HyperionDev

System Architecture

Task

Visit our website

Introduction

Designing a software system is like building a house – without a solid plan, the result can be unstable, inefficient, and costly. Software architecture provides the blueprint for organising components, managing interactions, and addressing system constraints. It forms the foundation for development, and ensures the system meets user and non-functional requirements like performance and security. By selecting the right architectural pattern, architects create systems that are scalable, maintainable, and efficient. This task explores the importance of software architecture, key architectural types, and common patterns that guide system design.

Software architecture

If you tried to build a house without planning it first, you can imagine things would not turn out very well. The same would happen if you tried to build a system with code without system architecture – it's the blueprint based on your previously established requirements that will form the basis of your designs for the system. This means that the decisions made in this phase of the design process will be the foundation for the development process later.

Software architecture aims to see how the components of a system are organised and put together, and how they communicate with each other, while being mindful of the constraints on the system as a whole. From there, the software architect can decide on an architectural pattern that would best suit the system.

The importance of software architecture

As mentioned above, it can be challenging to build something without planning how to do it. It could result in wasted time, wasted resources, and spending more money than necessary. Bass, Clements, and Kazman (2012, as cited by Sommerville, 2016, p. 169) discuss three advantages of implementing software architecture:

1. **Stakeholder communication:** Because the architecture shows the big picture of a system, anyone involved in its creation should understand it and use it as a point of reference in discussions.
2. **System analysis:** Documenting the architecture process from early on helps to keep the process on track and ensures that all established requirements of the system are met. This is because the level of analysis needed to document the process will ensure that everything is carefully considered and planned accordingly.
3. **Large-scale reuse:** If an architectural model is designed and documented correctly, it could be reused for future systems with similar requirements.

Architectural types and patterns

An architectural pattern is a description of a system of organisation developed informally over time (Garlan and Shaw, 1993, p. 5). It can be thought of as a stylised, abstract description of good practice that has been tried and tested in different systems and environments. As stated in Sommerville (2016, p. 167-195), the pattern chosen will depend on the non-functional requirements of the system, such as:

- **Performance:** keeping operations localised
- **Security:** keeping sensitive data buried
- **Safety:** keeping safety components centralised and secure
- **Availability:** including redundancies that are available for hot-swapping (implementing new components without needing to stop the system)
- **Maintainability:** creating components that are replaceable when updating or upgrading is needed

Architecture types

There are several different types of architectures that can be used when system planning and programming applications and their interfaces, each with its strengths and weaknesses. Some common types of application architecture include:

1. **Monolithic:** In a monolithic architecture, all components of an application are combined into a single, self-contained unit. This can make it easy to develop and deploy applications, but also difficult to scale and update individual components.
2. **Microservices:** In a microservices architecture, an application is broken down into smaller, independent components that can be developed and deployed separately. This can make it easier to scale and update individual components, but can also make it more complex to manage the overall application.
3. **Service-oriented:** In a service-oriented architecture, an application is composed of independent services that communicate with each other through well-defined interfaces. This can make it easy to integrate with other applications and systems, but make it more difficult to maintain consistency and integrity across the different services.

Architecture patterns

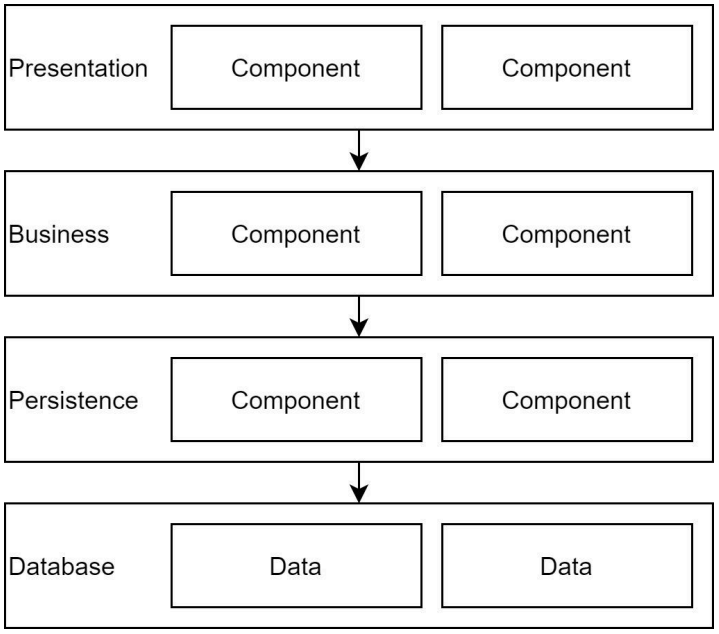
Architectural patterns solve specific problems, while types classify these patterns. Engineers and architects collaborate to select the best pattern based on system requirements, component interactions, and constraints. Proper planning and documentation streamline development and leads to a higher-quality system.

Below are summaries of some of the more common patterns used in system design.

Layered architecture

This pattern separates components into layers, where each layer only accesses the one below. This allows incremental development and makes outer layers easier to change, as only inner layers interact with hardware or the operating system (OS).

Most systems will have four layers: presentation, business, persistence, and database (Richards, 2015, p. 1-8). Each layer has a designated purpose: The **presentation** layer is the user interface component; the **business** layer is responsible for aspects like user authentication or authorisation; the **persistence** layer deals with functional components; and the **database** layer, as the name suggests, is the back-end component that contains the database. Below is an illustration based on Richards' description (2015, p. 1-8):



Layered architecture (Richards, 2015, pp. 1–8)

Note that each layer contains the components that apply to that particular element of the system. Further, note how each layer has a level of separation and independence – this makes the components of systems with a layered architecture easy to develop and test independently from the rest of the system.

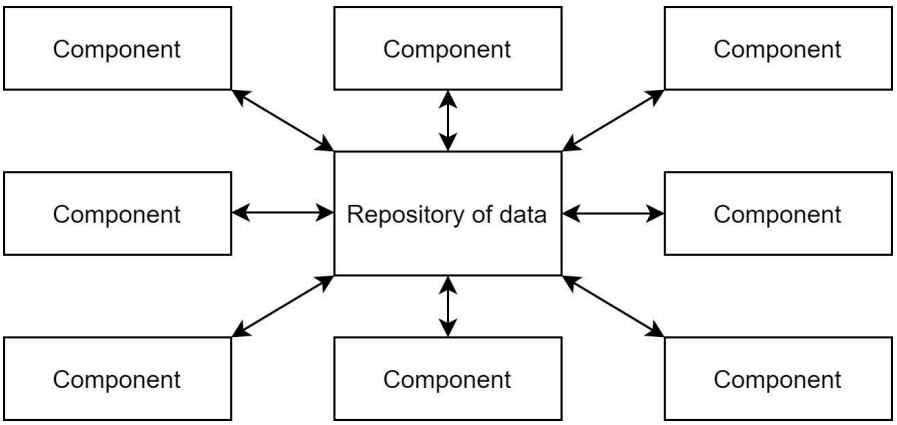
Because of how these layers are separate, this architecture can be an example of a service-oriented one. The presentation, business, persistence, and database all work independently but communicate with each other through their defined interfaces.

Take note

The diagram is a **block diagram**: each block represents a component, with nested blocks showing subcomponents. Arrows indicate data flow. It provides a broad system overview, useful for stakeholders, but developers need more detailed diagrams for implementation.

Repository architecture

In this architecture pattern, all data is kept in a central repository and all components are separated – they can, however, interact through the repository. This is a particularly useful pattern if you have a lot of data. Have a look at the block diagram of the pattern below:

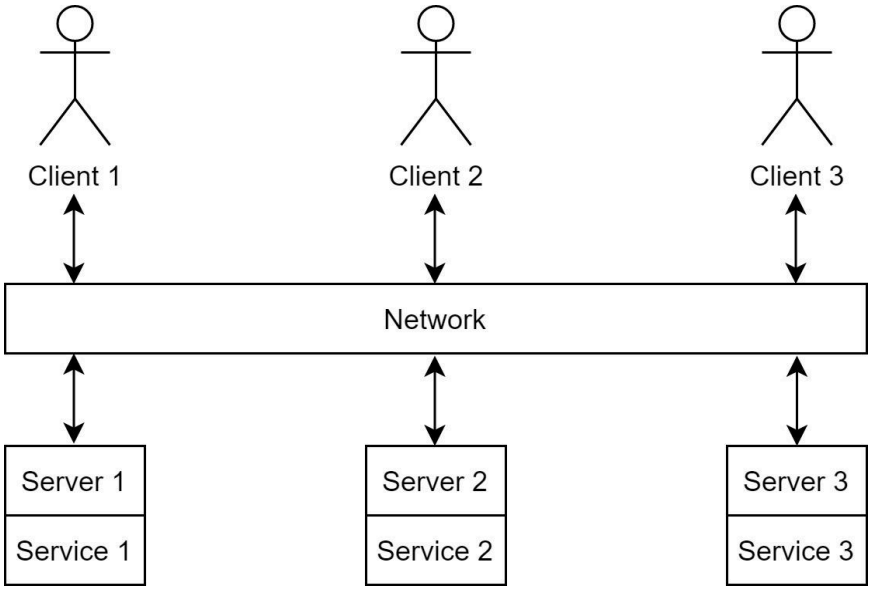


Repository architecture (Richards, 2015, pp. 1–8)

As you can see, there is a **bidirectional** flow of data to and from the repository, which means it is easy to input and output data as well as for components to interact with one another via the repository. However, one drawback is that the repository is a single point of failure. This means that a problem occurring in the repository could impact the whole system. This is the crux of a microservice-based architecture; lots of small parts work together, but if one component fails for any reason, it can ripple out to the rest of the system.

Client-server architecture

In this pattern, a set of services is developed to run on servers. The clients then access the servers over a network to make use of the system. The network usage allows multiple clients to access the servers at one time to access the services the system provides. Look at the block diagram of the pattern below:

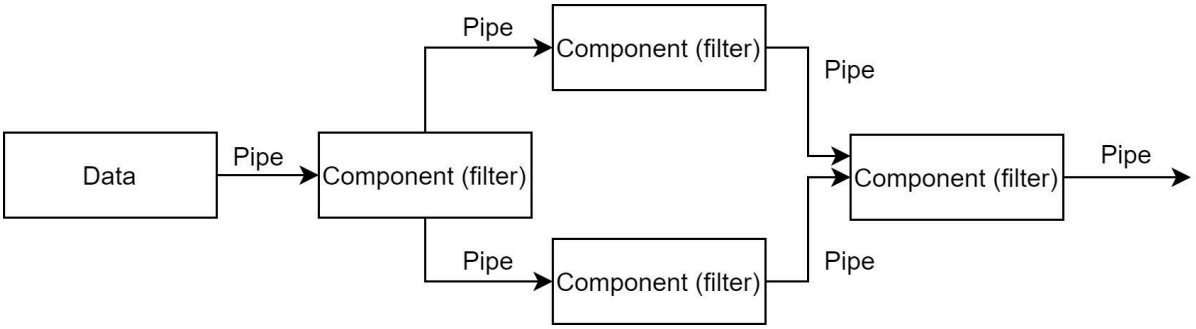


Client–server architecture (Richards, 2015, pp. 1–8)

You can see from the diagram above how vital a reliable network connection is. Without it, clients are unable to access the servers and, therefore, the system's services. This gives the client–server pattern a single point of failure, like the repository pattern.

Pipe and filter architecture

In this pattern, the data provided goes through a series of transformations (filters) as it flows from one component to another. This pattern is particularly useful for data processing. Have a look at the block diagram of the pattern below:



Pipe and filter architecture (Richards, 2015, pp. 1–8)

Looking at the diagram, you can see how transformations can occur in series as well as in parallel. This type of structure works well with the way that data-driven businesses already work, so the system would be fairly simple to integrate. However, because this system requires that input is parsed (broken up into smaller components) as input and unparsed for output through the transformations, this could lead to increased system computation time. This means that the system might be difficult to upgrade or update.

Yes, it's tempting to jump straight into coding, but that's like trying to build a house without a blueprint. You might end up with walls in the wrong place, a roof that doesn't fit, and a structure that falls apart under pressure. Poor planning leads to wasted time, resources, and frustration. Software architecture is your blueprint – it ensures everything fits, works together, and stands strong. Skip it, and you risk building a system as unstable as a poorly planned house.

Practical task

Answer the following questions:

- Do some research to find real-world examples of **each** of the architecture patterns covered above.
    - Note **when** would be most appropriate to use them.
    - Next, state your reasons for **why** each pattern would be most appropriate for each of the examples you found.

- Give two examples of systems combining two or more patterns, and state the strengths and limitations of these combined systems.

Save and submit your answers in a file titled **architecture.pdf**.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

Reference list

Garlan, D., & Shaw, M. (1993). An introduction to software architecture. In V. Ambriola & G. Tortora (Eds.), *Advances in software engineering and knowledge engineering* (pp. 1–39). World Scientific Publishing Co.

Richards, M. (2015). *Software architecture patterns* (1st ed. pp. 1–8). O'Reilly Media, Inc.

Sommerville, I. (2016). *Software engineering* (10th ed. pp. 167–195). Pearson Education Limited.