Software architecture

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.
- 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

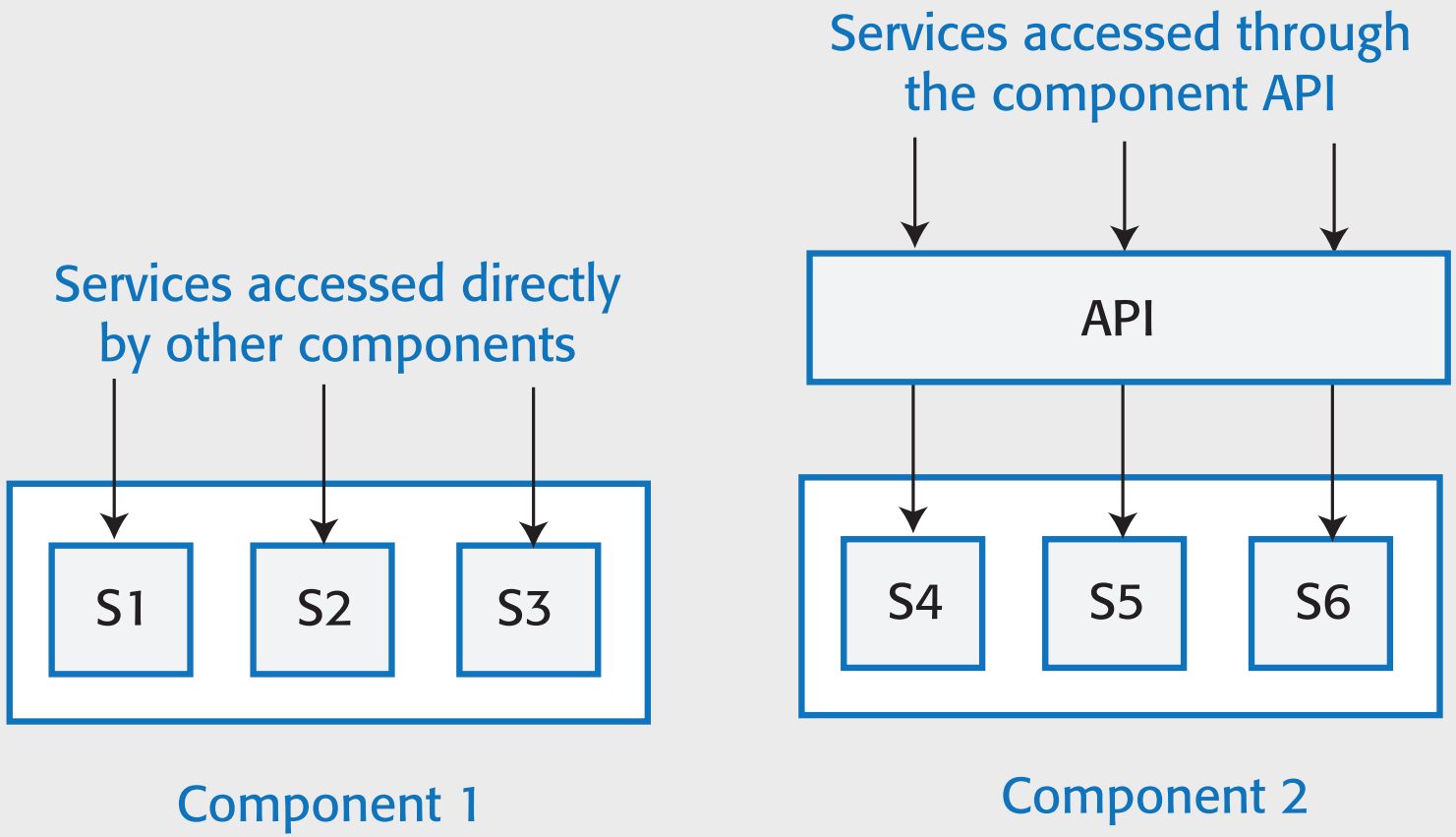
Architecture is the fundamental organization of a software system embodied in:

- *its components*
- *their relationships to each other and to the environment*
- *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, 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



Why is architecture important?

- Architecture of a system has a fundamental influence on the non-functional system properties (explained in the next slide).
- It shows the critical components of the system and their relationships.
- Minimizing complexity is an important goal for architectural designers.
  - The more complex a system, the more difficult and expensive it is to understand and change.
  - Programmers are more likely to make mistakes and introduce bugs and security vulnerabilities when they are modifying or extending a complex system.

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?

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?

The influence on architecture of system security

A centralized security architecture

In the Star Wars prequel Rogue One (https://en.wikipedia.org/wiki/Rogue_One), the evil Empire have stored the plans for all of their equipment in a single, highly secure, well-guarded, remote location. This is called a centralized security architecture. It is based on the principle that if you maintain all of your information in one place, then you can apply lots of resources to protect that information and ensure that intruders can't get hold of it.

Unfortunately (for the Empire), the rebels managed to breach their security. They stole the plans for the Death Star, an event which underpins the whole Star Wars saga. In trying to stop them, the Empire destroyed their entire archive of system documentation with who knows what resultant costs. Had the Empire chosen a distributed security architecture, with different parts of the Death Star plans stored in different locations, then stealing the plans would have been more difficult. The rebels would have had to breach security in all locations to steal the complete Death Star blueprints.

Centralized security architectures

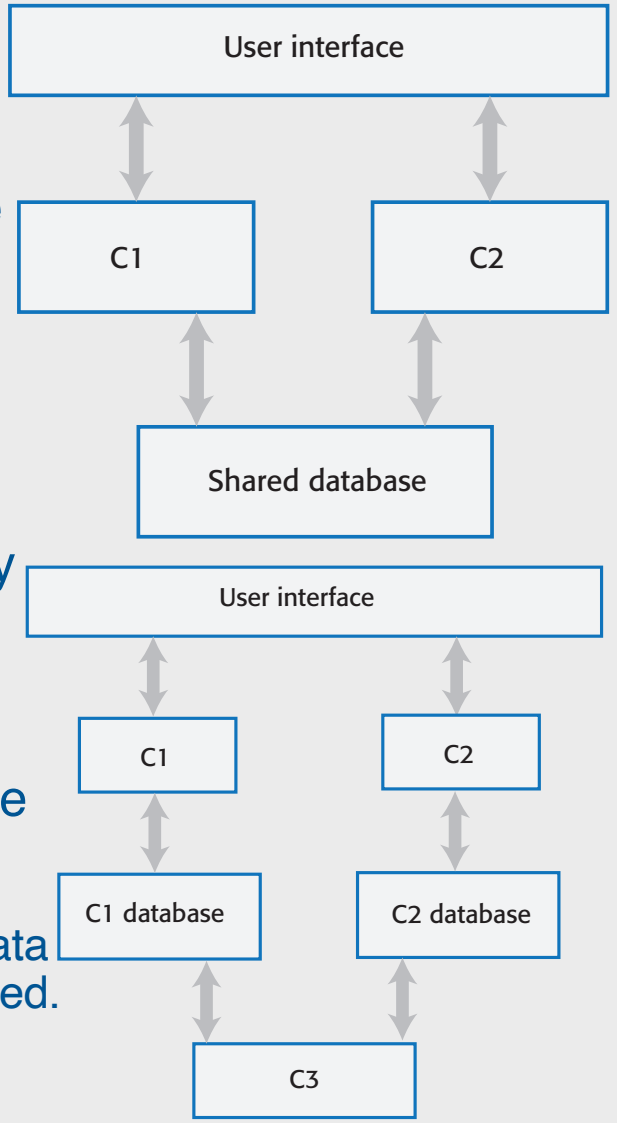
- The benefits of a centralized security architecture
  - it is easier to design and build protection
  - the protected information can be accessed more efficiently.
- However, if your security is breached, you lose everything.

- If you distribute information, it takes longer to access all the information and costs more to protect it.

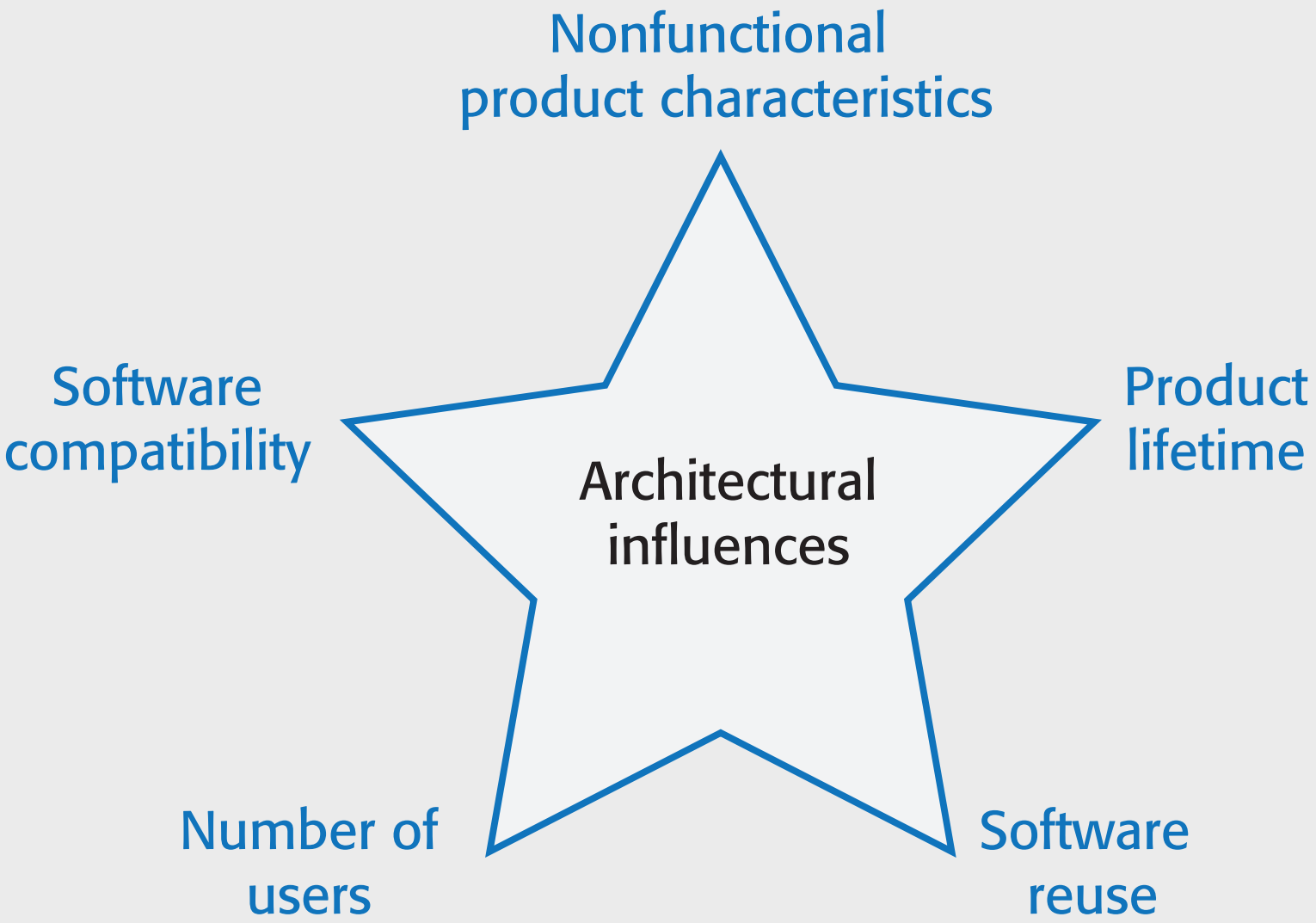
- However, if security is breached in one location, you only lose the information that you have stored there.

Maintainability vs performance

- Consider a system with two modules (C1 & C2) that share a common database.
  - Assume C1 runs slowly because it has to reorganize the information in the database before using it.
  - To make C1 faster we need to change the database.
  - This means that C2 also has to be changed, which may, potentially, affect its response time.
- Now, consider that each component has its own copy of the parts of the database that it needs.
  - Changing the database for C1 does not affect C2
- However, a multi-database architecture may run more slowly and may cost more to implement and change.
  - Also a mechanism (eg, C3) needed 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

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

- If you get these wrong, your product will not be a commercial success.
- However, some characteristics are conflicting, so you can only optimize the most important ones.

Product lifetime

- If you anticipate a long product lifetime, you will need to create regular product revisions.
- You need an architecture that is evolvable and can accommodate new features.

Software reuse

- You can save time and effort via reusing components from other open-source products.
- This constrains the architectural choices: you must fit your design to the reused software.

Number of users

- For the Internet-based software, the number of users can change very quickly.
- This can lead to performance degradation unless you design your architecture to quickly scaled up and down.

Software compatibility

- For some products, it is important to maintain compatibility with other software.
- This may limit architectural choices, such as the database software that you can use.

Trade off: Maintainability vs performance

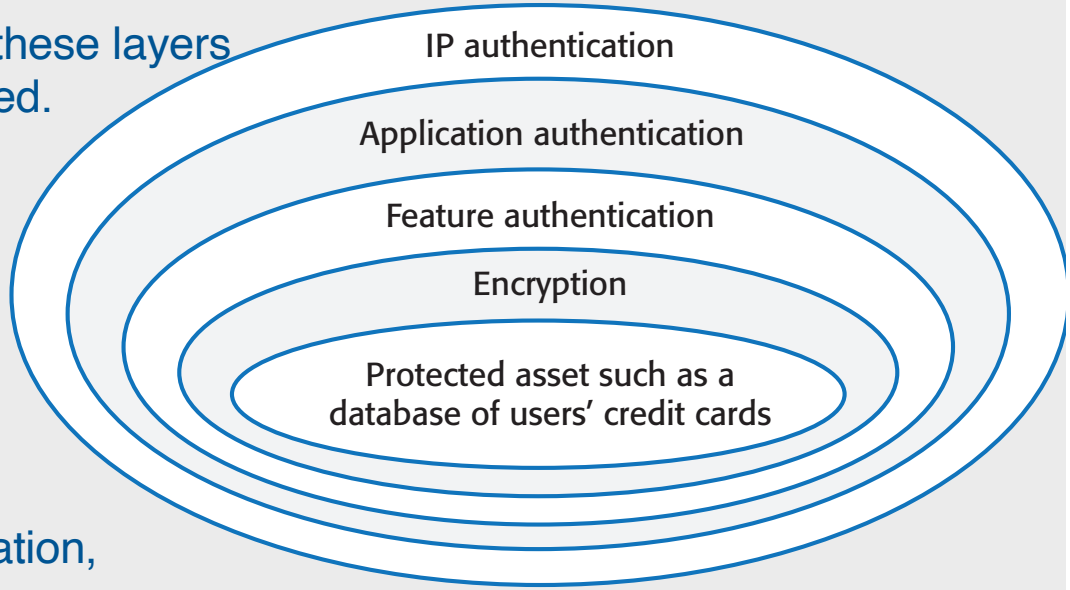
- System maintainability reflects how difficult and expensive it is to make changes to a system after it has been released to customers.

- How to improve maintainability?
  - Build the system from small self-contained parts, each of which can be replaced or enhanced if changes are required.
  - For 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

- To achieve security, we can design the system with a series of protection layers
  - An attacker has to penetrate all these layers before the system is compromised.



- A layered approach to security affects the software usability
  - Users have to remember information, like passwords.
  - Their interaction with the system is slowed down by the 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.

Security vs Usability issues

- 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.

Trade off: Availability vs time-to-market

- Availability of a system is the amount of ‘uptime’ of that system.
  - Availability is expressed as a percentage of the time that a system is available to deliver user services.
  - Availability is important in enterprise products, e.g. finance industry, where 24/7 operation is expected.
- Architecturally, you achieve availability by having redundant components in a system.
  - For redundancy, you include “failure detection”, and "switching" components that switch operation to the redundant component when a failure is detected.
- Implementing extra components:
  - takes time and increases the cost of system development
  - adds complexity to the system and therefore increases the chances of introducing bugs and vulnerabilities.

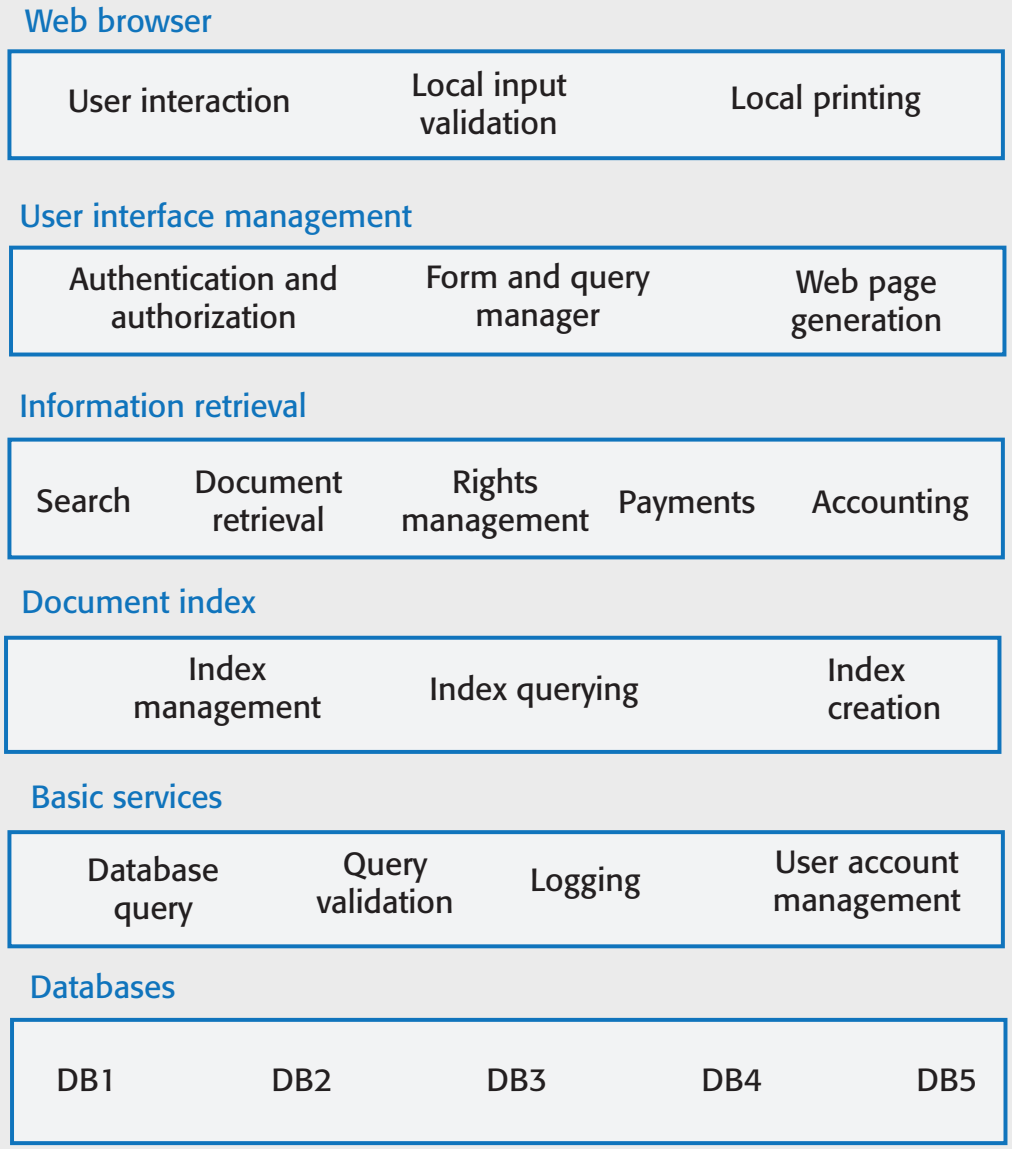
Architectural design questions

- How should the system be organized as a set of architectural components, where each of these components provides a subset of the overall system functionality?
  - The organization should deliver the system security, reliability and performance.
- How should these architectural components be distributed and communicate with each other?
- What technologies should you use in building the system and what components should be reused?

Component organization

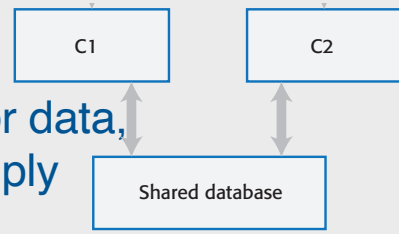
- Abstraction in software design means to focus on the essential elements of a system or software component without concern for its details.
- At the architectural level, your concern should be on large-scale architectural components.
- Decomposition involves analyzing these large-scale components and representing them as a set of finer-grain components.
- Layered models are often used to illustrate how a system is composed of components.

Example: An architectural model of a document retrieval system



Architectural complexity

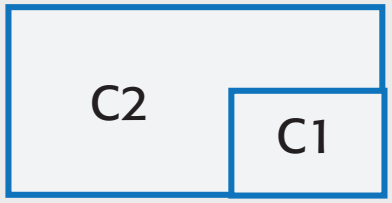
- Complexity in a system architecture arises because of the number and the nature of the relationships between components in that system.
- When decomposing a system into components, try to avoid unnecessary complexity.
  - *Localize relationships*
    If there are relationships between components A and B, these are easier to understand if A and B are defined in the same module.
  - *Reduce shared dependencies*
    Where components C1 and C2 depend on some other component or data, complexity increases because changes to the shared component imply understanding how these changes affect both C1 and C2.



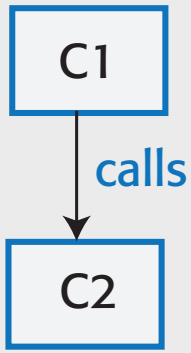
- It is always preferable to use local data wherever possible and to avoid sharing data if you can.

Examples of component relationships

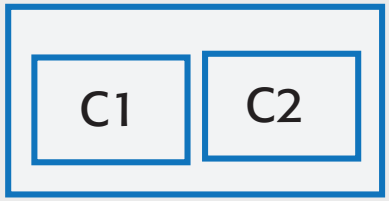
C1 is-part-of C2



C1 uses C2



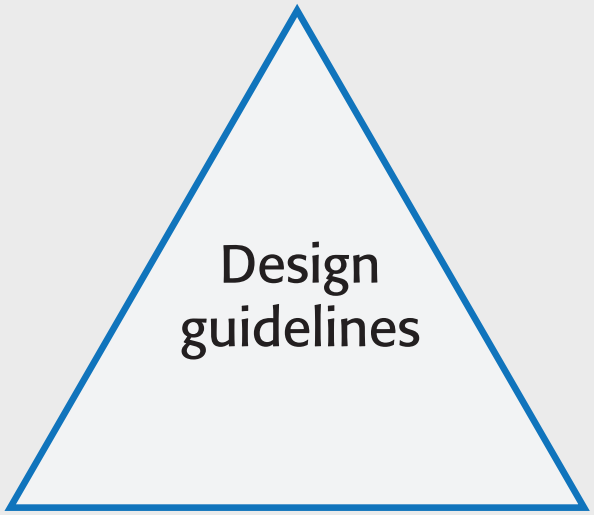
C1 is-located-with C2



C1 shares-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 change slowly

Implement once
Avoid duplicating functionality at different places in your architecture

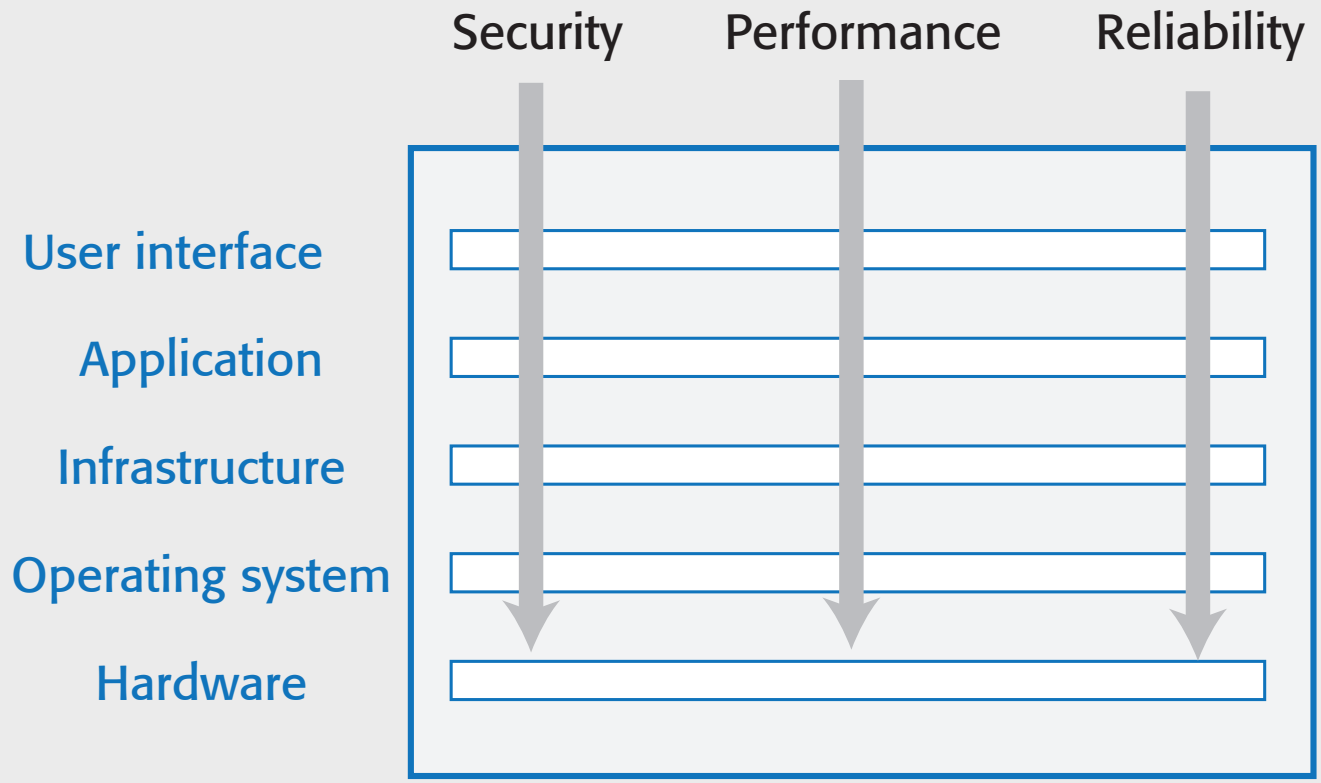
Design guidelines and layered architectures

- Each layer is considered separately from other layers.
  - The top layer is concerned with user interaction
  - The next layer down with user interface management
  - The third layer with information retrieval and so on…
- Within each layer, components are independent and do not overlap in functionality.
  - The lower layers include components that provide general functionality, so there is no need to replicate this in the components in a higher level.
- Ideally, components at level X (say) should only interact with the APIs of the components in level X-1.
  - That is, interactions should be between layers and not across layers.

Cross-cutting concerns

- Cross-cutting concerns are concerns that affect the whole system.
- Cross-cutting concerns are different from the functional concerns represented by layers in a software architecture.
- Every layer has to take them into account and there are inevitably interactions between the layers because of these concerns.
- The existence of cross-cutting concerns is the reason why modifying a system after it has been designed to improve its security is often difficult.

Cross-cutting concerns



Security as a cross-cutting concern

Vulnerability at multiple layers

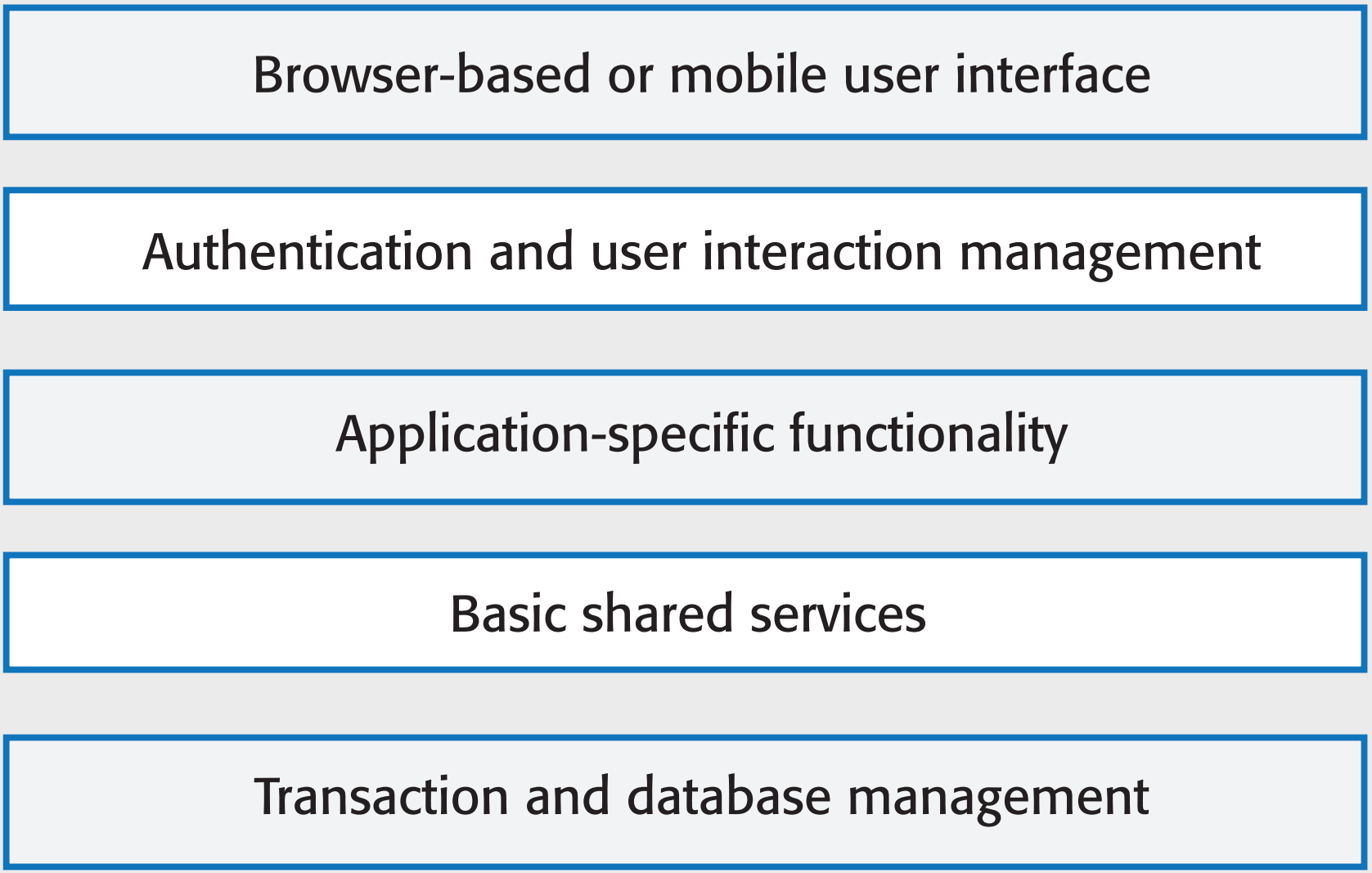
- Attackers can try to use vulnerabilities at various layers (DB, browser, etc.) to gain access.
- Protection from attacks at each layer is needed.

Having a single security component in a system is a critical vulnerability.

- If it stops working properly or is compromised in an attack, then you have no reliable security in your system!

By distributing security across the layers, the system is more resilient to attacks and software failure

A generic layered architecture for a web-based application



Layer functionality in a web-based application

Browser-based or mobile user interface
A web browser with HTML forms are used to collect user input. Javascript components for local actions, such as input validation, should also be included at this level. Alternatively, a mobile interface may be implemented as an app.

Authentication and UI management
A user interface management layer that may include components for user authentication and web page generation.

Application-specific functionality
An 'application' layer that provides functionality of the application. Sometimes, this may be expanded into more than one layer.

Basic shared services
A shared services layer, which includes components that provide services used by the application layer components.

Database and transaction management
A database layer that provides services such as transaction management and recovery. If your application does not use a database then this may not be required.

iLearn architectural design principles

Replaceability
It should be possible for users to replace applications in the system with alternatives and to add new applications. Consequently, the list of applications included should not be hard-wired into the system.

Extensibility
It should be possible for users or system administrators to create their own versions of the system, which may extend or limit the 'standard' system.

Age-appropriate
Alternative user interfaces should be supported so that age-appropriate interfaces for students at different levels can be created.

Programmability
It should be easy for users to create their own applications by linking existing applications in the system.

Minimum work
Users who do not wish to change the system should not have to do extra work so that other users can make changes.

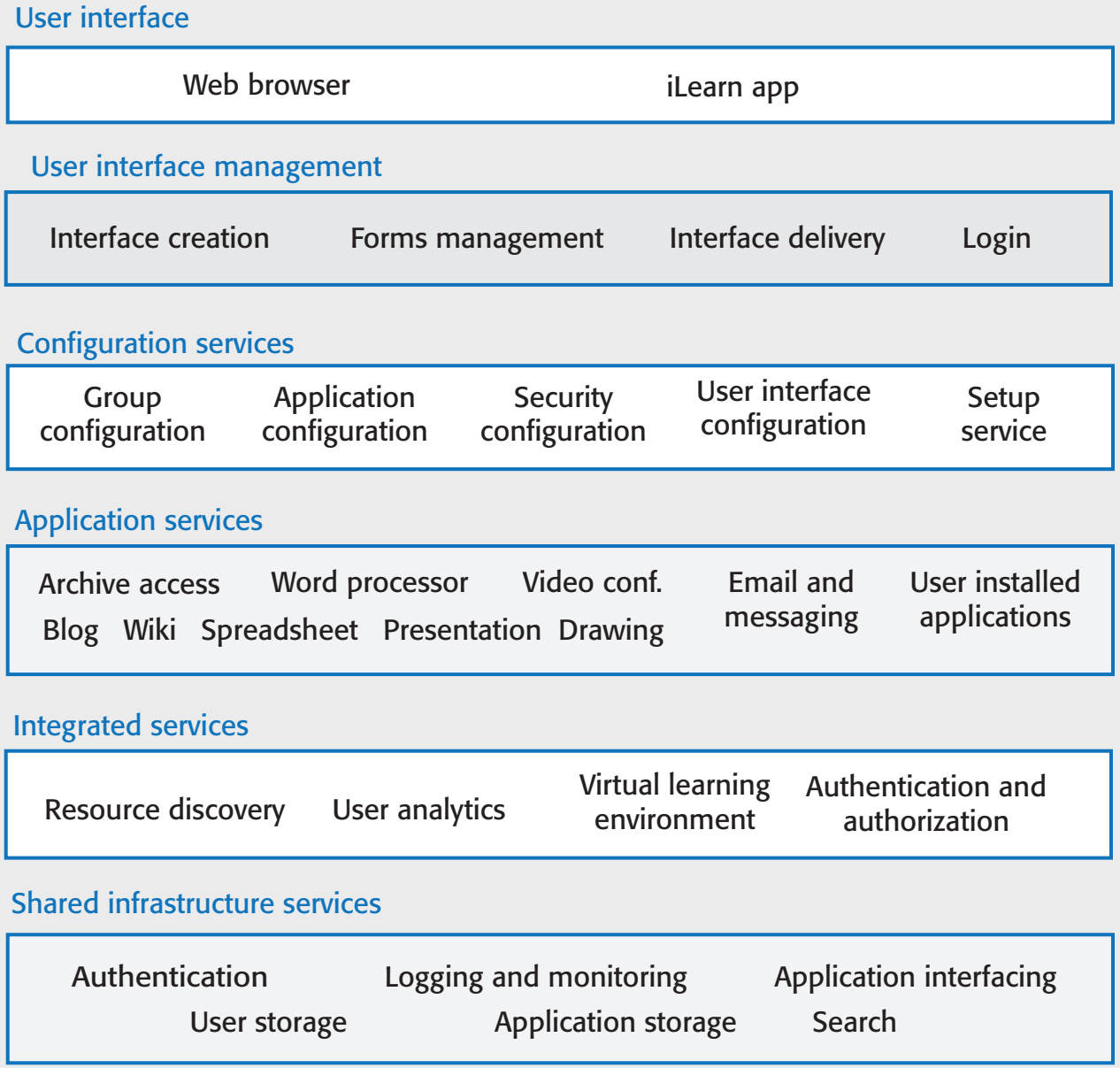
iLearn design principles

- Our goal in designing the iLearn system was to create an adaptable, universal system that could be easily updated as new learning tools became available.
  - This means that it must be possible to change and replace components and services in the system (principles (1) and (2)).
  - Because the potential system users spanned an age range from 3 to 18, we needed to provide age-appropriate user interfaces and to make it easy to choose an interface (principle (3)).
  - Principle (4) also contributes to system adaptability and principle (5) was included to ensure that this adaptability did not adversely affect users who did not require it.

Designing iLearn as a service-oriented system

- These principles led us to an architectural design decision that the iLearn system should be service-oriented.
- Every component in the system is a service. Any service is potentially replaceable and new services can be created by combining existing services. Different services delivering comparable functionality can be provided for students of different ages.
- Service integration
  - *Full integration* Services are aware of and can communicate with other services through their APIs.
  - *Partial integration* Services may share service components and databases but are not aware of and cannot communicate directly with other application services.
  - *Independent* These services do not use any shared system services or databases and they are unaware of any other services in the system. They can be replaced by any other comparable service.

A layered architectural model of the iLearn system



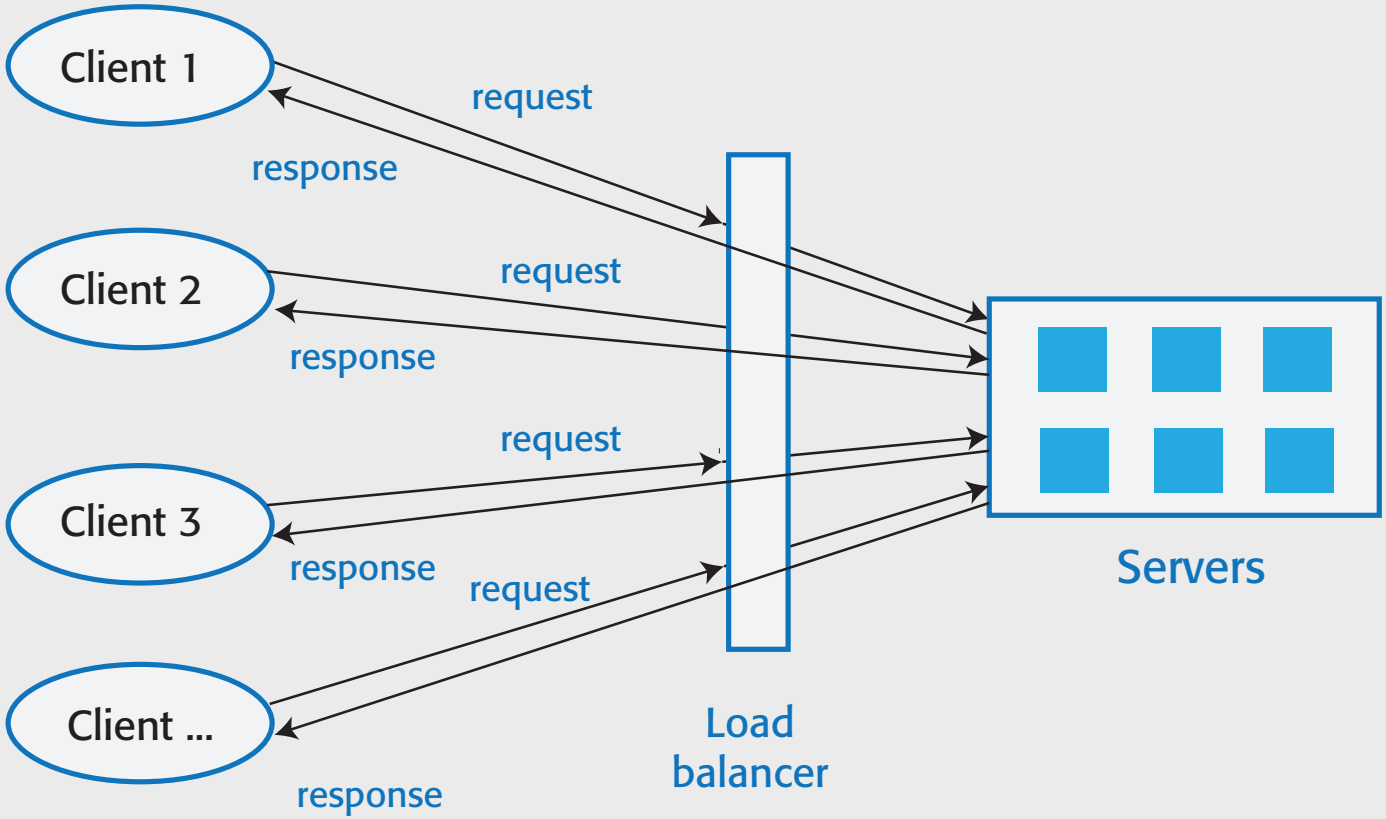
Distribution architecture

- 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.

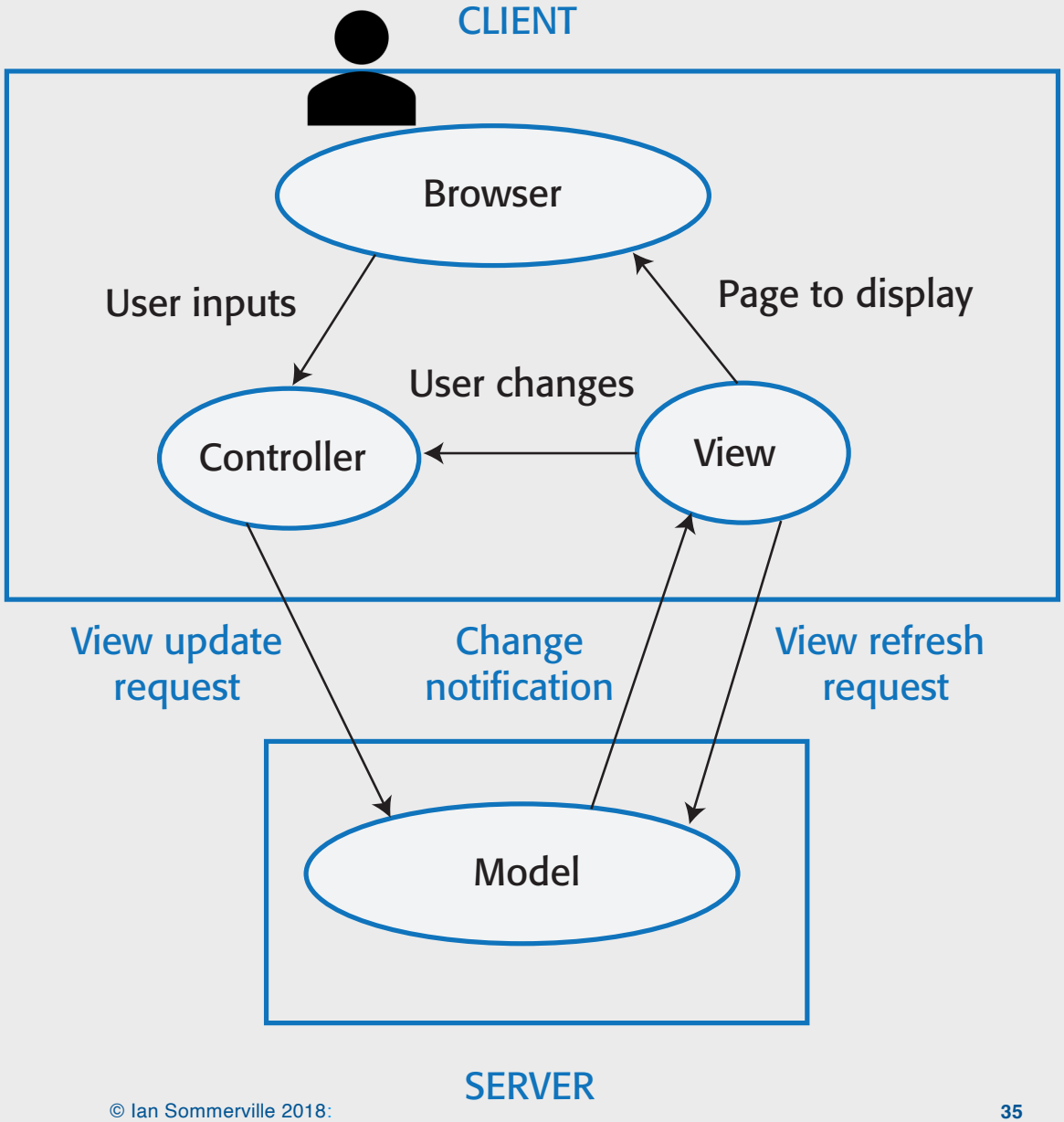
Client-server architecture



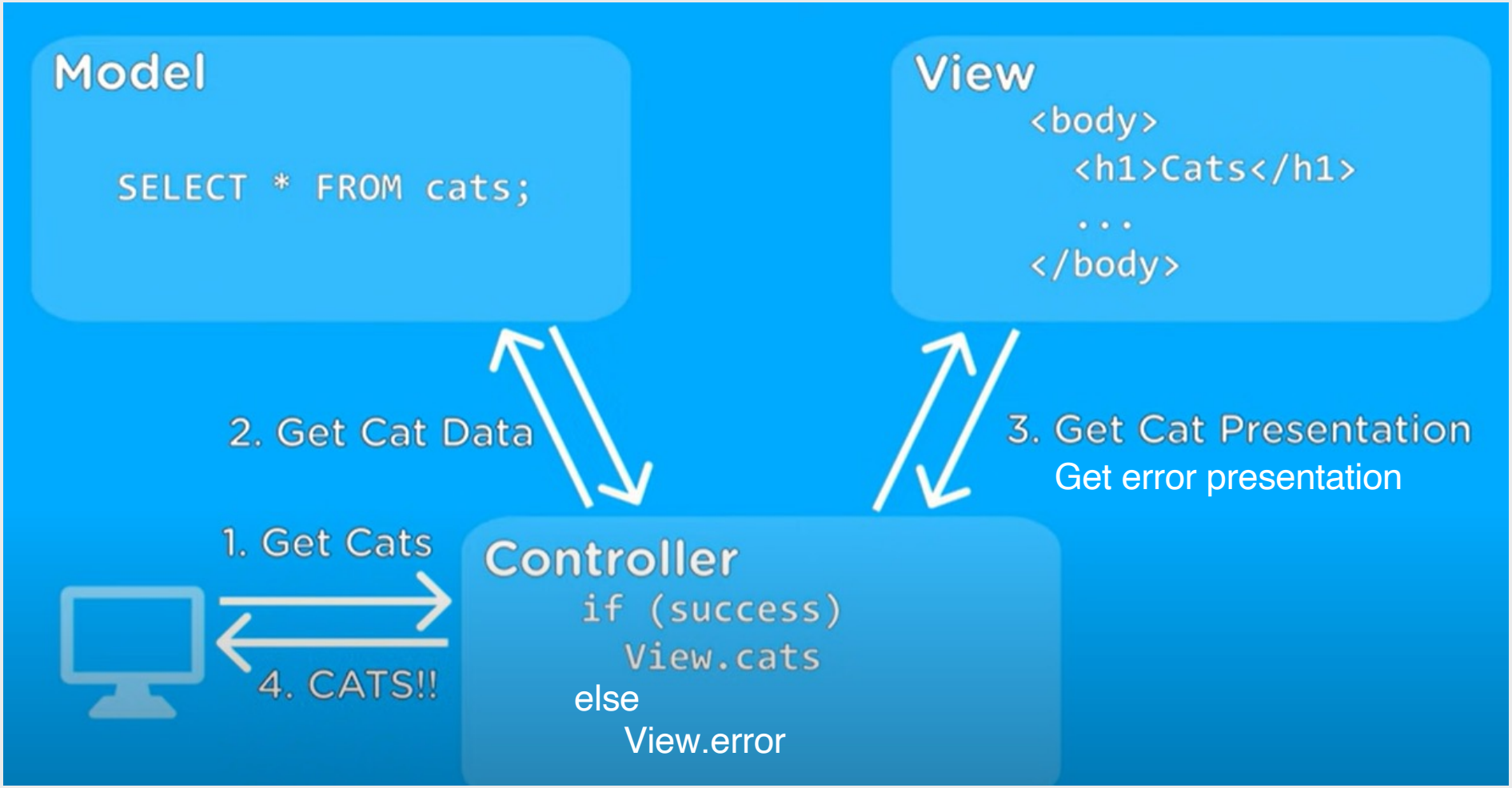
The model-view-controller architectural pattern

MVC is all about separation of concerns:

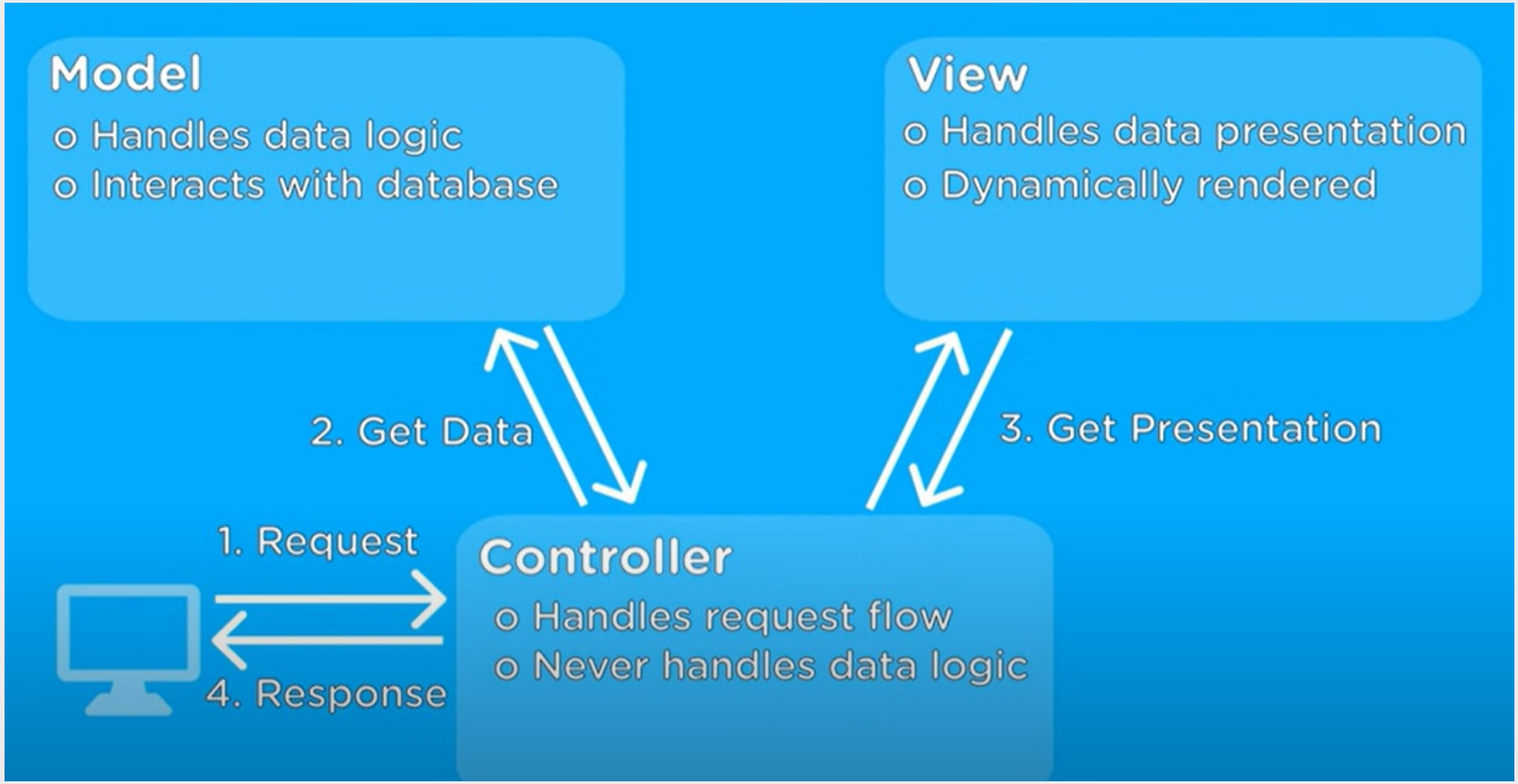
- Model
- View
- Controller



MVC: example for retrieving cats information



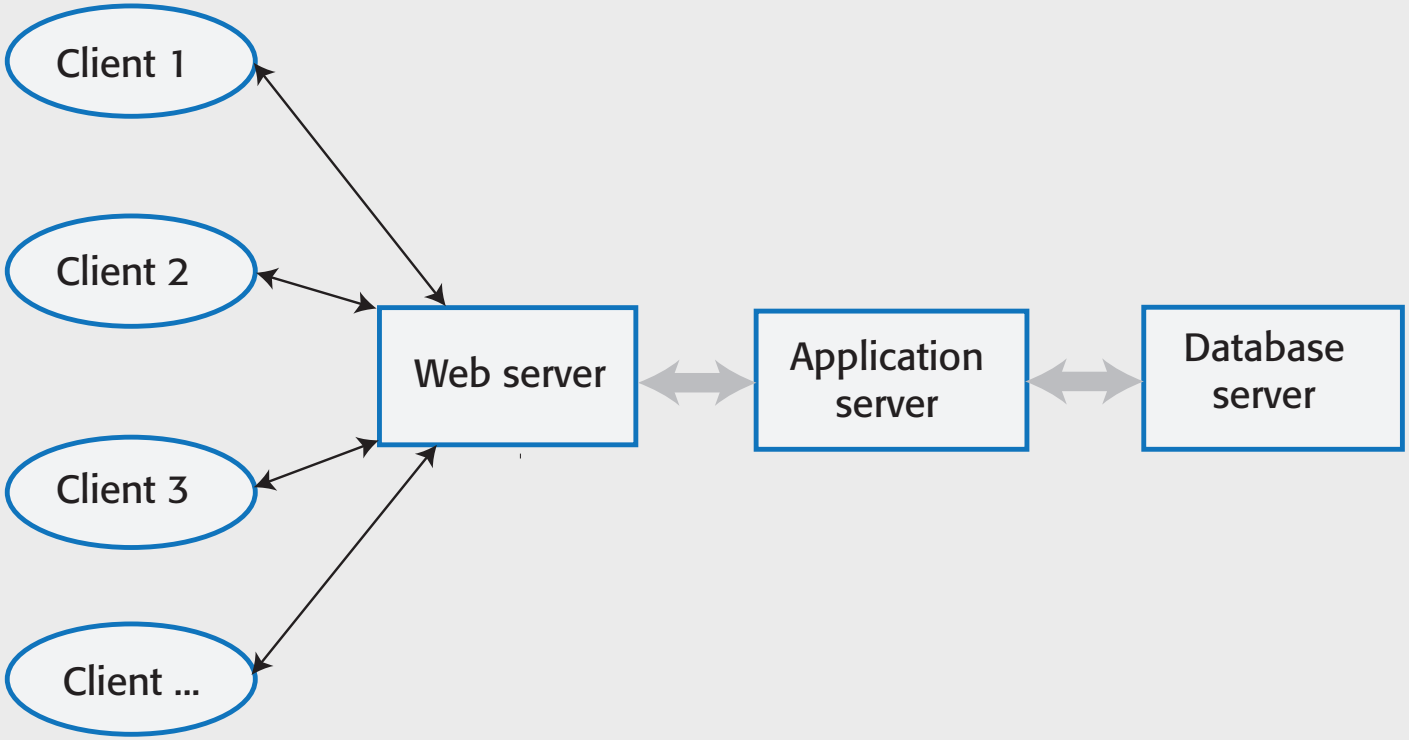
MVC: a popular architecture for web applications



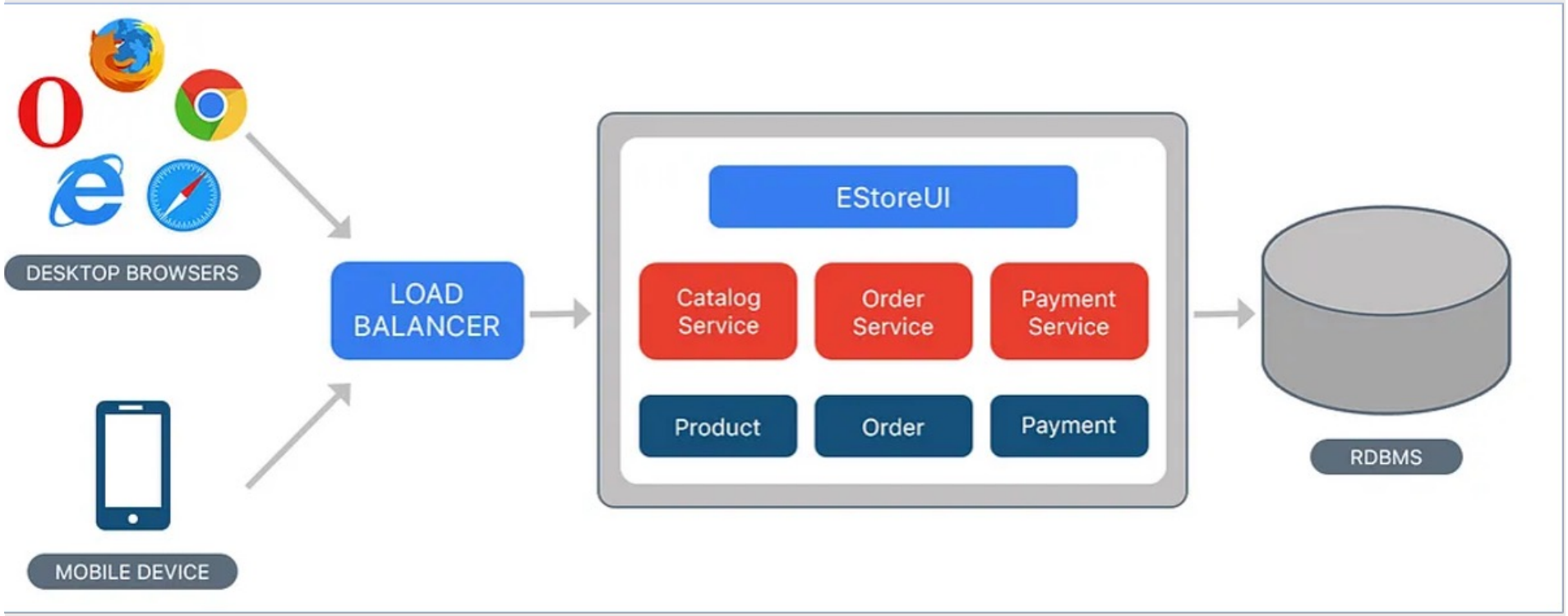
Client-server communication

- Client-server communication normally uses the HTTP protocol.
  - The client sends a message to the server that includes an instruction such as GET or POST along with the identifier of a resource (usually a URL) on which that instruction should operate. The message may also include additional information, such as information collected from a form.
- HTTP is a text-only protocol so structured data has to be represented as text. There are two ways of representing this data that are widely used, namely XML and JSON.
  - XML is a markup language with tags used to identify each data item.
  - JSON is a simpler representation based on the representation of objects in the Javascript language.

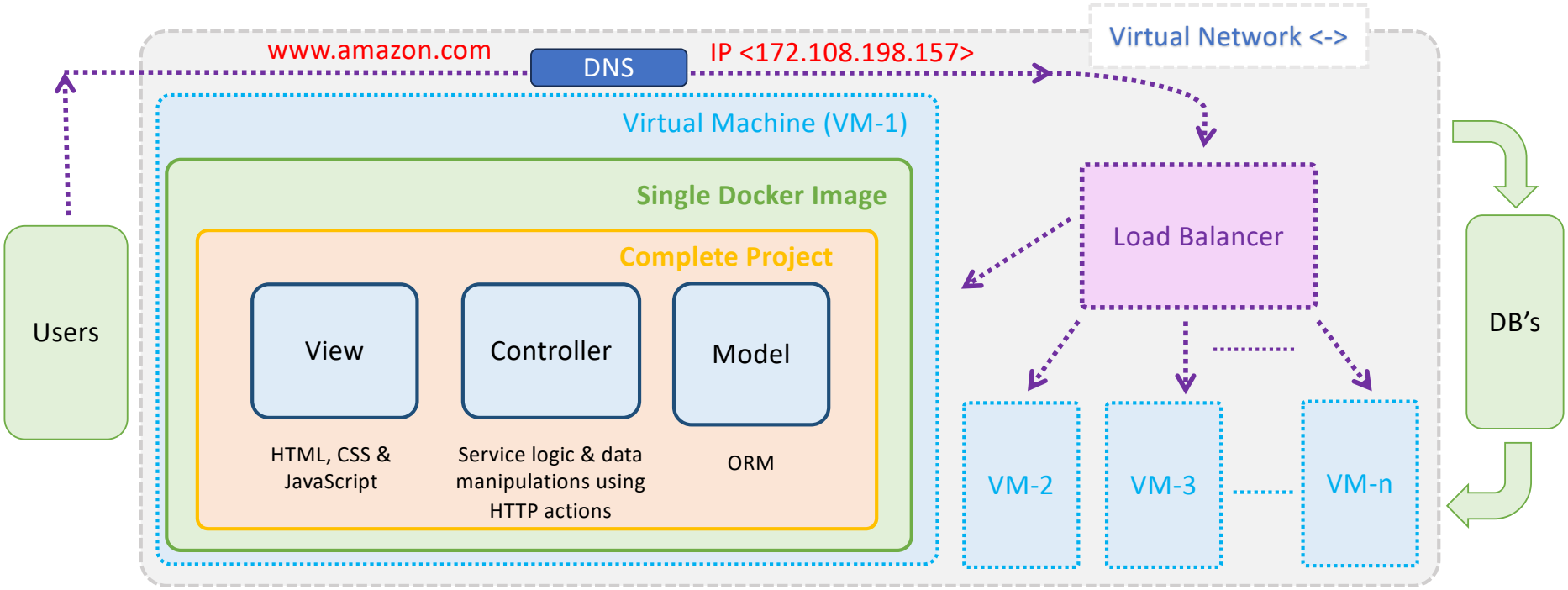
Multi-tier client-server architecture



Monolithic Architecture



Monolithic MVC



Advantages:

- Simple design and easy monitoring.
- Low network latency and easy deployment.

Dis-advantages:

- More resource utilization and Low scalability.
- Difficult to upgrade or revert the code and technologies.

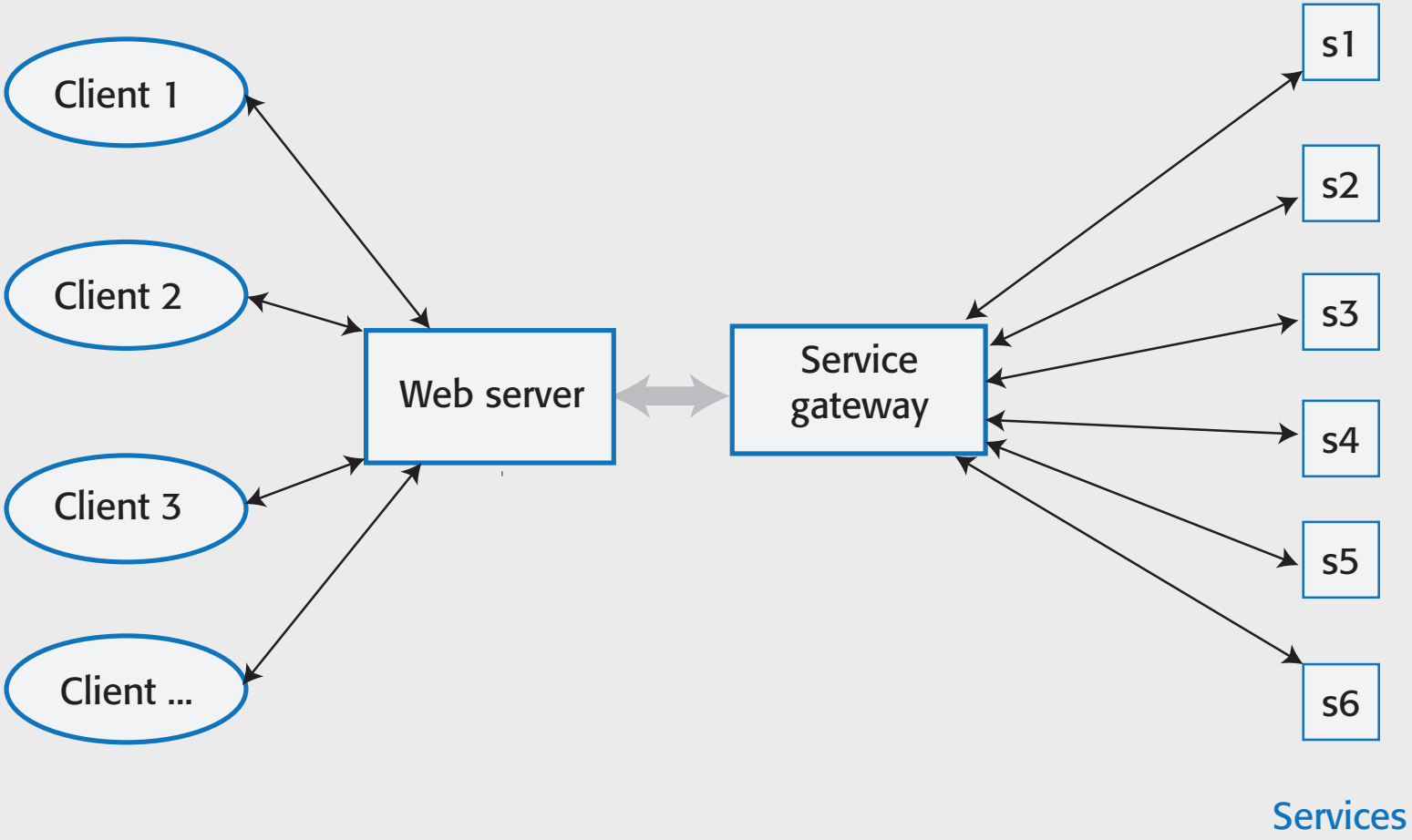
Service-oriented architecture (SOA)

- 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.

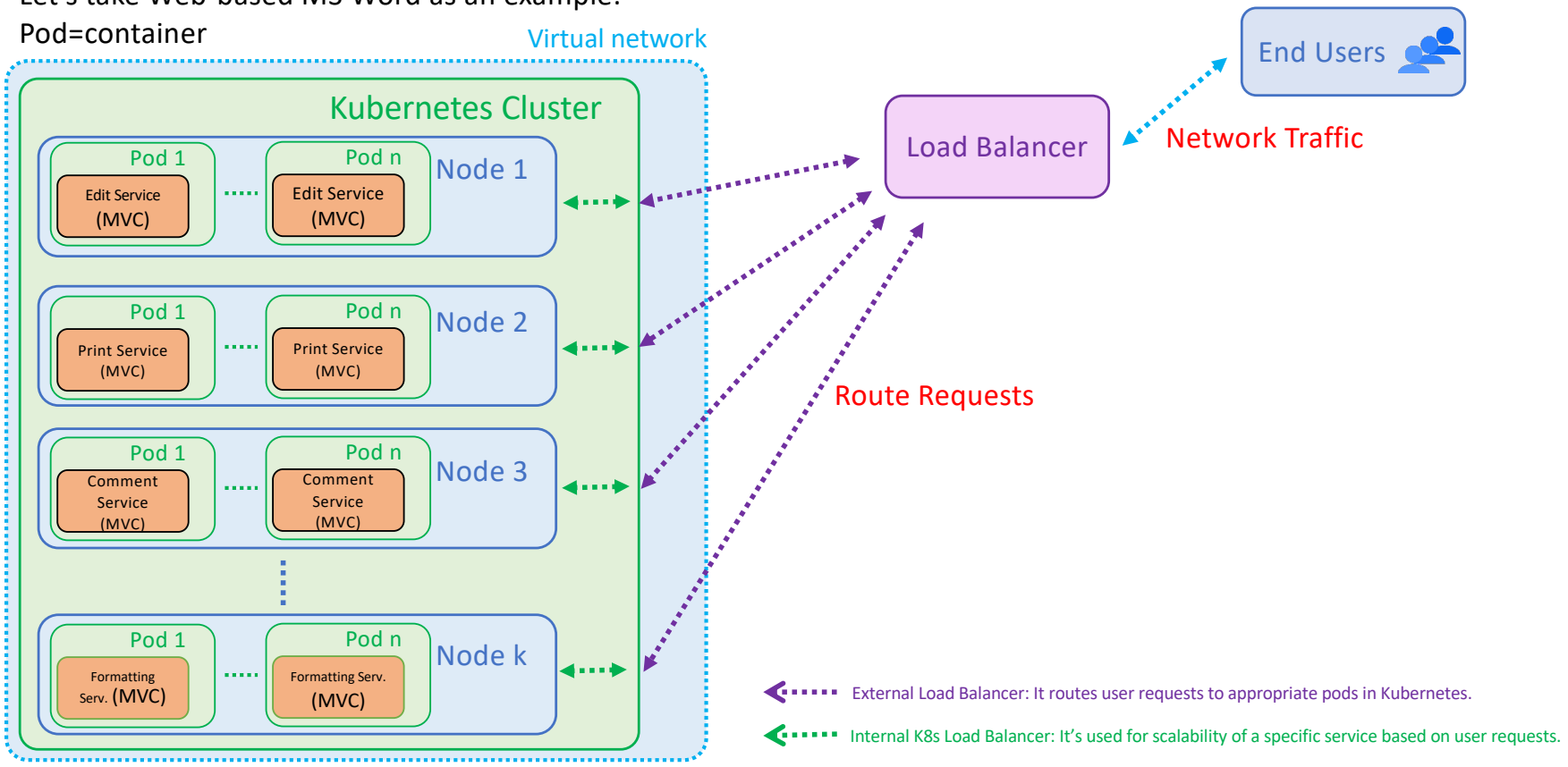
Service-oriented architecture (SOA)



SOA

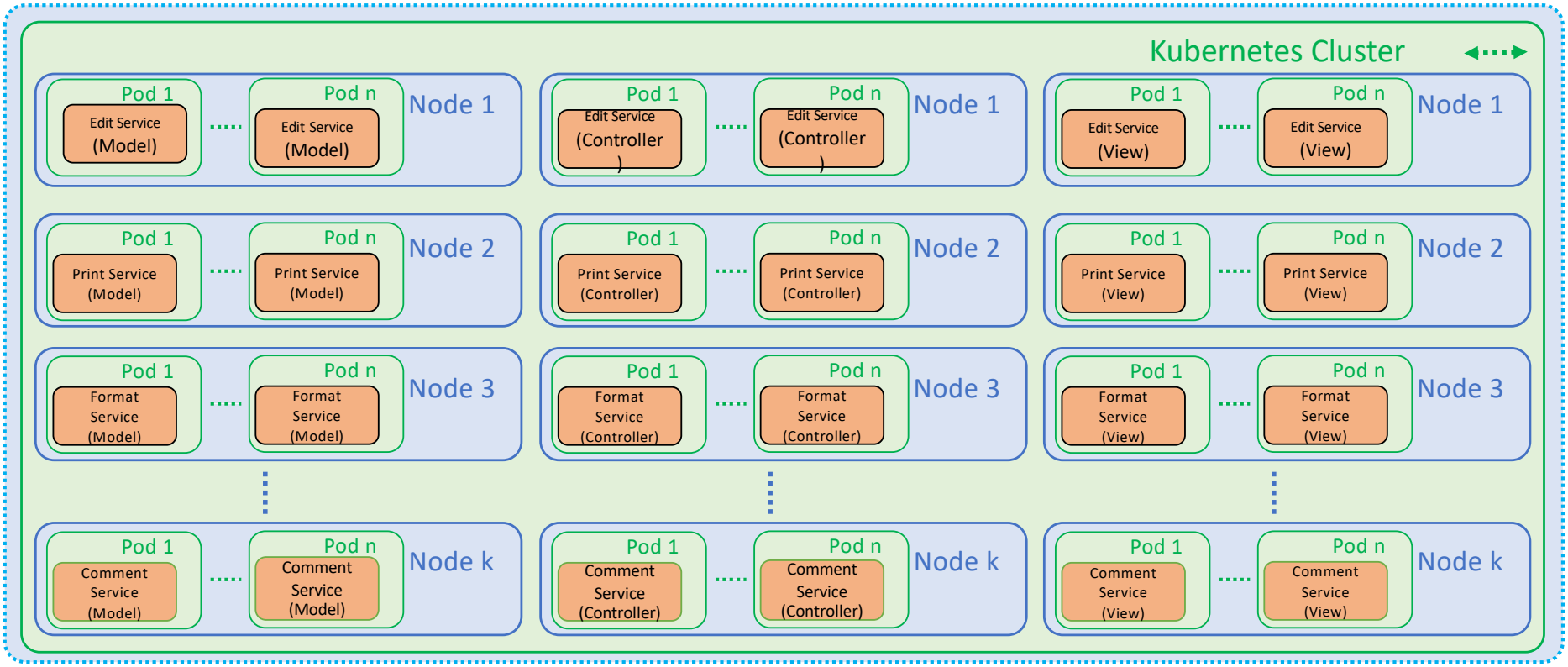
Service Oriented Architecture (SOA)

Let’s take Web-based MS Word as an example:
Pod=container



Microservice Architecture

Let's take web-based MS Word as an example:



SOA vs MVC

Purpose:

- SOA is for large applications by breaking them down into smaller, loosely coupled services that are developed, deployed, and scaled independently.
- MVC is a design pattern used for organizing the components and logic of a software application, especially in the context of web applications.

Granularity:

- SOA: Microservices are typically fine-grained, with each service focused on a specific function or feature.
- MVC doesn't prescribe a specific granularity for components. It can be applied at the application level, module level, or even within UI components

Independence:

- Microservices are developed, deployed, and scaled independently of each other. Each microservice can have its own technology stack, database.
- MVC separates concerns within one application but doesn't necessarily enforce the same level of independence between different components.

Scalability:

- Microservices provide inherent scalability as individual services can be scaled independently based on demand.
- MVC doesn't inherently provide scalability at the architectural level.

Technology Stack:

- Each microservice can use its own technology stack and database, which can be chosen based on the specific requirements of that service.
- MVC: MVC is a design pattern that can be applied within various technology stacks, including Java, .NET, Ruby on Rails, and more.

- Use Cases:

- Microservices: Microservices are suitable for large, complex applications with multiple functionalities that can be split into independent services, such as e-commerce platforms, social media networks, and distributed systems.
- MVC is often used for structuring web applications, desktop applications, and other software systems where a clear separation of concerns is needed.

Issues in architectural choice

- Data type and data updates
  - For structured data, it is usually best to have a single shared database that provides locking and transaction management.
  - For distributed data across services, you need a way to keep it consistent and this adds overhead to your system.
- Change frequency
  - If some system components will be regularly updated, then isolating these components as separate services simplifies those changes.
- The system execution platform
  - For cloud-based applications, use SOA, because scaling the app is simpler
    - This is known as “distributed software architecture”!
  - For business apps running on local servers, a multi-tier architecture is suitable

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? (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 include assumptions about the software being developed that limit your architectural choices?

Database

- There are two kinds of database that are now commonly used:
  - Relational databases, where the data is organized into structured tables
  - NoSQL databases, in which the data has a more flexible without transaction!

- Relational databases (e.g., MySQL) are suitable for situations where you need transaction support, and the data structures are known and simple.

- NoSQL databases, such as MongoDB, are more flexible and potentially more efficient than relational databases for data analysis.
  - NoSQL databases allow data to be organized hierarchically rather than as flat tables and this allows for more efficient concurrent processing of ‘big data’.

Delivery platform: Web? Mobile? Both?

- Mobile issues:
  - *Intermittent connectivity* You must be able to provide a limited service without network connectivity.
  - *Processor power* Mobile devices have less powerful processors, so you need to minimize computationally-intensive operations.
  - *Power management* Mobile battery life is limited so you should try to minimize the power used by your application.
  - *On-screen keyboard* On-screen keyboards are slow and error-prone. You should minimize input using the screen keyboard to reduce user frustration.
- To deal with these differences, you usually need separate browser-based and mobile versions of your product front-end.
  - You may need a different architecture in these different versions to ensure that performance and other characteristics are maintained.

Server: Local vs Cloud

- A key decision is whether to design your system to run on customer servers or to run on the cloud.
- For "consumer products" that are not simply mobile apps it almost always makes sense to develop for the cloud.
- For business products, it is a more difficult decision.
  - Some businesses are concerned about cloud security and prefer to run their systems on in-house servers.
  - They may have a predictable pattern of system usage so there is less need to design the system to cope with demand changes
- An important choice you have to make if you are running your software on the cloud is which cloud provider to use.

Open source

- Open-source software is software that is available freely, which you can change and modify as you wish.
  - The advantage is that you can reuse rather than implement new software, which reduces development costs and time to market.
  - The disadvantages of using open-source software is that you are constrained by that software and have no control over its evolution.
- The decision on the use of open-source software also depends on the availability, maturity and continuing support of open-source components.
- Open-source license issues may impose constraints on how you use the software.

Development tools

- Development technologies, such as a mobile development toolkit or a web application framework, influence the architecture of your software.
  - These technologies have built-in assumptions about system architectures and you have to conform to these assumptions to use the development system.
- The development technology that you use may also have an indirect influence on the system architecture.
  - Developers usually favor architectural choices that use familiar technologies that they understand.
  - For example, if your team have a lot of experience of relational databases, they may argue for this instead of a NoSQL database.

Key points 1

- 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.
- 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.

Key points 2

- 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.
- 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.

1. Layered Architecture - Smart Home Automation System

- **User Scenario:** Imagine a homeowner who wants to enhance the convenience, security, and energy efficiency of their home. They desire to control and monitor various smart devices such as thermostats, lights, security cameras, and door locks through a centralized system.

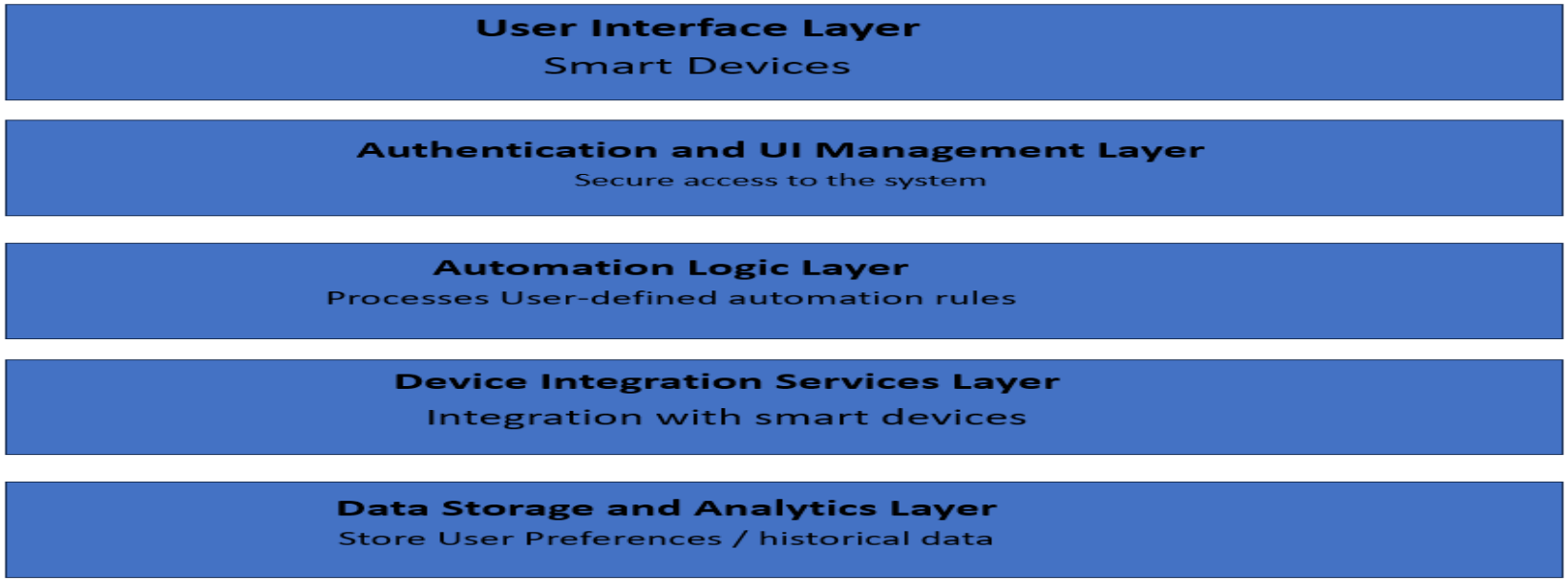
- **Use Case Scenarios:**

1. **Morning Automation:** The system adjusts the thermostat, opens smart blinds, and turns on lights based on the time of day when the homeowner wakes up.

2. **Away Mode:** When the homeowner leaves, the system ensures all lights are turned off, the thermostat is set to an energy-efficient mode, and security cameras are activated.

3. **Energy Optimization:** The system analyzes energy consumption patterns and suggests optimizations, such as adjusting thermostat settings or recommending energy-efficient lighting schedules.

4. **Security Alerts:** The system sends alerts to the homeowner's mobile app in case of unusual activity, such as unexpected door openings or motion detected when the house is supposed to be empty.



❖ Layered Architecture Explanation:

1. **User Interface:** You use a simple mobile app to control your smart home—changing settings, turning things on or off, all at your fingertips. Allows homeowners to control and monitor smart devices, set preferences, and receive notifications.

2. **Authentication and UI Management: :** It ensures your app is secure, like having a digital key. It also makes sure the app looks neat and organized for you. Ensures secure access to the system, manages the rendering of UI components displaying real-time home status and device controls.

3. **Automation Logic Layer:** This layer is like the smart brain of your home. It follows rules you set, making decisions to create a comfy and efficient living space. Processes user-defined automation rules based on factors like time, occupancy, and external conditions to optimize home automation.

4. **Device Integration Services:** It's the translator that makes your smart devices talk to each other. Your lights, thermostat, and cameras work together seamlessly. Integrates with various smart devices, facilitating seamless interaction between the smart home system and connected devices.

5. **Data Storage and Analytics:** This layer remembers your preferences and suggests ways to make your home even better. It's like a helpful assistant, always learning from your choices. Stores user preferences, historical data, and provides analytics services for homeowners to review energy consumption patterns and optimize their smart home system.

❖ Justification for Layered Architecture in Smart Home Automation System:

1. Modularity: Each layer has a specific responsibility, making the system modular and easy to understand and modify. New devices or features can be added without affecting the entire system.

2. Scalability: Individual layers can be scaled independently based on the number of connected devices or increasing user load.

3. Flexibility: The layered architecture accommodates a variety of smart devices and allows for easy integration of new devices into the system.

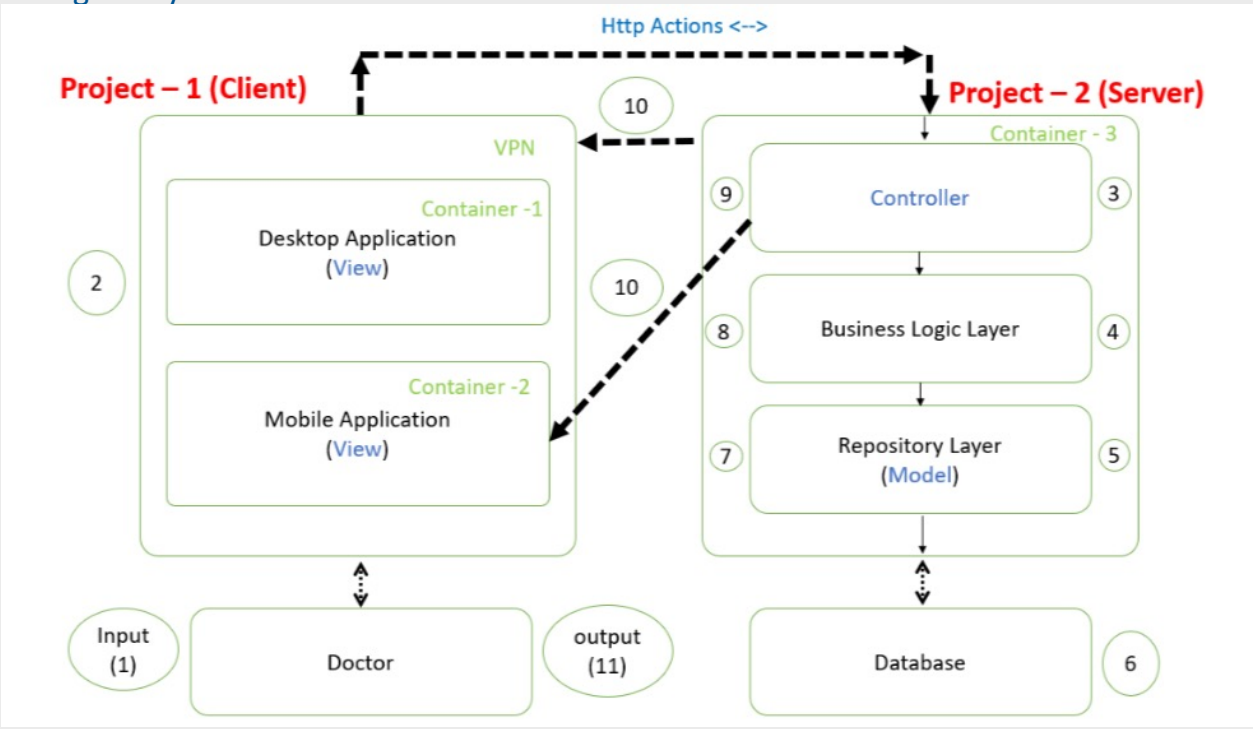
4. User Experience: The separation of the user interface and automation logic layers ensures a responsive and intuitive user experience for homeowners.

5. Security: The authentication layer ensures secure access, and the device integration services layer facilitates secure communication between the system and connected devices, enhancing overall security.

Clinical Decision Support System (CDSS)

User Scenario: Imagine a healthcare setting where doctors need quick and accurate assistance in making clinical decisions for their patients. The Clinical Decision Support System helps them by providing evidence-based recommendations, relevant patient data, and alerts about potential issues. The app must support both the mobile and web versions.

- **Medication Decision Support:** The CDSS assists doctors in prescribing medications by cross-referencing patient health records, allergies, and potential drug interactions.
- **Diagnostic Assistance:** When doctors are unsure about a diagnosis, the CDSS analyzes patient symptoms, medical history, and test results to suggest potential diagnoses and relevant investigations.
- **Alerts for Critical Conditions:** The system generates alerts for critical conditions or potential complications based on real-time patient data, ensuring timely intervention.



3. Inventory Management System

- **Software System:**

The Inventory Management System is designed to streamline the tracking, management, and control of an organization's inventory. This system serves as a centralized platform where businesses can effectively monitor their stock levels, manage orders, and optimize supply chain operations. User access controls and seamless integration with other business systems contribute to enhanced data security and a cohesive workflow. This system is pivotal for businesses seeking efficient inventory operations, accurate order fulfillment, and optimized supply chain processes.

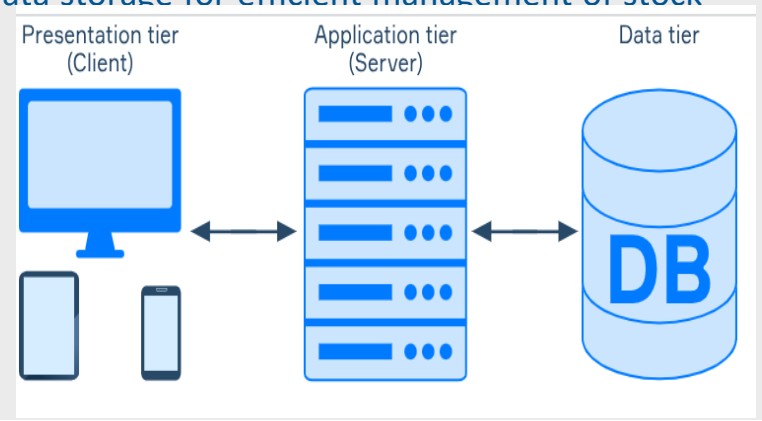
3. Multi-Tier Client-Server Architecture for Inventory Management

1. **Client Tier (User Interface):** Web or desktop interfaces for users to manage inventory, place orders, and track stock levels. Users use an application or website to check product availability, place orders, and monitor inventory status. Users interact with the Inventory Management System through a user-friendly app or desktop interface. They manage inventory, place orders, and monitor stock levels.

2. **Application Tier (Inventory Management Engine):** Central server hosting the inventory management engine and business logic. The intelligent system that tracks stock levels, processes orders, and manages inventory across multiple locations. The central server processes user requests, manages stock levels, and handles order processing. It ensures efficient inventory management across multiple locations.

3. **Database Tier (Data Storage):** Server storing product information, stock levels, order records, and supplier details. The secure storage room where product details, stock levels, and order history are stored for efficient inventory management. Product information, stock levels, order records, and supplier details are securely stored on a dedicated server. This ensures data integrity and provides a centralized repository for inventory-related information.

This example demonstrates the multi-tier client-server architecture in the context of an Inventory Management System, highlighting user interactions, inventory tracking, and secure data storage for efficient management of stock and orders.

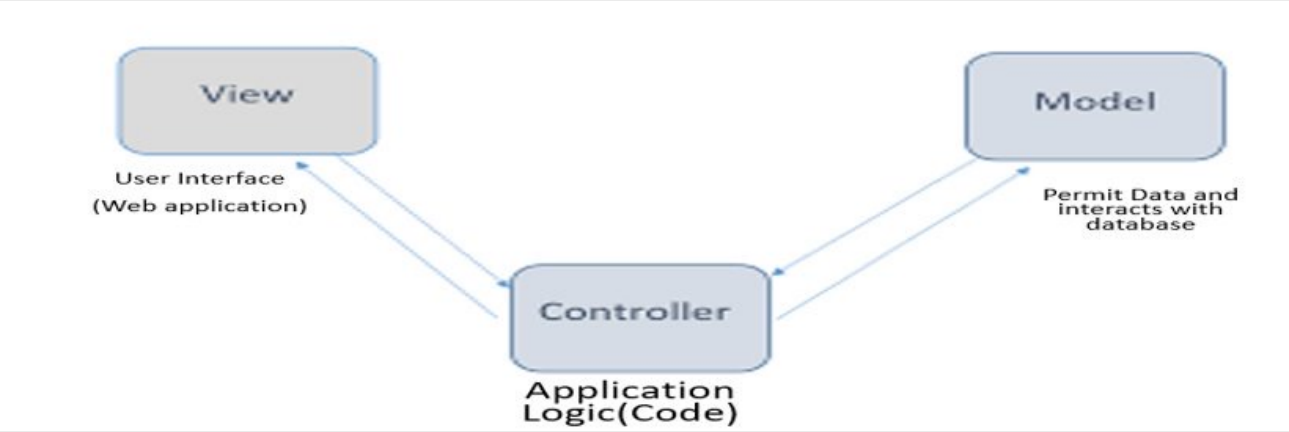


4. IPermit System

An iPermit System is an online platform designed to streamline and manage the process of obtaining permits. The iPermit System revolutionizes permit processes, offering a seamless digital experience for applicants and authorities. Users submit applications, upload documents, and receive automated updates, ensuring efficiency and transparency. The system streamlines approvals, facilitates secure online payments, and maintains a comprehensive digital record for monitoring and reporting, making it indispensable for permit management. The system is typically used in various contexts, such as by local authorities, educational institutions, or organizations that require a structured and efficient way to handle permit requests.

4. Model View Controller Architecture - IPermit System

- **Model (Permit Data / Interacts with database):** The Model in the iPermit System manages the permit data, and interactions with the database. It ensures that permit requests are processed accurately, data is validated, and information is securely stored. The brain of the iPermit System that handles permit requests, validates data, and communicates with the database for storage.
- **View (User Interface):** Users interact with the iPermit System through a user-friendly app or website. The View allows users to submit permit requests, track the status, and receive notifications about their permits. The screens users interact with to fill out permit applications, check the status, and receive updates.
- **Controller (Application Logic):** The central server acts as the Controller, coordinating between the Model and View. It handles user input from the View, processes it using the Model's logic, and updates the View accordingly. The Controller ensures a smooth flow of information and actions within the system. The coordinator that takes user input from the view, processes it using the model's logic, and updates the view accordingly.



Justification for MVC in Ipermit:

- **Component Isolation:** Permit data management and business logic (Model) are handled independently from user interfaces (View) and application logic (Controller). This separation makes the system modular, easier to understand, and maintain.
- **Scalability and Reusability:** As the IPermit System evolves, developers can scale or modify the Model, View, or Controller without affecting the other components. This makes it easier to adapt to changing permit requirements or user interface enhancements.
- **Maintainability and Testability:** Maintenance becomes more straightforward since modifications to the permit data logic (Model) or user interface (View) won't impact the entire system. Additionally, testing becomes more efficient as individual components can be tested in isolation.
- **User Interface Flexibility:** The IPermit System can have various interfaces (web, mobile) presenting permit information in a way that suits each platform. The View component can be adapted or extended without affecting the underlying permit data management (Model) or application logic (Controller).
- **Improved Collaboration:** Developers working on permit data logic can collaborate independently of those working on user interfaces or application logic. This division of labour enhances productivity and allows for parallel development.

5. Ticketing System

- A Ticketing System is a digital platform designed to efficiently manage and streamline the process of handling service requests, support issues, or inquiries. It serves as a centralized hub where users can submit, track, and resolve various types of tickets or requests. The system is commonly used in customer support, IT help desks, and environments to ensure a structured and organized approach to handling user queries.

❖ Service Oriented Architecture Explanation:

- **Services (Functional Components):** The Ticketing System is broken down into independent services, each responsible for a specific function. For example, one service might handle ticket creation, another manages status updates, and so on. This modular approach allows for flexibility and scalability.

- **Service Registry (Directory of Services):** The Service Registry acts as a centralized directory that keeps track of all available services within the system. It helps components discover and communicate with each other efficiently. A master list that tells the system which services are available and what tasks they can perform.

- **Service Consumer (User Interface):** Users interact with the Ticketing System through a user-friendly app or website. This component, known as the Service Consumer, connects with the relevant services to create tickets, check status, and manage support requests.

❖ Justification for SOA in Ticketing System:

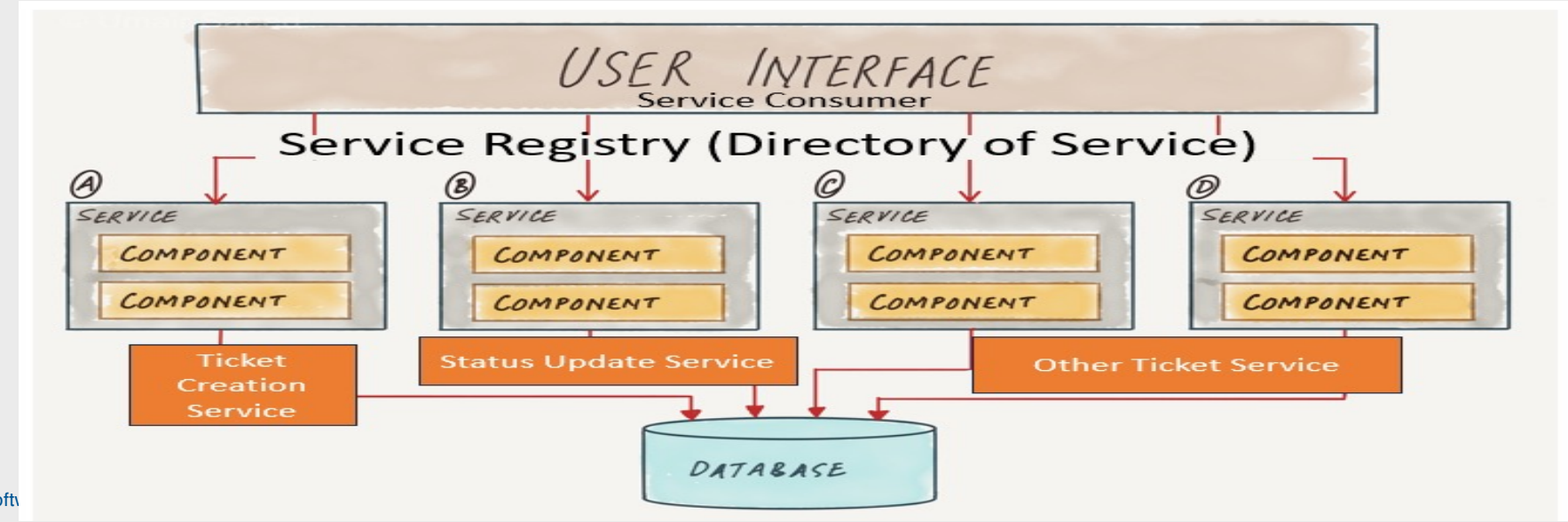
1. **Flexibility and Scalability:** As the ticketing system evolves, new services can be added or modified without affecting the entire system. This flexibility supports the growth and adaptation of the system over time.

2. **Interoperability:** Different services, such as ticket creation and status updates, can work together cohesively, enhancing the overall functionality of the ticketing system.

3. **Reusability of Services:** Common functionalities, like user authentication or notification services, can be reused across various parts of the ticketing system, reducing redundancy and improving maintenance.

4. **Easier Maintenance:** Developers can make changes or improvements to specific services without disrupting the entire system, making maintenance more efficient and manageable.

5. **Scalable User Interface:** As new features or functionalities are introduced, the user interface can seamlessly integrate with additional services, providing an enhanced experience for users.



Architecture 1 3tier deployment

Example

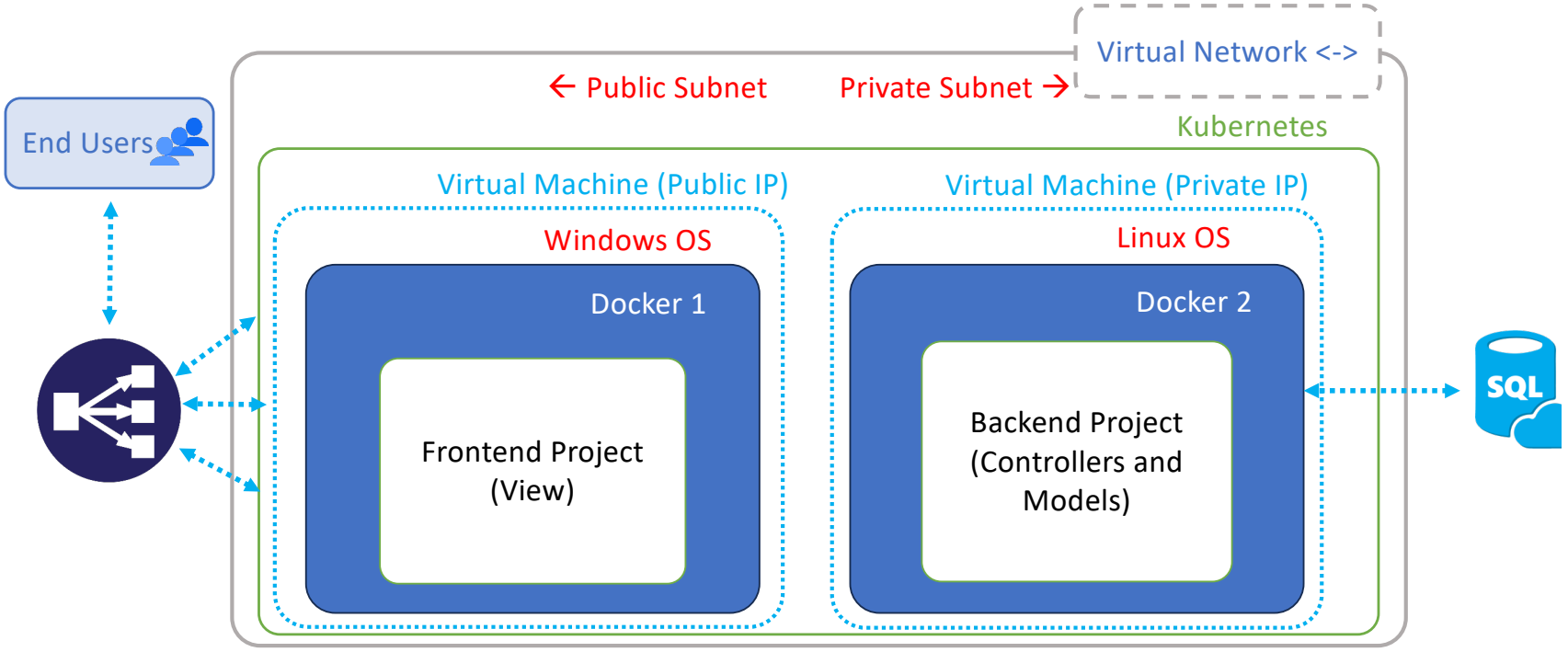
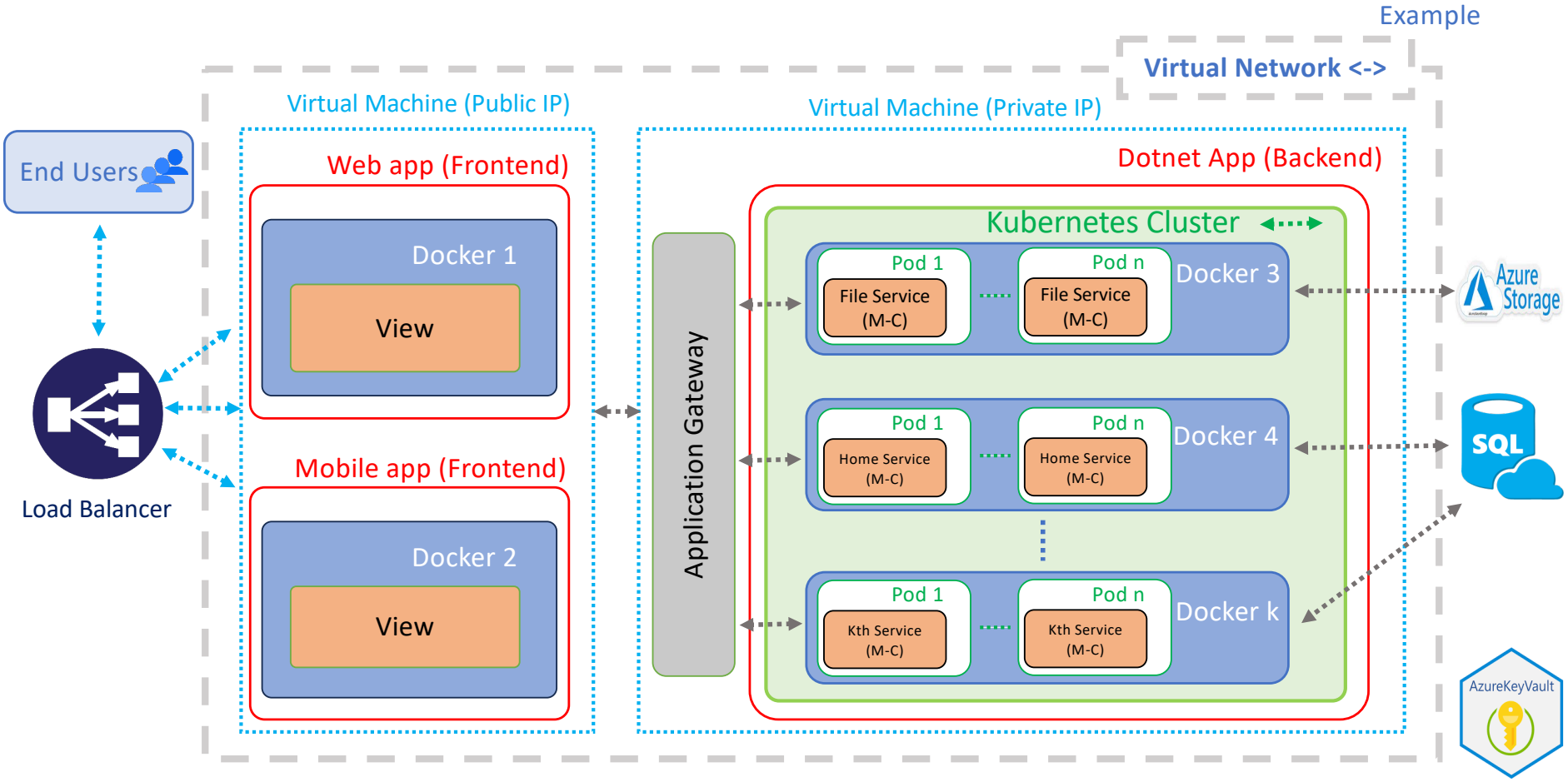


Fig: Combination of monolithic and Distributed Architecture.

Explanation of Architecture 1

- By this architecture, we retrieved the advantage of monolithic architecture which in terms of simple design, easy maintenance, low network latency and easy deployment. Along with this, we also achieved microservice architecture advantages like scalability with internal Load balancer which is configured in Kubernetes.
- By this architecture we can serve our application in cross platforms or single operating system with different versions which will helps when our different packages or frameworks in our application need to compatible with different OS or OS Versions.
- Segregation of API from UI with Private IP will give more security to our database. SQL server only interacts with API private IP which cannot be directly accessible with browser.

Architecture 2 (Enhanced): Hybrid of 3tier and SOA



- This architecture is a combination of monolithic UI and distributed (SOA) based API projects achieved by microservices. By this architecture, we achieved all of advantages of multi architecture discussed in above example along with code or service reusability and less resource utilization.