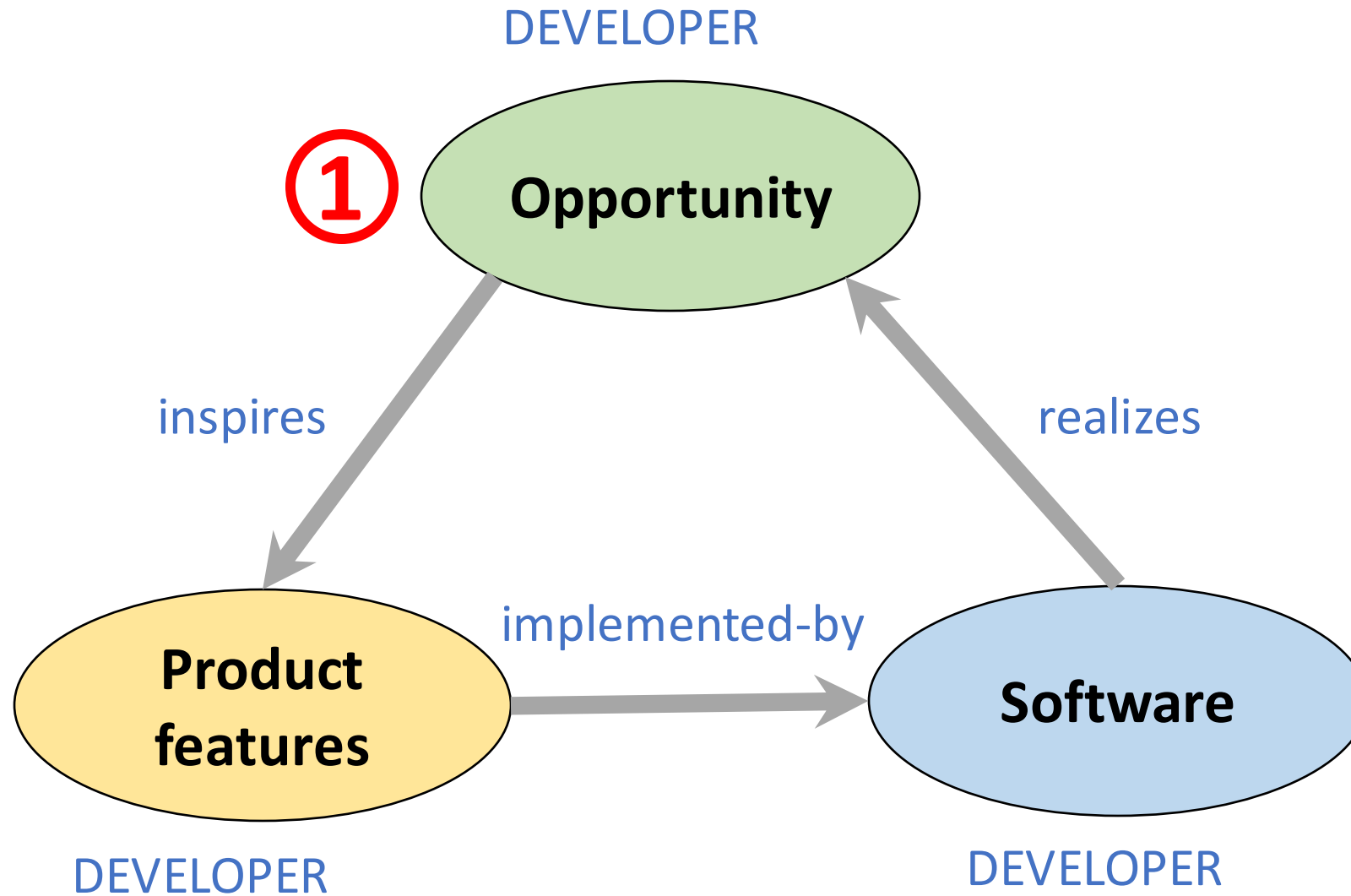




# Project-based software engineering

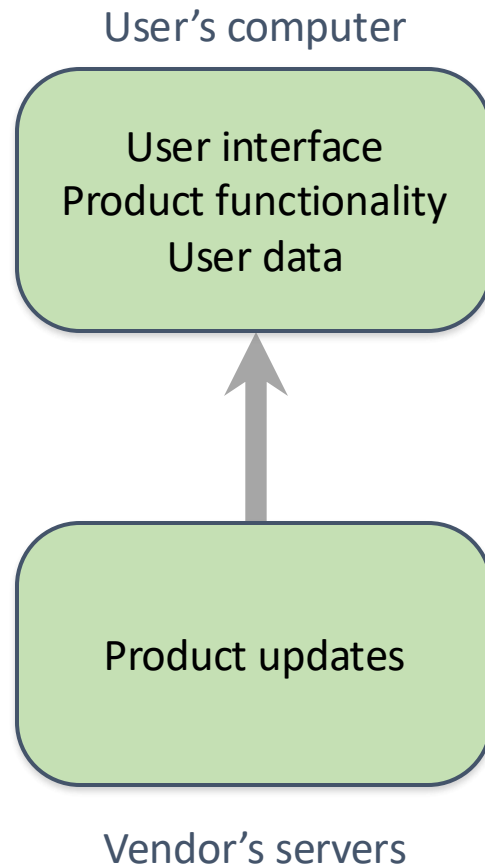


# Product software engineering

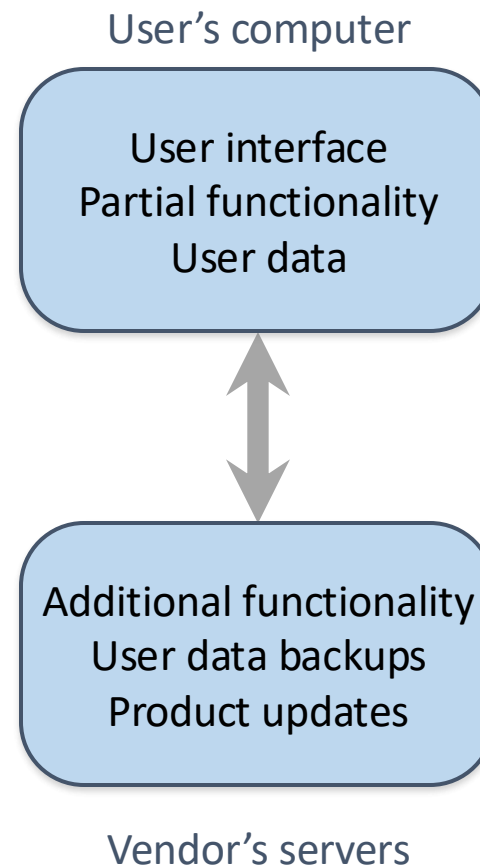


# Software execution models

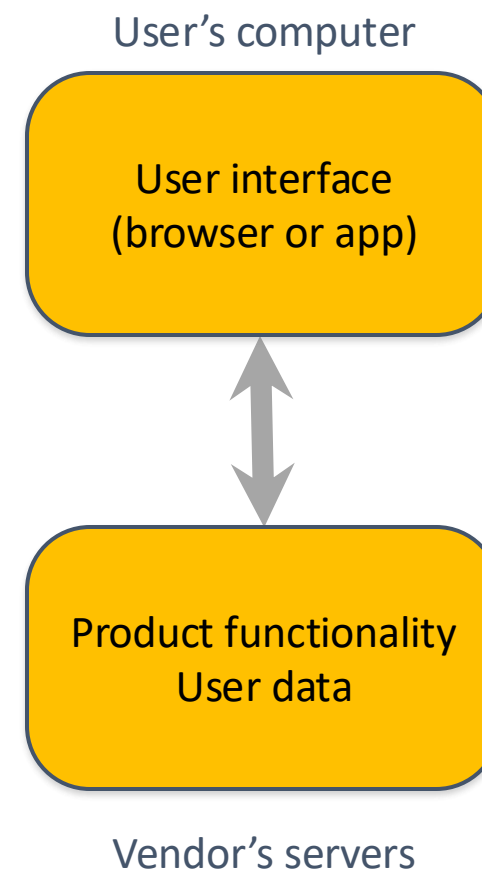
## Stand-alone execution



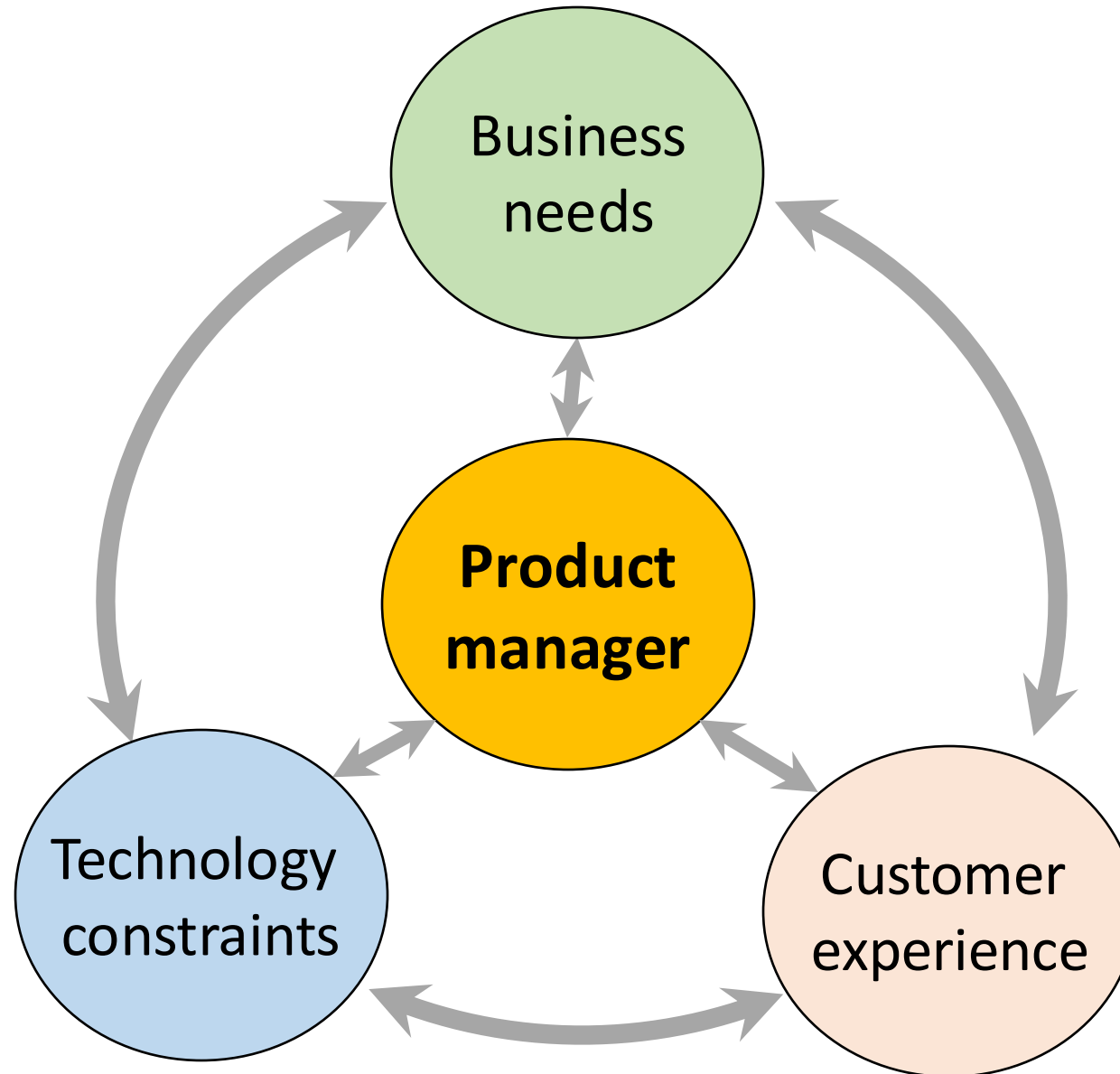
## Hybrid execution



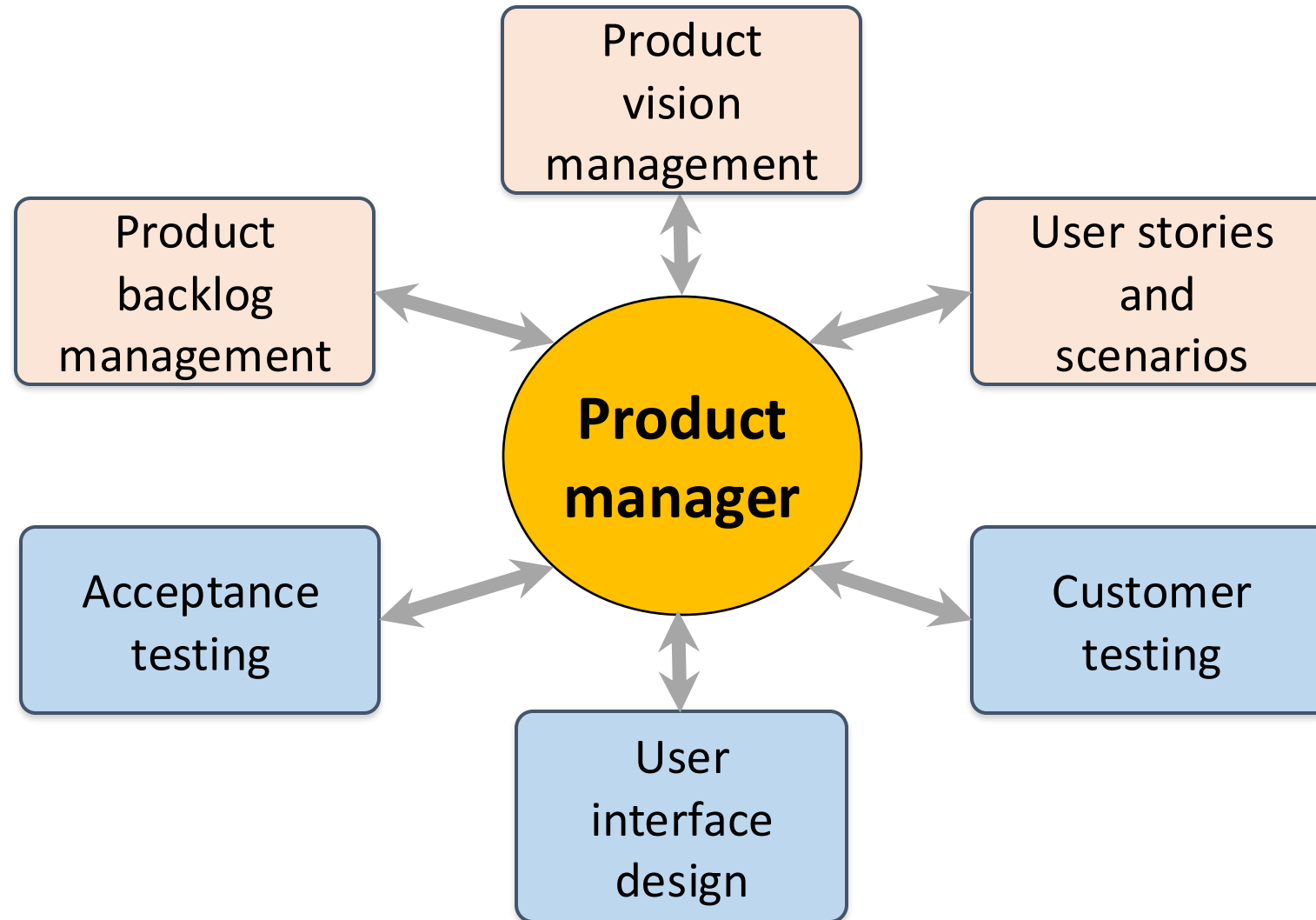
## Software as a service



# Product management concerns

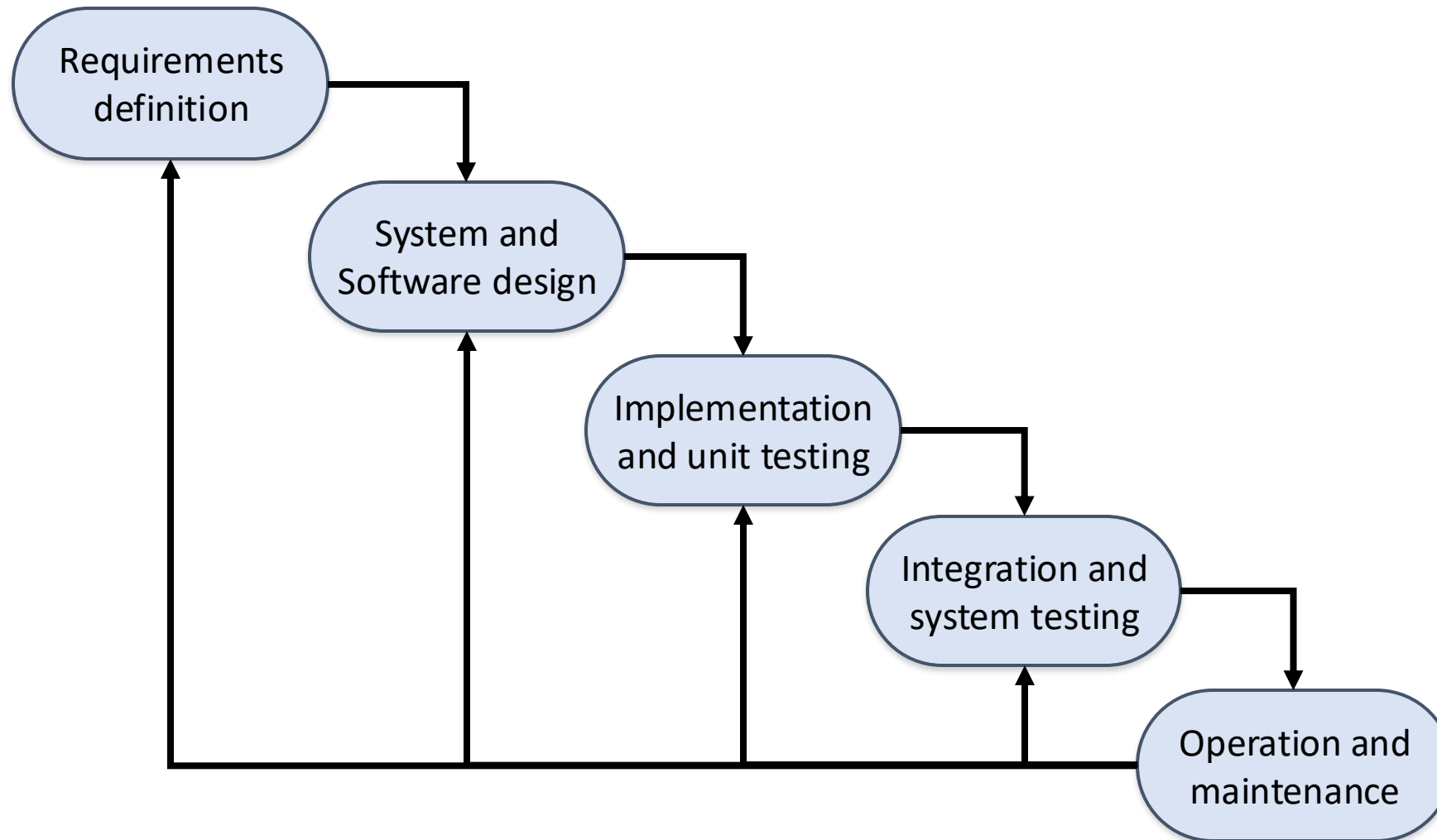


# Technical interactions of product managers



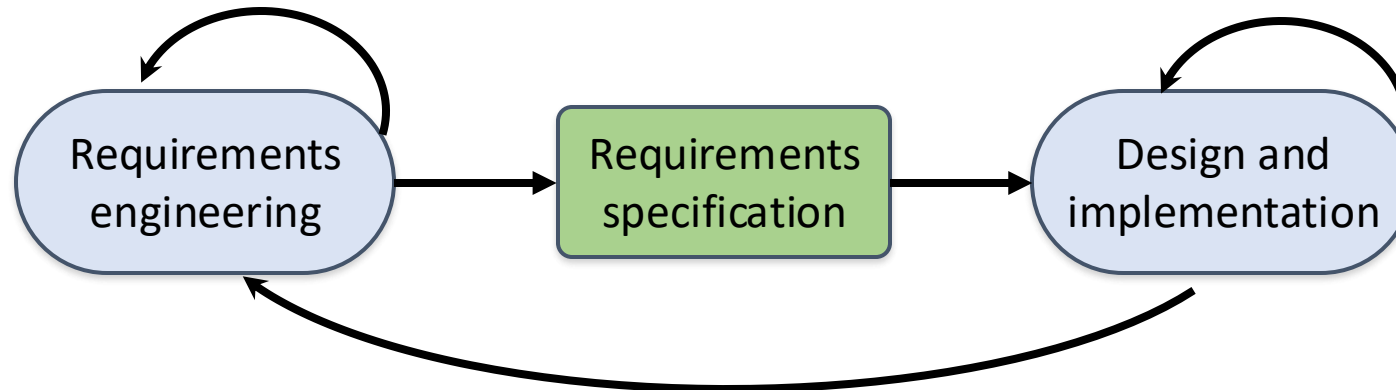
# Software Development Life Cycle (SDLC)

## The waterfall model



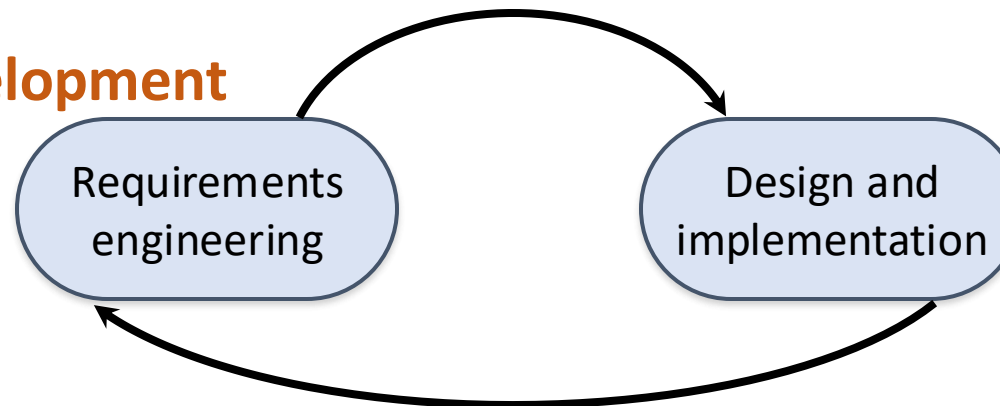
# Plan-based and Agile development

## Plan-based development

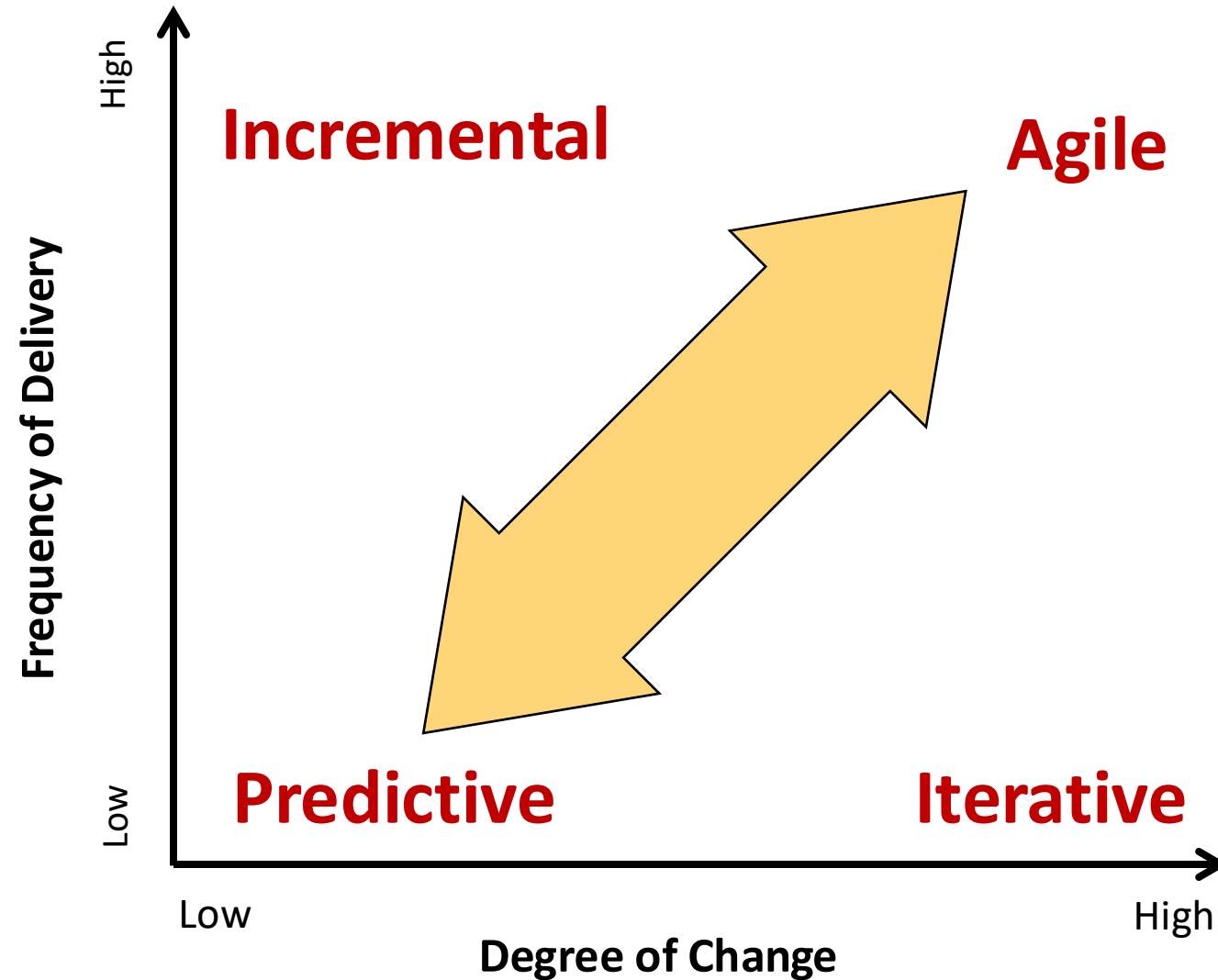


Requirements change requests

## Agile development

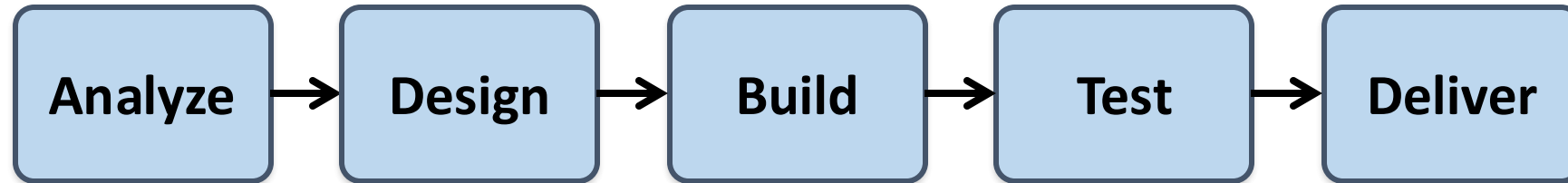


# The Continuum of Life Cycles

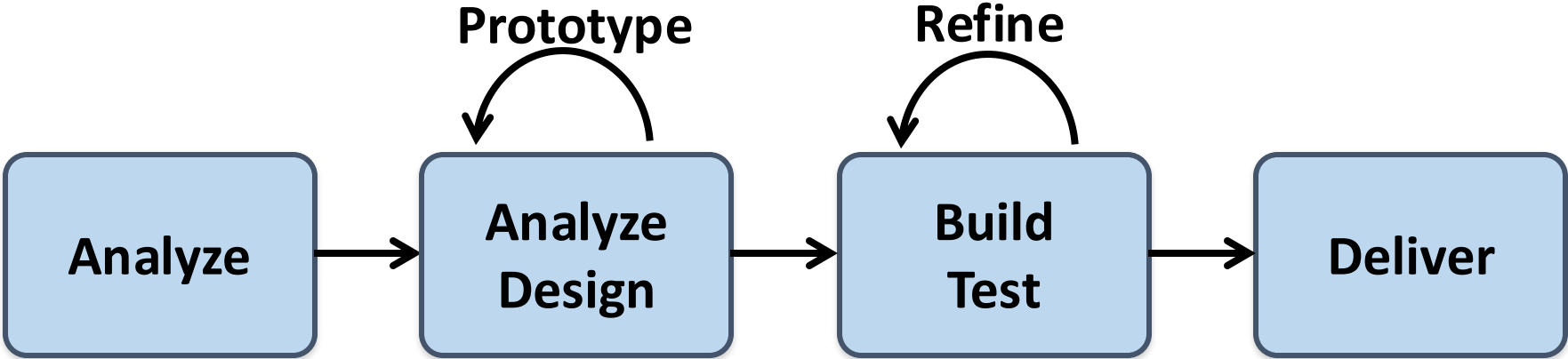




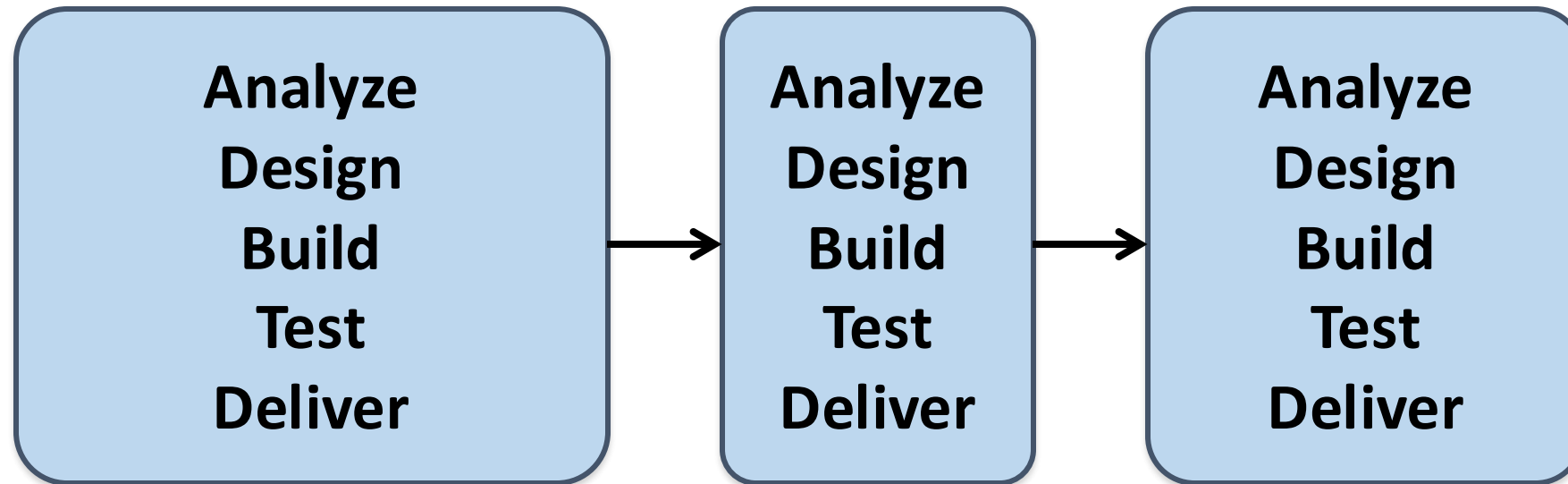
# Predictive Life Cycle



# Iterative Life Cycle

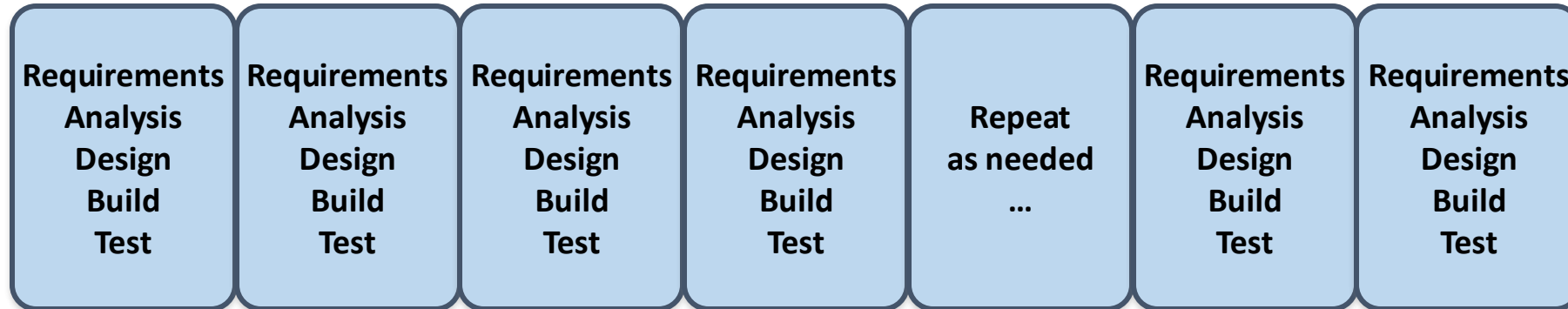


# A Life Cycle of Varying-Sized Increments

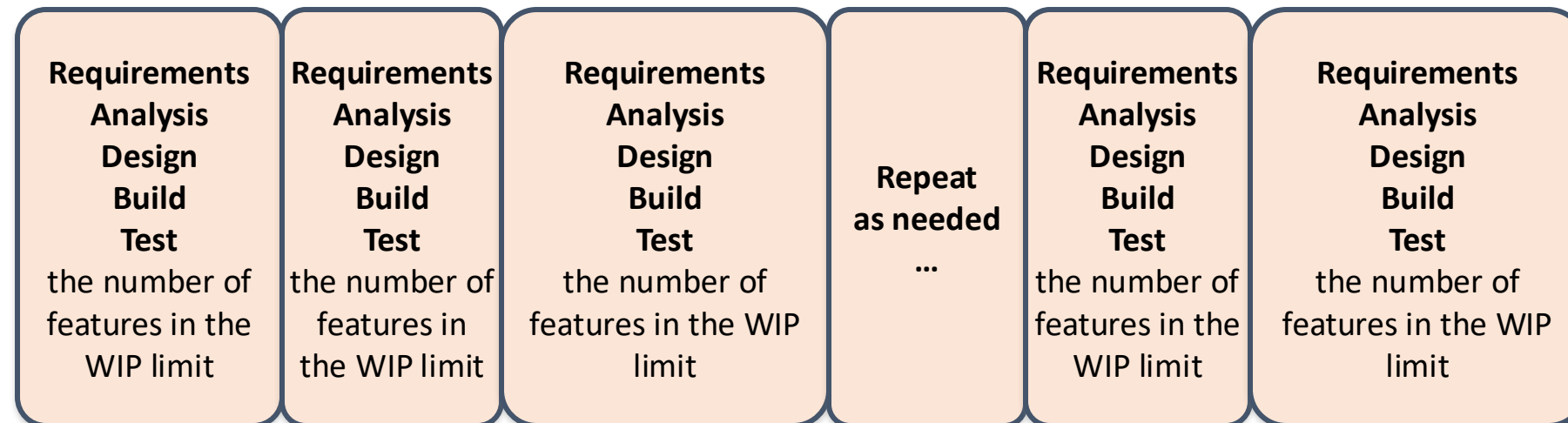


# Iteration-Based and Flow-Based Agile Life Cycles

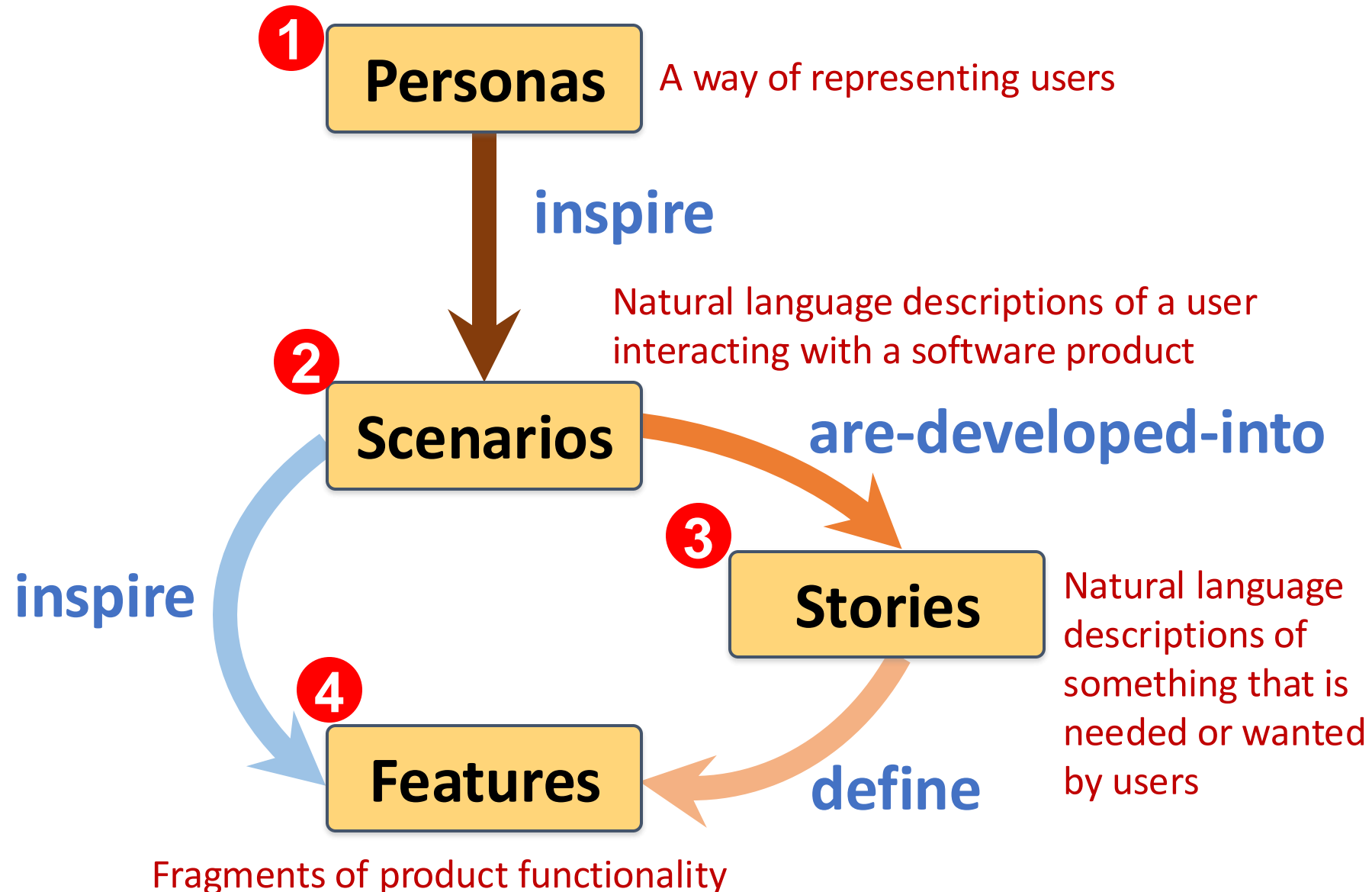
## Iteration-Based Agile



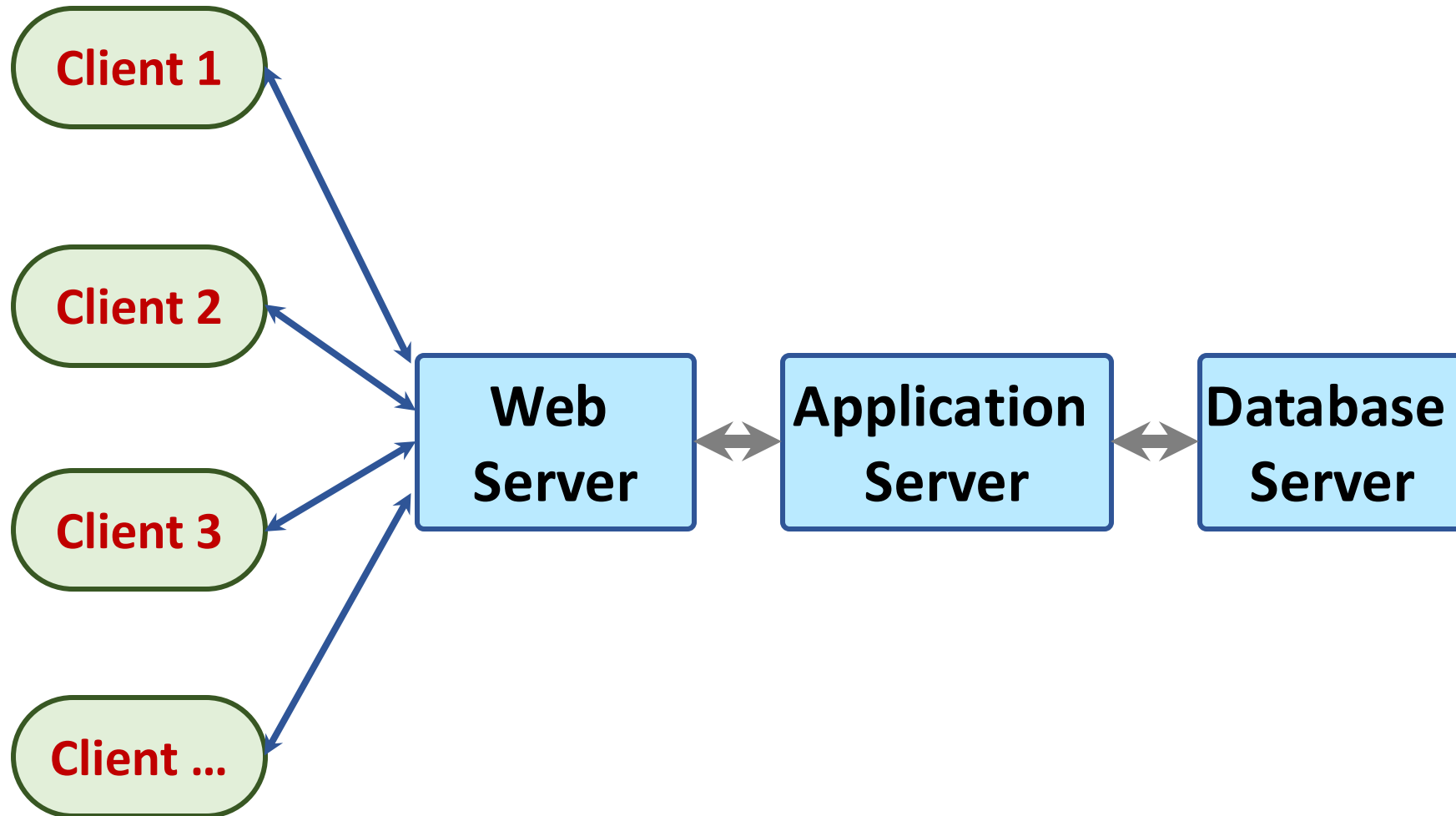
## Flow-Based Agile



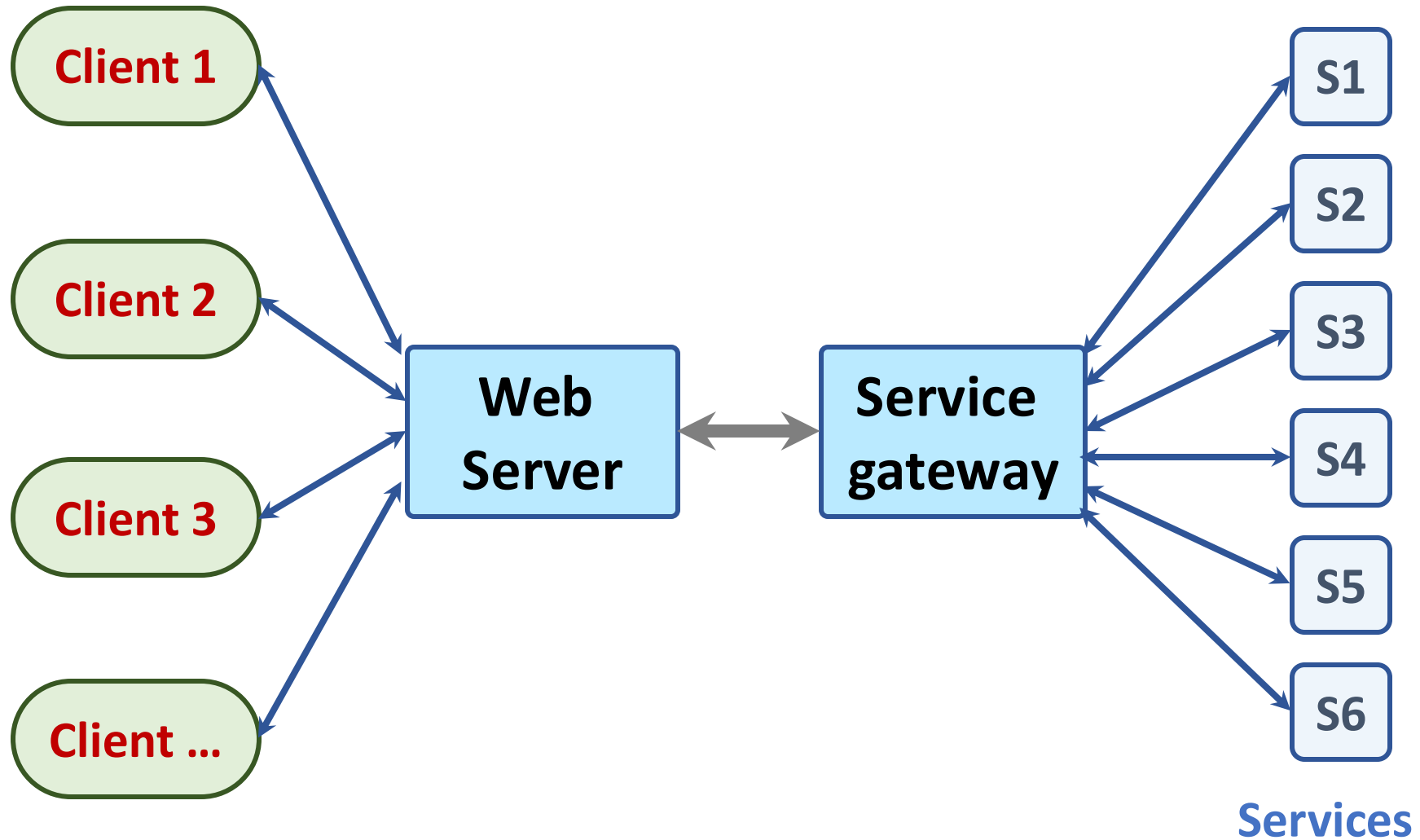
# From personas to features



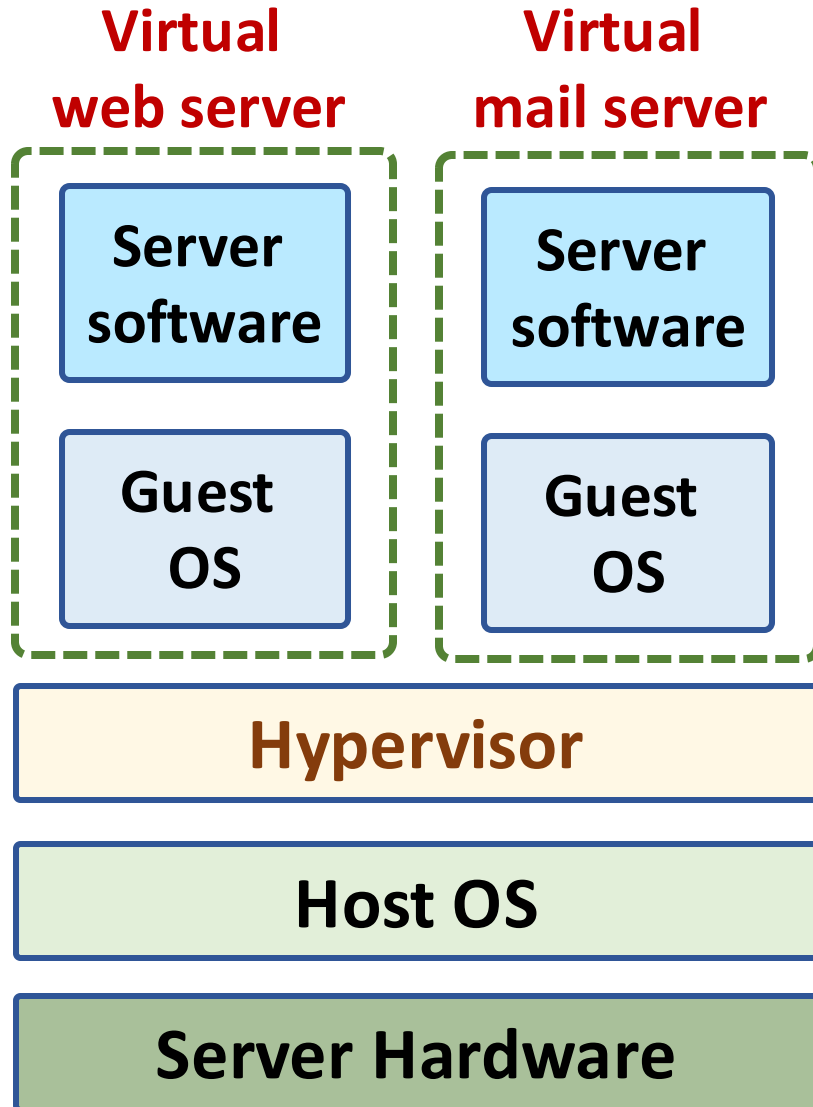
# Multi-tier client-server architecture



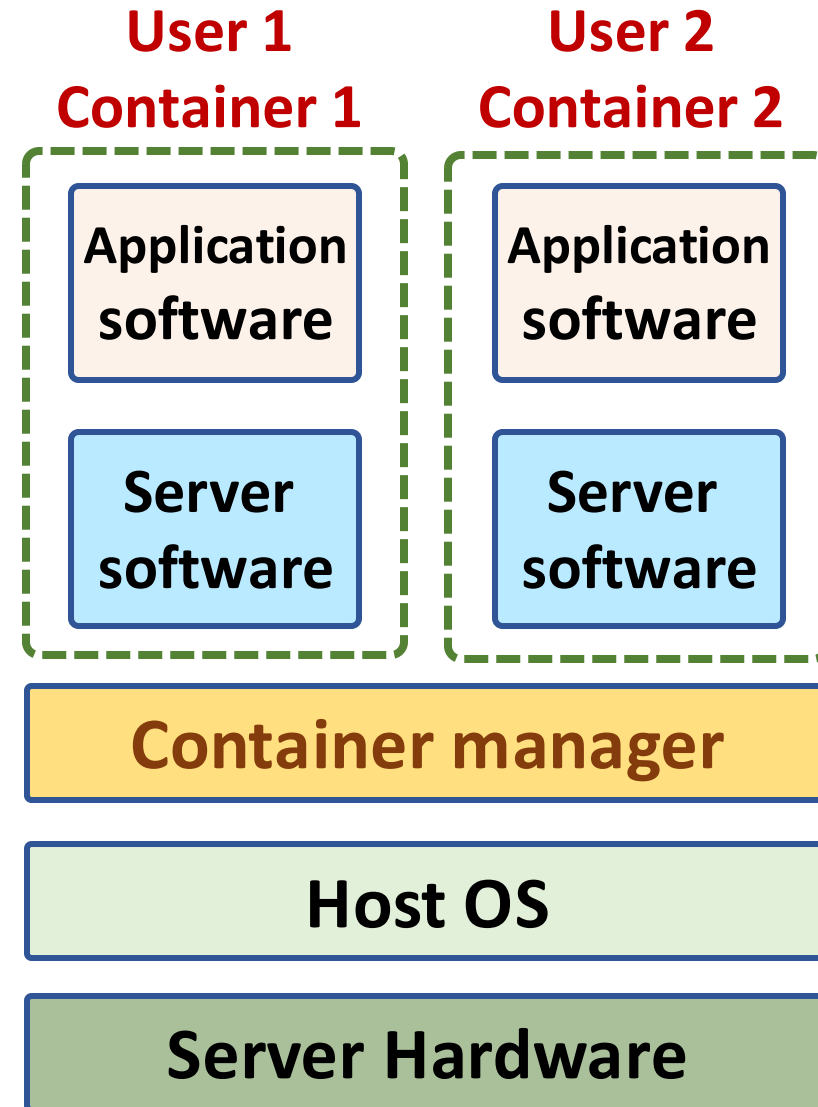
# Service-oriented Architecture



# VM

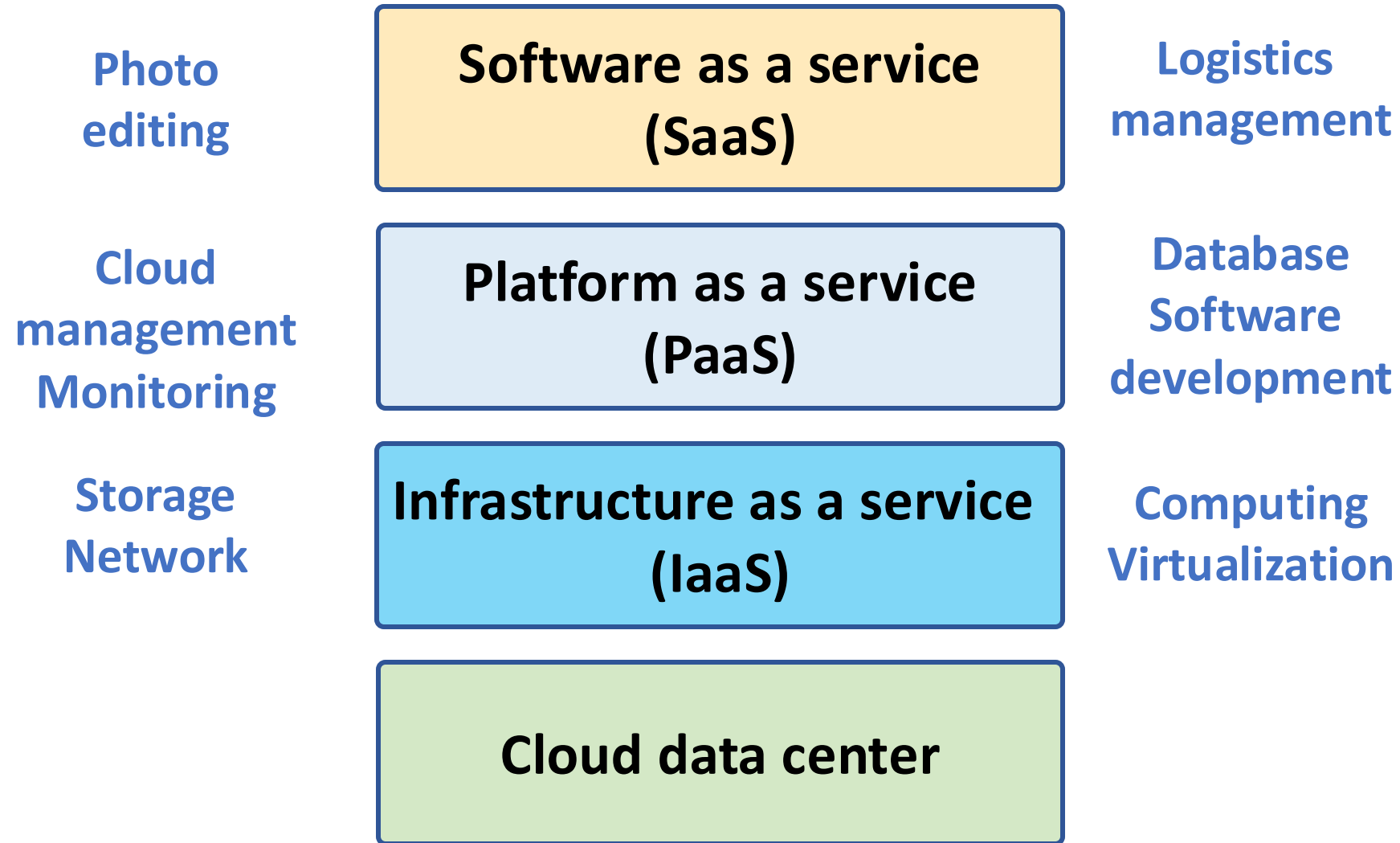


# Container

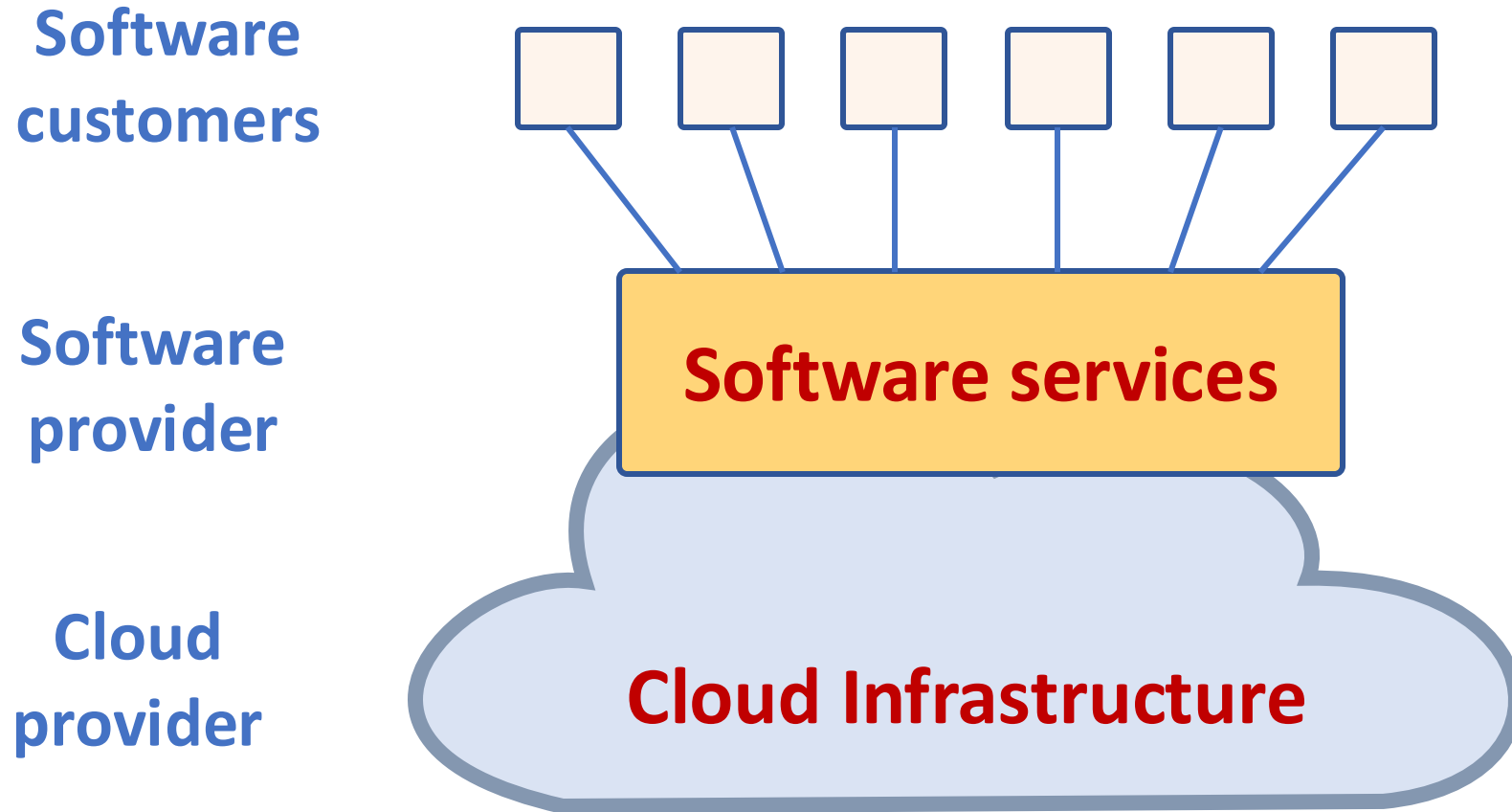




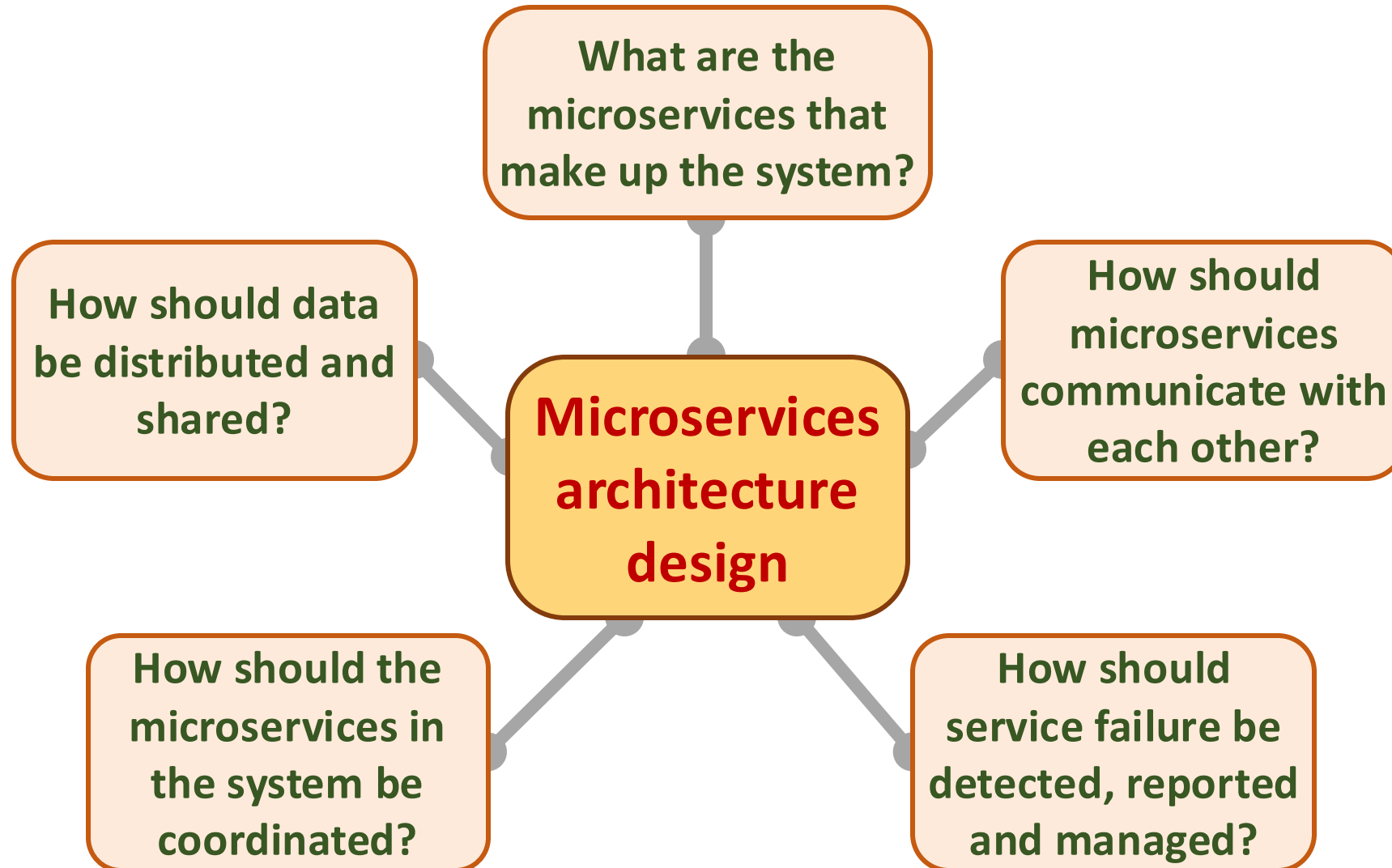
# Everything as a service



# Software as a service



# Microservices architecture – key design questions



# Types of security threat

An attacker attempts to deny access to the system for legitimate users

**Availability threats**

Distributed denial of service (DDoS) attack

An attacker attempts to damage the system or its data

**Integrity threats**

Virus

Ransomware

**SOFTWARE PRODUCT**

**PROGRAM**

**DATA**

Data theft

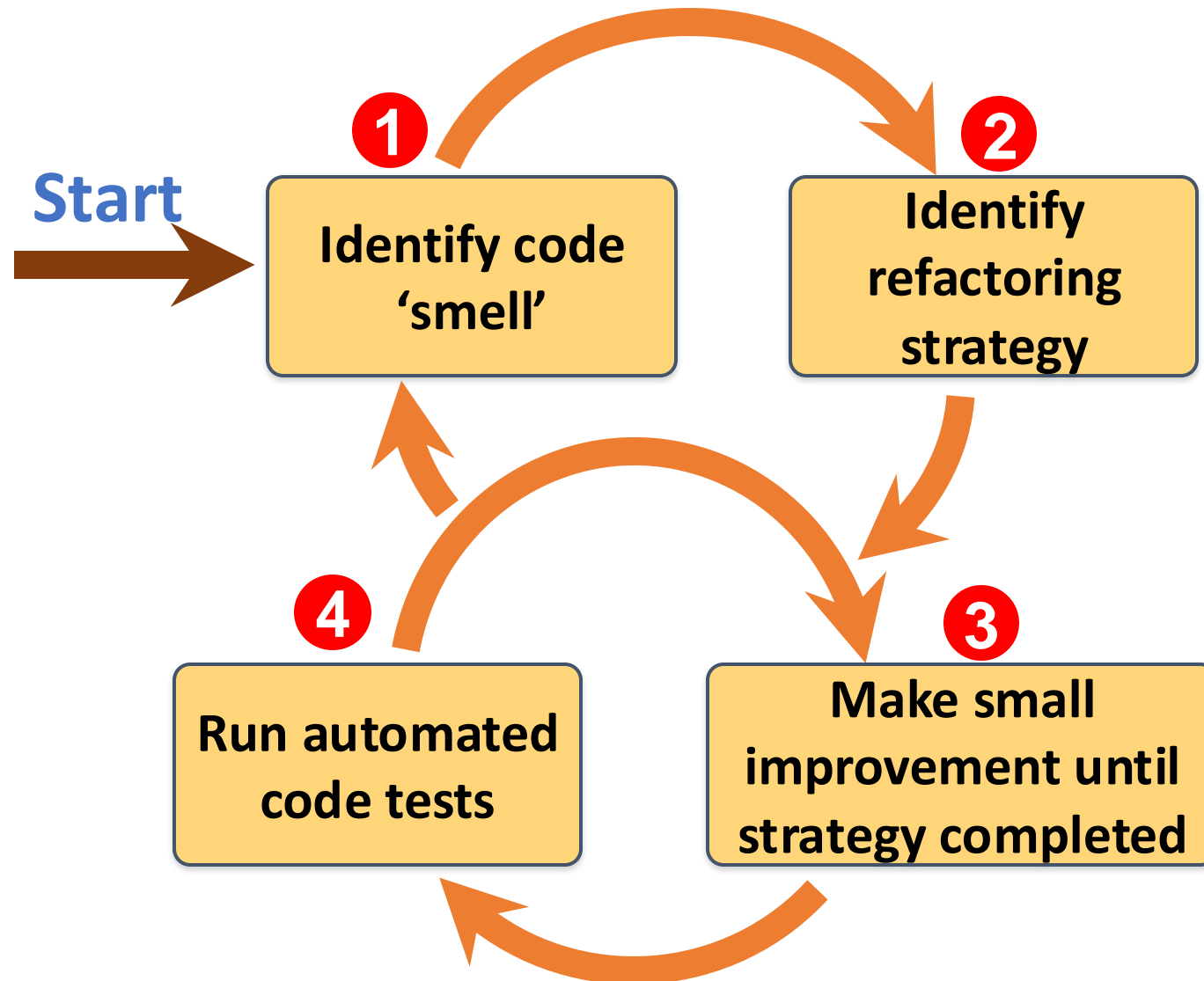
**Confidentiality threats**

An attacker tries to gain access to private information held by the system

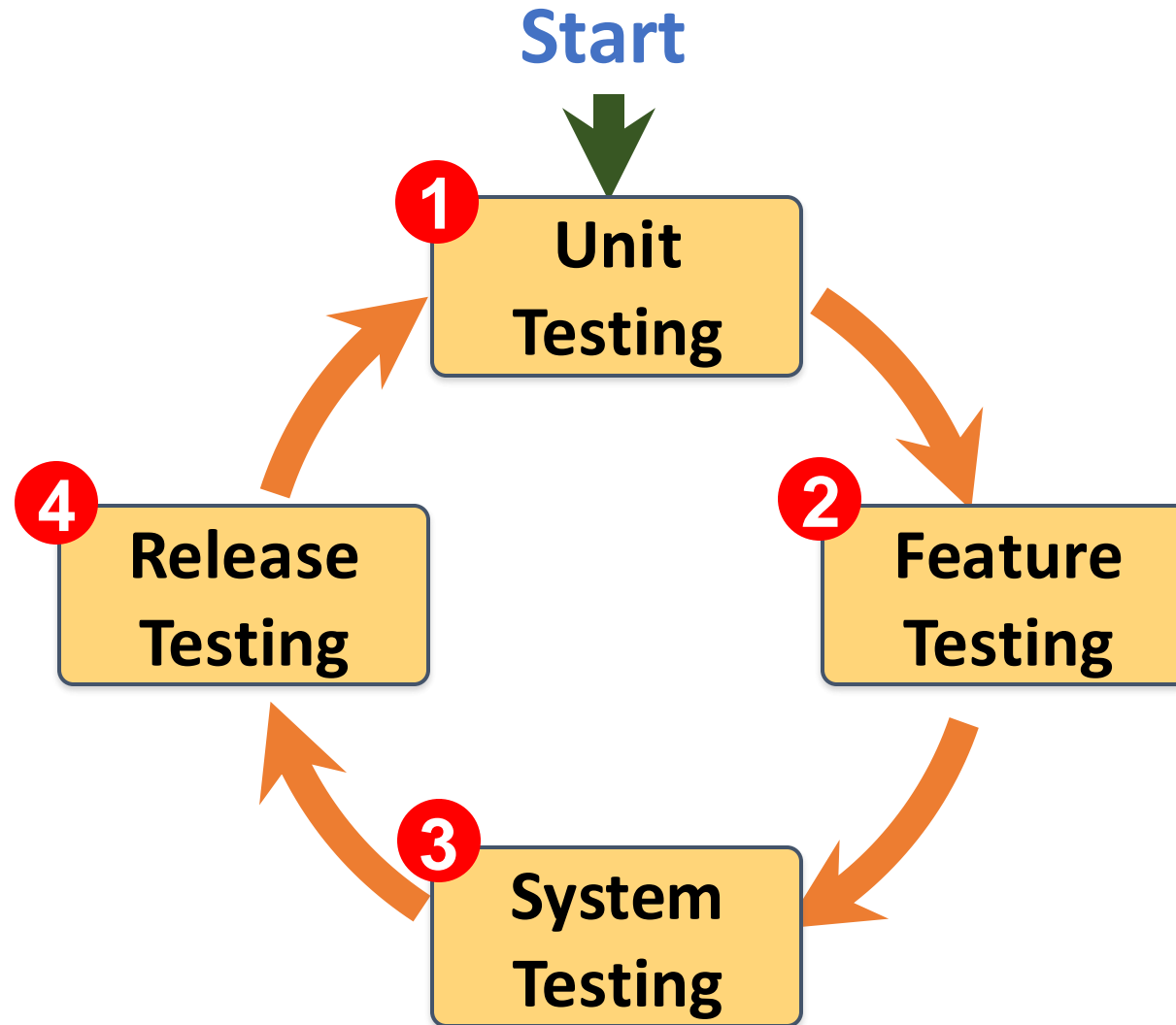
# Software product quality attributes



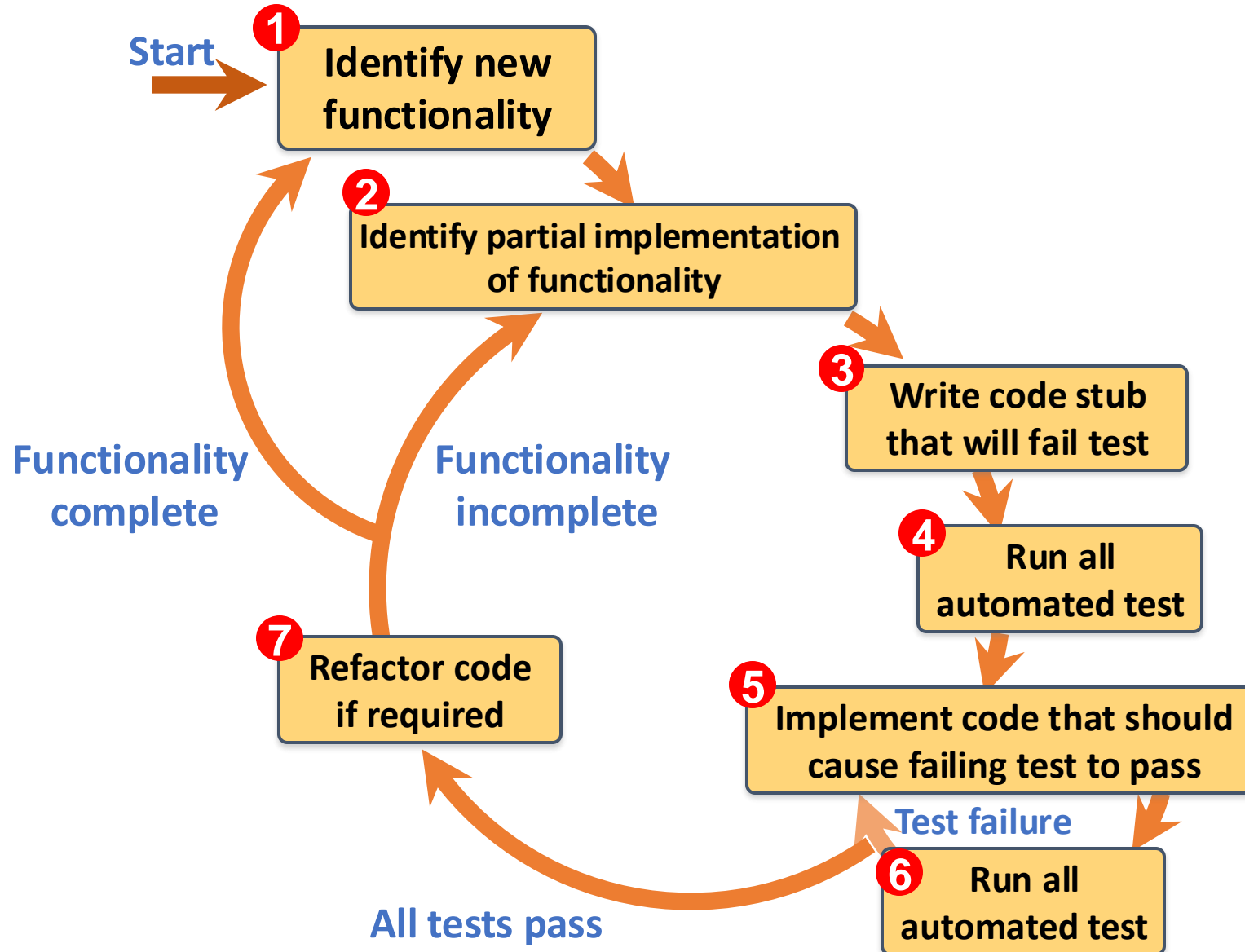
# A refactoring process



# Functional testing

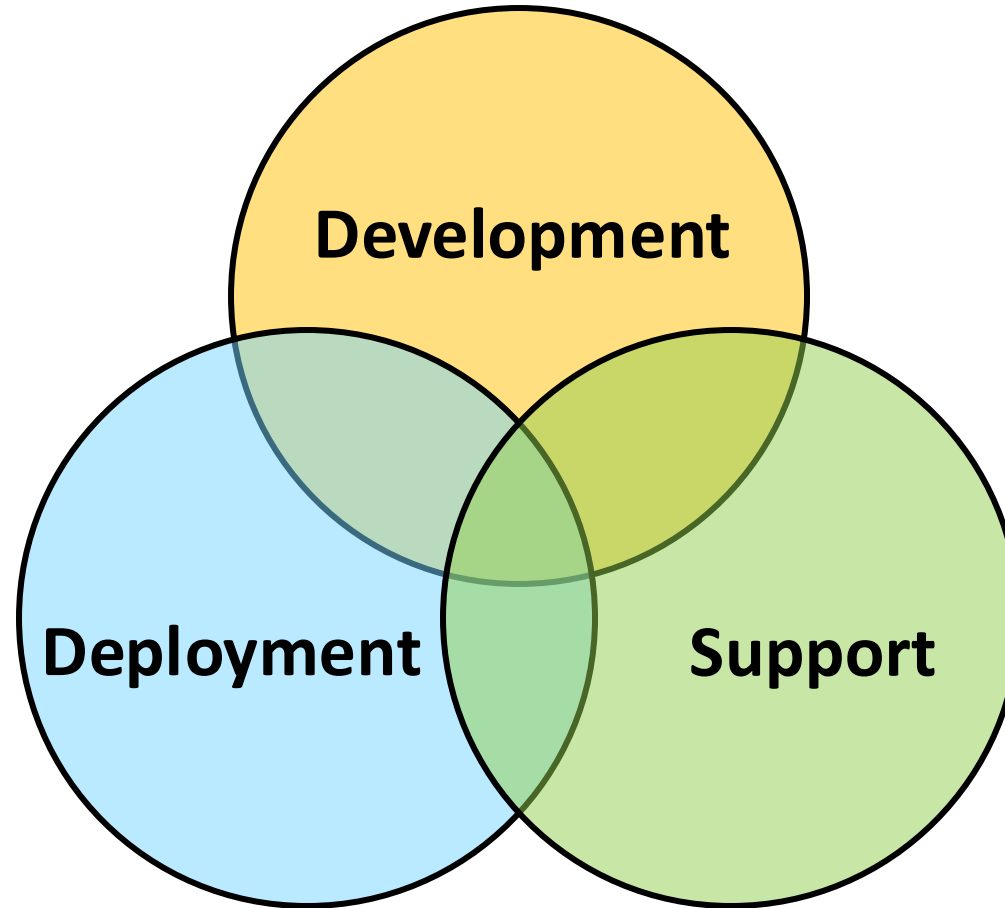


# Test-driven development (TDD)



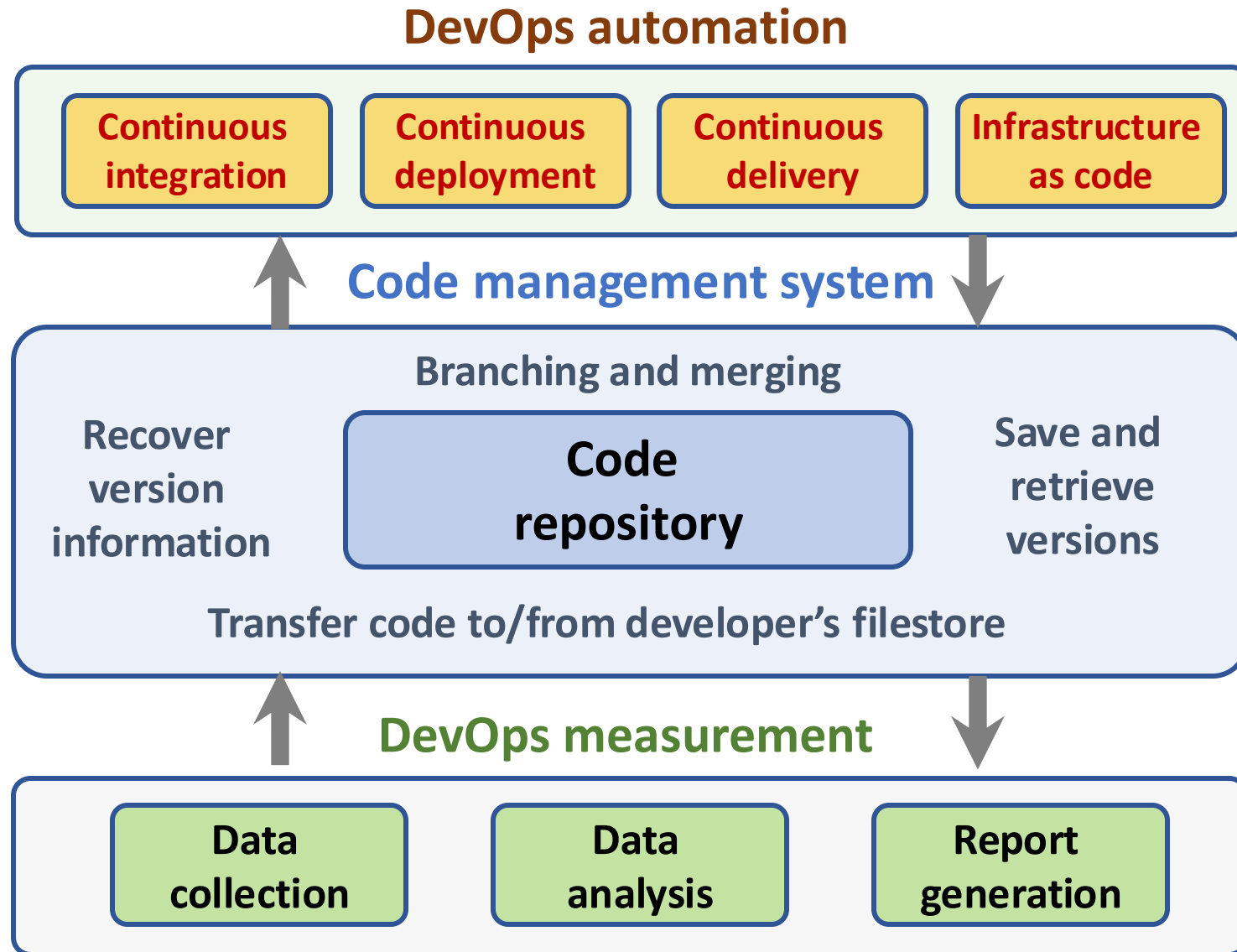


# DevOps



## Multi-skilled DevOps team

# Code management and DevOps



# **Software Architecture: Architectural design, System decomposition, and Distribution architecture**

# Agentic AI for Software Engineering Architecture

## Architectural Design

Design Pattern  
Suggestions

Technology Stack  
Recommendation

Diagram Generation

Trade-Off Analysis

## System Decomposition

Modulization Based on  
Function

Automated  
Component Design

Bottleneck Optimization

## Distribution Architecture

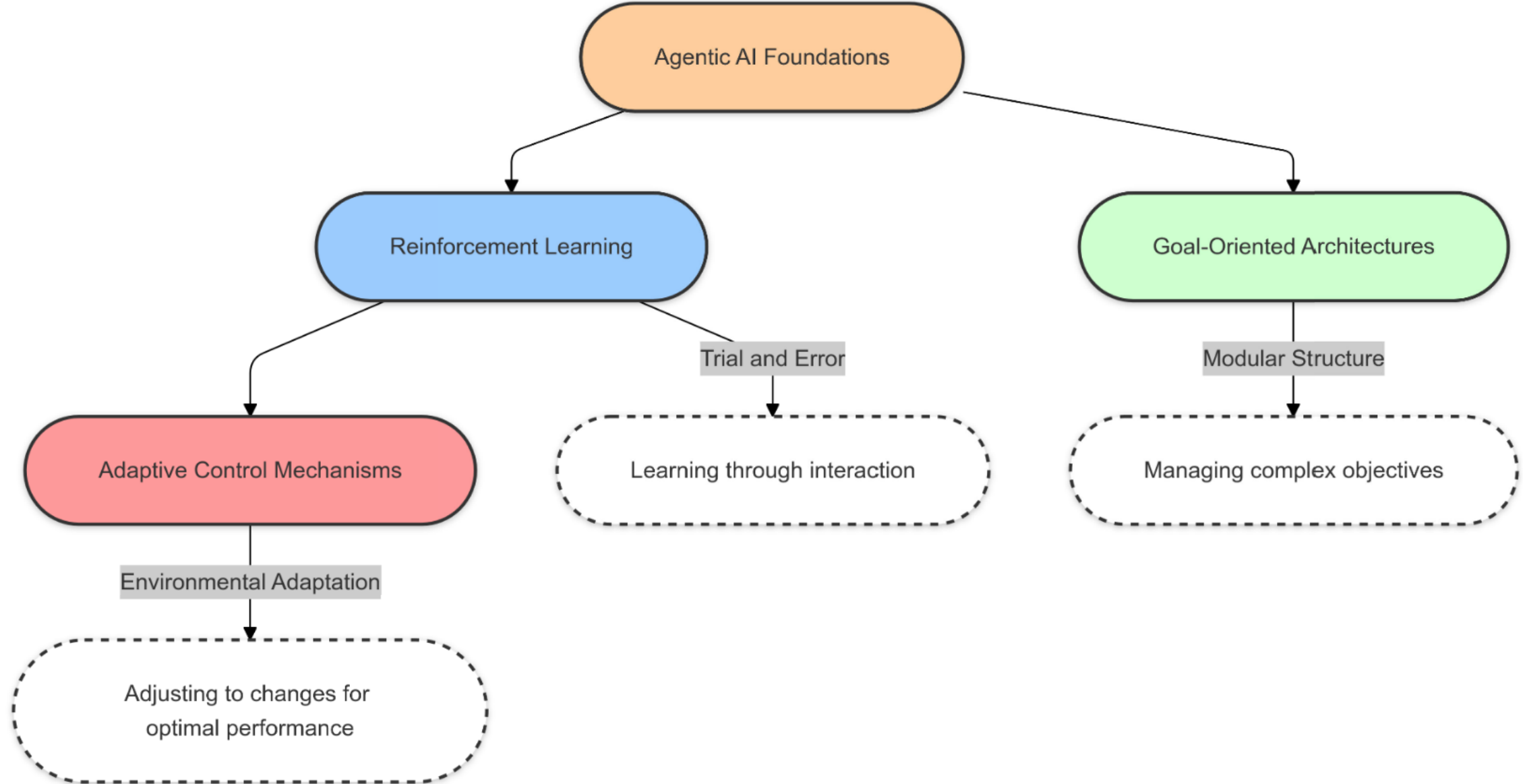
Designing  
Scalable Systems

Supporting  
Dynamic Scaling

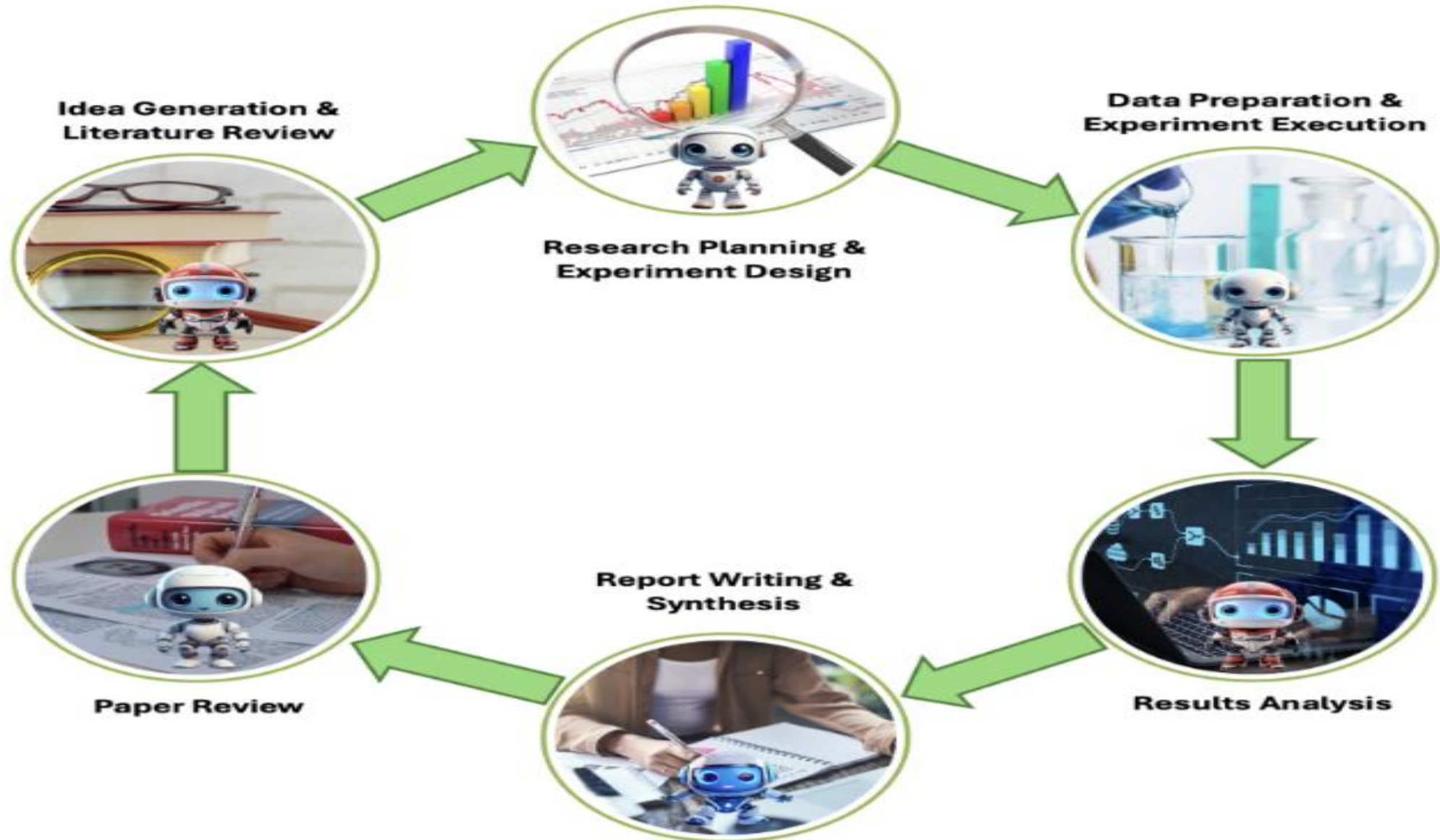
Optimizing Performance

**Key Benefits: Enhanced Efficiency, Data-Driven Decisions, Scalability**

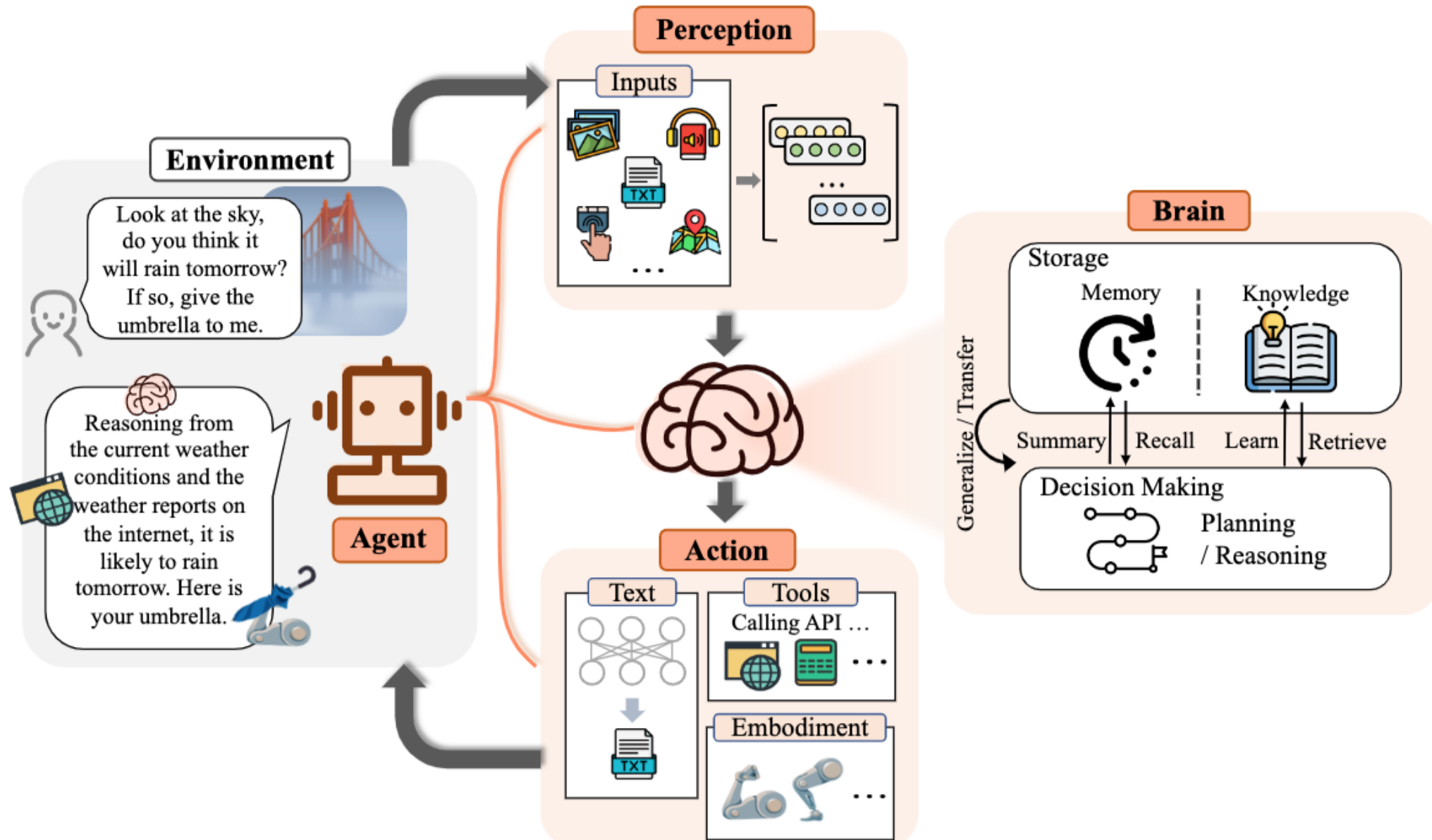
# Agentic AI Foundations



# Agentic AI Workflow for Scientific Discovery

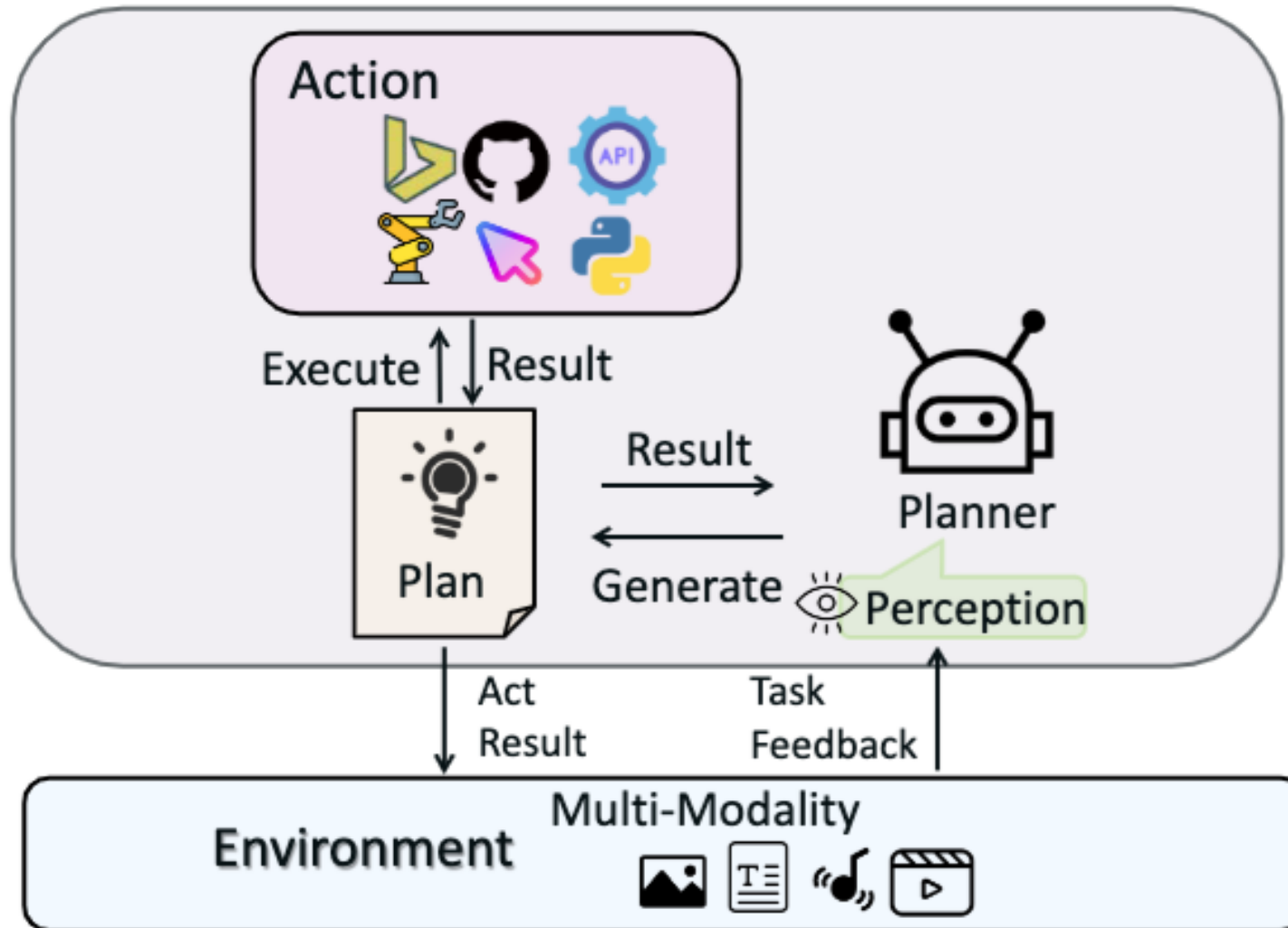


# Large Language Model (LLM) Based Agents



# Large Multimodal Agents (LMAs)

(a) Type I: **Closed-source LLMs as Planners** w/o Longterm Memory.

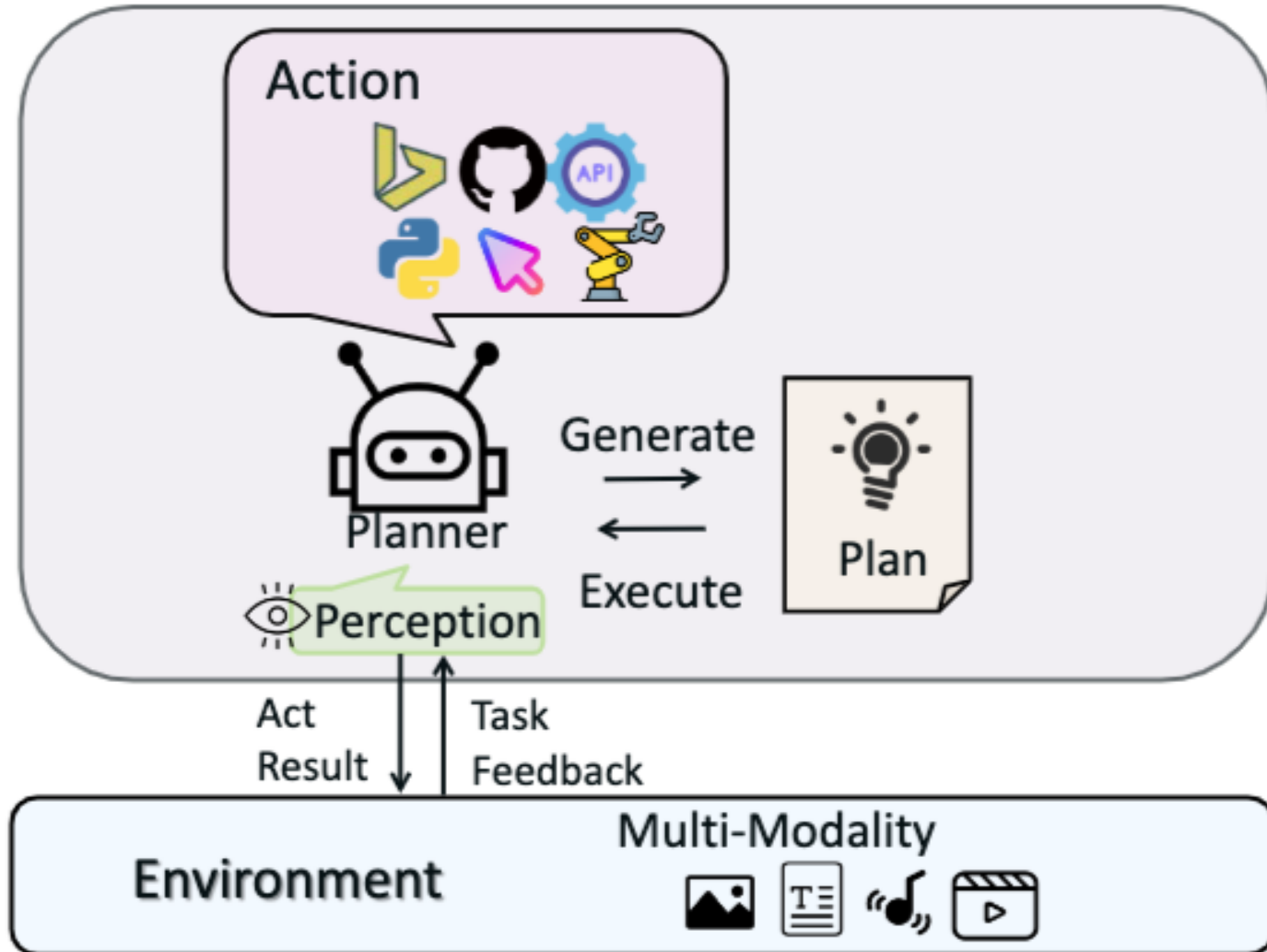


Use prompt techniques to guide closed-source LLMs in decision-making and planning to complete tasks without long memory.



# Large Multimodal Agents (LMAs)

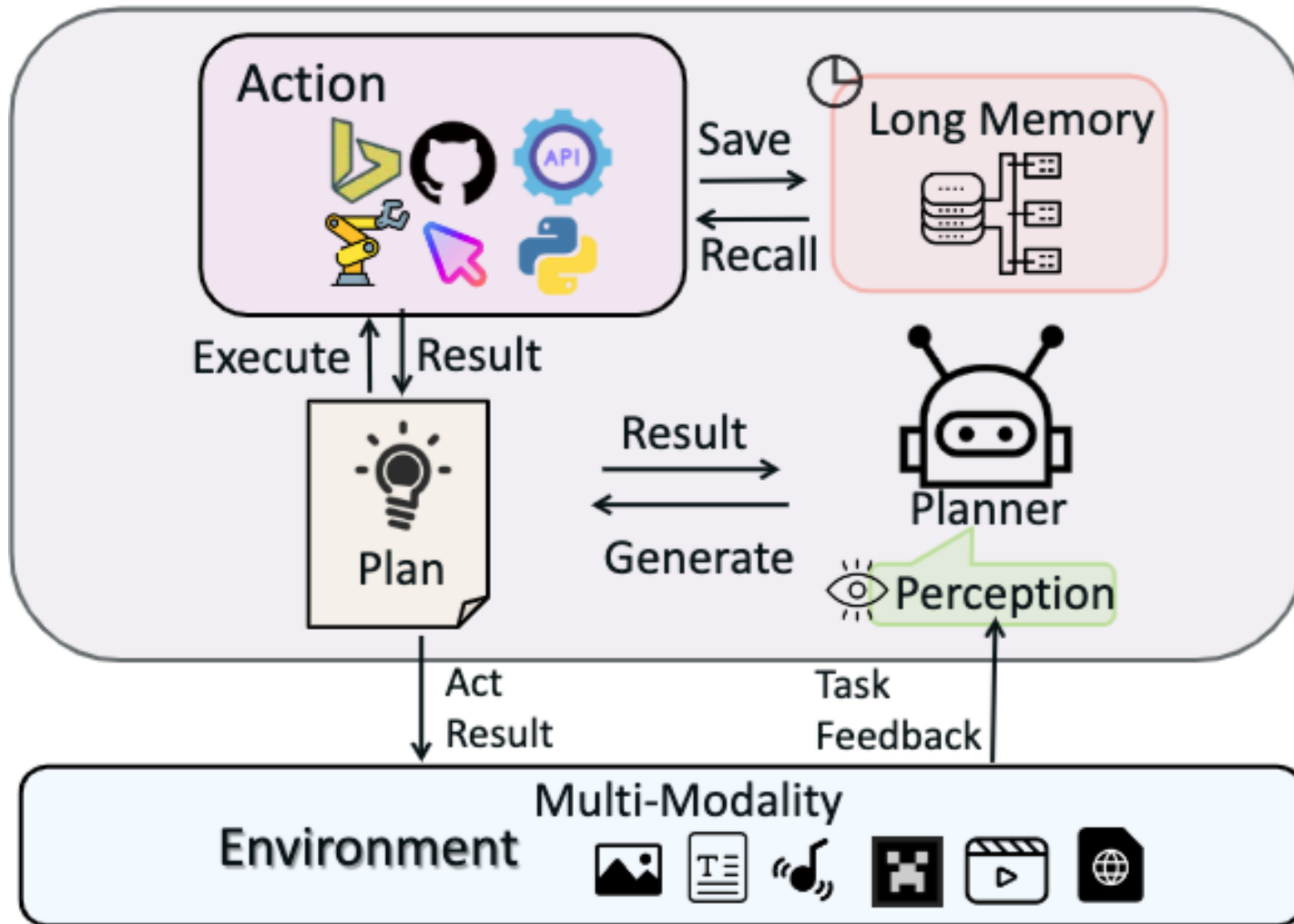
(b) Type II: **Finetuned LLMs** as Planners w/o long-term Memory.



Use action-related data to finetune existing open-source large models, enabling them to achieve decision-making, planning, and tool invocation capabilities comparable to closed-source LLMs

# Large Multimodal Agents (LMAs)

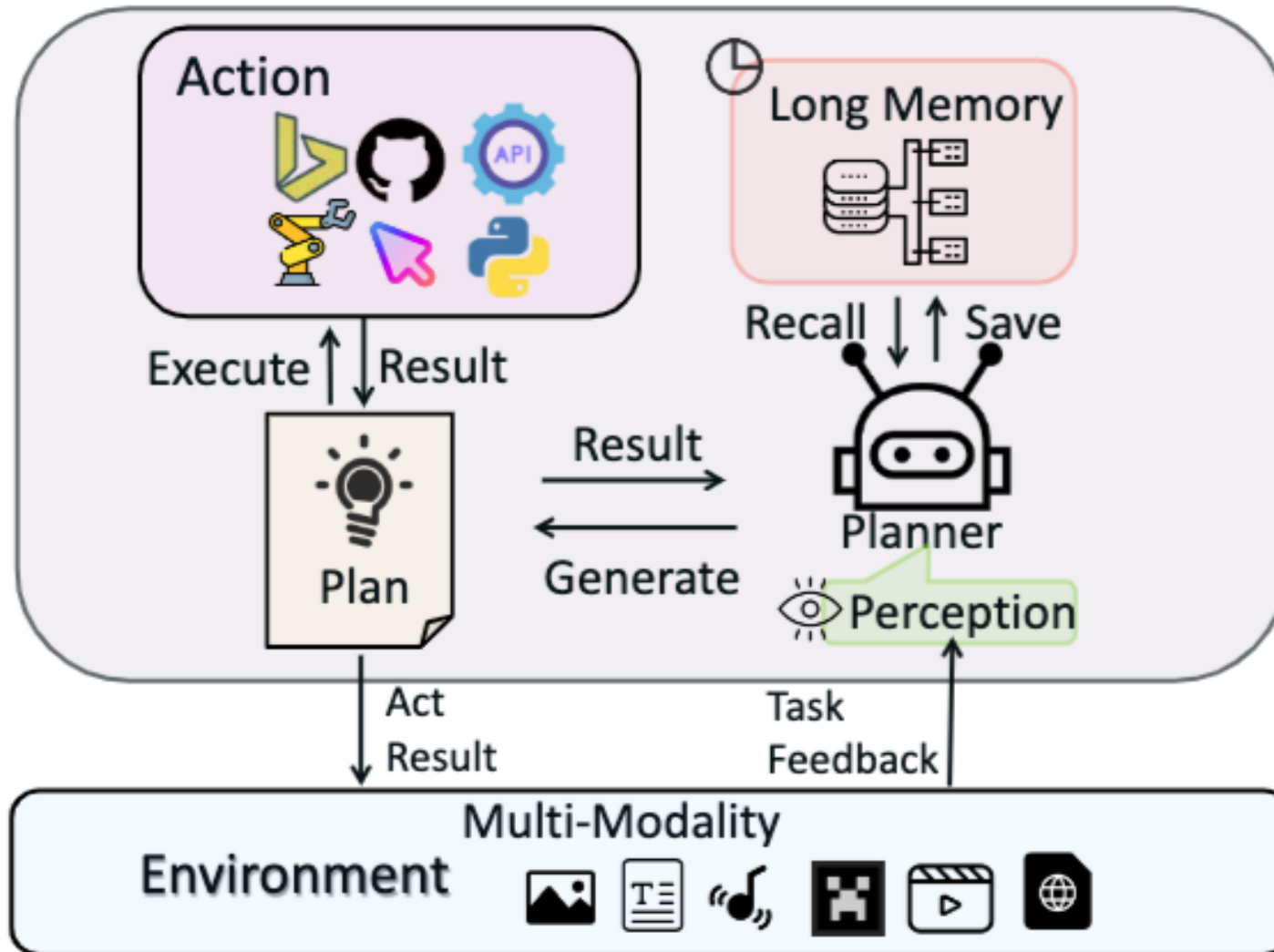
## (c) Type III: Planners with **Indirect Long-term Memory**



Introduce indirect long-term memory functions, further enhancing their generalization and adaptation abilities in environments closer to the real world.

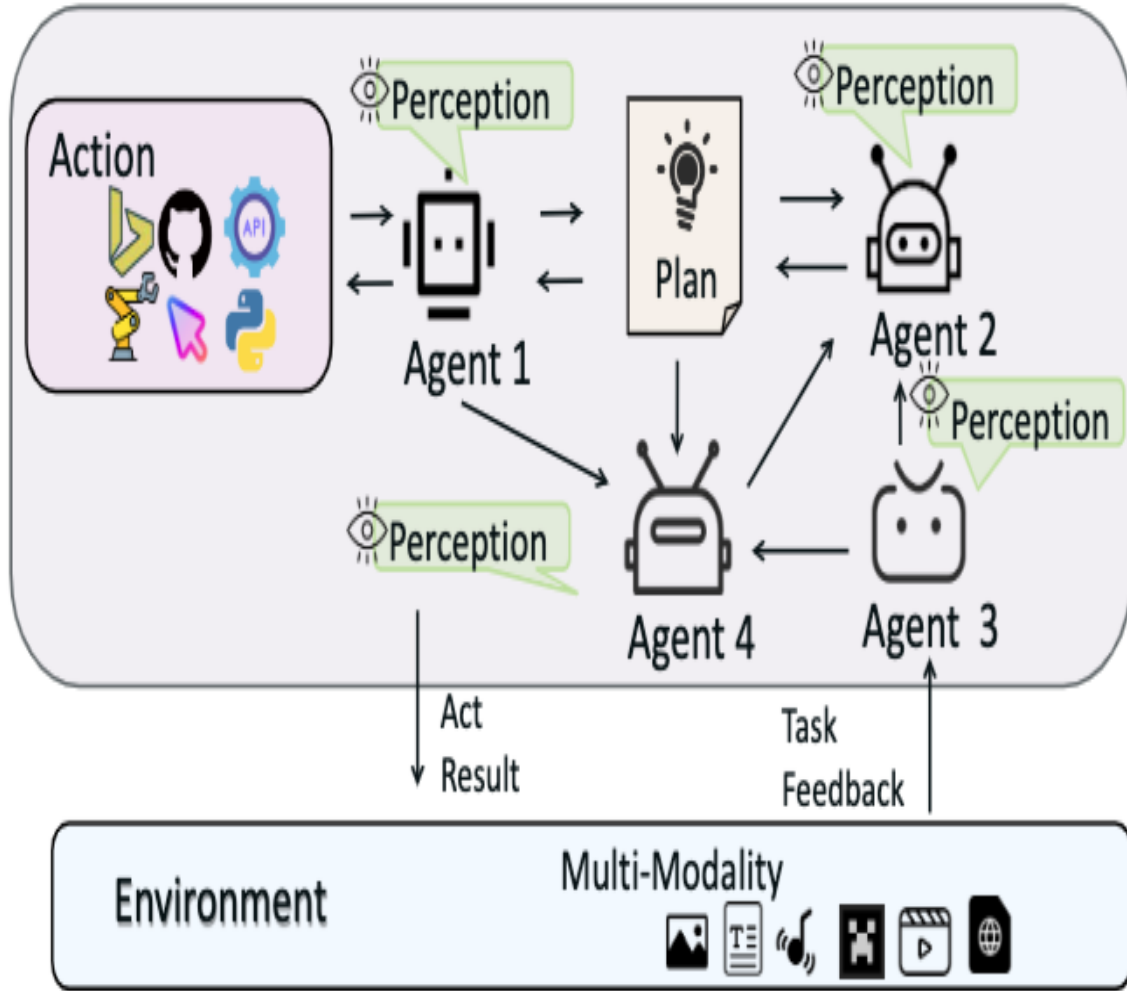
# Large Multimodal Agents (LMAs)

## (d) Type IV: Planners with **Native Long-term Memory**

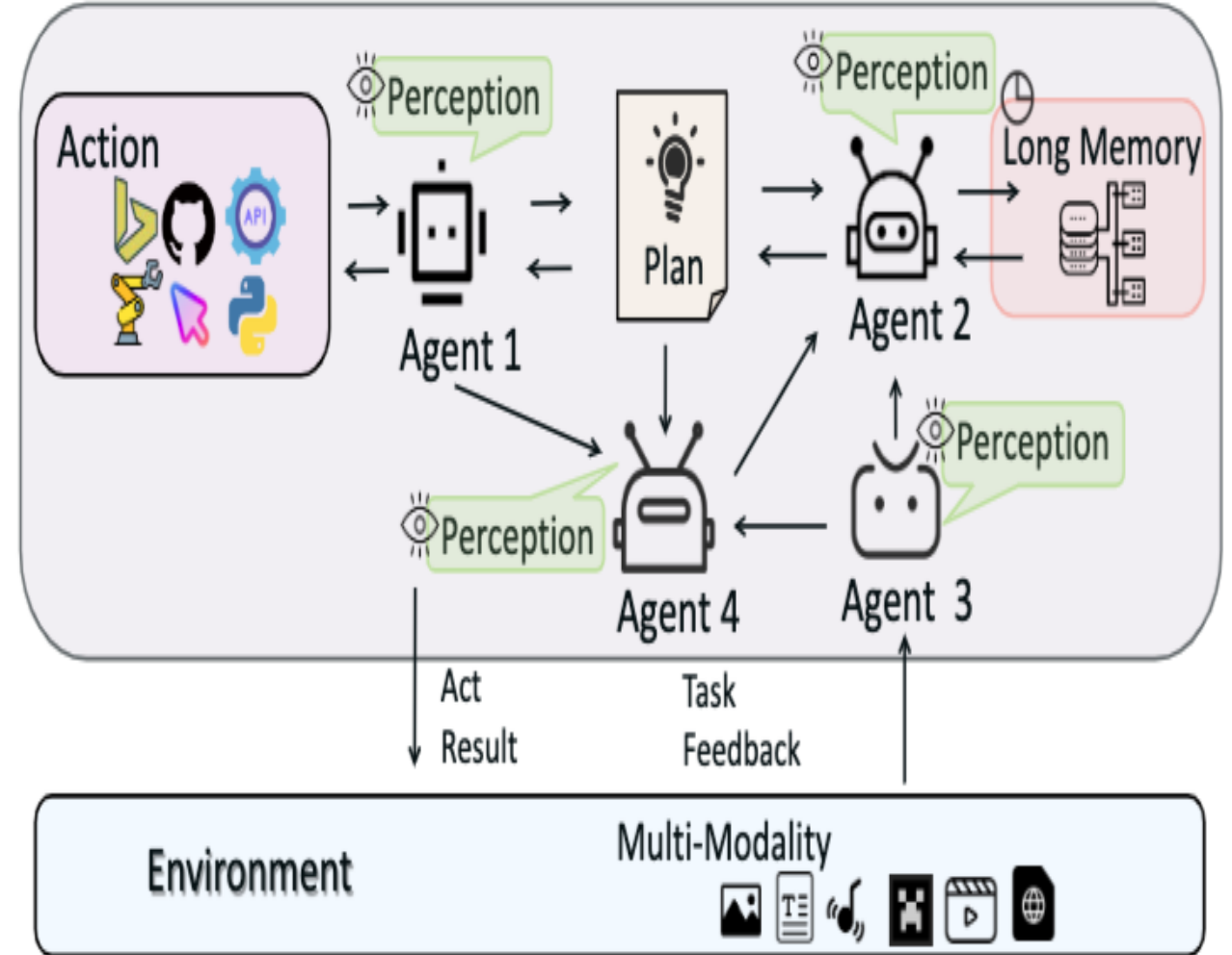


Introduce native long-term memory functions, further enhancing their generalization and adaptation abilities in environments closer to the real world.

# Multi-Agent Frameworks



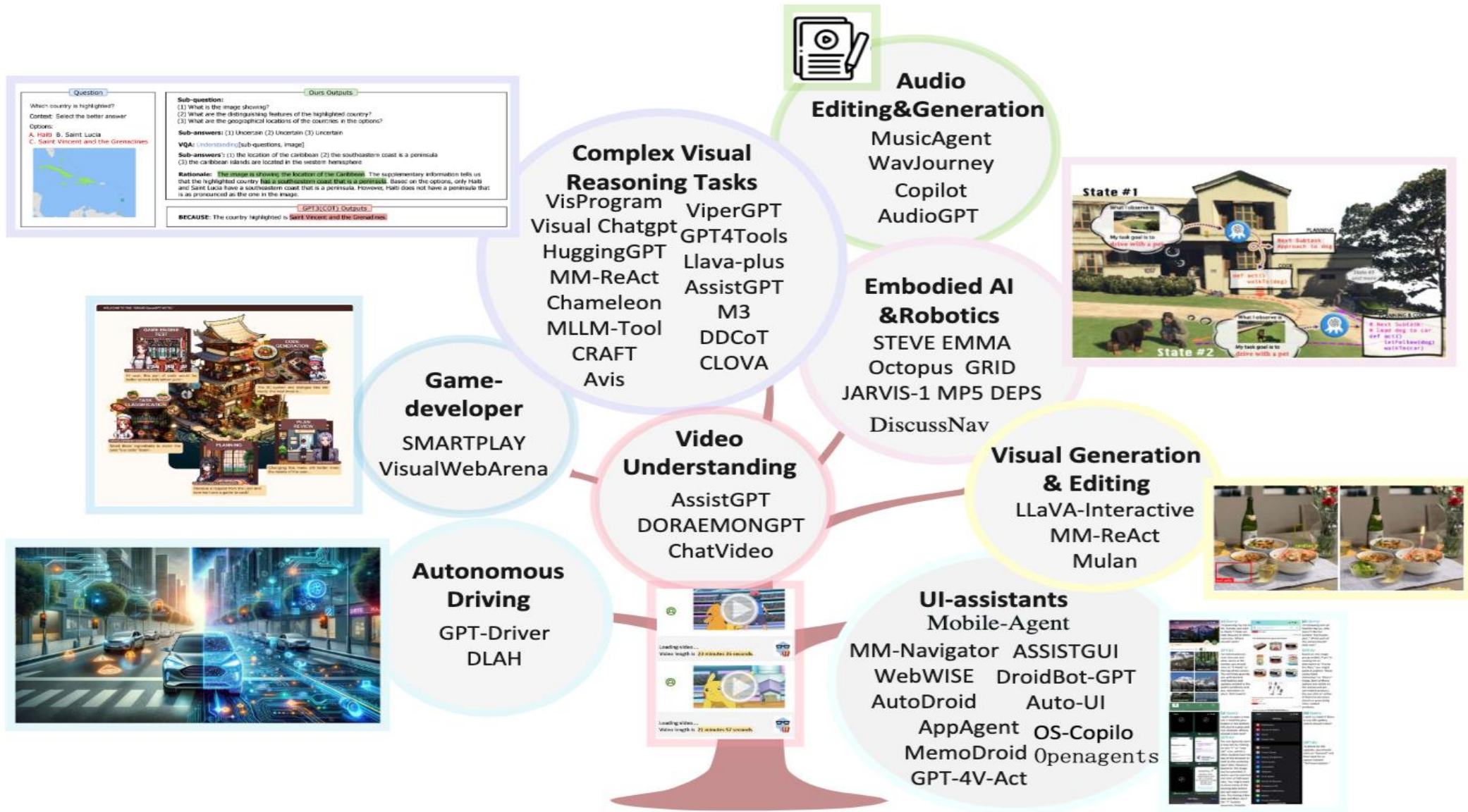
(a)



(b)

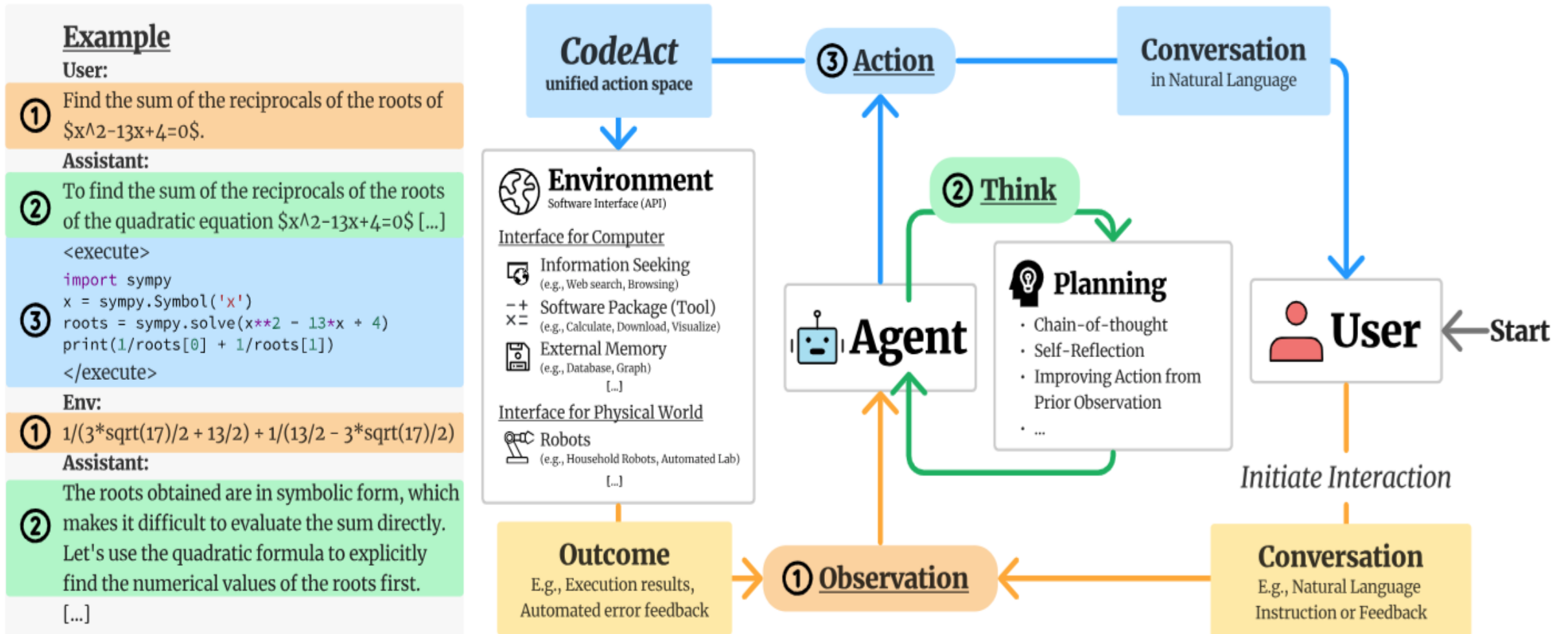


# Applications of Large Multimodal Agents (LMAs)



# CodeActAgent:

## General Agent Multi-turn Interaction Framework



# Software architecture

- To create a **reliable, secure and efficient** product, you need to pay attention to **architectural design** which includes:
  - its **overall organization**,
  - how the software is **decomposed into components**,
  - the **server organization**
  - the **technologies** that you use to build the software. The architecture of a software product affects its **performance, usability, security, reliability and maintainability**.

# Software architecture

- There are many different interpretations of the term **'software architecture'**.
  - Some focus on **'architecture'** as a **noun**
    - the **structure of a system**
  - and others consider **'architecture'** to be a **verb**
    - the **process of defining these structures.**



# The IEEE definition of software architecture

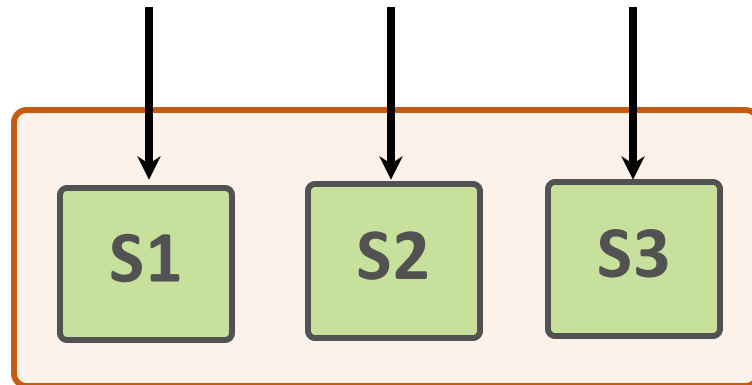
- **Architecture** is the fundamental organization of a software system embodied in its **components**, their **relationships** to each other and to the **environment**, and the **principles** guiding its design and evolution.

# Software architecture and components

- A **component** is an element that implements a coherent set of functionality or features.
- **Software component** can be considered as a **collection of one or more services** that may be used by other components.
- When **designing software architecture**, **you don't have to decide how an architectural element or component is to be implemented.**
- Rather, you **design the component interface** and leave the implementation of that interface to a later stage of the development process.

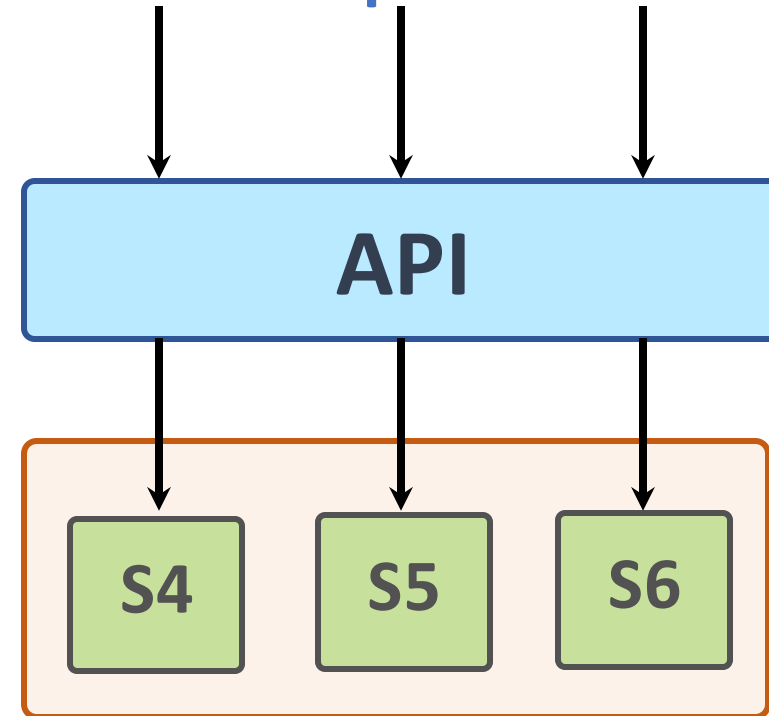
# Access to services provided by software components

Services accessed directly by other components



**Component 1**

Services accessed through the component API



**Component 2**

# Why is architecture important?

- **Architecture** is important because the architecture of a system has a fundamental influence on the **non-functional system properties**.
- **Architectural design** involves understanding the issues that affect the architecture of your product and creating an architectural description that shows the **critical components and their relationships**.
- **Minimizing complexity** should be an important goal for architectural designers.

# Non-functional system quality attributes

- **Responsiveness**

Does the system return results to users in a reasonable time?

- **Reliability**

Do the system features behave as expected by both developers and users?

- **Availability**

Can the system deliver its services when requested by users?

- **Security**

Does the system protect itself and users' data from unauthorized attacks and intrusions?

# Non-functional system quality attributes

- **Usability**

Can system users access the features that they need and use them quickly and without errors?

- **Maintainability**

Can the system be readily updated and new features added without undue costs?

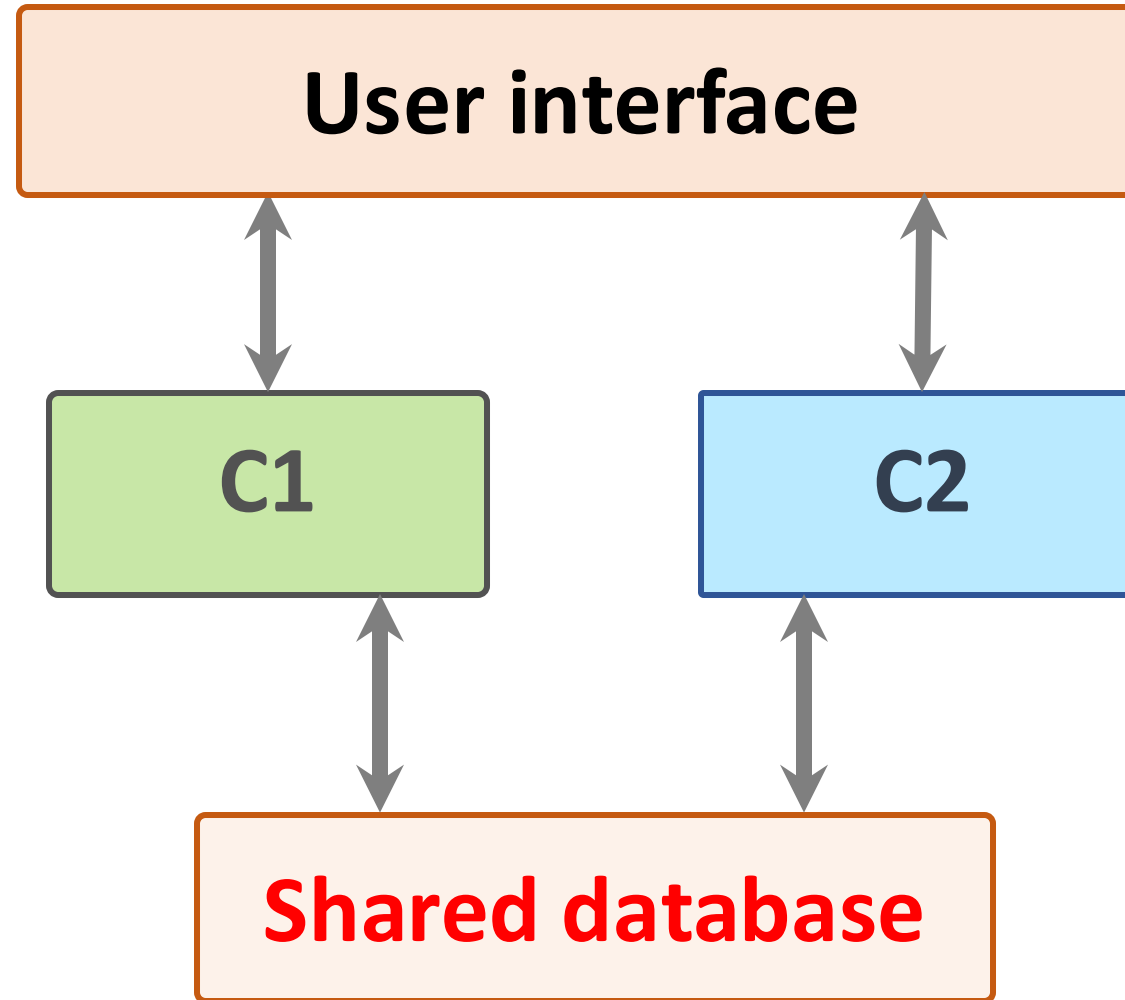
- **Resilience**

Can the system continue to deliver user services in the event of partial failure or external attack?

# Centralized security architectures

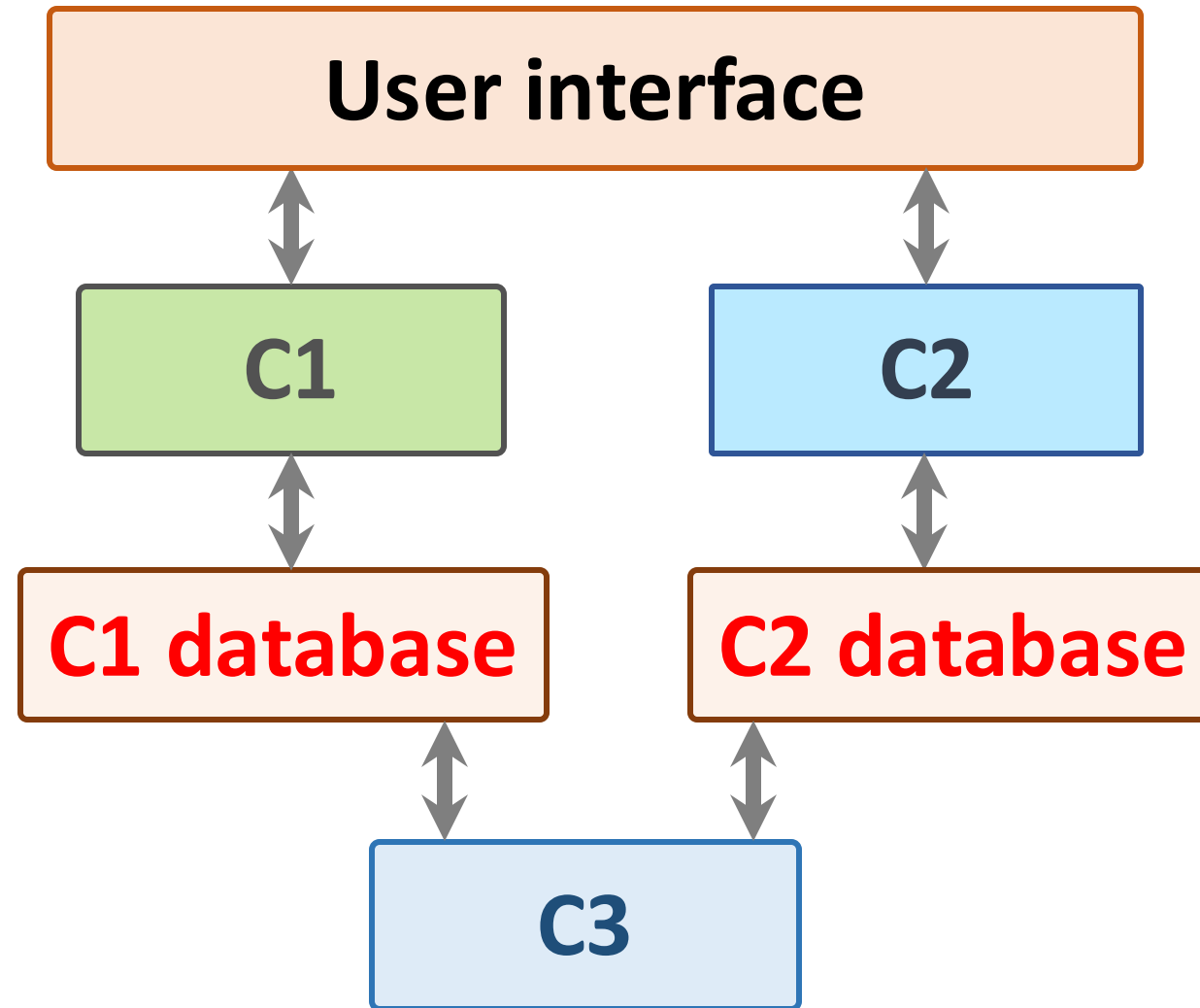
- The benefits of a **centralized security architecture** are that it is **easier to design and build protection** and that the protected information can be accessed more efficiently.
- However, if your security is breached, you lose everything.
- If you **distribute information**, it takes longer to access all of the information and **costs more to protect it**.
- If security is breached in one location, you only lose the information that you have stored there.

# Shared database architecture





# Multiple database architecture

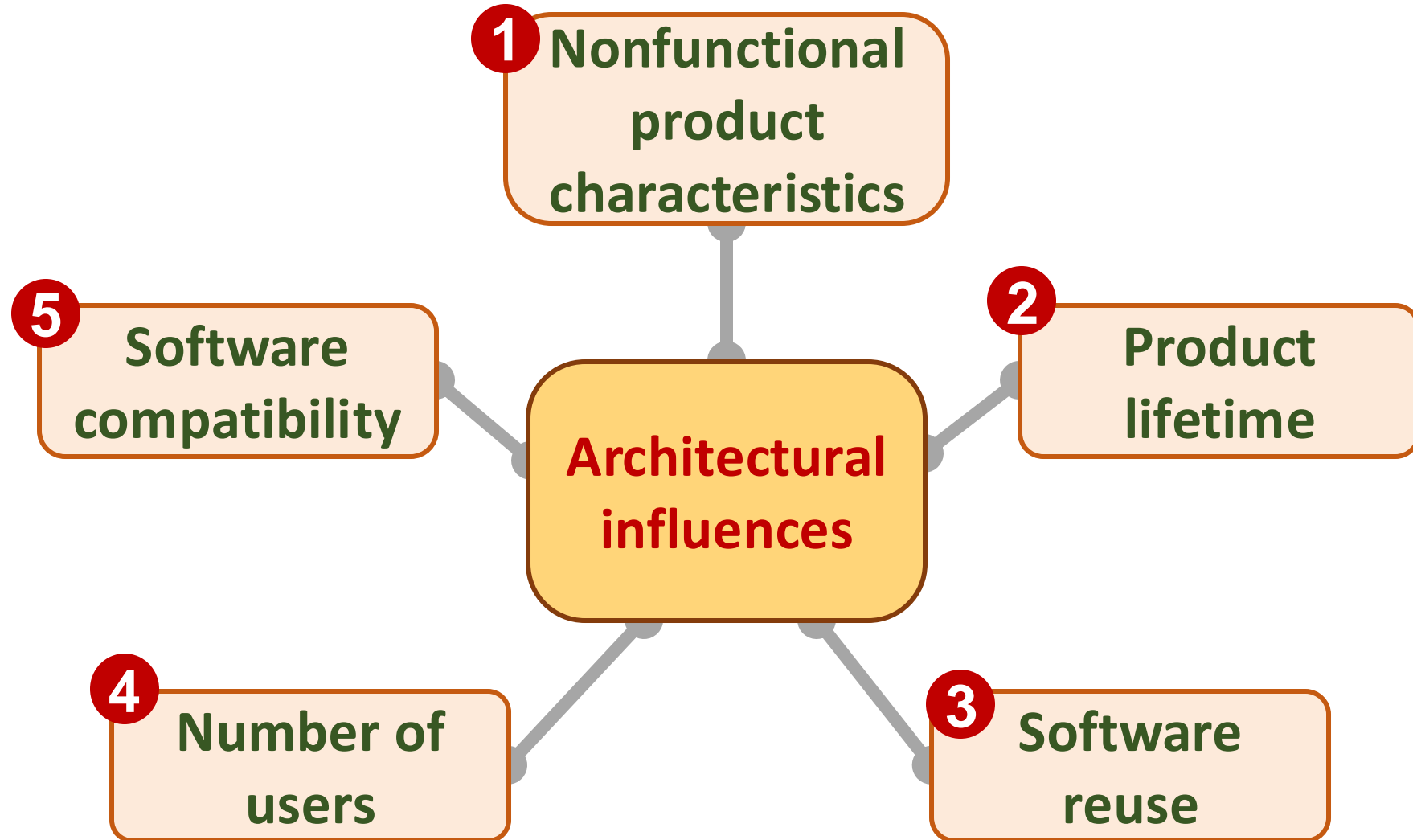


**Database reconciliation**

# Maintainability and performance

- **Shared database architecture:**
  - system with two components (C1 and C2) that share a common database.
- **Multiple database architecture:**
  - each component has its own copy of the parts of the database that it needs.
  - If one component needs to change the database organization, this does not affect the other component.
- A multi-database architecture may run more slowly and may cost more to implement and change.
  - A multi-database architecture needs a mechanism (component C3) to ensure that the data shared by C1 and C2 is kept consistent when it is changed.

# Issues that influence architectural decisions



# The importance of architectural design issues

1

- **Nonfunctional product characteristics**

**Nonfunctional product characteristics such as security and performance affect all users.**

**If you get these wrong,  
your product will be unlikely to be a commercial success.  
Unfortunately, some characteristics are opposing,  
so you can only optimize the most important.**

# The importance of architectural design issues

2

- **Product lifetime**

If you anticipate a long product lifetime, you will need to create regular product revisions. You therefore need an architecture that is evolvable, so that it can be adapted to accommodate new features and technology.

# The importance of architectural design issues

3

- **Software reuse**

**You can save a lot of time and effort, if you can reuse large components from other products or open-source software. However, this constrains your architectural choices because you must fit your design around the software that is being reused.**

# The importance of architectural design issues

- 4**
  - **Number of users**

If you are developing consumer software delivered over the Internet, the number of users can change very quickly. This can lead to serious performance degradation unless you design your architecture so that your system can be quickly scaled up and down.

# The importance of architectural design issues

5

- **Software compatibility**

For some products, it is important to maintain compatibility with other software so that users can adopt your product and use data prepared using a different system. This may limit architectural choices, such as the database software that you can use.



# Trade off:

## Maintainability vs performance

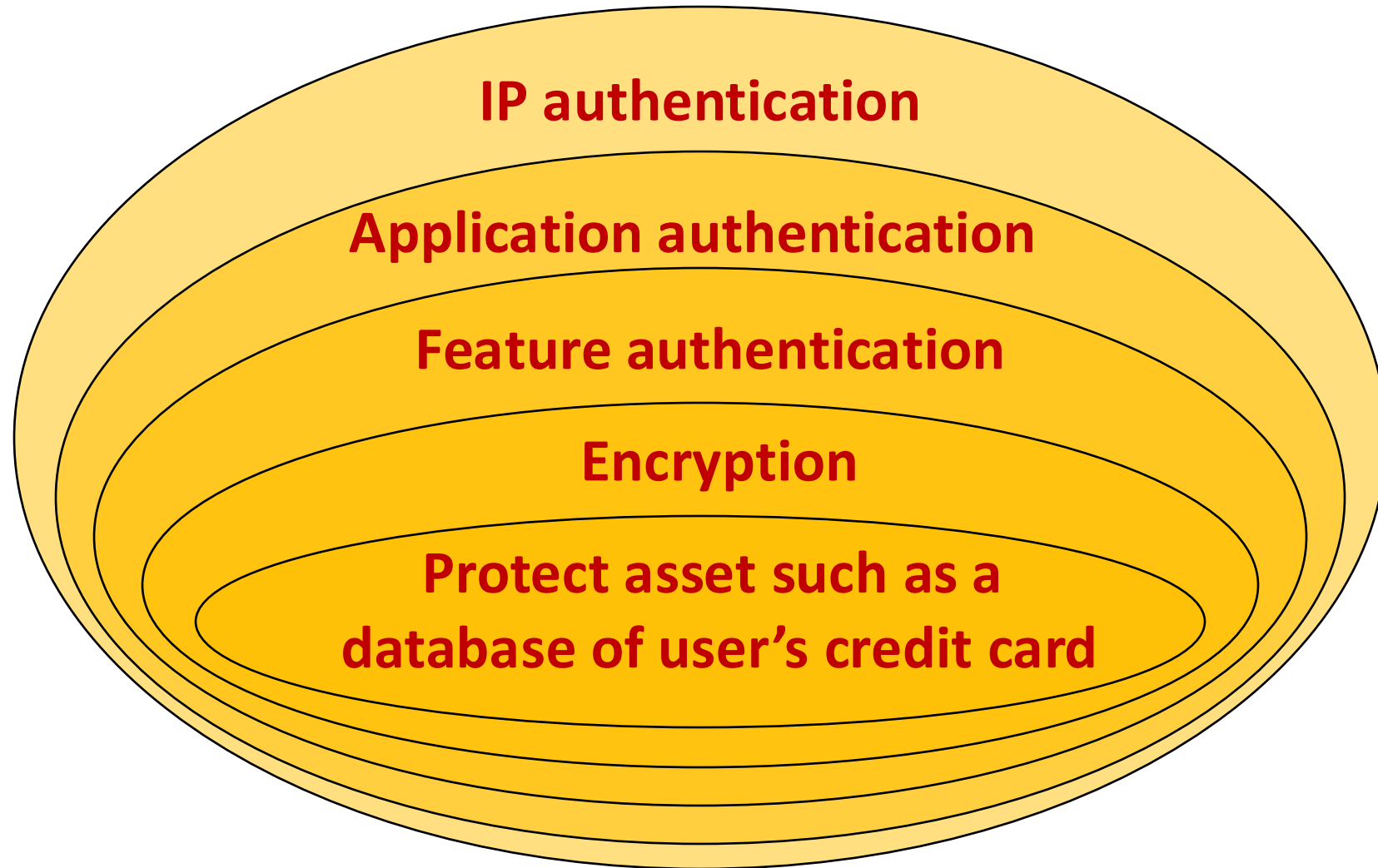
- System **maintainability** is an attribute that reflects how difficult and expensive it is to make changes to a system after it has been released to customers.
  - You improve maintainability by building a system from small self-contained parts, each of which can be replaced or enhanced if changes are required.
- In architectural terms, this means that the system should be **decomposed into fine-grain components**, each of which does one thing and one thing only.
  - However, it takes time for components to communicate with each other. Consequently, if many components are involved in implementing a product feature, the software will be slower.

# Trade off:

## Security vs usability

- You can achieve **security** by **designing the system protection as a series of layers**.
- An attacker has to penetrate all of those layers before the system is compromised.
- Layers might include **system authentication layers**, a separate critical **feature authentication layer**, an **encryption layer** and so on.
- Architecturally, you can implement each of these layers as separate components so that if one of these components is compromised by an attacker, then the other layers remain intact.

# Authentication layers



# Usability issues

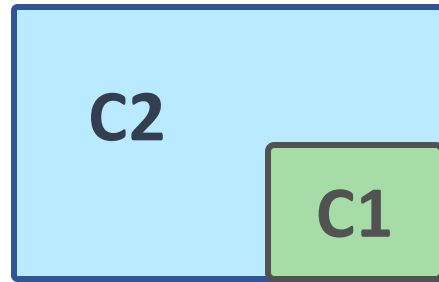
- **A layered approach to security affects the usability of the software.**
  - **Users have to remember information, like passwords, that is needed to penetrate a security layer.**  
**Their interaction with the system is inevitably slowed down by its security features.**
  - **Many users find this irritating and often look for work-arounds so that they do not have to re-authenticate to access system features or data.**
- **To avoid this, you need an architecture:**
  - **that doesn't have too many security layers**
  - **that doesn't enforce unnecessary security**
  - **that provides helper components that reduce the load on users**

# An architectural model of a document retrieval system

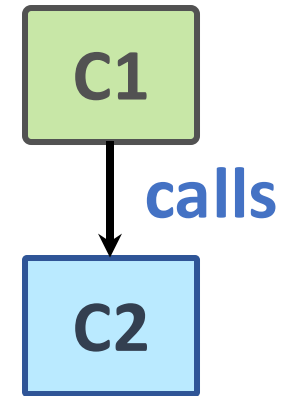
<b>Web browser</b>	User interaction	Local input validation	Local printing		
<b>User interface management</b>	Authentication and authorization	Form and query manager	Web page generation		
<b>Information retrieval</b>	Search	Document retrieval	Rights management	Payments	Accounting
<b>Document index</b>	Index management	Index querying	Index creation		
<b>Basic services</b>	Database Query	Query validation	Logging	User account management	
<b>Databases</b>	DB1	DB2	DB3	DB4	DB5

# Examples of component relationships

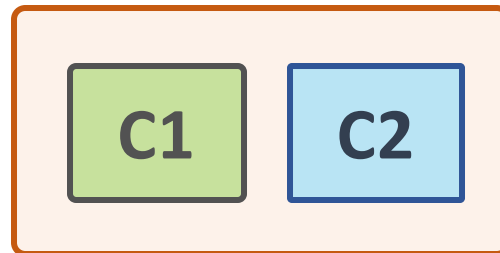
C1 is part of C2



C1 uses C2



C1 is-located-with C2



C1 shared-data-with C2



# Architectural design guidelines

## Separation of concerns

Organize your architecture into components that focus on a single concern



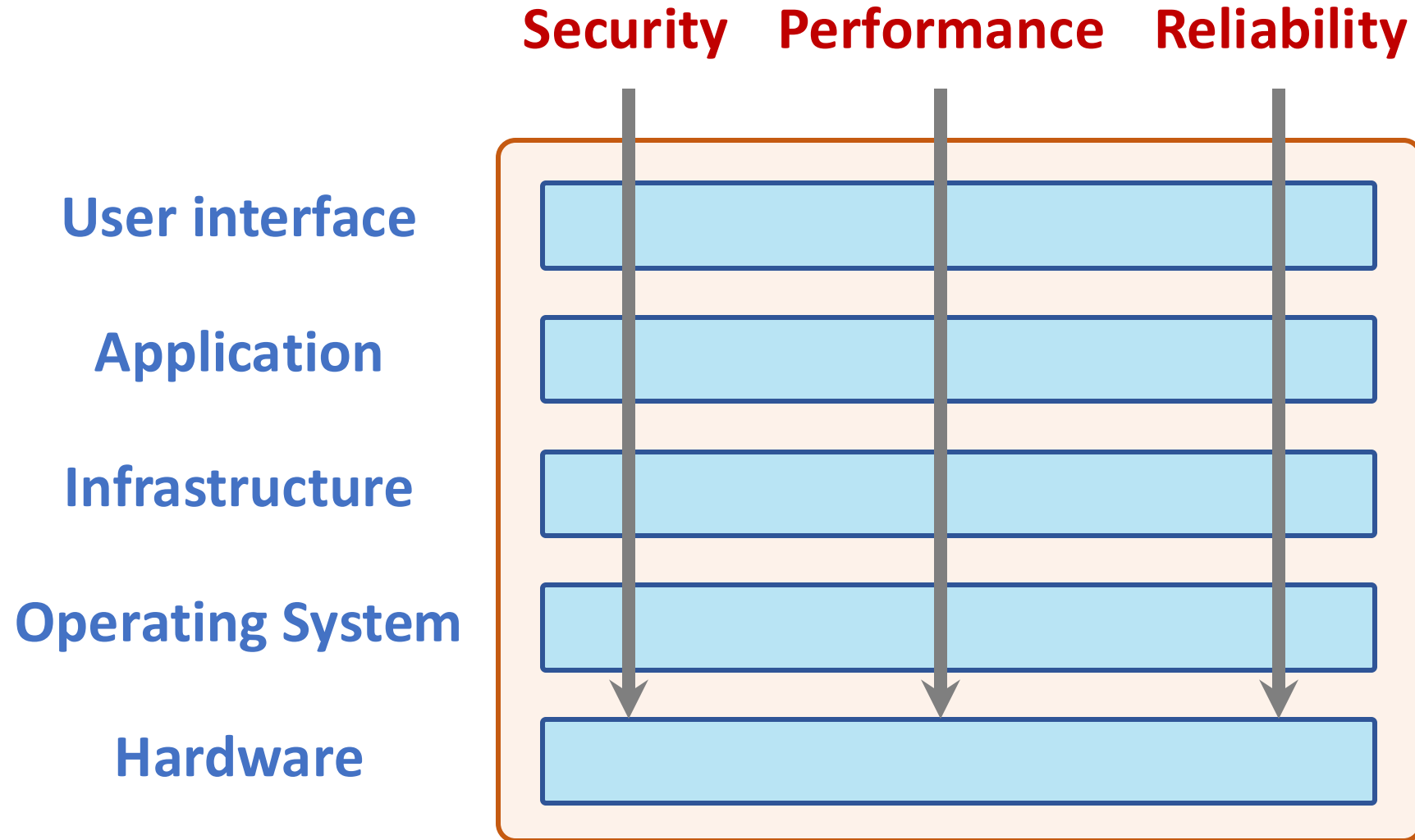
## Stable interfaces

Design component interfaces that are coherent and that changes slowly

## Implement once

Avoid duplicating functionality at different places in your architecture

# Cross-cutting concerns





# A generic layered architecture for a web-based application

**Browser-based or mobile user interface**

**Authentication and user interaction management**

**Application-specific functionality**

**Basic shared services**

**Transaction and database management**

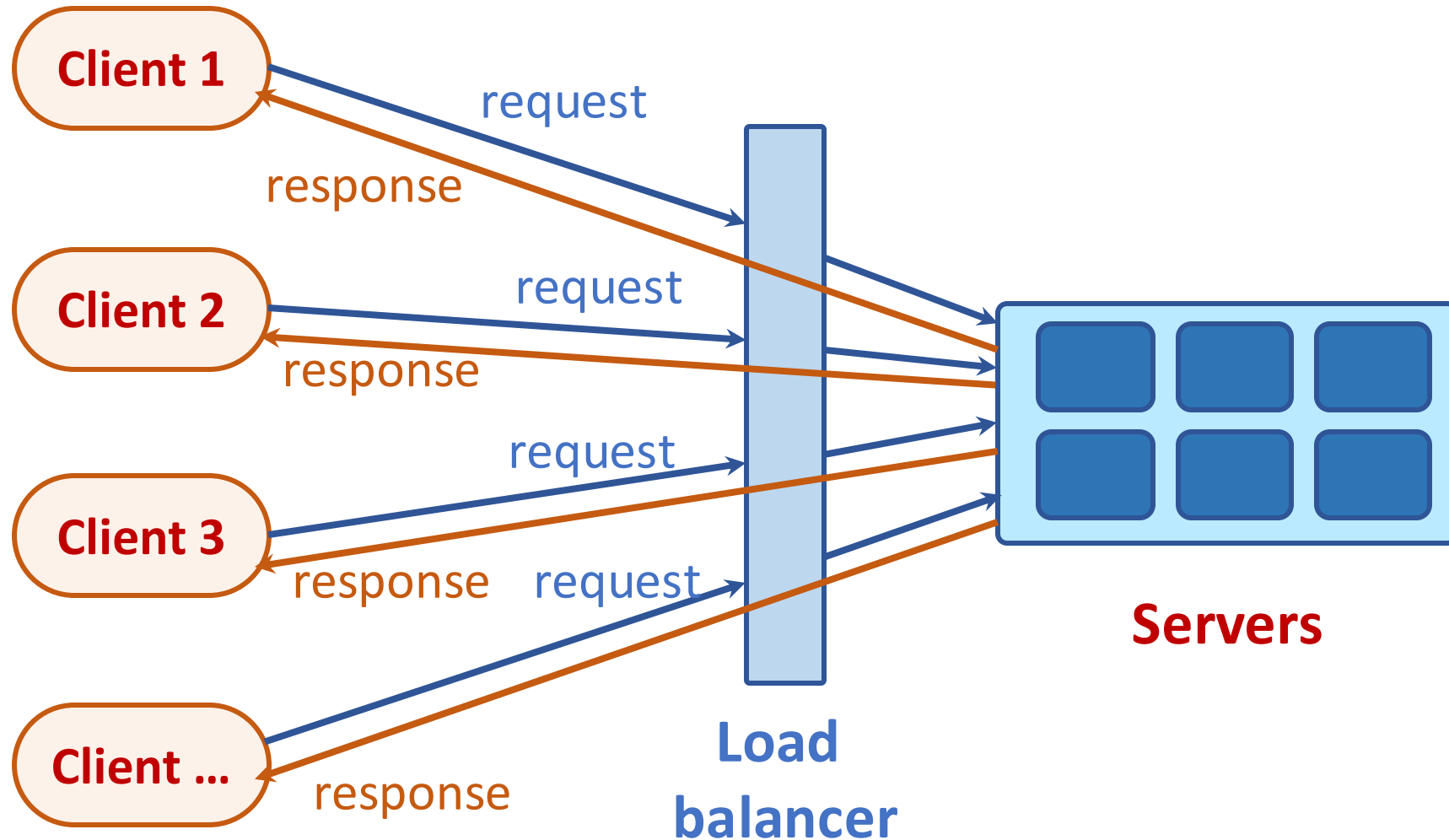
# A layered architectural model of the iLearn system

<b>User interface</b>	Web browser		iLearn app		
<b>User interface management</b>	Interface creation	Forms management	Interface delivery	Login	
<b>Configuration services</b>	Group configuration	Application configuration	Security configuration	User interface configuration	Setup service
<b>Application services</b>	Archive access Blog	Word processor Wiki Spreadsheet	Video conf. Presentation Drawing	Email and messaging	User installed application
<b>Integrated services</b>	Resource discovery	User analytics	Virtual Learning environment	Authentication and authorization	
<b>Shared infrastructure services</b>	Authentication	Logging and monitoring	Application interfacing		Search
	User storage	Application storage			

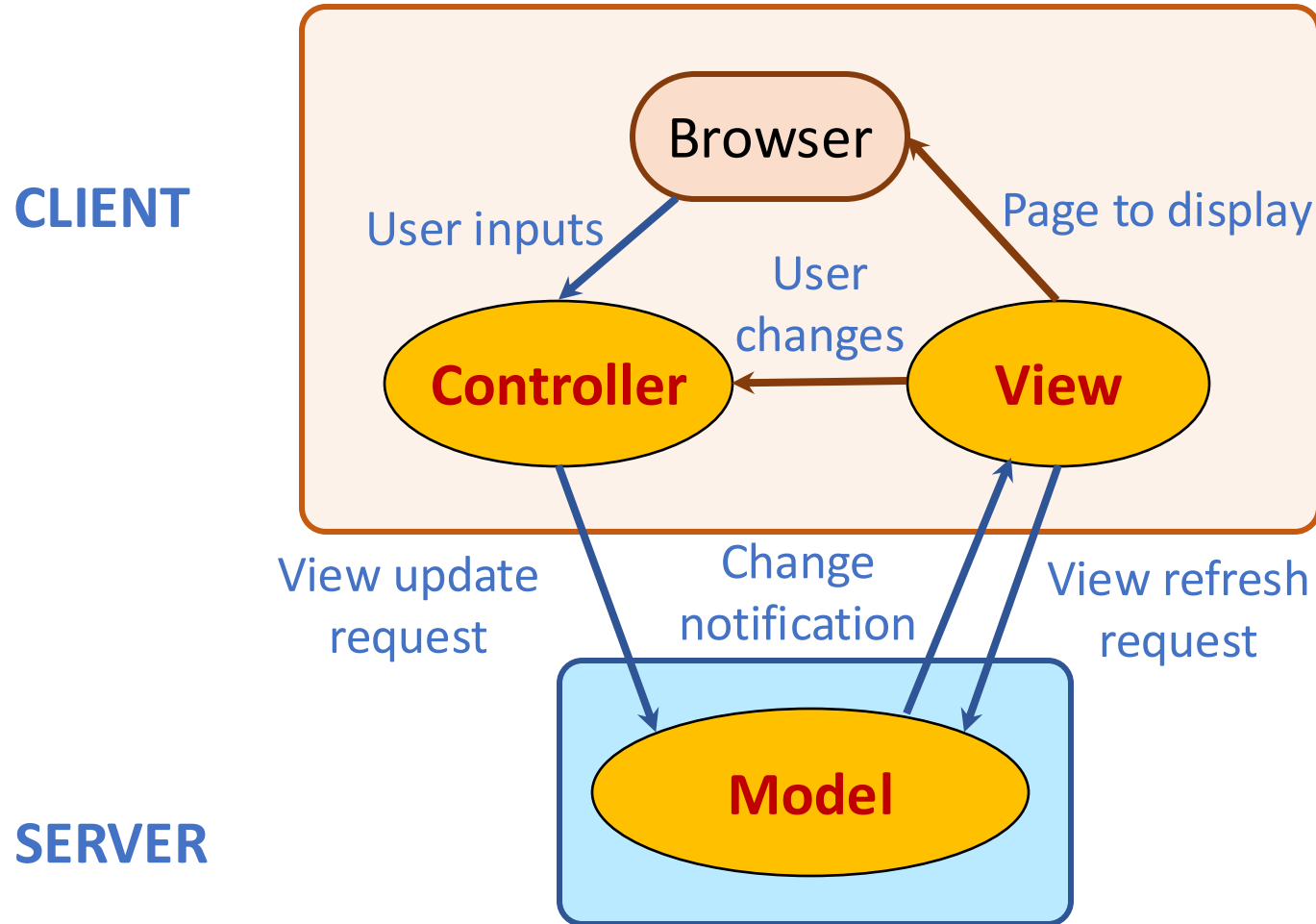
# Distribution architecture

- The **distribution architecture** of a software system defines the servers in the system and the allocation of components to these servers.
- **Client-server architectures** are a type of **distribution architecture** that is suited to applications where clients access a **shared database** and **business logic operations** on that data.
- In this architecture, the **user interface** is implemented on the **user's own computer or mobile device**.
  - **Functionality is distributed** between the client and one or more server computers.

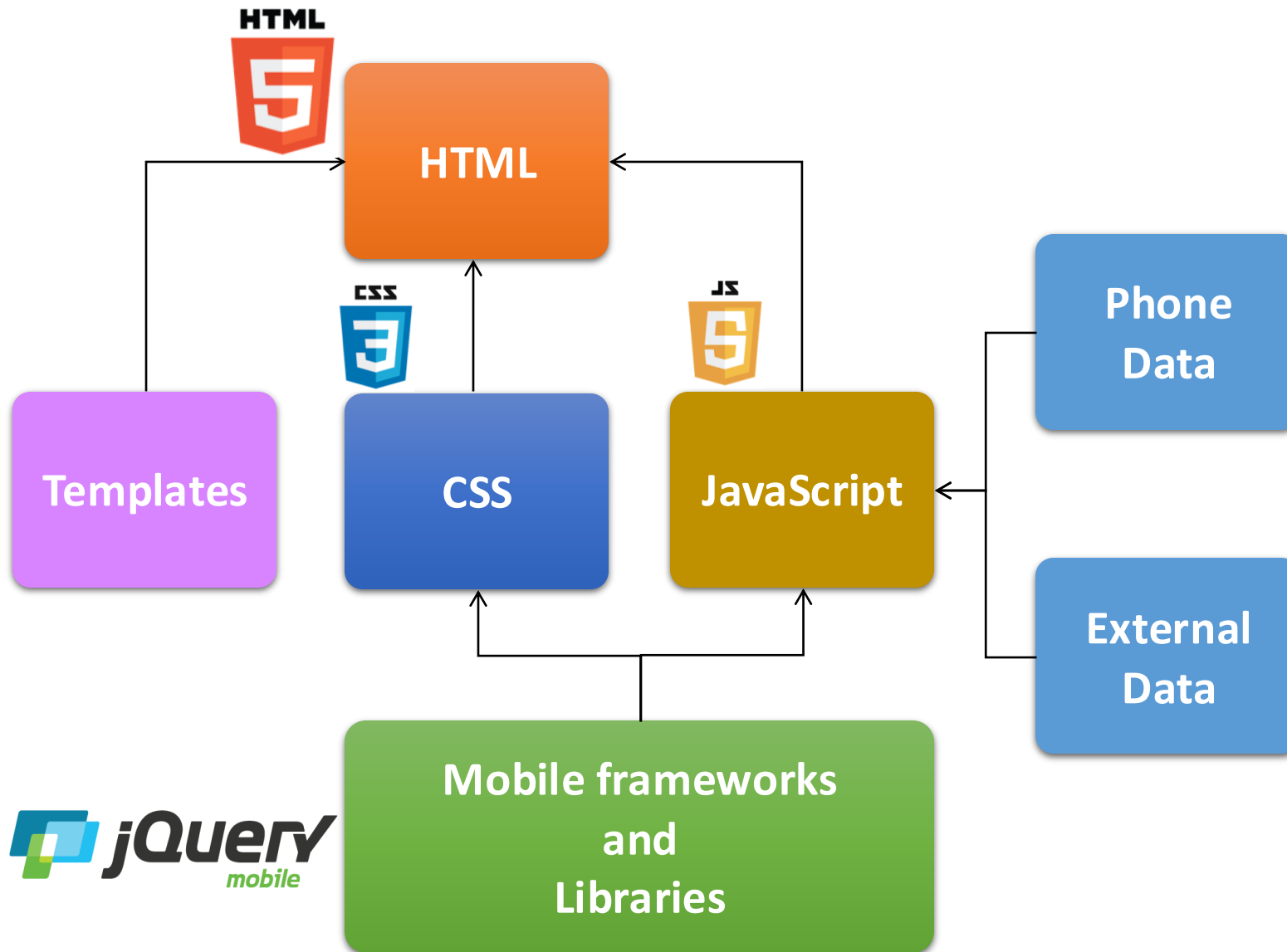
# Client-server architecture



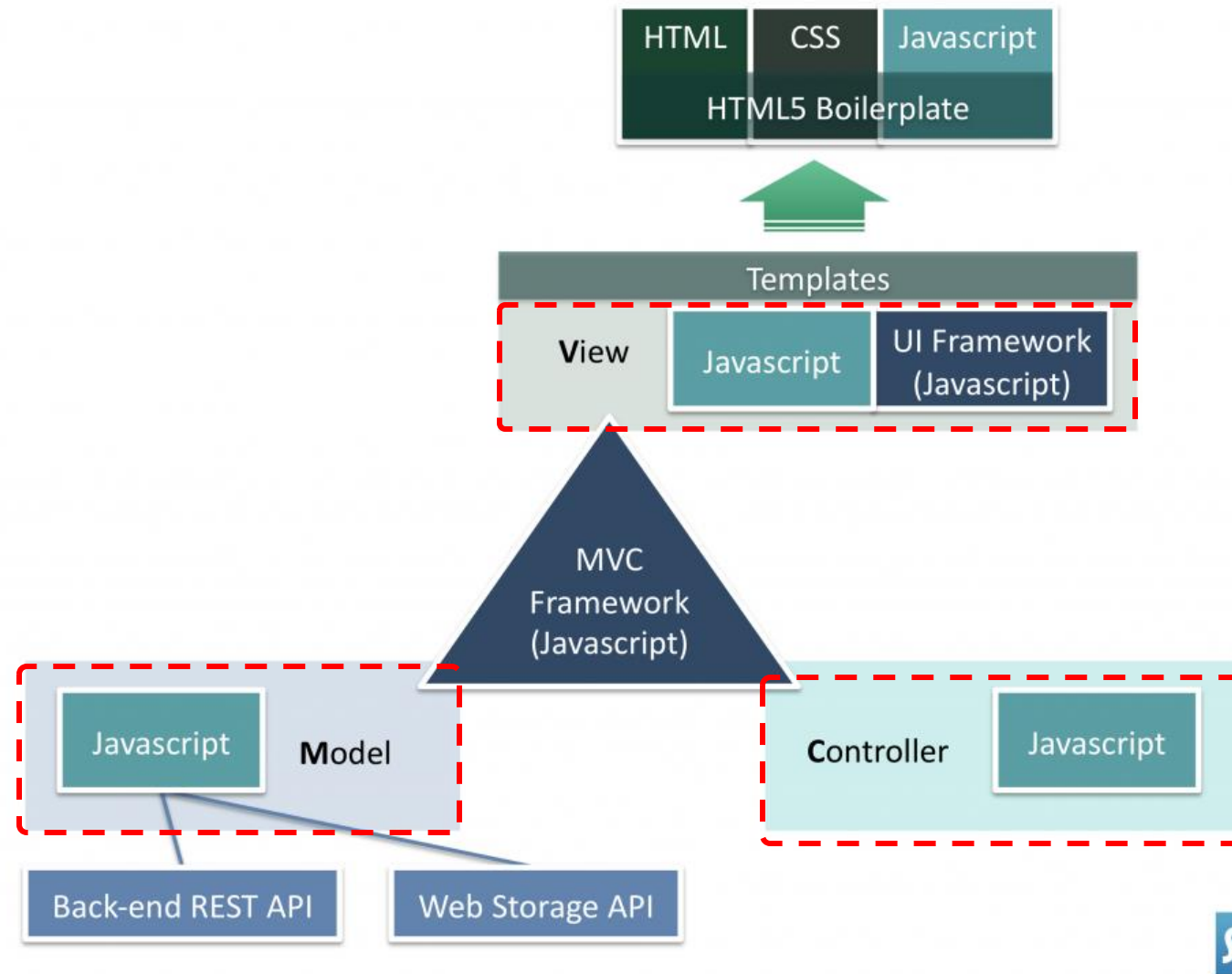
# The Model-View-Controller (MVC) pattern



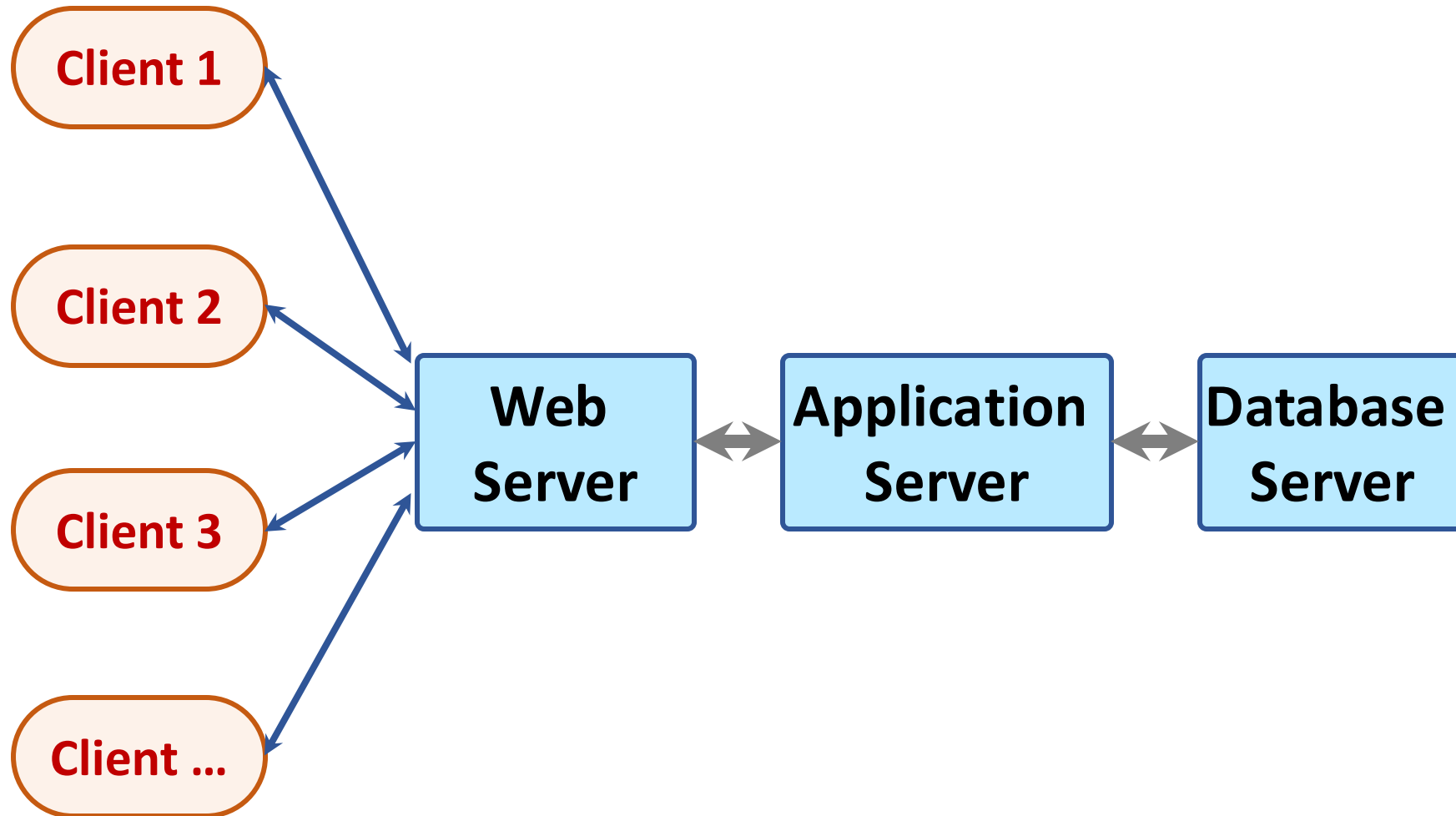
# Mobile Web App



# MVC Framework of Mobile Apps (HTML5, CSS3, JavaScript)



# Multi-tier client-server architecture

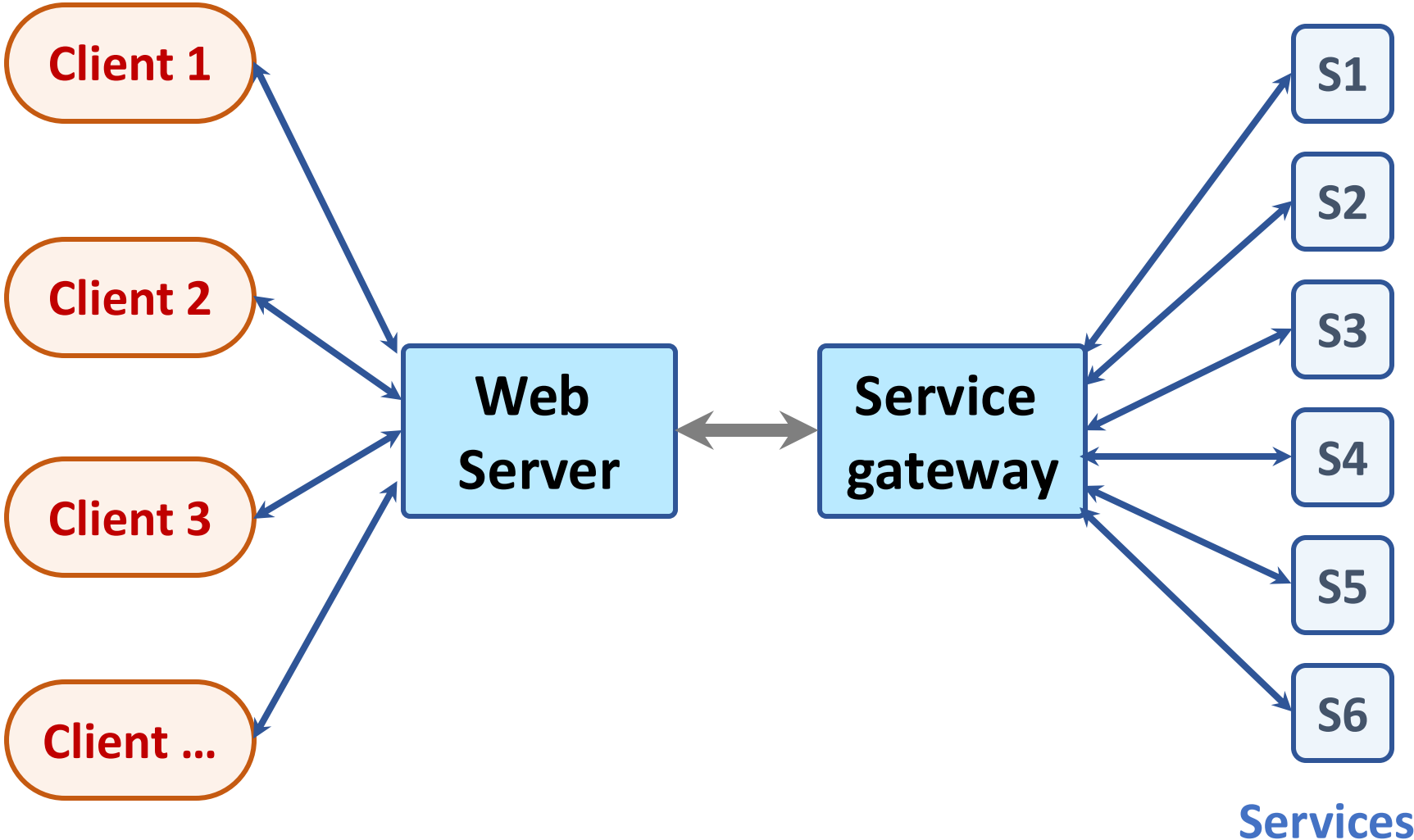




# Service-oriented Architecture

- Services in a **service-oriented architecture** are **stateless components**, which means that they can be replicated and can migrate from one computer to another.
- Many servers may be involved in providing services
- A **service-oriented architecture** is usually **easier to scale** as demand increases and is resilient to failure.

# Service-oriented Architecture



# Issues in architectural choice

- **Data type and data updates**
- **Change frequency**
- **The system execution platform**

# Issues in architectural choice

- **Data type and data updates**
  - **If you are mostly using structured data that may be updated by different system features, it is usually best to have a single shared database that provides locking and transaction management. If data is distributed across services, you need a way to keep it consistent and this adds overhead to your system.**

# Issues in architectural choice

- **Change frequency**

- **If you anticipate that system components will be regularly changed or replaced, then isolating these components as separate services simplifies those changes.**

# Issues in architectural choice

- **The system execution platform**
  - **If you plan to run your system on the cloud with users accessing it over the Internet, it is usually best to implement it as a service-oriented architecture because scaling the system is simpler.**
  - **If your product is a business system that runs on local servers, a multi-tier architecture may be more appropriate.**

# Technology choices

- **Database**

Should you use a relational SQL database or an unstructured NOSQL database?

- **Platform**

Should you deliver your product on a mobile app and/or a web platform?

- **Server**

Should you use dedicated in-house servers or design your system to run on a public cloud? If a public cloud, should you use Amazon, Google, Microsoft, or some other option?

- **Open source**

Are there suitable open-source components that you could incorporate into your products?

- **Development tools**

Do your development tools embed architectural assumptions about the software being developed that limit your architectural choices

# Summary

- **Software architecture** is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.
- **The architecture of a software system** has a significant influence on **non-functional system properties** such as reliability, efficiency and security.
- **Architectural design** involves understanding the issues that are critical for your product and creating system descriptions that shows components and their relationships.



# Summary

- The principal role of **architectural descriptions** is to provide a basis for the development team to discuss the system organization. Informal architectural diagrams are effective in architectural description because they are fast and easy to draw and share.
- **System decomposition** involves analyzing **architectural components** and representing them as a set of **finer-grain components**.

# Summary

- To **minimize complexity**, you should separate concerns, avoid functional duplication and focus on component interfaces.
- **Web-based systems** often have a common layered structure including **user interface layers**, **application-specific layers** and a **database layer**.
- The **distribution architecture** in a system defines the organization of the servers in that system and the allocation of components to these servers.

# Summary

- **Multi-tier client-server and service-oriented architectures** are the most commonly used architectures for web-based systems.
- Making decisions on technologies such as **database** and **cloud technologies** are an important part of the architectural design process.

# 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 (2021), A Guide to the Project Management Body of Knowledge (PMBOK Guide) – Seventh Edition and The Standard for Project Management, PMI.
- 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.