Composition of Textbooks and Books,
Translation Works and Compilation Works.
MFU Learning Innovation Institute

SOFTWARE MODELLING AND ARCHITECTURAL DESIGN

A Comprehensive Guide to Modelling, Designing, Analysing, and Evolving Techniques

Nacha Chondamrongkul

SOFTWARE MODELLING AND ARCHITECTURAL DESIGN

A Comprehensive Guide to Modelling, Designing, Analysing, and Evolving Techniques

Nacha Chondamrongkul

**Composition of Textbooks and Books,
Translation Works and Compilation Works.
MFU Learning Innovation Institute**

Book title	Software Modelling and Architectural Design
Author	Nacha Chondamrongkul
Publication Year	June 2024
Copies	5,000

National Library of Thailand Cataloging in Publication Data

Nacha Chondamrongkul.

Software modelling and architectural design.-- Chiang Rai : Learning Innovation Institute Mae Fah Luang University, 2024.
295 p.

1. Computer software -- Development. 2. Software architecture. I. Title.

005.1

ISBN (e-book) 978-616-470-090-1

Publisher	Composition of Textbooks and Books, Translation Works and Compilation Works. MFU Learning Innovation Institute 333 Moo 1, Thasud, Mueang Chiang Rai, Chiang Rai 57100 Tel: 0-5391-7897-8080 Email: mlii@mfu.ac.th Website: mlii.mfu.ac.th
Distributors	SE-EDUCATION Public Company Limited CU Book Center
Cover design	Pakakrong Panthanan
Format Editor	Rungarun Rungrattanakan
Price	250

Preface

Software architecture design plays a crucial role in the software development process, as it establishes the foundation for the subsequent stages. After many years of teaching and conducting research in this subject, I have observed that there is a lack of literature that offers a concept of software architecture design in the context of modern software systems. To build reliable and scalable software systems, it is essential to have a thorough understanding of software architecture design, but there are few books that do so in an effective manner.

As I recognise how crucial software architecture design is, I wrote this book to fill a gap in the field of study. As an experienced researcher and educator in the field, I have utilised my extensive knowledge to produce a comprehensive and upto-date guide to software architecture design. This book provides a comprehensive overview of the subject, covering both the fundamentals and the latest best practises. Additionally, it offers a concrete model for explaining software architecture in the context of modern software systems.

This book is intended for a diverse audience, including software architects, developers, and students. Whether you are just starting out in the field or seeking to expand your knowledge, this book serves as an indispensable resource for anyone interested in software architecture design.

Contents

Chapter 1

Introduction

There is an architecture to everything. Iron supports a house, while bones and skeletons support the human body. Generally, architecture provides an overall structure to support the shapes of things. Things are also supported by architecture in terms of their functions. For such reasons, software also requires architectural design. The architectural design of software focuses on representing the system at a high level, so not all designs are considered architectural designs. Software architecture design includes major design decisions, which are made at the beginning of the project. It helps to support reasoning and later guides development at the implementation stage by depicting the overall structure of the system. This chapter generally introduces the software architecture design. The objectives of this chapter can be summarised as follows:

- To provide a clear definition of software architecture design and distinguish between architectural-level design and system-level design.
- To emphasise the significance of software architecture and highlight the various roles of software architecture design in the software development project.
- To outline the comprehensive process of creating software architecture design.
- To identify the various factors that serve as inputs to the architecture design process.
- To examine the impact of emerging technologies on software architecture design.

1.1 Multi-level of Software Design

Designing software is a challenging task as the software engineers need to consider both functional and non-functional requirements. At the design phase, the software architect must consider all requirements and decide what components should be included in the system. Software architects usually create a model that depicts the overall structure of the system. Then, the system design is created by following the structure in the architecture design. In this section, we will examine both the architecture design, which serves as an overall view of the project, and the system design, which breaks the project down into smaller, more detailed components.

1.1.1 Architectural Design

Software architecture is composed of high-level constructs such as subsystems and components. These constructs are related to one another in order to support a specific set of requirements [10]. These high-level constructs are independent of the programming language and paradigm, which means they need to be translated into implementation constructs that can be used in a specific programming language during the development process.

The architectural design may include both structural and behavioral aspects. The structural aspect addresses how different constructs are related to one another, while the behavioral aspect provides details on how they interact and work together to provide functionality. These two aspects can be represented in a variety of views [54].

The term "views" refers to different ways of representing and understanding the structure and behaviour of the system. These views can be used to provide different perspectives on the system and to facilitate communication and collaboration among stakeholders. The two main types of views that are typically used in software architecture are structural and behavioural aspects. Structural aspects focus on the organisation and relationships among the different components within the system. This includes the overall layout of the system, the components that make up the system, and the relationships between them. Behavioral aspects, on the other hand, focus on the interactions and processes that occur within the system. This includes the behaviour of the system's components and the interactions between them.

The hierarchical relationship of architectural constructs is an important aspect of software architecture design, as it helps to visualise the structure and composition

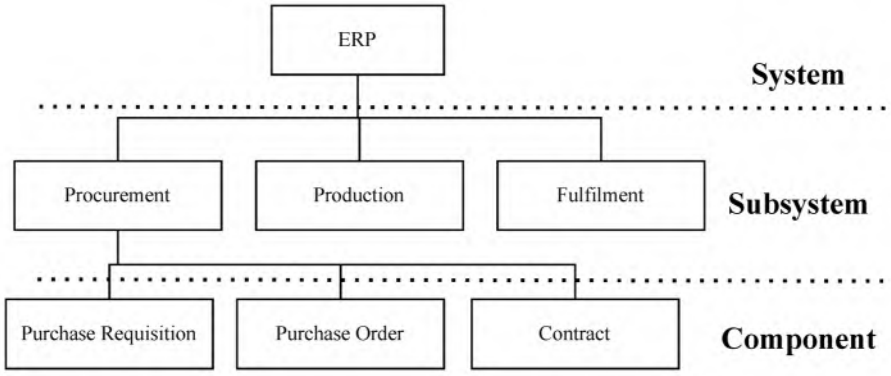

Figure 1.1: System, Subsystem and Component of ERP system

of a system. A system may consist of several subsystems, each of which is composed of related components that collectively support a specific area of system functionality.

Components are a key element in software architecture, as they encapsulate their content and functionality, providing an interface that can be utilized by other components or subsystems. For example, as shown in Figure 1.1 in an Enterprise Resource Planning (ERP) system, the hierarchy may include subsystems such as procurement, production, fulfilment, and accounting. Within the procurement subsystem, specific components such as purchase requisition and purchase order may be included.

These hierarchical relationships are important in understanding the structure of the system and identifying different parts of the system to support specific requirements. This hierarchical view is just one of many views that can be used to depict software architecture. Other views will be discussed later in Chapter 2. However, it is important to note that this section highlights the importance of the related constructs in the architectural design, and that the basic construct of architectural design is the component.

The task of designing software architecture is difficult and complex, requiring a great deal of knowledge and experience. In order to choose the best design configuration for a particular system, software engineers must carefully take into account a wide range of factors. This covers both the system's functional requirements and the various quality characteristics that must be supported, such as performance, availability, usability, security, and maintainability.

Software engineers must consider a wide range of factors when making these decisions, including the system's specific requirements, the technical potential and constraints of the selected technology stack, and the project's overall goals and

objectives. In order to find the best solution, engineers must evaluate and compare various design options, which can be a time-consuming and iterative process. To make these decisions, software engineers must take into account a wide range of considerations, including the specific requirements of the system, the technical capabilities and limitations of the chosen technology stack, and the overall goals and objectives of the project. This can be a time-consuming and iterative process, as engineers must evaluate and compare different design options in order to find the best solution.

Software engineers also need to be aware of the trade-offs that are essential when creating software architecture. A design that is highly optimised for performance, for instance, might cost more in terms of complexity and bug risk. Similar to this, a security-focused design might sacrifice performance. In order to ensure that the system satisfies both its functional requirements and its quality attributes, engineers must carefully consider these trade-offs and come to a wise decision.

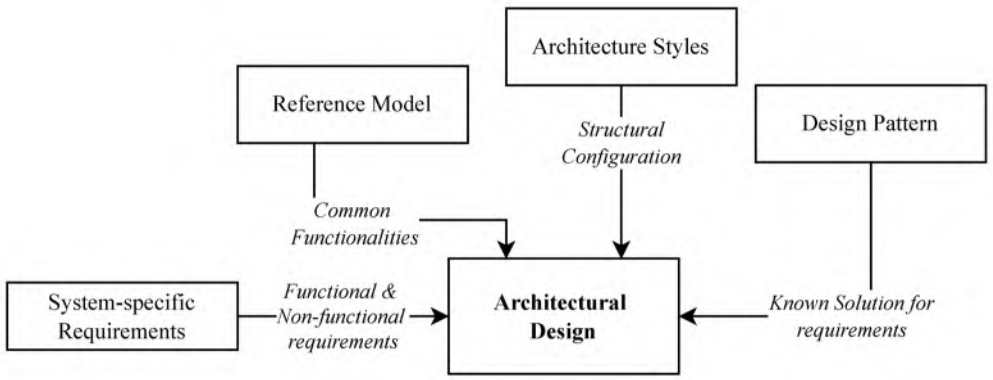

Figure 1.2: Overview of Designing Architecture

As shown in Figure 1.2, the process of designing software architecture involves several key elements, including architecture styles, architecture patterns, and reference models. Quality attributes are a critical component of this process, as they are non-functional requirements that do not directly serve the needs of users but rather characterise the system.

There are various collections of *architectural styles* [10] that can be applied to organise the structure of a design configuration, such as client-server, publishsubscribe, and peer-to-peer. These architectural styles provide a guideline for organising the design configuration in a way that facilitates the implementation of the system and supports specific requirements. For example, the pipe and filter style is well-suited for systems that require multiple steps of data processing.

Design patterns, on the other hand, are reusable solutions to common architectural problems. These design patterns provide a proven approach to solving specific problems and can be used to provide a proven approach to solving specific problems and can be used to guide the design. They provide a way of organising the design configuration, which facilitates the implementation of the system and supports the requirements. For example, MVC patterns advise to include three types of components namely model, view and controller to prevent the impacts of changes in user interface to the application logic. More details about design patterns and architectural styles are discussed in the Chapter 3.

Reference models play a crucial role in the design process. They are architectural designs of other software within the same domain as the software being designed. When designing the architecture, a reference model can be used as a starting point. This provides a common configuration that identifies the necessary key components to support key functionalities. For instance, if a reference model for an e-commerce system is available, it can give an overview of the system's general structure, including its various layers, components, and relationships among them. This can serve as a model for creating an architecture for a new e-commerce system with comparable features.

In practise, these elements are used together in the design process. Key configurations of the design can be outlined in detail using reference models as a starting point. Architecture styles are applied to provide a general structure for the system. Design patterns are then used to solve specific problems and guide the design of the system to support specific requirements.

1.1.2 System Design

System design is an important part of the software development process, as it provides the detailed instructions for implementing the software. It consists of specific implementation constructs that are dependent on the programming language and paradigm being used.

In object-oriented programming (OOP), the system design includes the details of classes, objects, interfaces, and the relationships among them. This allows the developer to understand how the different components of the system will interact and work together. In contrast, in structural programming, the system design may include different modules that contain subroutines that are used to perform specific tasks.

The system design is created after the software architecture design, which provides the high-level constructs of the system. Software engineers use the architecture design as a starting point and elaborate on it by breaking it down into the specific implementation constructs that can be used in the programming language and paradigm being used. In Chapter 5, we discuss more detail about this. The system design is a detailed representation of the software architecture design, and it guides the code development of the software.

The process of creating the system design, which includes two main stages: analysis and design, is a crucial one in the software development process. These stages are carried out to make sure that the system's requirements are completely understood and that the best possible solution is suggested.

The analysis stage is carried out to look into the system requirements. This involves creating an analysis model that contains information about the data that needs to be processed, use cases, and software functions. At the architectural design level, these specifics are not completely included. The analysis model provides a thorough understanding of the requirements that the system must meet and serves as a starting point for the design stage.

The design stage is where the solution for the system is proposed. The design stage transforms the analysis model into a design model that serves as a detailed plan for creating the software. The design model includes the specific implementation constructs that are dependent on the programming language and paradigm being used, such as class, object, interfaces and relationships among them.

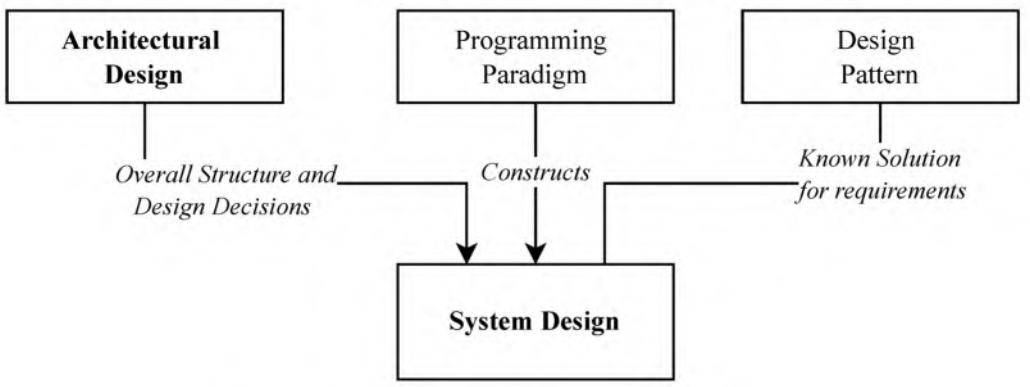

Figure 1.3: Overview of Designing System

Figure 1.3 shows the overview of how the system design is created.The architectural design serves as the foundation for the system design and provides an overview of the system. Each component in the architectural design must be detailed

in the system design stage. This includes breaking down the high-level constructs of the architectural design into specific implementation constructs that can be used in the programming language and paradigm being used.

The architectural design serves as the foundation for the system design and provides an overview of the system. Each component in the architectural design must be detailed in the system design stage. This includes breaking down the highlevel constructs of the architectural design into specific implementation constructs that can be used in the programming language and paradigm being used.

The programming paradigm is an important consideration in the system design stage, as it concerns how the code is organised, as well as the grammar and style of coding. There are several programming paradigms, such as procedural programming and object-oriented programming (OOP).

Procedural programming aims at organising the code as subroutines or procedures. A programme is constructed by making sequential calls to procedures. On the other hand, OOP aims at organising the code into classes and using concepts such as encapsulation, abstraction, inheritance, and polymorphism [80]. Therefore, the system design according to OOP includes details of how classes are implemented and how they are related. An object model is also created to represent a snapshot of how data is managed by the system at a point in time.

It is important to note that this book does not cover every details of system design. However, it demonstrates how the architectural design is transformed into the system design according to OOP in Chapter 5.

1.2 Why Architecture Matters

Having proper architectural design have many benefit as it manifest the early design decisions. Therefore, the architectural design is used as a guideline for the development as well as to facilitate communication among stakeholders in the software projects.

1.2.1 Early Design Decision

Software development is a difficult and intricate process that frequently results in delays, higher costs, and a failure to satisfy user needs and expectations. Software architecture design is typically established at the beginning phases of the software development process to reduce these risks and guarantee the success of a project. This

entails considering a variety of factors, such as functional and non-functional requirements, as well as the overall goals and objectives of the business.

Important design decisions that will have a big impact on the project's later stages must be made during the architectural design phase. Technical factors, such as the use of particular styles and patterns, as well as more general business factors, like time to market, must both be taken into account when making these decisions. It is crucial that these design decisions are made with care and consideration because they are challenging to change and have a significant impact on the overall system.

Different technical alternatives must be compared and evaluated in order to make the best design decisions, keeping in mind the advantages and disadvantages of each option. Risks can be reduced and the project's overall success can be maximised by making the right architectural decisions. It is crucial that the design choice be made after weighing and comparing various technical alternatives. This will help reduce risk while maximising the development's benefits.

1.2.2 Communication Support

The architectural design, as illustrated by graphical models, communicates how the software will be put together, as was previously stated. It represents the design choices made during the development process and is typically used as a communication tool for project stakeholders to comprehend and analyse the high-level design of software.

According to previous research [86], it can be challenging for stakeholders with different roles and backgrounds, such as customers, designers, product managers, user interface designers, developers, and system analysts, to communicate the architectural design in the software development projects. These stakeholders might have various viewpoints, objectives, and specialities, which can cause confusion and disagreements regarding the design. For instance, a developer may prioritise technical viability and scalability while a customer may prioritise functionality and usability. It may be difficult to come to an agreement on the architectural design and to effectively communicate it to all stakeholders as a result of these differences. Communication may also be hampered by a lack of common terminology and understanding of the design due to the diverse backgrounds of these stakeholders.

Effective communication between stakeholders in a project is heavily dependent on a clear and consistent architectural design. One way to achieve this is by using

consistent terminology throughout the design process, such as the same names for components and subsystems. This allows all stakeholders to understand and refer to the design in the same way, which can help prevent misunderstandings and confusion. Additionally, using common terms and patterns within the design can help developers understand how the software should be built. This is because design patterns are proven solutions to common software design problems, and using them in the architectural design can help developers quickly and effectively understand how the design should be implemented. By providing a clear and consistent architectural design with common terms and patterns, stakeholders can communicate effectively, and developers can understand and implement the design correctly.

Given how complex the process of building software can be, it is difficult to communicate the understanding of how it should be done without these common terms. To make sure that everyone working on the project has a shared understanding of the system and how it should be built, it is helpful to have an architectural design that is clear and consistent and has well-defined terms and concepts.

1.2.3 Development and Changes Management

Software architecture, which provides design details for various system elements or components, can also be used as a foundation to organise development tasks and set up the development team. It is easier for developers to understand the overall system and how their individual tasks fit into the bigger picture when the architecture of the system is defined, as it helps to clearly identify the various components and their responsibilities. When a software component needs to be reused or improved, the specification can be used as a guide. It is simpler for developers to comprehend how a component operates and interacts with other components in the system when it is clearly defined and well documented. This could speed things up and

Additionally, the architecture provides a high-level view of the software, allowing developers to determine what changes are necessary and how they will impact future improvements [17]. Developers can more accurately predict the potential effects of changes they make on the system as a whole by having a clear understanding of the overall structure and organisation of the system. This can lessen the chance of introducing new bugs or breaking existing functionality and help them make more informed decisions about how to make changes. For instance, a developer will have a better understanding of how various system components

interact and depend on one another if they are familiar with the architecture of the system. This can assist them in foreseeing the possible effects of changes they make to one component on other components.

1.3 Architectural Process

The architectural design is the high-level abstraction of a system. It is one of the early development artefacts to be developed at the design phase. In traditional waterfall software development process, architectural design of the system are created upfront before starting the development phase. Whereas, in the projects that apply agile methodology [65], the architectural design is created incrementally and evolves overtime. The architectural designs are created and used according to a certain process. There are some processes proposed for this, such as the Architecture Business Cycle (ABC). [10].

ABC gives a guideline of how software architecture design should be created and used. As shown in Figure 1.4, ABC is a cycle process as the software and its architecture evolves over time. Modern software development usually applies agile methodology so this cycle serves designing software in such way. It starts with creating the business case and after the architecture is implemented, the checking is performed to ensure that the design and implementation are aligned. The details of steps can be explained as follows:

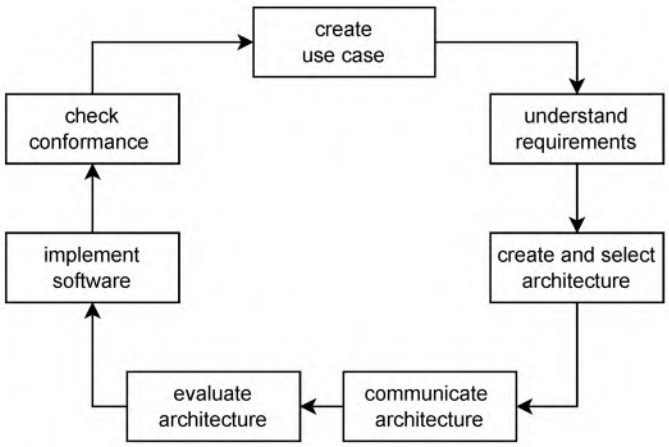

Figure 1.4: Architecture Business Cycle (ABC)

Creating the business case: This steps involves not only market assessment but it concentrate on gathering rough details of future requirement, as well as time and cost. The objective of business case is to determine whether the software is

justified to develop. The preliminary questions need to be answered such as the software's development cost, target market, development time, integrations to other systems. These questions need to be answered by different stakeholders to ensure that business goals can be achieve.

Understand requirement: is a crucial step in the software development process that allows to design a software architecture that effectively addresses the problem at hand and aligns with stakeholders needs. To accomplish that, it is important to gather information through various means such as understanding the problem domain, communicating with stakeholders, identifying use cases, and taking into account technical constraints. We discuss how us case can be identified and described in Chapter 2. Prioritizing and validating the requirements is a must to ensure alignment with the project goals, and in case of difficulties, prototyping and reviewing the design can help to clarify unclear requirements and validate design decisions.

Creating or selecting the architecture: The ultimate goal of software development is to create a system that meets the defined functional and non-functional requirements, ensuring the software works as intended, and is maintainable, scalable and secure. This is where the software architecture design comes into play, it is an essential process in the software development process, as it provides the overall structure and organization of the software system. The software architecture can be designed from scratch, considering the requirements and constraints, or it can also be based on already existing architectural styles, design patterns and tactics, which are proven and well-established solutions for common software development problems. These styles, patterns and tactics provide a starting point for the design and help to ensure that the architecture is robust and effective. With a solid software architecture, the developers can be assured that the system meets the required functional and non-functional requirements and will be able to evolve over time to meet the changing needs of the stakeholders. These styles, patterns and tactics are explained in Chapter 3.

Communicating the architecture: As the software architecture design has to be thoroughly analysed to ensure that it meets the predefined goals, the design need to be communicated among stakeholders. System analysts need to understand the architectural design to further create the system design. Likewise, developers need to understand their assignments and how software will be constructed. Therefore, the documentation of architectural design should be instructive, precise and clear for everyone in the development team. Proper diagrams and notation are required to establish effective communication, as we explain in Chapter 2.

Analysing or evaluating the architecture: When it comes to designing software architecture, there are often multiple design alternatives that can be considered. It is important that the design alternatives are carefully evaluated before making a decision at the architectural-level. This is because the success of software development relies heavily on making effective choices in a rational way. We need to ensure that the system is constructed in a way that it meets quality attributes and all other requirements. This can be achieved by systematically evaluating the software product using different methods such as formal verification, code review and testing, and by considering the trade-offs between different design options. By taking a methodical approach and carefully evaluating the design alternatives, the team can make informed decisions that will result in a robust and high-quality software architecture that can effectively meet the needs of the stakeholders. We discuss different methods to analyse software architecture design in Chapter 4.

Implementing the software: Developers can start writing the software's code once the software architecture design is complete. It is crucial that the system design, which ought to be in line with the architectural design, serves as the foundation for developing the code. The development of the code in accordance with the architectural design can be ensured by clear documentation and communication of the software architecture design. It is highly advised to set up a test environment and infrastructure that supports cooperation between developers and testers in order to ensure that the software complies with the requirements. This increases the likelihood that the software will meet the requirements and be prepared to handle the production loads by enabling testing and changes to the software before it is deployed to the production system. We discuss how we design the system based on the architectural design and how to effectively deploy the system in Chapter 5.

Ensuring conformance to an architecture: When software enters into maintenance phase, it is important to ensure that the ground-truth architecture, or the actual software architecture, is conformed to the original software architecture design. This conformance checking is important when many changes are made to the software system, as the differences of design and actual system may cause architectural erosion and can make it difficult for the software to accept any changes [58]. The importance of keeping the software architecture design in-tact and well-documented is discussed in further detail in the context of maintaining and evolving the software in Chapter 6. In situations where the architecture documentation is not well-kept, it's important to recover the current architectural design and ensure that it comply with the original design [35].

1.4 Design Inputs

There are inputs that are required when designing the software architecture. When designing software architecture, these inputs support decision making and trading-off technical alternative. They can be discussed as follows:

1.4.1 Functionalities

A specific set of is what software architecture is made to achieve. As a result, the primary factors guiding the design of software architecture are functional requirements. Functional requirements are a crucial part of the software development process and are typically described as what the system should be able to do. User preferences, business rules, administrative tasks, permission levels, audit tracking, external interfaces, certification and legal requirements, reporting requirements, and historical data are just a few examples of the many different sources that contribute to these requirements.

User preferences are typically the source of functional requirements. The system must provide for the specific needs of its users. For instance, a user might demand that the system quickly search for specific data or that it have a certain level of security. When creating the architecture of the system, these requirements must be taken into consideration.

Business rules are another source of functional requirements. Business operating procedures and policies are known as “business rules. A company might have a rule stating that all transactions must be recorded or that all data must be encrypted, for instance. To guarantee that the system complies with the business’s policies and procedures, these regulations must be reflected in the architecture of the system.

Administrative tasks also produce functional requirements. The tasks required to manage and maintain the system are referred to as “administrative functions. An administrative function might include, for instance, the capacity to monitor system performance or perform data backups. For the system to be effectively managed and maintained, these functions must be reflected in the system architecture.

Authorization levels, audit tracking, external interfaces, certification and legal requirements, reporting requirements, and historical data are additional sources of functional requirements. To make sure that the system satisfies all necessary requirements, these must also be taken into account when designing the system's architecture.

The domain of application that software is built for helps to understand what data, the amount of data, and the relationship among them will be processed and stored on the software. For example, if the software is designed for the healthcare industry, it must be able to process and store significantly more accurate of sensitive data and adhere to any necessary certifications and laws, like HIPAA. Understanding the data model and the data storage options that must be used in the software architecture is made easier with this knowledge.

1.4.2 Quality Attributes

The are non-functional requirements that software engineers need to consider when designing software architecture. There are various quality attributes, which can be categorised into three groups namely system quality attribute, business quality attribute and architecture quality attribute [10]. The system quality attributes directly determine the characteristics of the system. The business quality attributes related to the business stakeholders, such as sales, marketing and management. The business quality attributes represents the business in relation to the system, such as cost, delivery schedule and market. The architecture quality attributes are the design constraints that characterise the architectural design, such as conceptual integrity and buildability. The conceptual integrity is how the design should be consistent from the architectural-level to system-level. The buildability is an ability to complete the development in timely manner.

There are numerous quality attributes proposed by academia and industry. In this book, we focus on important system quality attributes that are commonly applied in many software such as availability, performance, security, testability, usability and security. Because they fundamentally influence the architectural design in both structural and behavioural aspect.

Availability is an ability of the system that can operate when the users need. The concerns the system that is noticeable by the users or other integrated systems. The availability is usually defined by percentage. For example, 99.9% availability

means that there is a 0.01% probability that the system will not be operated when needed.

When a system fails, the time it takes to repair it is an important concern. This is because it is time that users and external system have to wait until the system become available again. The availability percentage is calculated based on the service hours that are predefined. This service hours usually define the operation time of users such as 8 hours on 5 working days or 24 hours on 7 working days.

Performance identifies how responsive the system is in processing and responding to events. Having events from the requests of users and the integrated external system complicates the ability to predict when the system has a high or low load of computation. The measurements for are such as latency and throughout. *Latency* is the time taking between the arrival of request and the system responds. Throughput is the number of requests that the system can process in a certain period of time.

Security characterises the capability of resisting unauthorised access or malicious attacks, while the system can still serve authorised access. Security attacks are usually attempting to breach the policy. These attempts allow the unauthorised individual to access and/or modify the data and services. There are subcategories of quality attributes under security. *Nonrepudiation* defines the ability to provide an access to users who have authority to access. *Confidentiality* is an ability to protect from unauthorized access. *Integrity* is an ability to protect data from unauthorised changes. *Assurance* is an ability to ensure that only authorised users are involved to the transaction.

Modifiability concerns the cost of making modifications to the software. The modification may support any changes at non-functional and/or functional requirements. During the maintenance phase, some functional requirements may be changed or added according to evolving business operations so some parts of software require modification to support. For non-functional requirements, the software may require some modification to support changes in other quality attributes. For example, when the security policy is changed, the system must be reconfigured to support it. The number of concurrent operations and users may increase, which affects the system's performance. The modification in the software can be classified into three levels. Firstly, at the user level, the users can modify by themselves. For example, to create an ad-hoc report, the user can select the options to specify the data fields queried from the database and the data field to sort. Secondly, at the administrative

level, the change can be made by modifying some parameters in the configuration to reflect the changes in the business rules. These modification carried out by administrators are intentional and require to be addressed during the design phase. Lastly, at the developer level, the source code needs to be modified to support new or changed requirements. The modification of the source code usually requires re-building and redeploying to the production system. According to the study [36], the modification by the developer has the highest cost and risk, in comparison with the modification at the other two levels.

Usability is the degree to how easy the software can be used by the users to accomplish their goals. There are different views to assess the as follows. Firstly, the software should support the users in self-learning the usage of its functionalities. Secondly, the users should be able to operate the software with minimum time and effort. Thirdly, user interfaces should prevent the error caused by the users. Lastly, the users are confident in operating on the software.

There are other quality attributes in software architecture design, such as explainability, interoperability, scalability, reliability, etc. These attributes are important to ensure that the system is robust and meets the needs of end-users. Some of these attributes overlap and are related to each other, and understanding these relationships is crucial for designing a successful software system.

Explainability is an important quality attribute that involves usability, as it is how computation in the software can be explained. The aims at building trust in using the system, as users should be able to understand how the system works and how it produces results. Explainability is particularly important in systems that are used in critical domains, such as healthcare or finance, where users need to have confidence in the results produced by the system.

Interoperability is another important quality attribute that refers to the ability of different systems to work together seamlessly. The is important in today's world, where systems often need to integrate with other systems in order to achieve a common goal. Interoperability can be achieved by using open standards and protocols, which can help ensure that different systems can work together seamlessly.

Scalability is another important quality attribute that refers to a system's ability to handle an increasing amount of load and users without degradation in performance. The is important for systems that are expected to grow in popularity or usage over time. Scalability can be considered modifiable, which focuses on accepting the changes in a load of requests. A good architecture should be designed in such a way that it can easily be scaled to accommodate the increasing load.

Reliability is used interchangeably with availability. It refers to a system's ability to work correctly over time. This is important for systems that are used in critical domains, such as healthcare or finance, where users need to have confidence in the system's ability to work correctly over time. The can be improved by using techniques, such as fault tolerance and redundancy, which can help to ensure that the system continues to work correctly even when faced with unexpected failures.

1.5 Emerging Technological Environment

Given that they are methods for addressing a specific issue or requirement, the technologies used to create the software have a significant impact on how the software architecture is designed. It is the architect's responsibility to comprehend the fundamentals of these technologies at the beginning of the development project, including what requirements they help to achieve and how they are implemented and deployed. Software has evolved to be more dynamic, intelligent, and decentralised in recent years. We briefly discuss prominence technologies, which dramatically impact on how software architectures are designed. These technologies include microservices, edge computing, blockchain, and artificial intelligence.

1.5.1 Microservices

In conventional software development, the software is handled as a single unit in the development, testing and deployment. Even though they consist of multiple components and subsystems, they are highly dependent on each other and cannot work without one another. Therefore, different parts of software need to be developed together and executed on the same platform. This style of development produces a software system as a monolith. However, modern software systems become more complex and yet require constant changes. The development and maintenance of modern software systems as monolith are challenging, as making changes on a part of the software requires a tremendous amount of effort to rebuild and test the whole system. Microservice [74] can help to resolve this problem.

1.5.1.1 Overview

Developing the software as can help to minimise the risk and effort in developing and maintaining the complex software system. The development of microservices applies the single-responsibility principle, in which the system is

divided into multiple components loosely dependent on each other. A microservice component has a single responsibility that it does best by providing one or more services to other microservices. Different microservices on the same system may be developed by different teams and run on different platforms. Therefore, it is flexible to select the platform, such as application framework, programming language and database, which is best suited for the required responsibility.

Each microservice usually manages its own database. Figure 1.5 shows the sample of microservices for an online shopping system, which consists of four microservice components. It can be seen that each service is developed with a different programming language and there is no centralised database. The microservices such as Account Service, Product Catalogue and Payment Service manage their own database. To support a high load of requests, each microservice can be individually upgraded. In contrast, a software system as a monolith requires the whole system to be upgraded.

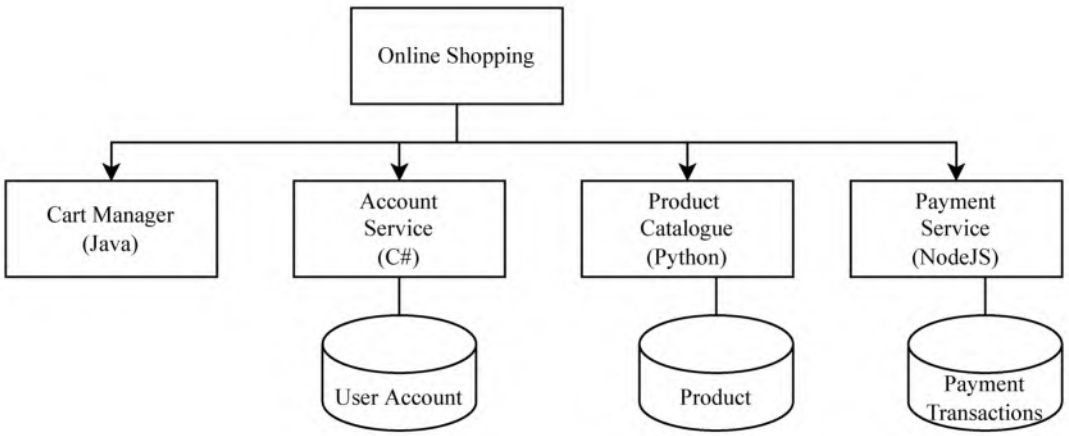

Figure 1.5: Sample Microservice

Building microservice architecture starts with gathering the functional requirements. After that, the services can be defined for the requirement. Each service must be small and take a single responsibility so a boundary of service must be clearly defined. Then, a development team consisting of no more than seven engineers is assigned to build each service. After the services are built, each must be prepared to deploy in a separate environment. The containerisation technology is usually applied to deploy the microservice. The container is a standard package of software that includes the code and all dependencies to execute the service. It can be transported to

run on a different physical system without setting up a new environment to support the service. To support a high load of requests, we can run multiple instances of a container. This allows the requests to be distributed on different instances.

1.5.1.2 Architectural Impacts

Applying microservices as a development principle raises a number of challenges. Firstly, a complex system may consist of many microservices. Therefore it is difficult to manage the number of service instances and their locations, which may be changed dynamically; not to mention the new microservices that may be added in the future. Secondly, the format of services input may be different due to the development platform that the service used to build and execute.

Thirdly, as microservices maintain their own data, the same set of data may be stored in more than one database for different purposes such as recording and analytic reporting. Therefore, data consistency becomes an issue. Traditional techniques, such as transactions, only ensure consistency in a centralised database setting. Fourthly, testing and monitoring are essential in the microservices architecture as services run independently. When a failure occurs, it is a challenge to find the source of the failure without proper monitoring. Also, as a microservice relies on other microservice, if one fails, it could lead to a cascading failure. The cascading failure occurs when the component spends a period of time making repeated requests until time-out. During this period, the critical resources such as memory, thread and database, may be blocked and cause other related parts of the system to fail. Lastly, as the microservices are usually deployed dispersedly in the multi-cloud infrastructure, the number of attack surfaces increases. Without proper security mechanisms applied to the design, it could allow malicious activities that affect the confidentiality and integrity of data.

1.5.2 Artificial Intelligence

Some modern software are required to solve complex problems that no static algorithms can solve. Finding the solution may involve analysing a large set of data, in order to elicit certain patterns. It may also involve performing a repetitive task to exhaustively explore many alternatives. This kind of task would take a tremendous amount of time for humans to perform. Artificial Intelligence (AI) can help to achieve this in an automated manner.