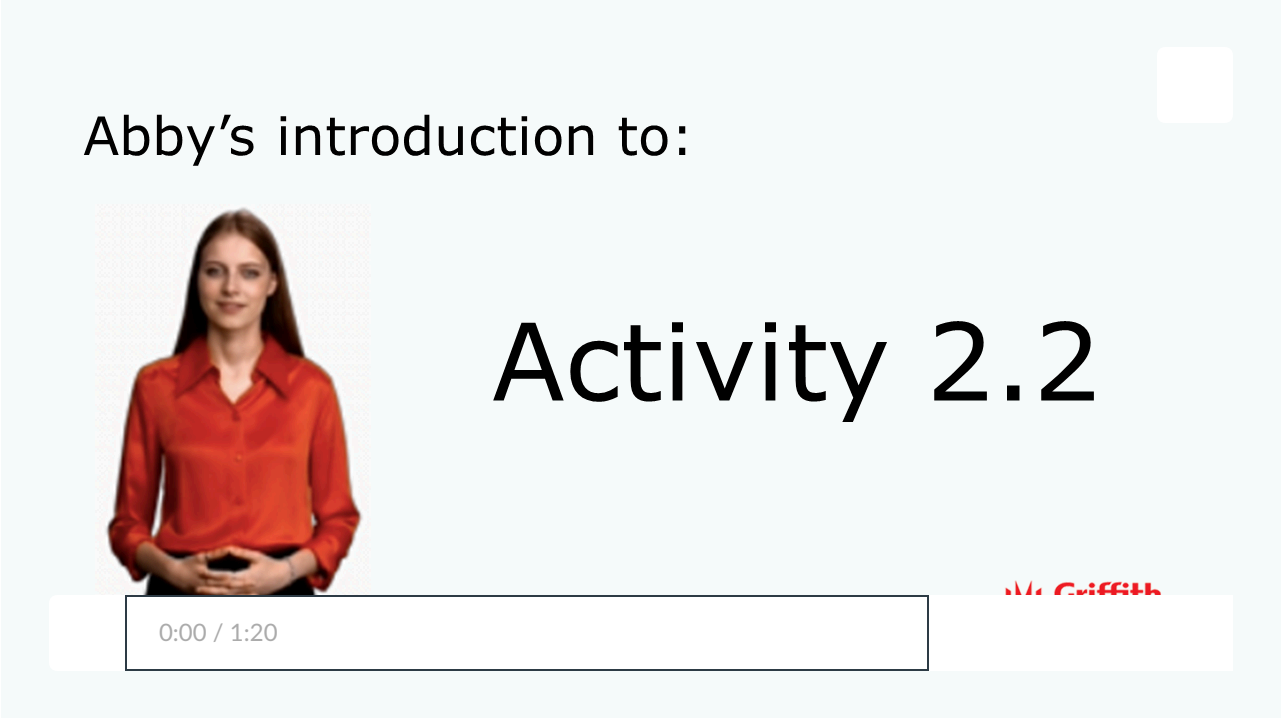
Activity 2.2 Design a system architecture

Access course FAQ chatbot (https://lms.griffith.edu.au/courses/24045/pages/welcome-to-the-course-chatbot)

Module 2 - Design system architecture and select application type

---

(https://lms.griffith.edu.au/courses/24045/pages/review-progress-ready-to-move-on-to-work-task-3?module_item_id=634641)



What is this activity?

In Activity 2.2, you will design a system architecture for the SmartShelf inventory management system project. Building upon the knowledge and skills you acquired in Activity 2.1, you will now apply various architectural patterns and design principles to create a high-level system architecture that meets the functional and non-functional requirements of the project. This activity will involve analysing the project requirements, evaluating different architectural approaches, and creating a design document that showcases the system architecture and justifies the design decisions.

The final output of Module 2 is a detailed report section that designs the system architecture and selects the appropriate application type for your chosen assignment scenario (https://lms.griffith.edu.au/courses/24045/assignments/93487) . It should justify your choices and provide a clear rationale for the selected technology stack.

Why is this activity important?

By engaging in this activity, you will gain hands-on experience in applying architectural patterns and design principles to a real-world project scenario. You will develop your ability to analyse complex project requirements and translate them into a coherent system architecture. This activity will also teach you how to evaluate and justify the selection of an appropriate application type based on the project's specific needs and constraints. Furthermore, you will enhance your skills in creating clear and concise design documentation that communicates your architectural vision to stakeholders and development teams.

Case study

▶ SmartShelf - Inventory Management System

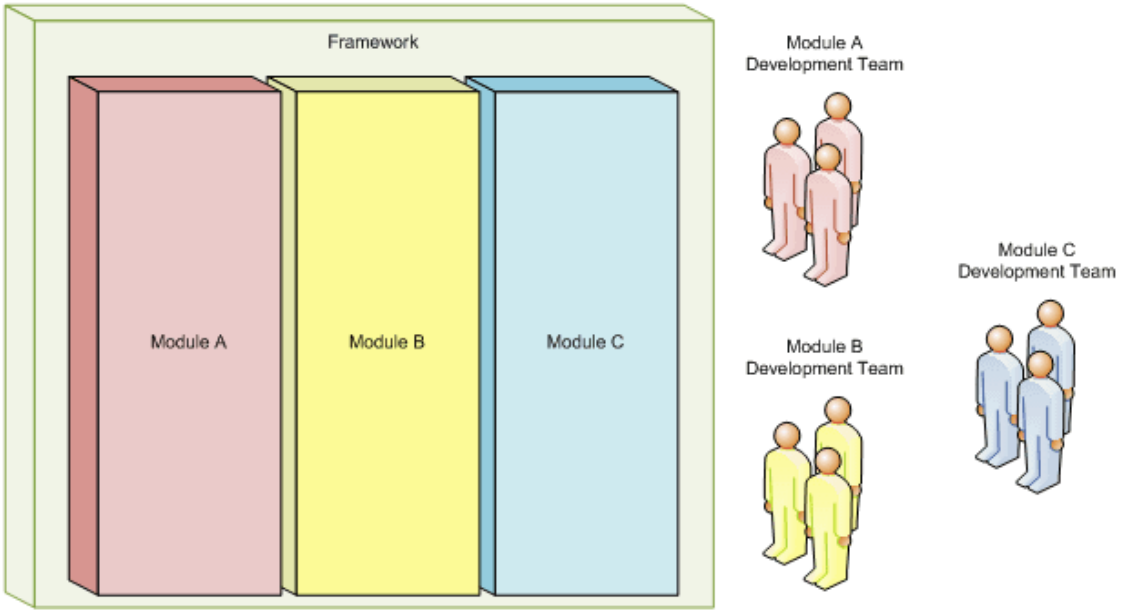
Supporting content for this activity

You should then work through the content elements below. These will reinforce the principles and elements from the case study exercise and will provide you with the knowledge and tools that you need to complete this activity.

▼ Supporting content A - Architectural patterns and design principles

Separation of Concerns

Separation of Concerns (SoC) is a design principle that encourages the division of a computer program into distinct sections, such that each section addresses a separate concern and overlaps in functionality are reduced to a minimum. The principle is closely related to the **Single Responsibility Principle** of object-oriented design, which states that a class should have only one reason to change, or that it should have only one job. By separating concerns, developers can build more **maintainable and scalable** systems, as changes in one part of the system are less likely to require changes in others. This can also lead to better testability, as individual components can be tested in isolation.

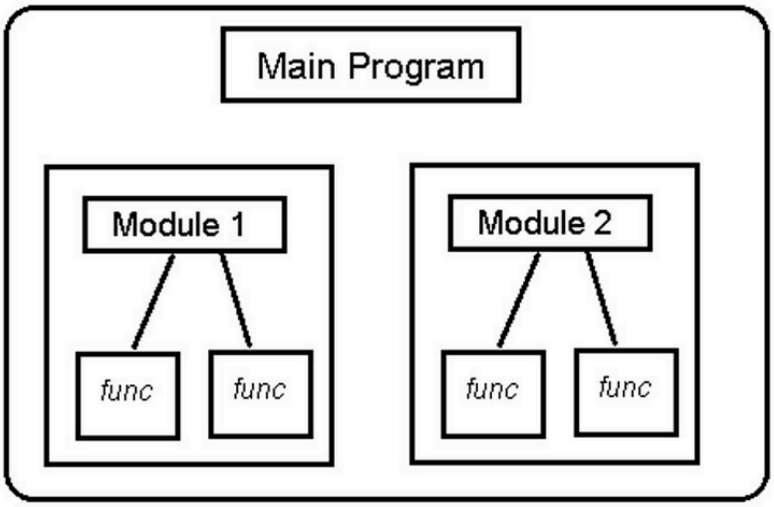


Separation of concerns (Image source (https://dev.to/suspir0n/soc-separation-of-concerns-5ak7))

In the context of system architecture, applying the Separation of Concerns principle can lead to the creation of **modular components** that are responsible for specific parts of the system's functionality. For example, in a web application, concerns such as data access, business logic, and presentation can be separated into different layers or services. This separation allows teams to work on different aspects of the application concurrently without causing conflicts. It also facilitates the reuse of components across different parts of the application or even across different projects, promoting a more efficient development process.

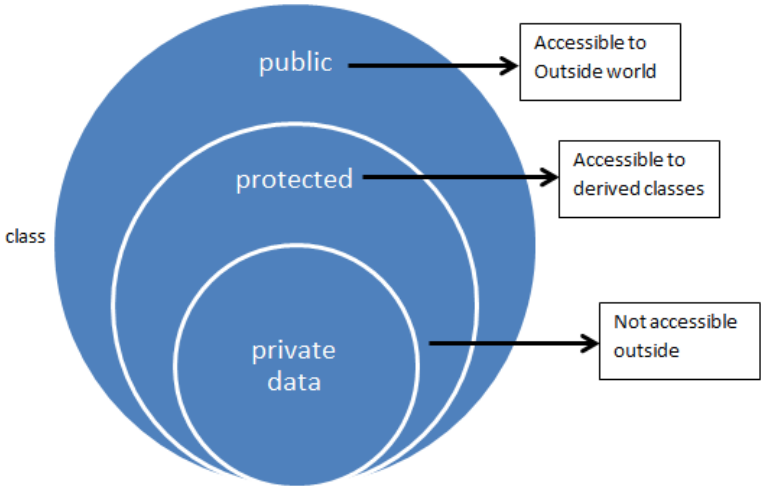
Moreover, Separation of Concerns supports the concept of loose coupling and high cohesion, which are fundamental to good software design. **Loose coupling** means that components of a system are interconnected in a way that changing one component has a minimal impact on other components. **High cohesion** refers to the grouping of related functionality within a component, making it more focused and easier to understand. By adhering to SoC, architects and developers can create systems that are not only easier to maintain and extend but also more resilient to changes, as the impact of modifications is contained within the scope of the specific concern being addressed.

Modularity and Encapsulation



Modularity (Image source (https://resources.altium.com/p/product-design-trends-in-2020-modular-hardware-vs-modular-software))

Modularity and encapsulation are two fundamental concepts in software engineering that contribute to the creation of robust, maintainable, and scalable systems. **Modularity** refers to the division of a system into smaller, independent modules, each with a specific functionality. These modules can be developed, tested, and deployed independently, which simplifies the overall complexity of the system. By breaking down a system into modules, developers can work on different parts simultaneously, reducing the time to market and facilitating concurrent development.



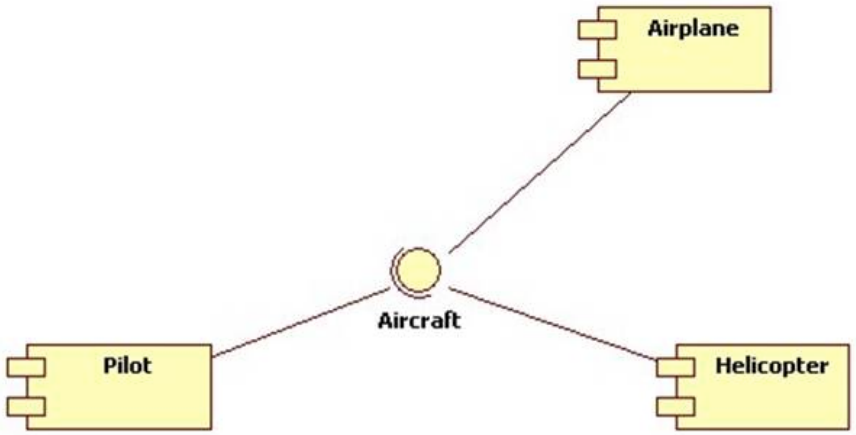
Encapsulation (Image source (https://www.quora.com/Computer-Programming-How-does-encapsulation-provide-control-over-the-program-data))

Encapsulation, on the other hand, is the principle of hiding the internal state and functionality of a module behind a well-defined interface. This means that the details of how a module operates are not exposed to other parts of the system; instead, other modules can only interact with it through a set of public functions or methods. Encapsulation promotes information hiding, which is the idea that each module should only expose what is necessary for other modules to interact with it, and keep the rest hidden. This reduces the interdependencies between modules and makes it easier to manage changes within a module without affecting others.

The combination of modularity and encapsulation leads to a system architecture that is **highly cohesive** within modules and **loosely coupled** between them. Cohesion refers to the degree to which the elements within a module belong together, and loose coupling means that modules have minimal dependencies on one another. This architecture allows for greater flexibility and adaptability, as changes to one module do not ripple through the entire system. It also supports code reuse, as well-encapsulated modules can be easily plugged into different parts of the system or even used in other systems altogether.

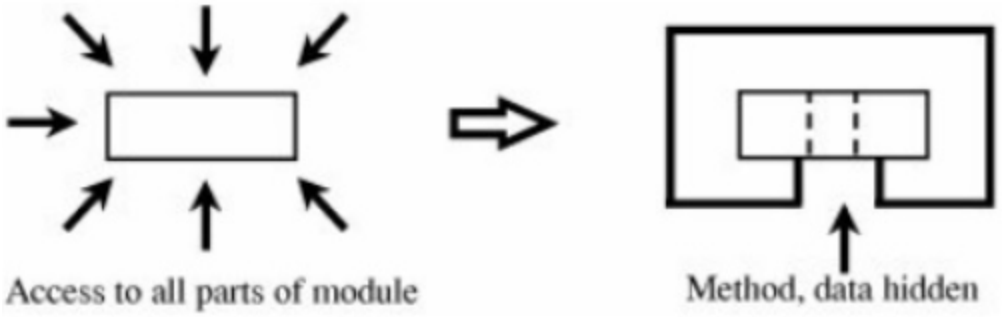
Moreover, modularity and encapsulation support the principle of **separation of concerns**, where different modules handle different concerns of the system. This separation allows for more focused development and debugging, as developers can concentrate on a specific module without being overwhelmed by the complexity of the entire system. It also aids in the maintenance of the system, as it is easier to locate and fix issues within a specific module. Overall, the application of modularity and encapsulation in system design leads to more organised, efficient, and reliable software systems.

Abstraction and Information Hiding



Abstraction (**Image source** **(https://www.cs.sjsu.edu/~pearce/modules/lectures/ood/principles/Abstraction.htm)**)

Abstraction and information hiding are two interrelated concepts in software engineering that are essential for managing complexity and promoting reusability in system design. **Abstraction** involves creating a simplified representation of a system or a component by exposing only the necessary details and omitting unnecessary complexity. This allows developers to work at a higher level of thinking, focusing on what the system does rather than how it does it. By abstracting away the internal details, developers can think in terms of the problem domain rather than the implementation details, which makes the system easier to understand and reason about.



Information hiding (Image source (https://www.brainkart.com/article/Modularity,-Encapsulation,-and-Information-Hiding_9592/))

Information hiding, often referred to as encapsulation in object-oriented programming, is the practice of hiding the internal state and workings of a module or component from the outside world. This is achieved by providing a public interface that other parts of the system can interact with, while keeping the implementation details private. The goal of information hiding is to protect the integrity of the module by preventing direct access to its internal data and functionality. This not only reduces the risk of misuse but also allows the implementation to be changed without affecting the code that uses it, as long as the public interface remains consistent.

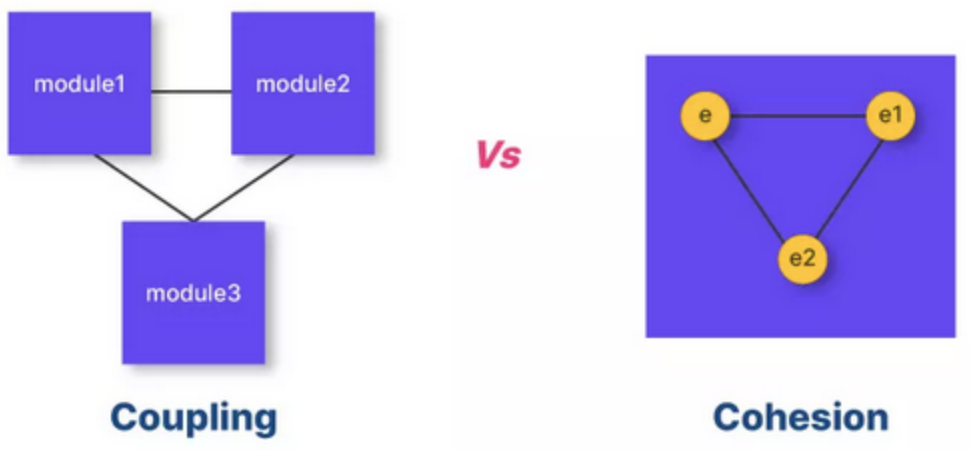
The combination of abstraction and information hiding is powerful in system design because it allows for the creation of **modular and maintainable** systems. By abstracting the functionality of a system into well-defined interfaces, developers can build components that are self-contained and have clear responsibilities. This modularity makes it easier to manage the complexity of the system, as each module can be developed, tested, and maintained independently. Furthermore, information hiding ensures that the dependencies between modules are minimised, which reduces the risk of changes in one module causing unintended consequences in others.

In practice, abstraction and information hiding are often implemented through the use of abstract data types, interfaces, and classes in object-oriented programming languages. These constructs allow developers to define the public-facing behavior of a component while keeping the implementation details private. This **separation of concerns** enables the development of systems that are not only easier to understand and maintain but also more resilient to change. As the requirements of a system evolve, the internal workings of a component can be modified without affecting the rest of the system, as long as the abstraction layer remains consistent. This flexibility is crucial for the long-term success of any complex software system.

Coupling and Cohesion

Coupling and cohesion are two critical concepts in software engineering that are used to measure the quality of interaction within and between components of a system. **Coupling** refers to the degree of interdependence between software modules, while **cohesion** measures how closely related the

responsibilities of a single module are. Both concepts are fundamental in designing systems that are maintainable, scalable, and easily adaptable to change.



Coupling vs cohesion (Image source (https://unstop.com/blog/difference-between-cohesion-and-coupling-in-software-engineering))

Low coupling is desirable in system design because it indicates that modules are relatively independent of each other. When modules are loosely coupled, changes in one module are less likely to require changes in another, which makes the system more maintainable and reduces the risk of introducing new bugs when modifications are made. **Loose coupling** is achieved by designing modules that interact with each other through simple and stable interfaces, rather than sharing complex details of their implementations. This can be facilitated by the use of design patterns such as dependency injection, where a module receives its dependencies from an external source rather than creating them internally.

High cohesion, on the other hand, means that the elements within a module are strongly related and focused on a single task or responsibility. A cohesive module is easier to understand, test, and reuse because its functionality is well-defined and encapsulated. **High cohesion** is achieved by carefully organising the code so that functions and data that are closely related are grouped together within the same module. This often involves refactoring code to separate different concerns into distinct modules, each with a clear and singular purpose.

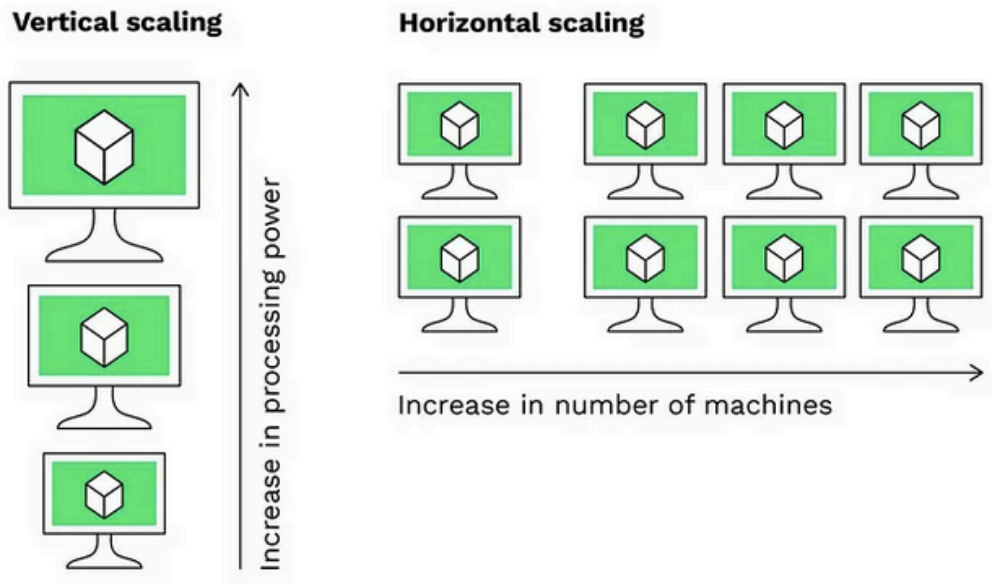
The relationship between coupling and cohesion is inversely proportional; **as coupling decreases, cohesion tends to increase**. A system with low coupling and high cohesion is considered well-designed because it consists of modules that are independent and focused, which makes the system as a whole more flexible and easier to manage. Such a system can accommodate changes more readily, as modifications are localised to specific modules that have minimal impact on the rest of the system.

In practice, achieving the **right balance** between coupling and cohesion requires thoughtful design and continuous refactoring. It involves making conscious decisions about how to group functionality into modules and how those modules should interact with one another. By striving for low coupling

and high cohesion, software developers can create systems that are not only robust and efficient but also adaptable to the ever-changing demands of the software development lifecycle.

Scalability and Performance

Scalability and performance are critical aspects of system design that are often interrelated but distinct in their focus. **Scalability** refers to the ability of a system to handle increased load or throughput by adding resources, such as more servers or processing power. A scalable system can grow or shrink in response to demand, ensuring that it remains responsive and reliable even under heavy usage. This is particularly important for applications that experience fluctuating user loads, such as e-commerce sites during holiday sales or social media platforms during major events.



Vertical vs horizontal scalability (Image source (https://medium.com/javarevisited/difference-between-horizontal-scalability-vs-vertical-scalability-67455efc91c))

There are two main types of scalability: vertical and horizontal. **Vertical scalability**, or scaling up, involves adding more power to an existing server, such as upgrading the CPU or adding more RAM. **Horizontal scalability**, or scaling out, involves adding more servers to the system to distribute the load. Cloud computing platforms often provide services that facilitate horizontal scalability, allowing systems to automatically add or remove servers based on demand.

Performance, on the other hand, is concerned with the efficiency of a system's execution and its ability to process requests quickly. A system with **good performance** will have minimal latency and high throughput, meaning it can handle a large number of requests in a short amount of time. Performance is influenced by many factors, including the system's architecture, the efficiency of the algorithms used, the database design, the network infrastructure, and the hardware capabilities.

To achieve both scalability and performance, system designers must carefully plan and optimise various components of the system. This includes implementing **caching strategies** to reduce database load, using **asynchronous processing** to handle long-running tasks, and designing the

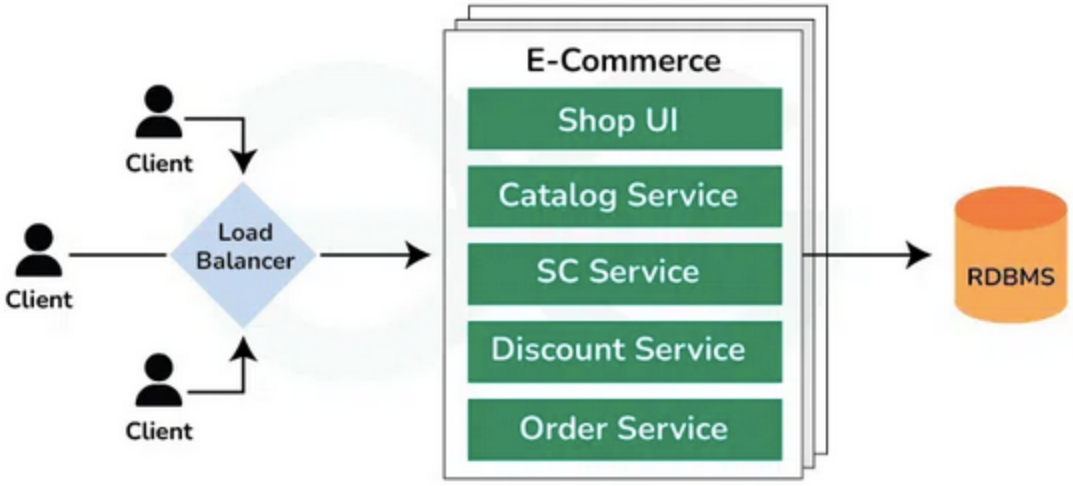
system to be **stateless**, which allows for easier horizontal scaling. Additionally, continuous monitoring and profiling of the system's performance can help identify bottlenecks and guide further optimisation efforts.

In summary, scalability and performance are key considerations in system design that require a thoughtful approach to ensure that a system can handle growth and provide a seamless user experience. Balancing these aspects involves making strategic decisions about the system's architecture, resource allocation, and ongoing optimisation efforts.

▼ Supporting content B - Application types and their suitability

Monolithic Applications

Monolithic applications are traditional software systems where all components are interconnected and work together as a single, unified service. These applications are typically built, deployed, and scaled as a single unit. Monolithic architectures have been the standard for many years, especially for applications that started their life cycle before the widespread adoption of microservices and cloud-native architectures. They are characterised by a single codebase, a single database, and a tightly-coupled set of functionalities that are all part of the same deployment package.



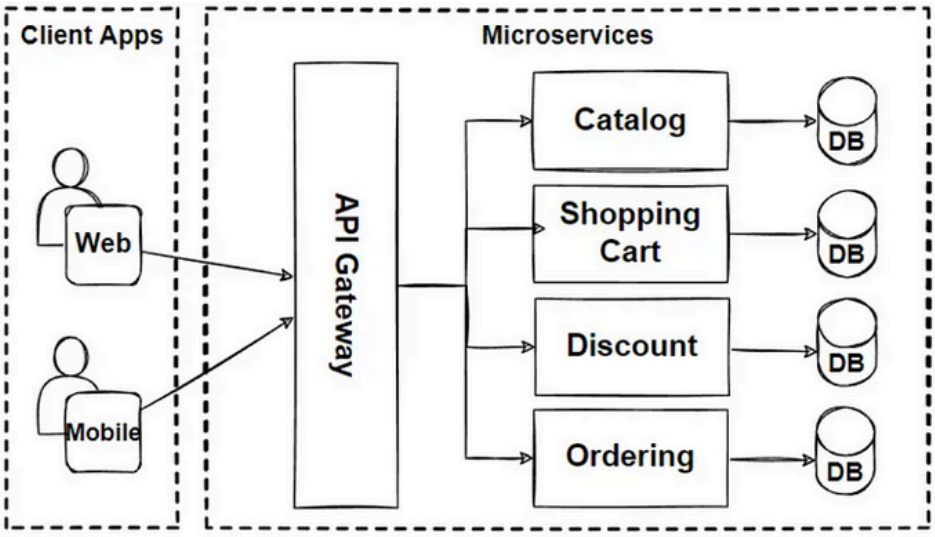
Monolithic applications (Image source (https://www.geeksforgeeks.org/monolithic-architecture-system-design/))

One of the main advantages of monolithic applications is their **simplicity**. Since all components are integrated into a single system, it can be easier to understand, develop, and test. This can lead to faster development cycles, especially for smaller teams or projects with straightforward requirements. Additionally, monolithic applications can be more **performant** in certain scenarios because they avoid the network overhead and latency that can be introduced by inter-service communication in distributed systems.

However, monolithic applications also have significant **drawbacks**, particularly as they grow in size and complexity. As new features are added, the codebase can become unwieldy, making it difficult to maintain and extend. This can lead to what is often referred to as a **"monolithic ball of mud"**, where the architecture degrades over time due to the accumulation of quick fixes and patches. Furthermore, **scaling** a monolithic application can be challenging because the entire application must be scaled as a single unit, which can be inefficient and costly. This can limit the application's ability to handle increased load or to be resilient in the face of failures.

Microservices

Microservices architecture represents a shift from the traditional monolithic approach, where applications are decomposed into small, **independent services** that perform specific business functions. Each microservice is developed, deployed, and scaled independently of the others, often using different programming languages, databases, and hardware. This architectural style promotes a more agile and flexible development process, as teams can work on different microservices concurrently without stepping on each other's toes.



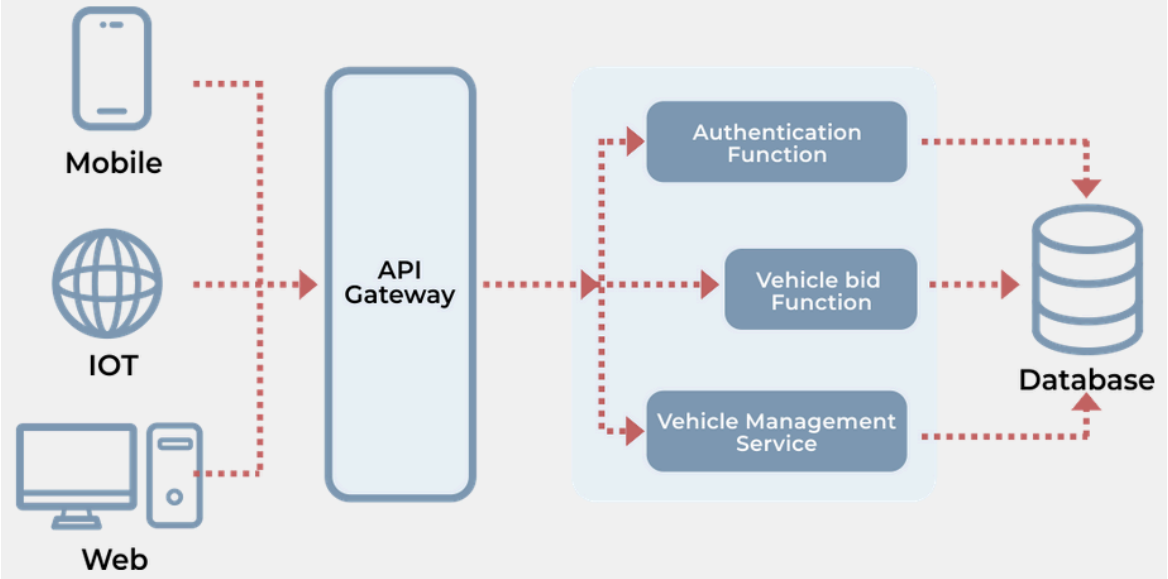
Microservices architecture (Image source (https://medium.com/design-microservices-architecture-with-patterns/microservices-architecture-for-enterprise-large-scaled-application-825436c9a78a))

One of the key benefits of microservices is the **ability to scale** each service independently based on demand. This granular control over scaling allows organisations to optimise resource usage and reduce costs by allocating resources only where they are needed most. Furthermore, the independence of microservices facilitates **continuous integration and delivery (CI/CD)** practices, enabling faster time-to-market for new features and improvements. Since each microservice is smaller and more focused, it is also easier to understand, maintain, and replace than the components of a monolithic application.

Despite these advantages, microservices also introduce new challenges. The distributed nature of microservices can lead to **complexities** in terms of service discovery, load balancing, and inter-service communication. Ensuring high availability and fault tolerance requires careful design and the implementation of **resilience patterns** such as circuit breakers and retries. Additionally, the operational overhead of managing a large number of services can be significant, necessitating the use of sophisticated orchestration tools like Kubernetes. Moreover, the **distributed data management** and the need for consistent transactions across services can be complex to handle. Overall, while microservices offer many benefits, they are best suited for organisations with the expertise and processes to handle the increased complexity that comes with this architectural style.

Serverless Architectures

Serverless architectures represent a further evolution in the way applications are designed and deployed, abstracting away the need for developers to manage the underlying servers and infrastructure. In a **serverless model**, applications are broken down into event-driven functions that are executed in response to specific triggers, such as HTTP requests, database events, or messages from other services. These functions are managed by a cloud provider, which automatically scales the execution capacity up or down based on demand.



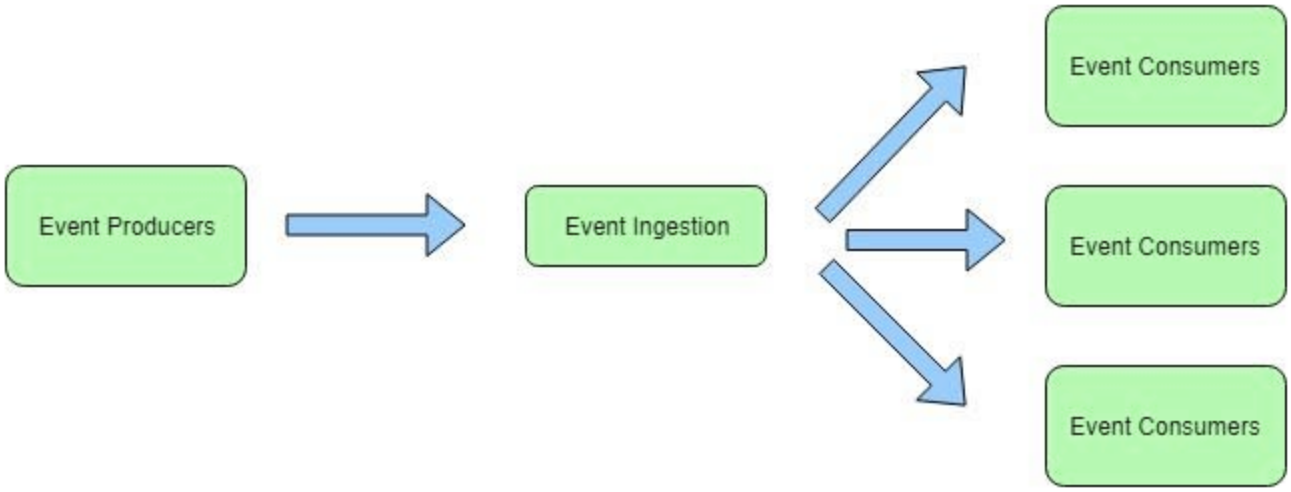
Serverless architectures (Image source (https://www.systango.com/blog/serverless-architecture-smart-choice))

One of the primary **advantages** of serverless architectures is the ability to pay only for the actual compute time used by the application, rather than paying for idle server capacity. This can lead to significant cost savings, especially for applications with variable or unpredictable workloads. Additionally, the cloud provider is responsible for maintaining the server infrastructure, which frees developers from tasks such as server provisioning, patching, and scaling, allowing them to focus on writing code that delivers business value.

Serverless architectures also promote a **microservices** approach, as each function can be developed, deployed, and scaled independently. This can lead to more **agile development** practices and **faster iteration** cycles. However, serverless computing also introduces its own set of challenges. The stateless and ephemeral nature of serverless functions can make state management and coordination between functions **more complex**. Additionally, the **cold start latency**, where a function may take longer to start up if it hasn't been invoked recently, can be a concern for applications requiring immediate response times. Furthermore, the serverless paradigm requires a rethinking of application design, as developers must adapt to the constraints and best practices associated with event-driven, function-as-a-service (FaaS) platforms.

Event-Driven Architectures

Event-driven architectures are designed around the production, detection, consumption, and reaction to events. An **event** is a significant change in state or a noteworthy incident that triggers an activity. In this architectural style, components or services are loosely coupled and communicate asynchronously by emitting events when something happens and listening for events from other components to trigger their actions.



Event-driven architectures (Image source (https://dev.to/shehanravindu/event-driven-architecture-4p7f))

One of the key benefits of event-driven architectures is their ability to **support complex, real-time, and reactive systems**. By reacting to events as they occur, systems can respond quickly to changes and new information, making them suitable for scenarios that require high levels of responsiveness and scalability. Event-driven systems are also **inherently decoupled,** which means that components can be developed, deployed, and scaled independently, leading to more maintainable and flexible applications.

However, event-driven architectures can introduce **complexity** in terms of **event management** and **consistency**. Ensuring that events are processed reliably, in the correct order, and without data loss can be challenging. Additionally, managing the flow of events and handling backpressure when the

event production rate exceeds the consumption rate requires careful design. Furthermore, **debugging** and **monitoring** event-driven systems can be more complex than traditional request-response systems, as the asynchronous nature of event processing can make it harder to trace the flow of data and identify bottlenecks or failures.

Despite these challenges, event-driven architectures are well-suited for modern distributed systems that need to handle a large number of concurrent operations and respond to changes in real-time. They are often used in scenarios such as **IoT (Internet of Things)**, real-time analytics, streaming data processing, and collaborative applications, where the ability to react to events quickly and efficiently is critical.

Hybrid Architectures

Hybrid architectures emerge as a blend of **different architectural styles**, combining the strengths of each to address the specific needs of complex applications. These architectures are particularly useful when a one-size-fits-all approach is insufficient, and a more tailored solution is required to meet diverse requirements such as performance, scalability, maintainability, and cost efficiency.

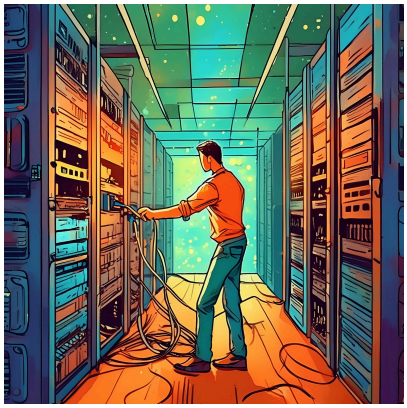
One common hybrid approach is the **combination** of **monolithic** and **microservices** architectures. In such a hybrid, the core functionalities that are stable and do not change frequently can remain in a monolithic block, while newer or more dynamic features are developed as microservices. This allows organisations to leverage the simplicity and performance of monolithic design for certain aspects of their application while benefiting from the agility and scalability of microservices for others. This approach can also facilitate a gradual transition from a legacy monolithic system to a more modern microservices-based architecture.

Another hybrid model is the **integration** of serverless functions with microservices or monolithic applications. In this scenario, serverless components can handle sporadic or event-driven workloads, such as file processing or background tasks, while the main application logic resides in a more traditional architecture. This allows for cost optimisation and scalability for specific parts of the application without committing the entire system to a serverless model.

Hybrid architectures offer **flexibility** and the ability to **optimise** different parts of the system for different concerns. However, they also **introduce complexity** in terms of integration, operation, and governance. Managing a system that spans multiple architectural styles requires careful planning and orchestration to ensure seamless interaction between components. Additionally, hybrid architectures may require developers and operations teams to be proficient in multiple technologies and practices, which can increase the learning curve and operational overhead. Nonetheless, for many organisations, the benefits of a tailored architecture that can evolve with changing business needs outweigh the challenges, making hybrid architectures a compelling choice for complex scenarios.

▼ Supporting content C - Designing system architectures

Identifying Key Components and Interactions



Identifying Key Components and Interactions is a critical phase in the design of any system architecture. This process involves the meticulous examination of the system's requirements and the identification of the essential building blocks that will constitute the system. **Key components** can range from hardware elements such as servers, storage devices, and networking equipment, to software elements including databases, application servers, and user interfaces. Each component must be carefully selected based on its ability to perform specific functions within the system, its compatibility with other components, and its scalability to accommodate future growth.

Once the key components are identified, the next step is to **understand the interactions** between them. These interactions are the pathways through which data and control signals flow within the system. They define how different components communicate, coordinate, and integrate to achieve the system's objectives. For instance, an interaction could be a data exchange between a user interface and a database, or a service call from a client application to a server. Mapping these interactions is crucial for ensuring that the system operates seamlessly and efficiently. It also helps in identifying potential bottlenecks, security vulnerabilities, and areas where performance can be optimised.

In the context of a complex scenario, the identification of key components and their interactions becomes even more critical. A **complex system** often involves a high degree of interdependence among components, with multiple layers of abstraction and numerous touchpoints. It is essential to have a clear understanding of how each component contributes to the overall functionality of the system and how changes in one component can affect others. This understanding is the foundation upon which the system's architecture is built, and it guides the selection of appropriate technologies, protocols, and design patterns that will ensure the system's reliability, maintainability, and scalability.

Defining Interfaces and Contracts

Defining interfaces and contracts is a fundamental aspect of system architecture design that ensures effective communication and integration between different components of a system. An **interface**, in this context, serves as a boundary or a point of interaction between two components, allowing them to exchange information without needing to understand each other's internal workings. This abstraction is crucial for maintaining the modularity and flexibility of the system, as it enables components to be developed, tested, and replaced independently.

Contracts, on the other hand, are formal specifications that define the rules and expectations governing the interactions between components via their interfaces. They outline what inputs are expected, what outputs will be produced, and under what conditions. Contracts can include details about data formats, protocols, error handling, and even performance benchmarks. By adhering to these contracts, components can interact reliably, even when developed by different teams or using different technologies. This predictability is essential for the stability and maintainability of the system.

In a complex scenario, defining interfaces and contracts becomes even more critical. As systems grow in size and complexity, the number of interactions increases exponentially. Without clear interfaces and contracts, managing these interactions becomes unwieldy, leading to integration issues, unexpected behavior, and increased difficulty in troubleshooting. By establishing **well-defined interfaces and contracts** early in the design process, architects can mitigate these risks and provide a framework that supports the orderly evolution of the system. This approach also facilitates the adoption of design-by-contract methodologies, where compliance with contracts is enforced through testing and validation, further enhancing the system's integrity and reliability.

Ensuring Scalability and Performance

Ensuring scalability and performance is a paramount concern in the design of system architectures, particularly in the face of growing data volumes, user demands, and market expectations. **Scalability** refers to the ability of a system to handle increased load without compromising its functionality or performance. This can be achieved through various strategies such as horizontal scaling (adding more machines to a system) or vertical scaling (upgrading the existing machines with more powerful hardware). The choice between these approaches often depends on the nature of the system, the cost implications, and the scalability limits of the underlying technology.

Performance, closely related to scalability, is the measure of how well a system executes its operations in terms of speed and efficiency. It involves optimizing the system's resources to minimize response times and maximise throughput. This can include fine-tuning algorithms, optimising database queries, implementing caching mechanisms, and leveraging **content delivery networks (CDNs)** to reduce latency. Performance considerations are not limited to the system's peak load but also encompass its behavior under normal conditions, ensuring a smooth user experience at all times.

In a complex scenario, ensuring scalability and performance becomes a **delicate balancing act**. As systems become more intricate, with multiple layers and dependencies, changes in one area can

have unforeseen effects on others. Architects must therefore adopt a holistic approach, considering the entire ecosystem in which the system operates. This includes anticipating future growth, designing for extensibility, and continuously monitoring system metrics to identify bottlenecks and areas for improvement. Additionally, the use of modern technologies such as cloud computing, containerisation, and microservices can provide the flexibility needed to adapt to changing demands while maintaining high performance and scalability.

Addressing Security and Reliability Requirements

Addressing security and reliability requirements is an indispensable aspect of system architecture design, as it lays the foundation for protecting sensitive data, maintaining system integrity, and ensuring continuous operation. **Security measures** must be integrated into the architecture from the ground up, encompassing all layers of the system, from the network infrastructure to the application level. This includes implementing robust authentication and authorisation mechanisms, encrypting data in transit and at rest, and employing intrusion detection and prevention systems to safeguard against cyber threats.

Reliability, closely intertwined with security, refers to the system's ability to perform its intended functions consistently and without failure. To achieve high reliability, architects must design systems with redundancy and fault tolerance in mind. This can involve deploying multiple instances of critical components, utilising load balancers to distribute traffic, and establishing failover mechanisms that kick in when primary systems go offline. Additionally, comprehensive monitoring and logging are essential for quick identification and resolution of issues, minimizing downtime and maintaining user trust.

In a complex scenario, addressing security and reliability becomes even more challenging, as the **attack surface** expands and the potential **impact of failures** increases. Architects must adopt a defense-in-depth strategy, layering security controls to mitigate risks at every level. This includes regular security audits, penetration testing, and staying informed about the latest vulnerabilities and threats. For reliability, architects must design systems that are resilient to partial failures, using patterns such as circuit breakers and bulkheads to prevent local issues from cascading through the system. By prioritising security and reliability in the architecture, organisations can build robust systems that withstand the rigors of the modern digital landscape.

Considering Maintainability and Extensibility

Considering maintainability and extensibility is crucial in the design of system architectures, as it ensures that systems can be easily updated, repaired, and expanded over time. **Maintainability**

refers to the ease with which changes can be made to the system to fix bugs, improve performance, or adapt to changing requirements. This is heavily influenced by the clarity and organisation of the codebase, the quality of documentation, and the adherence to design principles that promote understandability and simplicity. Systems that are maintainable require **less effort and time** to modify, reducing the costs and risks associated with ongoing support.

Extensibility, on the other hand, is the system's ability to accommodate new features and capabilities without significant structural changes. It is about building systems that can grow and evolve with changing business needs. This often involves designing **modular components** with clear interfaces that can be swapped out or enhanced as needed. It also means considering future technologies and standards that may emerge, and designing the system to be adaptable to such changes. By prioritising maintainability and extensibility in the architecture, organisations can create systems that are not only robust in their initial deployment but also capable of thriving in the face of the inevitable changes that will come.

▼ Supporting content D - Justifying application type selection

Aligning with Project Requirements and Constraints

When **aligning with project requirements and constraints**, it is crucial to select an application type that not only meets the **functional needs** of the project but also adheres to the **limitations** imposed by the environment, budget, timeline, and other factors. For instance, if the project requires real-time data processing and has stringent latency requirements, a cloud-native microservices architecture might be the most suitable application type. This approach allows for scalable and efficient processing of data, ensuring that the application can handle high loads and maintain performance standards. Additionally, microservices facilitate easier maintenance and updates, which can be critical in projects with rapidly changing requirements or those that need to be frequently updated to stay relevant or competitive.

On the other hand, if the project is subject to strict security regulations or needs to operate in an offline environment, a different application type, such as a desktop application or an on-premises server-based solution, might be more appropriate. These applications can be designed to operate without constant internet connectivity and can include robust security features to protect sensitive data. Furthermore, the selection of the application type should also consider the existing infrastructure and the skill set of the development and maintenance teams to ensure a smooth integration and support process. By carefully considering these requirements and constraints, the

chosen application type can be optimised to deliver the desired outcomes while minimizing risks and costs.

Evaluating Trade-offs and Benefits

Evaluating trade-offs and benefits is a critical step in selecting the appropriate application type for a complex scenario. Each application type comes with its own set of **advantages** and **disadvantages**, and understanding these can help in making an informed decision. For example, a mobile application might offer the benefit of portability and accessibility, allowing users to interact with the system on the go. However, this convenience may come with trade-offs such as limited processing power and the need for a responsive design that can accommodate various screen sizes and input methods. Similarly, a web application can be accessed from any device with a web browser, providing a broad reach and ease of deployment. Yet, it may face performance limitations compared to native applications and require a stable internet connection.

When evaluating these trade-offs, it's essential to **prioritise** the project's **core objectives** and **user needs**. For instance, if the primary goal is to provide a high-performance, feature-rich experience for power users, a desktop application might be the best choice, despite the potential downsides of platform-specific development and distribution challenges. Conversely, if the focus is on quick deployment and broad accessibility, a progressive web application (PWA) could strike the right balance between native app-like features and web-based convenience. Ultimately, the decision should be based on a **thorough analysis** of how each application type aligns with the project's requirements, the expected user experience, and the long-term maintenance and scalability of the solution.

Considering Development and Operational Complexity

Considering development and operational complexity is paramount when selecting an application type for a complex scenario. The chosen application type should **balance the need** for robust functionality with the practicalities of development and maintenance. For instance, a distributed system based on microservices architecture can offer high scalability and resilience but may introduce significant complexity in terms of development, testing, and deployment. This complexity can lead to increased development time and operational overhead, requiring skilled personnel and sophisticated DevOps practices to manage effectively.

On the other hand, opting for a monolithic application might simplify the development process and reduce the initial complexity, especially for smaller projects or teams with limited resources. However, as the application grows, it can become more challenging to maintain and scale, potentially leading to performance bottlenecks and longer release cycles. It's essential to consider the **long-term implications** of the chosen application type, such as the ease of onboarding new developers, the ability to adapt to changing requirements, and the overall cost of ownership. Striking the right balance between development and operational complexity ensures that the application remains agile and sustainable throughout its lifecycle.

Assessing Scalability and Performance Implications

Assessing scalability and performance implications is a key factor in determining the most suitable application type for a complex scenario. **Scalability** refers to the ability of the application to handle increased loads without compromising performance, which is critical for systems that expect growth in user base or data volume over time. For instance, a cloud-based application that leverages containerisation and orchestration tools like Kubernetes can offer superior scalability, allowing the system to automatically adjust resources in response to demand. This can be particularly beneficial for applications with variable workloads, such as e-commerce platforms during peak shopping seasons.

Performance, on the other hand, is about the responsiveness and efficiency of the application under normal operating conditions. An application that is too complex or relies on excessive third-party services may suffer from performance issues, leading to a poor user experience. In such cases, a simpler, more streamlined application architecture might be preferable, even if it means sacrificing some flexibility. It's important to **benchmark** and **profile** the application during the design phase to understand its performance characteristics and to identify potential bottlenecks. By carefully assessing scalability and performance implications, developers can design an application that not only meets the current needs of the project but also has the capacity to grow and evolve without significant degradation in service quality.

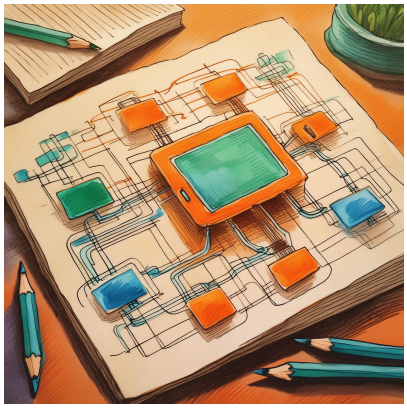
Ensuring Compatibility with Existing Systems and Infrastructure

Ensuring compatibility with existing systems and infrastructure is a critical consideration when selecting an application type for a complex scenario. The chosen application must be able to **integrate seamlessly** with the current technological landscape of the organisation, including databases, legacy systems, and other applications that it needs to interact with. This compatibility ensures that the new application can **leverage existing resources and data**, reducing the need for costly replacements or migrations and minimizing disruption to business operations.

For example, if an organisation has a significant investment in a particular programming language or platform, it may be prudent to select an application type that aligns with these choices. This could mean developing a new module within an existing monolithic application or creating a microservice that adheres to the same communication protocols and data formats as the rest of the system. Additionally, the new application should be designed with future compatibility in mind, using standardised interfaces and protocols that will allow it to evolve alongside the organisation's infrastructure. By ensuring compatibility, organisations can create a cohesive and efficient IT ecosystem that supports their long-term strategic goals.

▼ Supporting content E - Creating effective design documents

Structuring the Document for Clarity and Readability



Structuring a design document for clarity and readability is crucial for ensuring that all stakeholders, including developers, project managers, and clients, can understand the system's architecture and functionality. The document should be organised in a **logical flow**, starting with an introduction that outlines the purpose of the system, the problems it aims to solve, and the scope of the project. This sets the stage for the reader and provides context for the detailed information that follows.

The main body of the document should be divided into **clear sections**, each focusing on a specific aspect of the system. For example, one section could describe the system architecture, detailing the components, their interactions, and the data flow between them. Another section could focus on the user interface and experience, explaining how users will interact with the system. Each section should be further broken down into subsections as needed, with headings and subheadings that make it easy to navigate the document and find specific information.

To **enhance readability**, the use of diagrams, charts, and bullet points is highly recommended. **Visual aids** help to clarify complex concepts and relationships, making it easier for readers to grasp the information. Additionally, **concise language** should be used, avoiding unnecessary jargon or technical terms that might confuse non-technical readers. Definitions or explanations of specialized terms can be provided in a **glossary**. The goal is to create a document that is not only comprehensive but also accessible to everyone involved in the project, facilitating a shared understanding of the system's design.

Providing Sufficient Detail and Context

Providing sufficient detail and context in a design document is essential for enabling stakeholders to fully understand the system's architecture and the rationale behind design decisions. The document should delve into the **specifics of each component**, explaining their functionality, the technologies or frameworks they rely on, and how they contribute to the overall system. This level of detail ensures that developers have the information they need to implement the design faithfully and that other stakeholders can appreciate the complexity and capabilities of the system.

Context is equally important, as it helps readers understand why certain design choices were made and how they align with the project's goals and constraints. This includes discussing the problem domain, the target user base, and any relevant industry standards or best practices that influenced the design. By providing this background, the design document justifies its proposals and demonstrates that the design is well-considered and appropriate for the intended use cases.

Moreover, the document should not only focus on the current state of the system but also provide information on how it can be **extended or modified** in the future. This includes discussing potential growth vectors, areas that may require further development, and how the system can be scaled to accommodate increased usage or additional features. By offering this forward-looking perspective, the design document serves as a roadmap for future iterations of the system, ensuring that it remains adaptable and relevant over time.

Using Diagrams and Visuals to Communicate Architecture

Using diagrams and visuals is an effective way to communicate the architecture of a system within a design document. **Visual representations** can convey complex structures and relationships more clearly and concisely than text alone, making it easier for stakeholders to understand the system's components and their interactions. Diagrams such as UML (Unified Modeling Language) charts, entity-relationship diagrams, and flowcharts can depict class structures, data flows, and process sequences, providing a bird's-eye view of the system's design.

When creating diagrams, it is important to strike a **balance between detail and simplicity**. Too much detail can overwhelm the reader and obscure the main points, while too little detail can leave stakeholders with an incomplete understanding of the system. Effective diagrams are clear, well-labeled, and use a consistent notation or symbology that is explained within the document. This

ensures that all readers, regardless of their familiarity with the specific diagramming techniques, can interpret the visuals correctly.

In addition to static diagrams, the design document can also include **mock-ups**, **wireframes**, or **prototypes** of the user interface. These visuals help to illustrate how the system will look and function from the end-user's perspective. They are particularly useful for communicating design ideas to non-technical stakeholders, such as clients or product owners, who may not be as familiar with the technical aspects of the system but need to understand how it will meet user needs and business objectives. Overall, the strategic use of diagrams and visuals in a design document can significantly enhance the clarity and effectiveness of architectural communication.

Justifying Design Decisions and Trade-offs

Justifying design decisions and trade-offs is a critical aspect of creating a comprehensive design document. It involves **explaining the reasoning** behind choosing one approach over another, considering the project's requirements, constraints, and goals. This justification process helps stakeholders understand the thought process behind the design and the factors that influenced the decisions. It also provides a rationale for any compromises made, ensuring that all parties are aligned with the design direction and understand the implications of the chosen path.

When justifying design decisions, it is important to be **transparent** about the considerations that were taken into account. This includes discussing the trade-offs that were evaluated, such as performance versus scalability, simplicity versus flexibility, or cost versus feature completeness. By outlining the pros and cons of different approaches, the design document demonstrates a thorough analysis of the problem space and shows that the chosen design represents the best available solution given the circumstances.

Furthermore, justifying design decisions often involves **referencing external factors** such as industry standards, best practices, or existing research. Citing relevant literature, case studies, or expert opinions can strengthen the justification by providing empirical or theoretical support for the chosen design. Additionally, discussing how the design aligns with the organisation's strategic goals, technical capabilities, and resource availability helps to contextualise the decisions within the broader organisational landscape. Ultimately, a well-justified design document builds trust and confidence in the design, facilitating smoother project execution and stakeholder buy-in.

Addressing Stakeholder Concerns and Requirements

Addressing stakeholder concerns and requirements is a fundamental aspect of creating an effective design document. Stakeholders, including clients, users, developers, and other interested

parties, each have their own perspectives, needs, and expectations for the system. A design document must acknowledge and address these **diverse viewpoints** to ensure that the final product meets the project's objectives and satisfies all key stakeholders.

To address stakeholder concerns effectively, it is important to **engage with stakeholders** throughout the design process. This involves gathering input through interviews, workshops, or feedback sessions to understand their specific requirements and concerns. Once these are identified, the design document can explicitly address them by explaining how the proposed system design takes into account the various needs and expectations. This might involve detailing features that cater to user requirements, discussing how the system will scale to meet business growth, or outlining the measures taken to ensure security and compliance with industry standards.

Moreover, the design document should be **clear** and **accessible** to stakeholders with varying levels of technical expertise. This means using language and explanations that are appropriate for the audience, supplementing technical details with summaries or visual aids that convey the essence of the design. By making the document inclusive, stakeholders feel that their contributions are valued and that they have a clear understanding of how their concerns are being addressed. This inclusivity fosters a collaborative environment and helps to build consensus around the design, reducing the likelihood of misunderstandings or conflicts later in the project lifecycle.

This activity is complete when you have

- Engaged with the AI tutor in the SmartShelf case study and participated in class discussion to share your experiences and learn from others.
- Documented your analysis and recommendations for the SmartShelf case study in a short report (1-2 pages, or a copy of the chat transcript), which will form part of your **portfolio (https://lms.griffith.edu.au/courses/24045/pages/building-a-portfolio-for-assignment-2)** .
- Applied the concepts of Activities 2.1 and 2.2 to your **application system design report (https://lms.griffith.edu.au/courses/24045/assignments/93487)** .