Electronics and
Computer Science

University of Southampton

Software Architecture

COMP6226: Software Modelling Tools and Techniques for Critical Systems

Dr A. Rezazadeh (Reza)
Email: ra3@ecs.soton.ac.uk or ar4k06@soton.ac.uk

October 24

University of Southampton

Overview

- What Is Software Architecture?
- Why is software architecture important?
- The software architect role – Making architecture work through collaboration
- What Makes a “Good” Architecture?
- Why Architectural Design?
- Architecture and Abstraction
- Architecture and Modularity
- The Role of Architecture – Managing Complexity
- The Role of Architecture – Cohesion
- The Role of Architecture –Coupling

What Is Software Architecture?

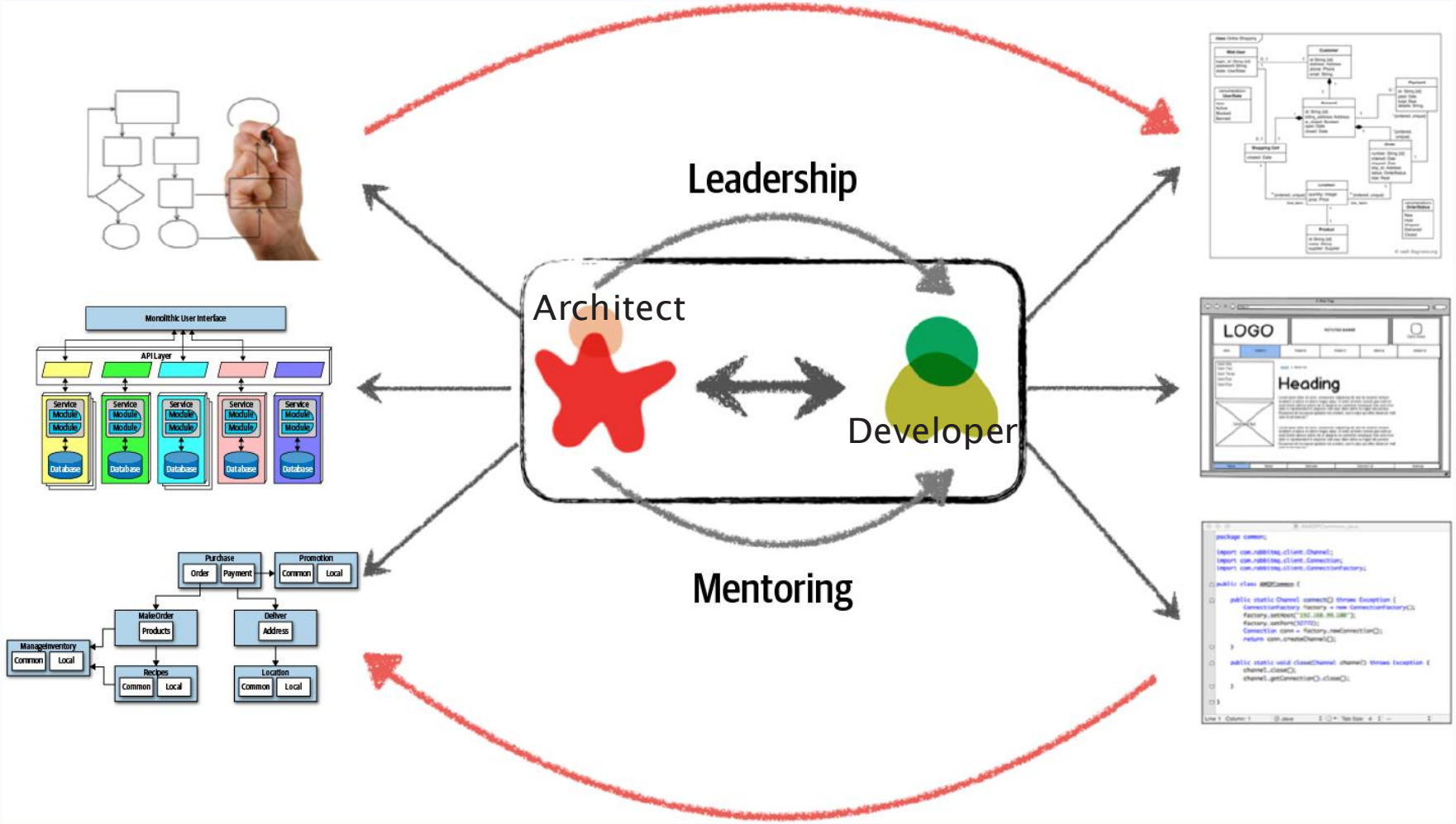
- The definition of software architecture has long been argued.
- For some people, it's the way essential compounds are wired together or the fundamental organisation of a system.
  - In this regard, the abstract concept proposed by Ralph Johnson, Associate Professor, University of Illinois, is something noteworthy:
  - "*Architecture is about the important stuff. Whatever that is.*"
- A high-level view of a software system, which describes aspects such as:
  - The software components and component dependencies and interconnections within the system.
  - Links and interfaces to any external systems.
  - The distribution structure

Why is software architecture important?

- Architectural designs can help to plan, manage and document a software development, to:
  - Aid project management.
  - Assign different component developments to different teams.
  - Centralise coordination of how the system components should fit together.
  - Document decisions about frameworks, platforms, structure, etc.
  - Document how non-functional properties of systems are achieved, such as performance, reliability, security, etc.
  - Document and modularise the system to aid in future maintenance and evolution.

University of Southampton

The software architect role – Making architecture work through collaboration



University of Southampton

What Makes a "Good" Architecture?

- There is no such thing as an inherently good or bad architecture.
- Architectures are either more or less fit for some purpose.
- For example, a three-tier layered service-oriented architecture may be just the ticket for a large enterprise's web-based B2B system but completely wrong for an avionics application.

Why Architectural Design?

- Making architecture explicit has main benefits or objectives:
  - Clarifying ideas and decisions: Architectural concepts provide a vocabulary that can aid our understanding of an application's high-level design (model).
  - Analysing properties of the architectures: The architecture can help with checking for consistency across different characteristics of a design.
  - Stakeholder communication and understanding: Sharing architecture information among members of a development team, or between different teams.
  - Large-scale reuse: the high-level components com- prising a system architecture may assist with identifying where there is scope for reuse.
  - Management: Anticipating how an application may evolve is (or should be) a factor in determining its architectural form.

Architecture and Abstraction

- At one level, architecture is about the structure of individual services, applications and programs.
  - At this level, we are concerned with the internal structure and decomposition of a single program.
- At another level, architecture is about sophisticated enterprise systems that comprise systems of systems.
  - Enterprise systems often involve interacting collections of programs harnessed to create some overarching and coordinated set of functions.
- Hence, we use architecture to simplify and clarify different levels of abstraction.
  - Abstraction is the process of hiding implementation details from users.

Architecture and Modularity

- If an architect designs a system without paying attention to how the pieces wire together, they end up creating a system that presents myriad difficulties.
- A module is defined as “each of a set of standardised parts or independent units that can be used to construct a more complex structure.”
- For discussions about architecture, we use modularity as a general term to denote a related grouping of code: classes, functions, or any other grouping.
  - This doesn’t imply a physical separation, merely a logical one;

Software Architecture – Modularity

- Modularity is important principle of software architecture.
  - This principle states that software systems should be designed as a collection of independent modules.
- Each module should be responsible for a specific task or function.
  - Modules should be loosely coupled, meaning that they should not depend on each other too much.
- Modularity helps to improve the scalability and maintainability of software systems.
  - It also makes it easier to test and debug the system.

University of Southampton

The Role of Architecture – Managing Complexity

- Complexity in designing software makes the software development process difficult.
- Reducing the complexity is the biggest challenge in software design.
- How is this done?
- The best way to reduce complexity is by dividing the entire design into manageable parts.

The Role of Architecture – Defining structure and behaviour

- Software architectures consist of three main parts:
  - Components – representing computational/data storage nodes.
  - Connectors – for information interchange between components.
  - Configurations – which instantiate components and connectors in particular arrangements.
- Having an overview of the architectural dynamics helps us answer questions such as the following:
  - Which are the main components of my system?
  - How do these components interact and evolve?
  - What resources will I need and what costs will I have in the development process?
  - What are the areas where I can predict change?

The Role of Architecture – Cohesion

- *Cohesion* refers to what extent the *parts of a module* should be *contained* within the *same module*.
- In other words, it is a measure of how *related the parts* are to one another.
- Ideally, a *cohesive* module is one where all the parts should be packaged together.
  - Because breaking them into smaller pieces would require *coupling* the parts together via calls between modules to achieve useful results.
- Attempting to divide a cohesive module would only result in increased *coupling* and decreased readability.

Types of Cohesion

- *Functional cohesion*
  - Every part of the module is related to the other, and the module contains everything essential to function.
- *Sequential cohesion*
  - Two modules interact, where one outputs data that becomes the input for the other.
- *Communicational cohesion*
  - Two modules form a communication chain, where each operates on information and/or contributes to some output.
  - For example, add a record to the database and generate an email based on that information.

Types of Cohesion – Cont.

- *Logical cohesion*
  - The data within modules is related logically but not functionally.
  - For example, consider a module that converts information from text, serialised objects, or streams.
  - Operations are related, but the functions are quite different.
  - A common example of this type of cohesion exists in virtually every Java project in the form of the StringUtils package: a group of static methods that operate on String but are otherwise unrelated.

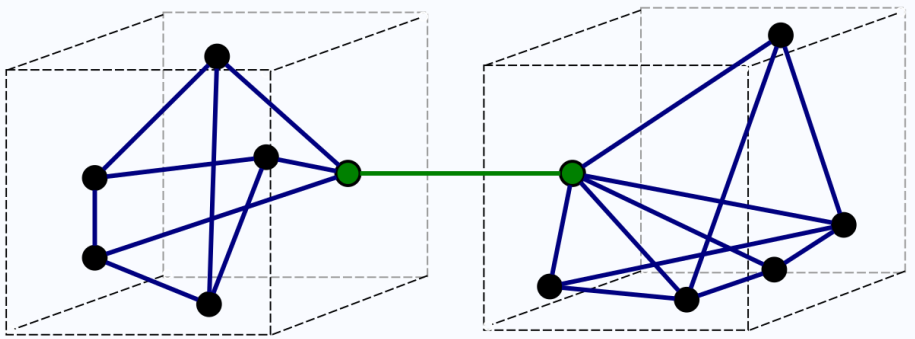
Types of Cohesion – Cont.

- *Procedural cohesion*
  - Two modules must execute code in a particular order.
- *Temporal cohesion*
  - Modules are related based on timing dependencies.
  - For example, many systems have a list of seemingly unrelated things that must be initialized at system startup; these different tasks are temporally cohesive.
- *Coincidental cohesion*
  - Elements in a module are not related other than being in the same source file; this represents the most negative form of cohesion.

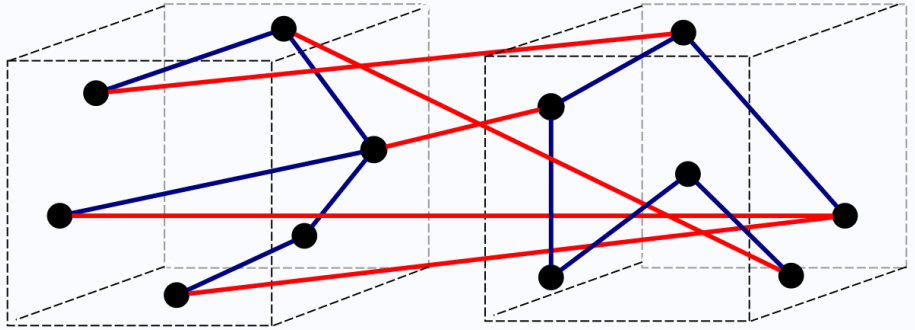
University of Southampton

The Role of Architecture –Coupling

- Coupling refers to the degree of interdependence between different modules, classes, or components of a software system.



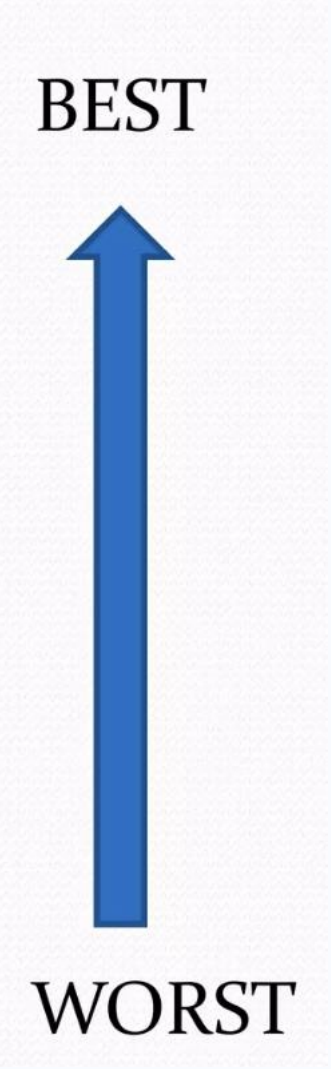
a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

Types of Coupling

1. Data Coupling: When modules shared primitive data between them.
2. Stamp Coupling: When modules shared composite or structural data between them, and It must be a non-global data structure. for example, Passing object or structure variable in react components.
3. Control Coupling: When data from one module is used to direct the structure of instruction execution in another.
4. External Coupling: When two modules shared externally imposed data type that is external to the software like communication protocols, device interfaces.
5. Common Coupling: When two modules shared the same global data & dependent on them, like state management in JavaScript frameworks.
6. Content Coupling: When two modules shared code and can modify the data of another module, which is the worst coupling and should be avoided



Software Architecture Patterns

- Software architecture patterns can be defined as solutions to mainstream and recurring software engineering problems.
- An architectural pattern is a set of architectural design decisions that address recurring design problems in various software development contexts.
  - It offers rules and principles for organizing the interactions between predefined subsystems and their roles.

Software architecture pattern vs. design pattern

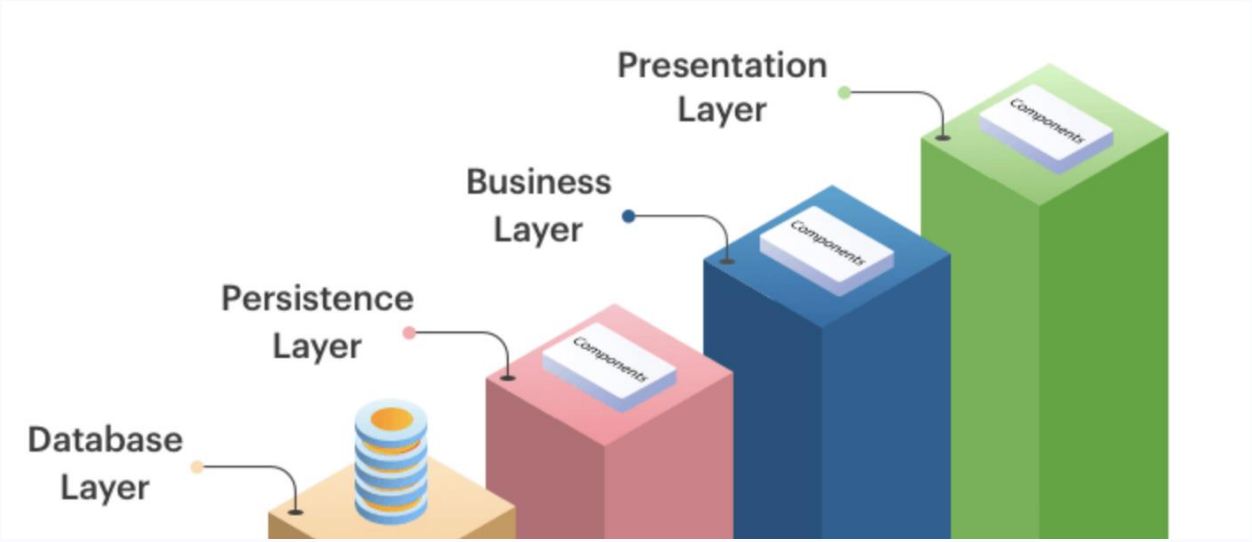
	Architecture Patterns	Design Patterns
Defination	Fundamental structural organization for software systems	Specification that could help in implementation of a software
Role	Conversion of software characteristics to a high-level structure.	Description of all the units of the software system to support coding
Example	Microservice, serverless and event-driven	Creational, structural and behavioral
Level	Large Level tool - concerns large scale components, global properties, and mechanism of the system	Small level tool- concerns schemes for refining and building smaller subsystems - structure and behavior of entities and their relationships
Problem Addressed	Distributed functionality, system partitioning, protocols, interfaces, scalability, reliability, security	Problems in software construction

Different types of software architecture pattern

- Layered Architecture Pattern
- Event-driven Architecture Pattern
- Microkernel Architecture Pattern
- Microservices Architecture Pattern
- Space-Based Architecture Pattern
- Client-Server Architecture Pattern
- Master-Slave Architecture Pattern
- Pipe-Filter Architecture Pattern
- Model-view-controller pattern
- Peer-to-Peer Architecture Pattern

Layered Architecture Pattern

- Multi-layered, aka tiered architecture, or n-tier architecture.
- This pattern stands out because each layer plays a distinct role within the application and is marked as closed.
  - It means a request must pass through the layer below it to go to the next layer.

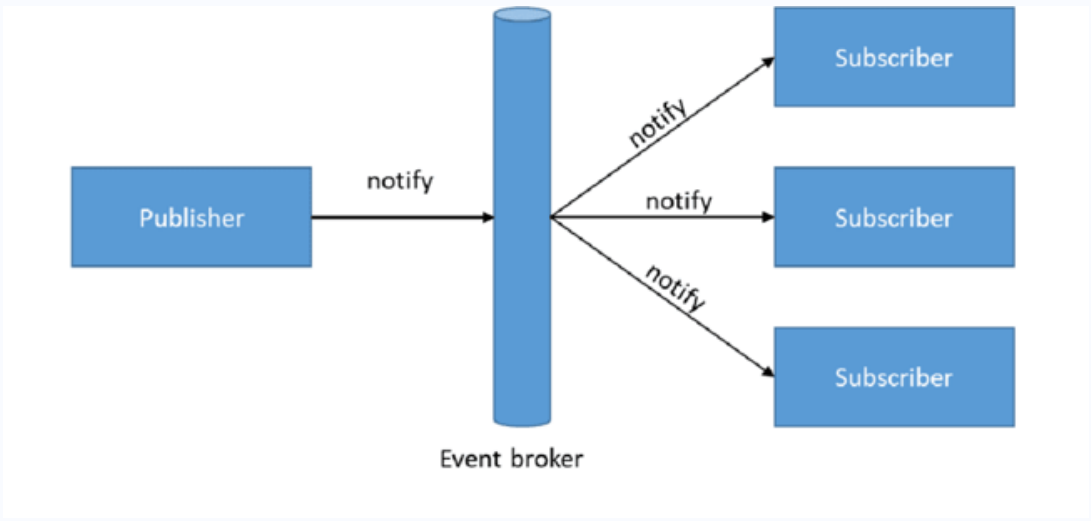


- Another one of its concepts – layers of isolation – enables you to modify components within one layer without affecting the other layers.

Layered Architecture Pattern

- Usage:
  - Applications that are needed to be built quickly.
  - Enterprise applications that require traditional IT departments and processes.
  - Appropriate for teams with inexperienced developers and limited knowledge of architecture patterns.
  - Applications that require strict standards of maintainability and testability.
- Shortcomings:
  - Unorganized source codes and modules with no definite roles can become a problem for the application.
  - Skipping previous layers to create tight coupling can lead to a logical mess full of complex interdependencies.
  - Basic modifications can require a complete redeployment of the application.

Event-driven Architecture Pattern

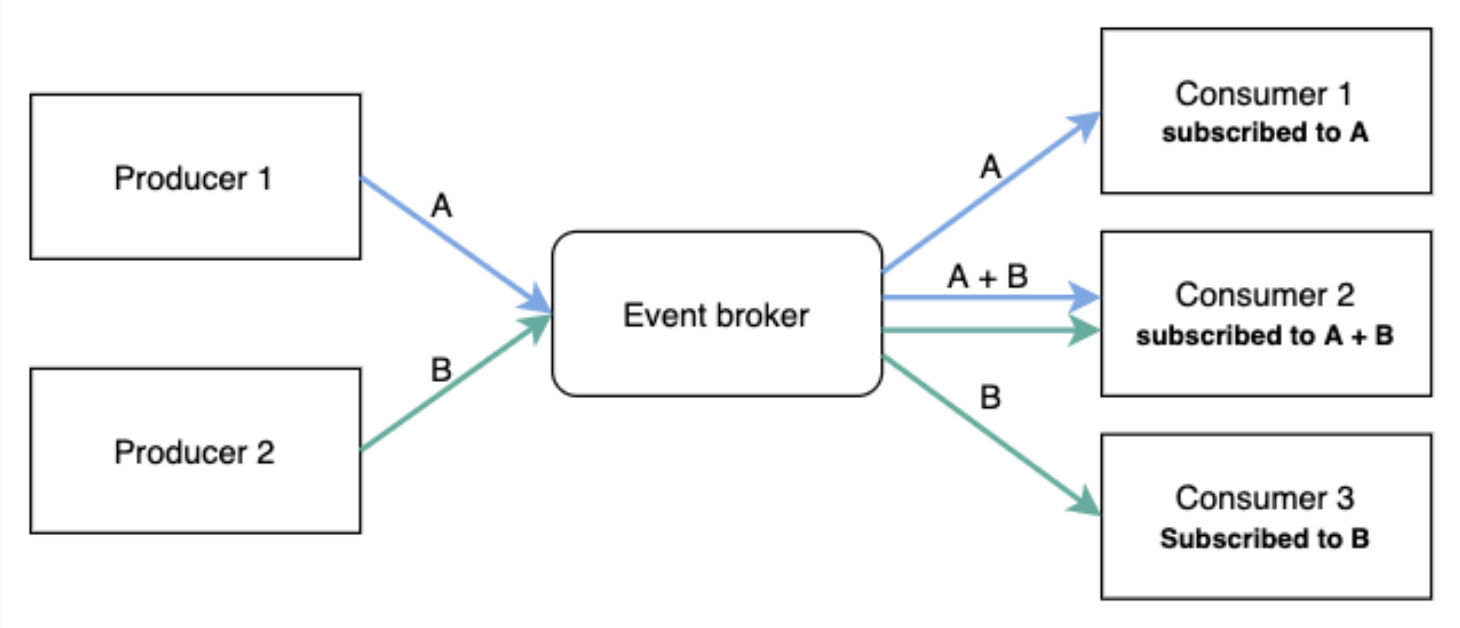


- If you are looking for an architecture pattern that is agile and highly performant, then you should opt for an event-driven architecture pattern.
- It is made up of decoupled, single-purpose event processing components that asynchronously receive and process events.
- This pattern orchestrates the behaviour around the production, detection, and consumption of all the events, along with the responses they evoke.

University of Southampton

Publisher/Subscriber Architecture

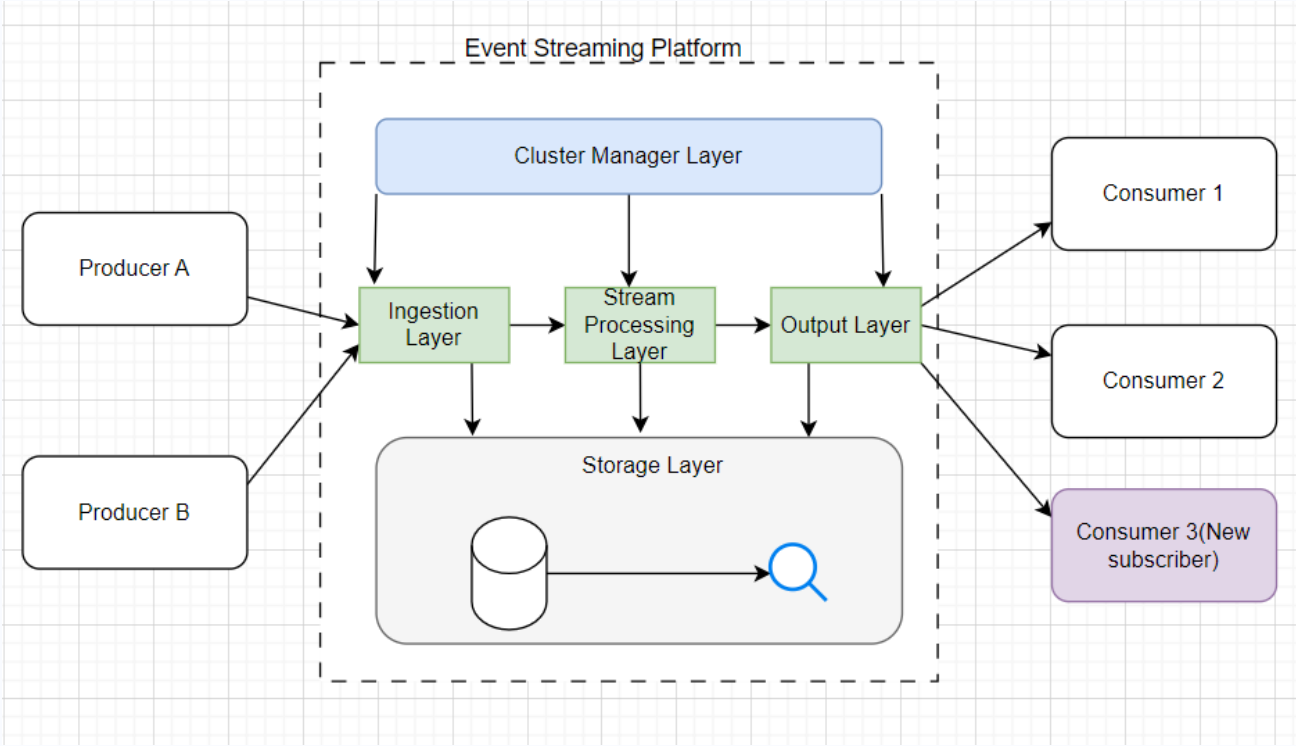
- There are two common patterns in EDA architecture:
  - Publisher/subscriber
  - and event streaming.



University of Southampton

Event Streaming Architecture

- In this approach, events are written to a log, and all consumers can read the log to identify relevant events.
- Consumers can read from any part of the stream, so they can replay past events.



Advantages of Event-Driven Architecture

- Event-Driven Architecture offers a wide range of benefits, making it a crucial tool for addressing complex modern software challenges:
  1. *Scalability*: EDA allows systems to scale horizontally by adding event consumers as needed. This ensures that your system can handle increased loads effortlessly, making it ideal for applications with fluctuating workloads.
  2. *Responsiveness*: EDA enables real-time responsiveness to changes and events. Systems can react immediately to user actions or external stimuli, providing an enhanced user experience.
  3. *Flexibility*: The decoupled nature of EDA makes systems more adaptable. You can modify or add new components without affecting the existing ones, making maintenance and updates more manageable.

Event-Driven Architecture – Advantages

- Event-Driven Architecture offers a wide range of benefits, making it a crucial tool for addressing complex modern software challenges:
  4. *Fault Tolerance*: In EDA, if an event consumer fails to process an event, the event broker can often retry or route the event to an alternative consumer. This enhances fault tolerance in distributed systems.
  5. *Data Integration*: EDA facilitates data integration by allowing various systems and applications to communicate seamlessly through a shared event infrastructure.
- Use Cases for Event-Driven Architecture
  - E-commerce
  - IoT (Internet of Things)
  - Financial Services
  - Logistics and Supply Chain
  - Social Media
  - Healthcare Monitoring

University of Southampton

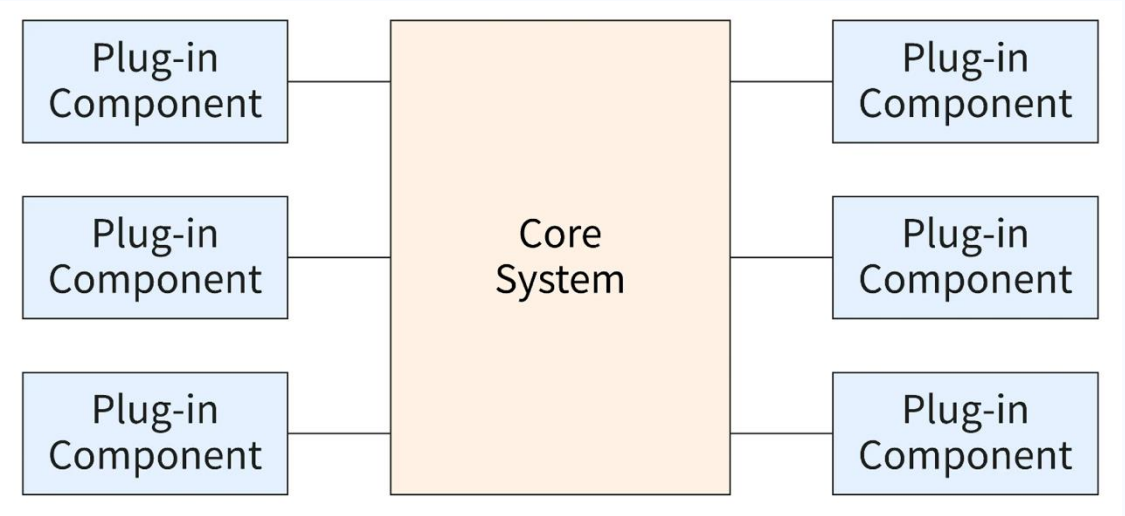
Event-Driven Architecture – Disadvantages

- Testing individual modules can only be done if they are independent, otherwise, they need to be tested in a fully functional system.
- When several modules are handling the same events, error handling becomes challenging to structure.
- Development of a system-wide data structure for events can become arduous if the events have different needs.
- Maintaining a transaction-based mechanism for consistency can become complex with decoupled and independent modules.

University of Southampton

Microkernel Architecture Pattern

- This architecture pattern consists of two types of components – a **core** system and several plug-in modules.
- While the core system works on minimal functionality to keep the system operational, the plug-in modules are independent components with specialised processing.
- A classic example of the microkernel architecture is the Eclipse IDE.



Microkernel Architecture Pattern – Advantages

- It can react to changes in plug-in modules while minimizing changes to the core system.
- Unlike layered architecture, having plug-in modules means it is easier to deploy thereby minimizing downtime.
- Testing also is easier as individual modules can be tested in isolation.
- While not generally the ideal pattern to be used in high-performance applications, it can perform well due to customising the application to only include features that are needed.

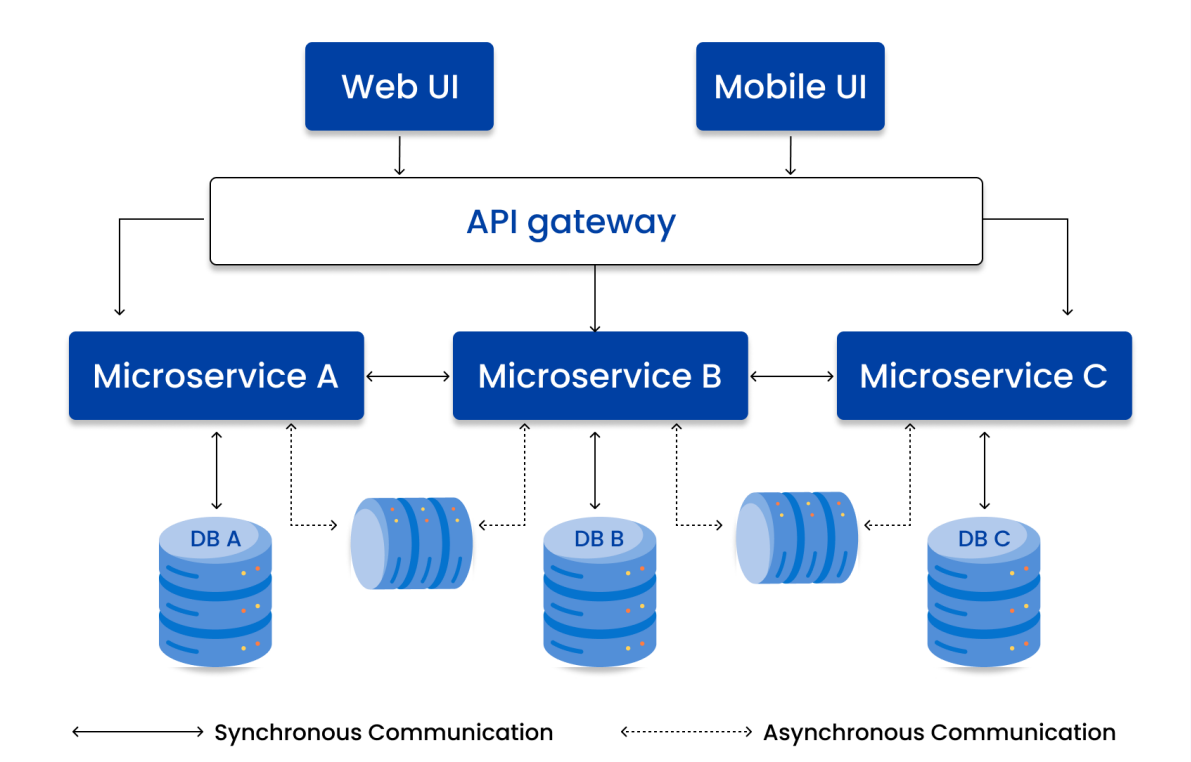
Microkernel Architecture Pattern – Disadvantages

- The plugins must have good handshaking code so that the microkernel is aware of the plugin installation and is ready to work.
- Changing a microkernel is almost impossible if multiple plugins depend on it.
- It is difficult to choose the right granularity for the kernel function in advance and more complex at a later stage.
  - Because most microkernel architecture implementations are product based and are generally smaller in size, they are implemented as single units and hence not highly scalable.

University of Southampton

Microservices Architecture Pattern

- *Microservices* architecture pattern is seen as a viable alternative to *monolithic* applications and *service-oriented* architectures.
- The *components* are deployed as separate units through an effective, streamlined delivery pipeline.
- The pattern's benefits are *enhanced scalability* and a *high degree of decoupling* within the application.



Microservices Architecture Pattern – Advantages

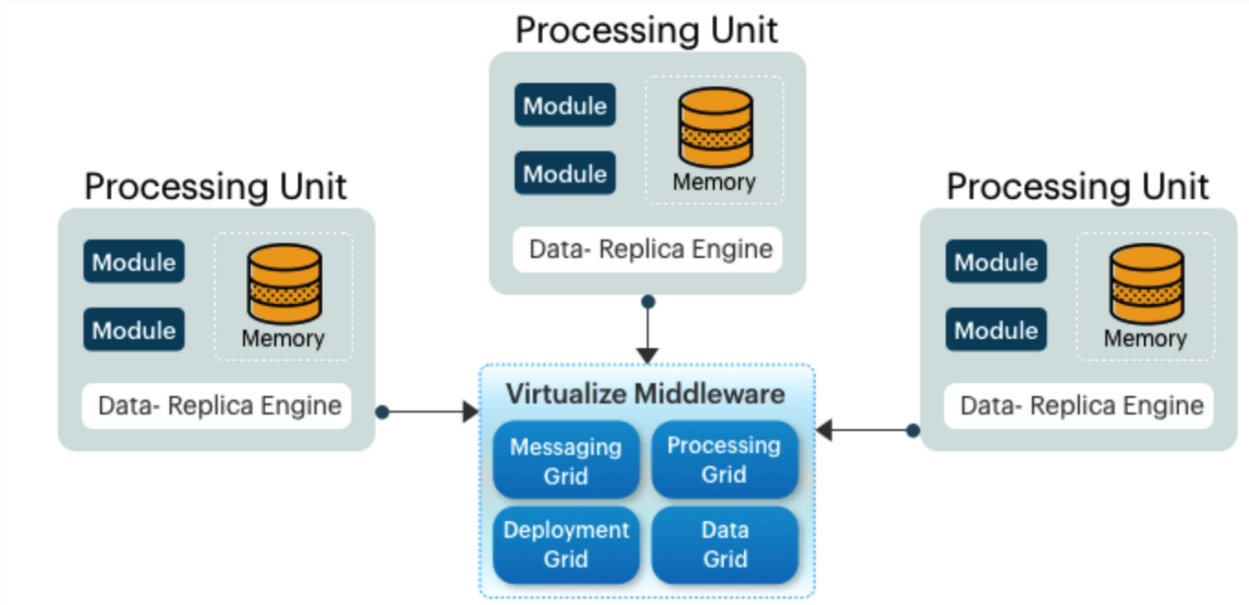
- *Accelerate scalability*: Development teams seamlessly introduce new components without causing any downtime, thanks to the independent operation of each service within the microservices architecture.
- *Improve fault isolation*: Microservices architecture is compartmentalized — if one service encounters a fault or failure, it doesn't propagate across the entire system.
- *Quicker deployment time*: Microservices architecture enables faster releases because each service evolves and deploys independently.
- *Increase Cost-efficiency*: Teams focus on specific functionality, ensuring resources are used efficiently without redundancy or excess capacity.

Microservices Architecture Pattern – Disadvantages

- *Increased complexity*: Because microservices are distributed, managing service communication can be challenging. Developers may have to write extra code to ensure smooth communication between modules.
- *Deployment and versioning challenges*: Coordinating deployments and managing version control across multiple services can be complex, leading to compatibility issues.
- *Testing complexity*: Testing microservices involves complex scenarios, mainly when conducting integration testing across various services. Orchestrating this task can be challenging.

Space-Based Architecture Pattern

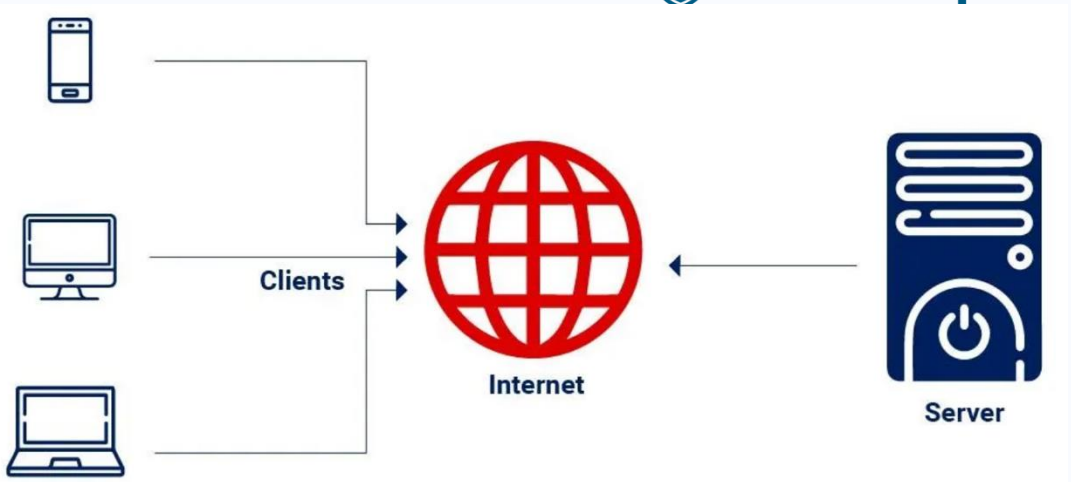
- The space-based pattern comprises two primary components – a processing unit and a virtualised middleware.
  - The *processing unit* contains portions of application components, including web-based components and backend business logic.
  - The *virtualized-middleware* component contains elements that control various aspects of data synchronization and request handling.



Space-Based Architecture Pattern

- Usage:
  - Applications and software systems that function with a large user base and a constant load of requests.
  - Applications that are supposed to address scalability and concurrency issues.
- Shortcoming:
  - It is a complex task to cache the data for speed without disturbing multiple copies.

Client-Server Architecture Pattern



- A *client-server* architecture pattern is described as a *distributed* application structure having *two main components* – a *client* and a *server*.
  - This architecture facilitates the *communication between* the *client* and the *server*, which may or may not be under the same network.
  - A *client* requests *specific resources* to be *fetched* from the *server*, which might be in the form of *data*, *content*, *services*, *files*, etc.
  - The *server* identifies the *requests* made and *responds* to the client *appropriately* by *sending over* the *requested resources*.

Client-Server Architecture – Usage

- Applications like emails, online banking services, the World Wide Web, network printing, file sharing applications, gaming apps, etc.
- Applications that focus on real-time services like telecommunication apps are built with a distributed application structure.
- Applications that require controlled access and offer multiple services for a large number of distributed clients.
- An application with centralized resources and services that has to be distributed over multiple servers.

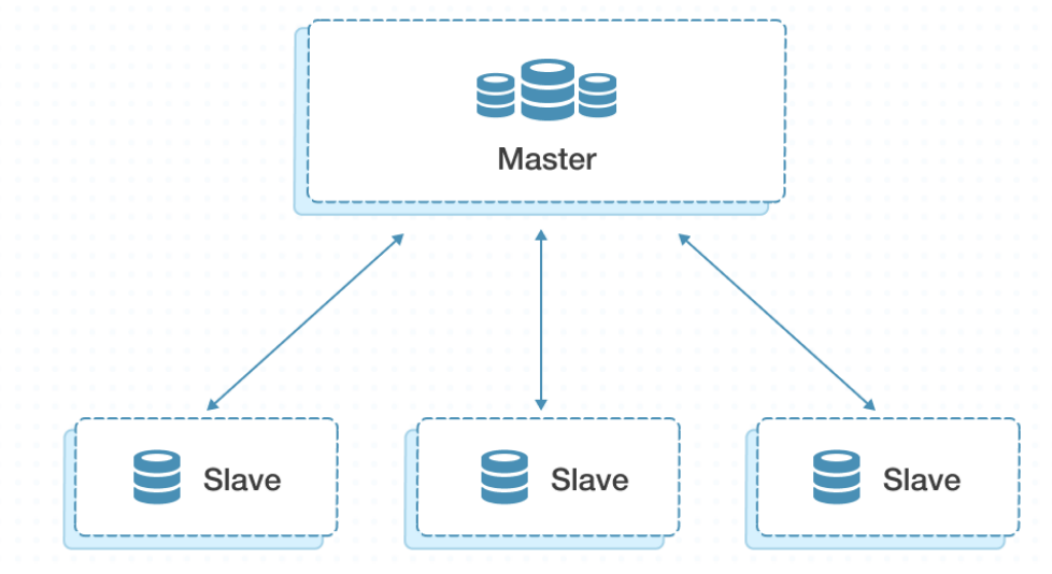
University of Southampton

Client-Server Architecture – Shortcomings

- Incompatible server capacity can slow down, causing a performance bottleneck.
- Servers are usually prone to a single point of failure.
- Changing the pattern is a complex and expensive process.
- Server maintenance can be a demanding and expensive tas

Master-Slave Architecture Pattern

- Imagine a single database receiving multiple similar requests at the same time.
  - Naturally, processing every single request at the same time can complicate and slow down the application process.
  - A solution to this problem is a master-slave architecture pattern that functions with the master database launching multiple slave components to process those requests quickly.



Master-Slave Architecture – Usage

- Development of Operating Systems that may require a multiprocessors compatible architecture.
- Advanced applications where larger services have to be decomposed into smaller components.
- Applications processing raw data stored in different servers over a distributed network.
- Web browsers that follow multithreading to increase its responsiveness.

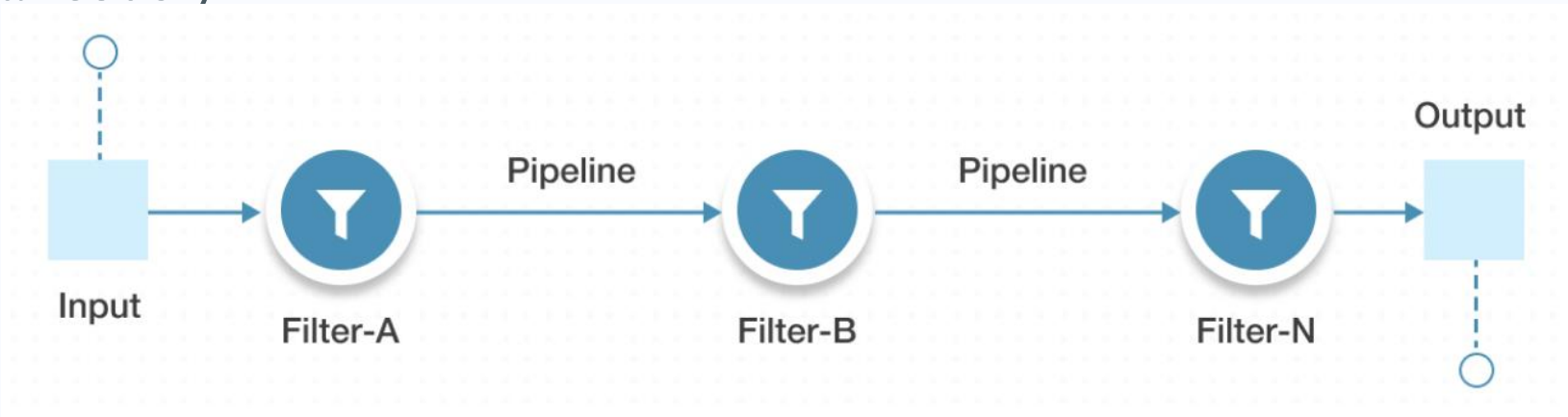
University of Southampton

Master-Slave Architecture – Shortcomings

- Failure of the master component can lead to a loss of data with no backup over the slave components.
- Dependencies within the system can lead to a failure of the slave components.
- There can be an increase in overhead costs due to the isolated nature of the slave components.

Pipe-Filter Architecture Pattern

- A pipe-filter architecture pattern processes a stream of data in a unidirectional flow where components are referred to as filters, and pipes are those which connect these filters.
  - The chain of processing data takes place where the pipes transmit data to the filters, and the result of one filter becomes the input for the next filter.
  - The function of this architecture is to break down significant components/processes into independent and multiple components that can be processed simultaneously.



Pipe-Filter Architecture – Usage

- It can be used for applications facilitating a simple, one-way data processing and transformation.
- Applications using tools like Electronic Data Interchange and External Dynamic List.
- Development of data compilers used for error-checking and syntax analysis.
- To perform advanced operations in Operating Systems like UNIX, where the output and input of programs are connected in a sequence.

Pipe-Filter Architecture – Shortcomings

- There can be a loss of data in between filters if the infrastructure design is not reliable.
- The slowest filter limits the performance and efficiency of the entire architecture.
- During transmission between filters, the data-transformation overhead costs might increase.
- The continuous transformational character of the architecture makes it less user-friendly for interactional systems.

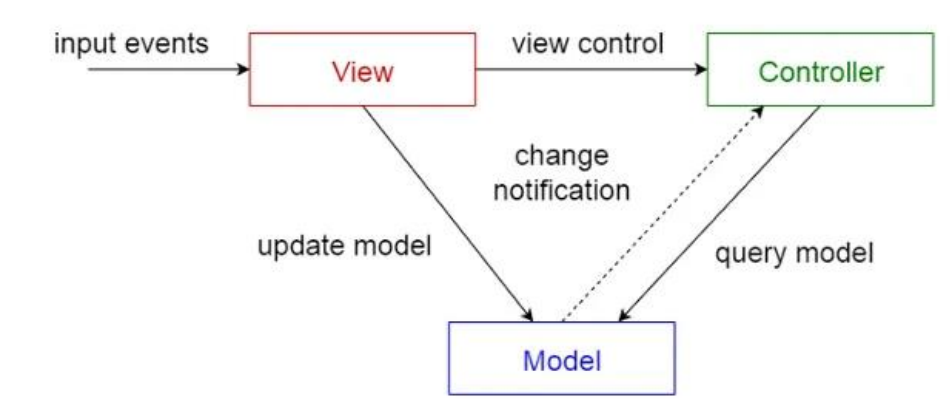
Model-view-controller pattern

- The model-view-controller (MVC) pattern divides an application into three components: A model, a view, and a controller.
  - The model, which is the central component of the pattern, contains the application data and core functionality for processing the data.
  - The view displays application data and interacts with the user. It can access data in the model.
  - The controller handles the input from the user and mediates between the model and the view. It listens to external inputs from the view or from a user and creates appropriate outputs.
  - The controller interacts with the model by calling a method on it to generate appropriate responses.

MVC – Communication between the Components

- **User Interaction with View:** The user interacts with the View, such as clicking a button or entering text into a form.
- **View Receives User Input:** The View receives the user input and forwards it to the Controller.
- **Controller Processes User Input:** The Controller receives the user input from the View. It interprets the input, performs any necessary operations (such as updating the Model), and decides how to respond.
- **Controller Updates Model:** The Controller updates the Model based on the user input or application logic.
- **Model Notifies View of Changes:** If the Model changes, it notifies the View.
- **View Requests Data from Model:** The View requests data from the Model to update its display.
- **Controller Updates View:** The Controller updates the View based on the changes in the Model or in response to user input.
- **View Renders Updated UI:** The View renders the updated UI based on the changes made by the Controller.

MVC – Communication between the Components



- **User Interaction with View:** The user interacts with the View, such as clicking a button or entering text into a form.
- **View Receives User Input:** The View receives the user input and forwards it to the Controller.
- **Controller Processes User Input:** The Controller receives the user input from the View. It interprets the input, performs any necessary operations (such as updating the Model), and decides how to respond.
- **Controller Updates Model:** The Controller updates the Model based on the user input or application logic.
- **Model Notifies View of Changes:** If the Model changes, it notifies the View.
- **View Requests Data from Model:** The View requests data from the Model to update its display.
- **Controller Updates View:** The Controller updates the View based on the changes in the Model or in response to user input.
- **View Renders Updated UI:** The View renders the updated UI based on the changes made by the Controller.

When to Use the MVC Design Pattern

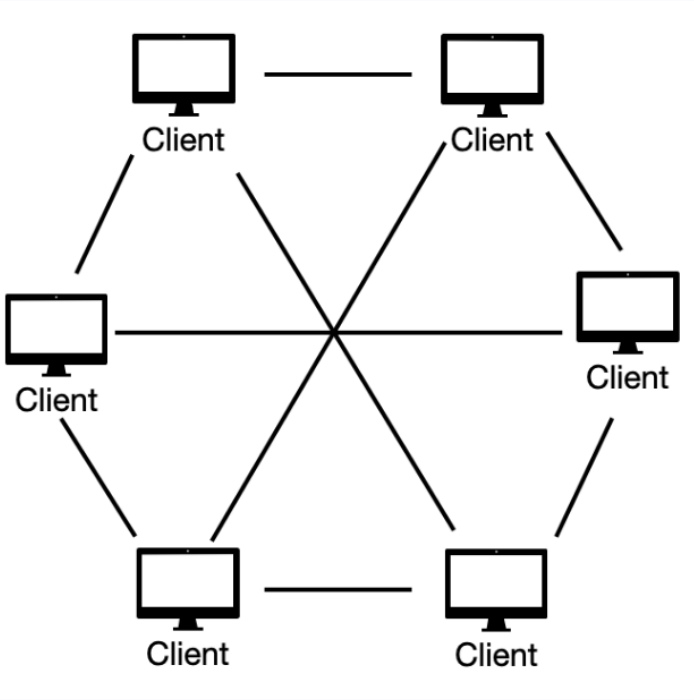
- Complex Applications: Use MVC for apps with many features and user interactions, like e-commerce sites. It helps organize code and manage complexity.
- Frequent UI Changes: If the UI needs regular updates, MVC allows changes to the View without affecting the underlying logic.
- Reusability of Components: If you want to reuse parts of your app in other projects, MVC's modular structure makes this easier.
- Testing Requirements: MVC supports thorough testing, allowing you to test each component separately for better quality control.

When Not to Use the MVC Design Pattern

- Simple Applications: For small apps with limited functionality, MVC can add unnecessary complexity. A simpler approach may be better.
- Real-Time Applications: MVC may not work well for apps that require immediate updates, like online games or chat apps.
- Tightly Coupled UI and Logic: If the UI and business logic are closely linked, MVC might complicate things further.
- Limited Resources: For small teams or those unfamiliar with MVC, simpler designs can lead to faster development and fewer issues.

Peer-to-Peer Architecture

- Peer-to-peer (P2P) architecture, in the context of networking, refers to a decentralised model where each node, or peer, in the network acts both as a client and server.
- This design allows the direct sharing of resources, be it bandwidth, storage, or processing power, creating cooperative networking and efficient resource discovery without relying on a central server.
- P2P technology has gained prominence over the years because of its benefits like flexibility, resilience, and scalability.



Advantages of P2P Architecture

- Scalability: one of the most significant benefits of P2P networks is their scalability.
- Efficiency: P2P networks have an edge over traditional client-server systems as well in terms of efficiency.
- Resilience: the decentralised nature of P2P networks allows them to remain functional even when one or more nodes fail.
- Flexibility: They allow devices to join and leave the network at will without disrupting overall operation.

Disadvantages of P2P Architecture

- Security: one of the primary concerns is security.
  - The lack of a central authority for managing security policies can make these networks vulnerable to various threats, including malware, illegal content sharing, and data breaches.
- Weak Content regulation: in P2P networks, there's no centralised control to prevent the sharing of copyrighted or illegal material.
  - This lack of regulation can lead to legal and ethical issues.
- Quality of Service (QoS) can also be a significant challenge for P2P systems.
  - Since each node operates autonomously, maintaining a constant quality or standard of service can be tricky.
  - For instance, the transfer speed can vary based on the peers' connection quality involved in a file-sharing process.

University of Southampton

YOUR QUESTIONS