

CHAPTER 2 - LITERATURE REVIEW

2.1 Swiss German University

Swiss German University (SGU) was established in 2000 and is the first international university in Indonesia with international double degree programs. Partnered with Germany, Switzerland, and Australia, SGU aims to bring strong international focus and promote knowledge and technology in the Indonesia's higher education system. SGU offers 12 Bachelor's Degree Programs and 4 Master's Degree Programs ranging from Engineering, Information Technology, and Business to Life Sciences and Social Sciences. SGU has partnered with 26 universities and 250 companies all across Europe.

After the Corona virus pandemics, SGU continues to accelerate its digital transformation efforts. SGU already has its own university management system to help streamline administrative processes and enhance the overall learning experience for students. After the MBKM program is implemented by the government, SGU realizes the need for a different portal to accommodate this initiatives. SGU realized the need for a separate portal to accommodate this initiative. This new portal will be used by SGU students, faculty, lecturers, partner companies, and exchange students.

2.2 Merdeka Belajar-Kampus Merdeka (MBKM)

The Merdeka Belajar-Kampus Merdeka (MBKM) policy, which has been implemented since 2020 and outlined in Permendikbud Number 3 of 2020, has become the basis for students to continue to develop talents and knowledge outside the campus. Students can choose to study outside their study program and campus for a maximum of up to 60 credits. Permendikbudristek Number 53 of 2023 concerning Quality Assurance of Higher Education emphasizes the right of students to be able to study outside their study program for 3 semesters. There are nine activities in the MBKM program that students can choose from, namely; internships, teaching in schools, humanitarian projects, independent studies/projects, entrepreneurship, research/research, student exchanges, village projects, and state defense.

2.3 Software Engineering

Software engineering is defined as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; the application of engineering to software" by the ISO/IEC/IEEE Systems and Software Engineering Vocabulary (SEVOCAB). Software engineering is about using a structured and methodical way to create, operate, and manage software. It involves applying engineering principles to ensure that the software is reliable, efficient, and meets the needs of users. This approach includes planning, designing, coding, testing, and maintaining software products. The goal is to produce high-quality software that is delivered on time and within budget. Next section will discuss software engineering based on Software Engineering Body of Knowledge (Swebok) guidance version 3 (IEEE Computer Society 2014)

2.3.1 Software Requirements

A software requirement defines what a software system must do or a constraint on how it should perform. Requirements are essential to establish a common understanding of the features and functionalities of the system among all the stakeholders involved. Software requirements should specify and define the needs across different departments and levels of an organization, therefore, requirements have to be specific and quantified as precisely and clearly as possible to ensure they can be verified.

Software requirements can be classified into product and process requirements. A product requirement lists the needs and constraints of the software, while a process requirement imposes constraints on the development process of the software. Software requirement can come directly from the stakeholders or indirectly from other requirements. Product requirements can be further classified into functional and non-functional requirements. Functional requirements, often known as capabilities or features, describe what the system should do. Non-functional requirements, also known as constraints or quality requirements, define how the system should be. Non-functional requirements can include performance, maintainability, safety, reliability, security, and interoperability.

Requirement Process

The requirements process describes the overall process of managing requirements. This process needs to be adapted to the specific context of the organization and project. Key activities include elicitation, analysis, specification, and validation. The requirements process involves various roles, including users, customers, market analysts, regulators, and software engineers called stakeholders. Each stakeholder has different needs which needs to be communicated and negotiated while considering budgetary, technical, and regulatory constraints. Identifying all stakeholders and understanding their needs, while also maintaining adequate support and management, is crucial to ensure the software meets user needs and is maintained properly throughout its lifecycle.

Requirement Elicitation

This involves gathering requirements from various sources like business goals, business rules, domain knowledge, organization operation and environments, or stakeholders, which can be done through various techniques like interviews, surveys, and observations, or through analyzing competitor's products. Software engineer need to keep track and manage these sources to ensure complete understanding of the requirement. It is the initial stage where the stakeholders are gathered, problem is analyzed, and the scope is defined. It is fundamentally a human activity where relationship between development team and customer are established

Requirement Analysis

Requirement analysis involves analyzing the collected requirements to detect conflicts or inconsistencies, categorize them, and prioritize them. Requirement analysis develops conceptual models, such as use case diagrams, data flow models, and state models, depending on the problem the software is trying to solve. Requirement analysis also creates a system architecture. A software engineer often acts as a software architect. It is the architecture and design components that will be responsible for satisfying the requirements. These models help visualization of the problem and their relationship, aiding in the

comprehensive understanding of the system. This process helps decision making based on cost-value analysis.

Requirement Specification

This is about documenting the requirements in a clear and detailed manner that can be reviewed, evaluated, and approved, often resulting in documents like the Software Requirements Specification (SRS), which include the system's overall objectives, target environment, constraints, assumptions, nonfunctional requirements, and conceptual models illustrating the system context, usage scenarios, and workflows. This documentation serves as a guide for developers and stakeholders. It should also provide a realistic basis for estimating product costs, risks, and schedules.

Requirement validation

The primary aim of requirements validation is to verify that the requirements define the right software, ensuring that the software that is to be built will meet the user's expectations and needs before resources are committed to adding to the requirements. One most common technique involves a group of reviewers examining the required documents to identify errors, unclear assumptions, lack of clarity, or deviation from standard practices. Model validation is also a common validation technique, validating models developed during the analysis phase.

Validation of the requirements can be performed during all phases of the software Lifecycle. Prototyping can be performed after the design phase to visualize the software functionality and provide feedback. They are particularly useful for clarifying complex or dynamic requirements such as user interfaces. Another common technique is acceptance test. Acceptance test is an essential part of requirements validation performed by end-users.

Practical considerations

The requirements process is not straightforward. Instead, it is iterative by nature and spans the entire software life cycle. This means requirements can evolve as new information emerges or as the project scope changes. It is impractical to

think of the requirements process as a one-time activity that ended after the requirement documents were handed over to the development team. Most software projects, especially large ones, never have requirements that are perfectly understood or specified from the outset.

Given that requirements can change over time, maintaining well-documented requirements becomes critical for assessing the impact of changes and ensuring continuity in understanding on the product being developed. Proper documentation often crucial as the organization grow and expand.

2.3.2 Software Design

Design in software engineering is the process of defining the architecture, components, interfaces, and other characteristics of the system. Software design involves analyzing software requirements to produce building blocks of the software, providing detail for construction. Software design includes two main activities: software architectural design and components, the top-level structure of the software, and detailed design, which specifies each component in detail for construction.

Software design principles

Software design involves key principles such as abstraction, coupling and cohesion, decomposition and modularization, encapsulation and information hiding, separation of interface and implementation, sufficiency, completeness, and primitiveness, and separation of concerns. Abstraction means focusing on relevant information and disregarding irrelevant details. Coupling measures, the relationship among software modules, while cohesion measures the strong module relationship. Decomposition and modularization mean the software must be divided into smaller components for different functionalities. Encapsulation and information hiding internal detail from external entities. Separation of interface defines a component for a public interface. Sufficiency, completeness, and primitiveness mean software components reflect the abstraction and are easy to implement. And separation of concerns means distinct separation of components.

Key Issues in Software Design

Key issues are issues that must be taken care of when designing software. These issues include concurrency, which concerns how designs can manage multiple computations at the same time. Control and handling of events, focusing on how the software can respond to external and internal stimuli through callbacks. Data persistence, the design have to addresses the need to manage long-lived data across operations, ensuring data is stored and can be retrieved accurately. Other critical aspects include the distribution of components, which concerns the allocation of software components across different hardware or networked systems, and error and exception handling, which aims to ensure that software can handle issues or errors without crashing. Interaction and presentation deal with user interface design and the organization of information, while security is focused on protecting the software from unauthorized access and ensuring the integrity and confidentiality of data. These key issues are interconnected and often overlap, requiring a comprehensive approach to design with key issue in mind.

Software Structure and Architecture

Designing software involves creating software structure and architecture. Software structure and architecture refer to the overall design and organization of a software system, which determines how different parts of the software are arranged and interact with each other. This involves defining the main components or modules of the software, such as databases, user interfaces, and logic components, and deciding how these components communicate and work together to achieve the software's goals. The architecture provides a high-level view that helps developers understand the system, manage its complexity, and ensure it meets requirements.

Designing software also involves deciding Architectural styles. Architectural styles are common patterns or templates used to organize a software system's structure. These styles, such as layered architecture or client-server architecture, provide a standard way of organizing components and their interactions. For example, a layered architecture separates software into different layers like the

user interface, business logic, and data storage, making it easier to manage and modify each part independently.

Design patterns must be decided as well. Design pattern are reusable solutions to common problems that developers face when designing software. They offer a way to solve specific design challenges in a consistent manner. For instance, the Model-View-Controller (MVC) pattern separates the software into three components—model (data), view (user interface), and controller (business logic)—which helps in organizing code and improving maintainability. Other patterns, like Singleton or Observer, provide solutions for managing object creation or communication between objects. Using design patterns helps developers write code that is more modular, easier to maintain, and adaptable to change.

User Interface Design

User interface design focuses on creating effective ways for users to interact with software. It aims to make the software easy to learn and use by employing principles like consistency, learnability, and minimal surprise. User interface design considers various interaction styles, such as direct manipulation (like dragging and dropping), menu selection, and natural language commands, to ensure the software meets user needs. It also involves presenting information clearly, whether through text or graphics, and using design strategies to separate the user interface from business logic.

The design process must also consider accessibility and usability for all users, including those with disabilities. Effective user interface design integrates user feedback and iterative prototyping to refine how information is displayed and how users interact with the software, ensuring the interface aligns with users' expectations and is adaptable to different devices and cultural contexts

Software Design Quality Analysis and Evaluation

Software design involves assessing how well a software design meets certain standards and requirements to ensure it is effective, efficient, and maintainable. The goal is to determine whether the design aligns with key quality attributes

such as performance, security, usability, scalability, and maintainability. Quality analysis uses various techniques, such as code reviews, architecture evaluations, and design inspections, to identify potential weaknesses or areas for improvement in the design before the software is fully developed. This process helps ensure that the software will perform as expected under different conditions and that it will be easier to update and maintain over time.

Evaluation also includes using specific metrics and measures to quantify different aspects of the design quality. For example, metrics might assess how well the design can handle increased user load (scalability) or how secure the design is against potential threats. By systematically evaluating these qualities, developers can make informed decisions to refine the design, address any identified issues, and ensure the software meets both technical requirements and user needs. This proactive approach to quality helps avoid costly errors later in development and supports the creation of a robust, reliable software system that can adapt to future changes.

Software Design Notation

Software design employs software design notation, which is visual or textual representations used to describe and communicate the structure, behavior, and interactions of a software system. These notations provide a clear and standardized way to document different aspects of the software design, making it easier for developers, stakeholders, and team members to understand and collaborate on the project. Common notations include diagrams and models that represent the static structure of the software, such as class diagrams or entity-relationship diagrams, which show the relationships between different components or data entities.

Other notations focus on the dynamic aspects of the software, such as sequence diagrams or state diagrams, which illustrate how the system behaves over time or in response to certain events. These diagrams help to visualize the flow of control and data within the software, making it easier to identify potential issues or inefficiencies in the design. Using design notations ensures that the

software's design is well-documented, reducing misunderstandings and improving communication among team members. It also supports better planning, analysis, and decision-making throughout the development process by providing a detailed and organized representation of the software's architecture and behavior.

Software Design Strategies and Methods

Common design strategies include object-oriented design, which focuses on organizing software around objects that represent real-world entities, and function-oriented design, which centers on defining functions or procedures to perform specific tasks within the software. These strategies help in breaking down complex problems into smaller, more manageable parts, making the software easier to develop and maintain.

Methods like component-based design emphasize building software using reusable components or modules that can be easily integrated and swapped out. This approach enhances flexibility and allows for faster development by reusing existing code. Another method, data structure-centered design, focuses on selecting the most appropriate data structures to efficiently manage and process data. Each of these strategies and methods provides a different way to think about and solve design challenges, allowing developers to choose the most suitable approach based on the specific needs and constraints of their project. By applying these strategies and methods, developers can create well-structured, high-quality software that meets user needs and is easier to maintain and extend over time.

2.3.3 Software Construction

Process of writing and compiling the source code that makes up a software system is called software construction. The goal of software construction is to create code that is correct, efficient, maintainable, and easy to understand. It involves various activities, such as writing code, debugging, testing, and integrating different parts of the software to ensure they work well together. core principles and practices of software construction involve in building software includes writing the actual source code, debugging, testing,

and integrating different software components. The goal is to create software that is not only functional but also efficient, reliable, and easy to maintain. Fundamental aspects of software construction include minimizing complexity, which means writing simple and clear code that is easy to understand and maintain, and anticipating change, which involves designing software that can easily adapt to new requirements or modifications over time.

Other key principles include constructing for verification, which ensures that the code can be easily tested and debugged to verify that it works as intended, and reuse, which involves using existing code and libraries whenever possible to save time and effort. Software construction also emphasizes following coding standards and best practices to ensure consistency and quality across the software project. These fundamentals are essential for creating robust, high-quality software that meets user needs and can be efficiently maintained and extended in the future.

Managing Construction

Managing construction involves overseeing the various activities and processes required to build software effectively and efficiently. This includes planning the construction phase, coordinating tasks among team members, and ensuring that the software is built according to specifications and quality standards. Effective management of software construction requires careful consideration of the development lifecycle, including choosing the right development model (such as Agile or Waterfall), defining milestones and deliverables, and allocating resources appropriately to ensure that construction stays on schedule and within budget.

It also involves construction planning, which covers identifying the specific tasks needed to build the software, setting timelines, and estimating the required effort and resources. Construction measurement is another key aspect, which involves tracking progress and quality through metrics such as code coverage, defect density, and build frequency. Additionally, managing construction includes handling practical considerations like integrating different components smoothly, using the right tools and environments for development, and ensuring

that team members follow coding standards and best practices. Overall, managing construction ensures that the software development process is organized, efficient, and capable of delivering high-quality software products.

Practical Considerations

software construction refer to the real-world factors that developers and teams must address to build effective, maintainable, and high-quality software. These considerations include choosing the right tools and environments, such as integrated development environments (IDEs), version control systems, and debugging tools, to support the development process efficiently. Developers must also consider the specific programming languages and libraries best suited for the project, balancing factors like performance, scalability, and ease of use.

Another important aspect is code construction practices, which focus on writing clean, readable, and well-documented code that is easy to understand and maintain. This includes following coding standards and conventions, implementing error handling and exception management, and ensuring code is modular and reusable. Additionally, practical considerations involve managing dependencies and integrations, ensuring that all software components work together seamlessly, and planning for testing and quality assurance to catch and fix defects early in the development process. Overall, practical considerations help ensure that the software construction process is smooth, efficient, and capable of delivering a reliable and robust software product.

Construction Technologies

Construction technologies in software development refer to the various tools, techniques, and practices used to support the construction phase of software development. These technologies are designed to improve the efficiency, quality, and maintainability of the code being written.

API Design and Use: Application Programming Interfaces (APIs) allow different software components to communicate with each other. Effective API design is crucial because it ensures that these components can interact seamlessly, promoting code reuse and modularity. Good APIs are easy to

understand, use, and extend, and they hide the underlying complexity of the system.

Object-Oriented Runtime Issues: Object-oriented programming (OOP) introduces specific runtime considerations, such as managing object lifecycles, memory allocation, and garbage collection. Understanding how objects are created, managed, and destroyed in memory is essential for optimizing performance and avoiding memory leaks.

Parameterization and Generics: These techniques involve creating code that can handle different types of data more flexibly. Parameterization allows developers to write functions or classes that operate on generic data types, enhancing code reuse and reducing redundancy.

Assertions, Design by Contract, and Defensive Programming: These are techniques used to ensure the robustness and reliability of software. Assertions are statements that check if certain conditions hold true at specific points in the code. Design by Contract involves defining formal, precise, and verifiable interface specifications for software components. Defensive programming is a practice where code is written to handle unexpected or invalid inputs gracefully, preventing crashes and errors.

Error Handling, Exception Handling, and Fault Tolerance: These are strategies for managing errors and exceptions that occur during the execution of a program. Proper error and exception handling ensure that software can recover from unexpected conditions without crashing, while fault tolerance refers to the ability of a system to continue operating properly in the event of a failure of some of its components.

Executable Models: These are models that can be executed directly, providing a way to simulate the behavior of a system before full-scale coding begins. They are useful for validating requirements and design decisions early in the development process, reducing the risk of costly changes later on.

Concurrency Primitives: Concurrency involves managing multiple operations or threads running simultaneously. Concurrency primitives, such as locks, semaphores, and monitors, help manage shared resources and ensure that operations do not conflict, improving performance and responsiveness in multi-threaded environments.

Middleware: Middleware is software that provides common services and capabilities to applications outside of what's offered by the operating system. It acts as a bridge between different software applications or components, facilitating communication and data management, especially in distributed systems.

Construction Methods for Distributed Software: Developing software that runs across multiple machines or locations presents unique challenges, such as data consistency, network latency, and fault tolerance. Construction methods for distributed software focus on addressing these challenges through techniques like replication, partitioning, and using distributed algorithms.

Performance Analysis and Tuning: This involves identifying bottlenecks in the software's performance and making adjustments to improve efficiency. Tools and techniques for performance analysis help developers understand where their code may be inefficient and provide strategies for optimizing speed, memory usage, and overall performance.

Platform Standards: These are guidelines and rules for building software that operates consistently across different hardware or software environments. Adhering to platform standards ensures compatibility and reduces the likelihood of issues when software is deployed in different environments.

Test-First Programming: This approach involves writing tests before writing the actual code that needs to be tested. Test-first programming, which includes techniques like Test-Driven Development (TDD), helps ensure that the code meets its requirements from the start and that new changes do not break existing functionality.

Overall, construction technologies encompass a broad range of tools, techniques, and practices that support the efficient and effective building of software systems. They help manage complexity, ensure quality, and facilitate maintenance, making them essential components of the software construction process.

2.3.4 Software Testing

Software testing is a critical part of the software development process that involves evaluating a software system to ensure it meets specified requirements and works correctly. The primary goal of software testing is to identify any defects, errors, or gaps in the software so that they can be fixed before the software is released to users. Testing helps improve the quality, reliability, and performance of the software, ensuring that it functions as expected under different conditions and scenarios.

There are different types of testing, such as unit testing, integration testing, system testing, and acceptance testing, each targeting different levels of the software to ensure comprehensive coverage. Testing can also be categorized based on its purpose, like functional testing, which verifies that the software's features work as intended, and non-functional testing, which assesses aspects like performance, usability, and security.

Other key principles in software testing fundamentals involve test planning, which defines the strategy and scope of testing, and test case design, which involves creating specific scenarios to test different aspects of the software. Effective software testing also includes the use of automated tools and frameworks to execute tests more efficiently and the continuous process of identifying and fixing defects to enhance software quality. Overall, software testing is essential to deliver reliable, high-quality software that meets user expectations and operates effectively in real-world environments.

Test Levels

Test levels refer to the different stages of testing that a software product undergoes to ensure its quality and functionality at every stage of development. Each test level focuses on a specific aspect of the software, and together they

provide a comprehensive approach to detecting defects and verifying that the software meets its requirements.

Unit Testing: This is the first level of testing, where individual components or pieces of code, such as functions or methods, are tested in isolation. The goal of unit testing is to ensure that each part of the software works correctly on its own. Unit tests are typically written by developers and help catch errors early in the development process, making it easier and less costly to fix them.

Integration Testing: After unit testing, integration testing checks how different components or modules of the software work together. The goal is to identify issues that may arise when combining these components, such as interface mismatches, data format issues, or incorrect interactions. Integration testing helps ensure that the integrated parts of the system function as intended and that the software works well as a whole.

System Testing: This level of testing involves testing the complete, integrated software system to verify that it meets its specified requirements. System testing is performed in an environment that closely simulates the real-world use of the software. It includes various types of testing, such as functional testing, performance testing, and security testing, to assess the overall behavior, reliability, and quality of the software.

Acceptance Testing: The final level of testing, acceptance testing, is conducted to determine whether the software is ready for release to end users. It is typically performed by the end users or clients and focuses on validating that the software meets their needs and requirements. Acceptance testing ensures that the software is fit for its intended purpose and meets the criteria for delivery and deployment.

By conducting testing at multiple levels, organizations can identify and fix defects early, reduce the risk of software failures, and ensure the final product is of high quality and meets user expectations.

Test Techniques

Test techniques are methods used to design test cases and ensure that different aspects of the software are thoroughly tested. These techniques help testers systematically identify defects and verify that the software behaves as expected under various conditions. Different test techniques target different testing needs, such as checking functionality, performance, or security.

Black-Box Testing: This technique focuses on testing the software's functionality without considering its internal structure or code. Testers provide inputs and verify outputs against expected results, ensuring that the software meets its requirements. Examples of black-box testing techniques include equivalence partitioning, boundary value analysis, and decision table testing. These techniques are useful for validating user interfaces, external APIs, and overall system behavior.

White-Box Testing: Also known as structural or glass-box testing, white-box testing involves examining the internal workings of the software. It requires knowledge of the code and focuses on testing specific code paths, conditions, and loops. Techniques like statement coverage, branch coverage, and path coverage are used to ensure that all possible paths through the code are tested. White-box testing helps identify logical errors, security vulnerabilities, and issues related to code complexity.

Gray-Box Testing: This technique combines elements of both black-box and white-box testing. Testers have partial knowledge of the internal structure of the software but also focus on testing its functionality. Gray-box testing is particularly useful for testing complex systems, such as web applications, where understanding the underlying architecture (e.g., database structure) helps identify potential issues more effectively.

Exploratory Testing: Unlike formal testing techniques, exploratory testing is an informal, flexible approach where testers explore the software's functionality without predefined test cases. This technique relies on the tester's experience and intuition to identify potential defects and issues that might not be covered

by structured tests. Exploratory testing is useful for discovering unexpected behaviors and improving the software's usability and robustness.

Model-Based Testing: In model-based testing, testers create models that represent the desired behavior of the software. These models help generate test cases automatically based on various input combinations and scenarios. Model-based testing is effective for complex systems where creating manual test cases would be time-consuming and error-prone.

Regression Testing: Regression testing is performed to ensure that recent changes or updates to the software have not introduced new defects or broken existing functionality. This technique involves re-running previously executed test cases and is often automated to quickly identify any unintended side effects of code modifications.

Performance Testing: This technique assesses how well the software performs under different conditions, such as varying loads, network speeds, or hardware configurations. Performance testing includes load testing, stress testing, and scalability testing, aiming to identify bottlenecks and ensure the software can handle expected user loads.

Security Testing: Security testing involves evaluating the software's resilience against potential threats and vulnerabilities, such as unauthorized access, data breaches, or malicious attacks. Techniques like penetration testing, vulnerability scanning, and static code analysis help identify security flaws and ensure the software protects user data and maintains integrity.

Test related measures

Test-related measures are metrics used to evaluate both the software being tested and the effectiveness of the testing process itself. These measures help determine whether the software meets its requirements and how well the tests are identifying defects.

Evaluation of the Program Under Test: This involves measuring aspects of the software, such as the number and types of faults found during testing (fault density), and classifying these faults to understand common failure modes. Metrics like fault density (the ratio of discovered faults to the size of the software) help in assessing the quality and reliability of the software. Reliability growth models are also used to predict how software reliability improves over time as faults are discovered and fixed.

Evaluation of the Tests Performed: This focuses on assessing how thoroughly the tests cover the software. Coverage measures, such as code coverage (the percentage of code executed by the tests) and functional coverage (the percentage of functional requirements tested), are used to gauge the thoroughness of testing. Other techniques, like fault seeding (intentionally adding faults to measure test effectiveness) and mutation testing (making small changes to code to see if tests can detect the errors), provide insights into the robustness of the test suite and the likelihood of undiscovered defects remaining in the software.

Test process

The test process is a structured approach to planning, executing, and managing software testing to ensure that the software meets its requirements and is free from defects. The test process typically includes several key stages:

Test Planning and Control: This stage involves defining the overall strategy and objectives for testing. Test planning includes determining the scope of testing, selecting the testing techniques and tools to be used, estimating the effort and resources required, and setting up a schedule for testing activities. Test control involves monitoring the progress of testing activities and making adjustments to the plan as needed to address any issues or changes in the project.

Test Analysis and Design: In this stage, testers analyze the software requirements and specifications to identify what needs to be tested. They then design test cases and test scripts that will be used to verify the software's

functionality and performance. Test design also includes creating test data and defining the expected results for each test case.

Test Implementation and Execution: During this stage, the designed test cases are implemented in a test environment that closely simulates the production environment. Testers execute the test cases, document the results, and compare the actual outcomes against the expected outcomes. If any discrepancies are found, they are reported as defects, and the development team is notified to fix them.

Evaluating Exit Criteria and Reporting: This stage involves assessing whether the testing objectives have been met and if the software is ready for release. Exit criteria, such as achieving a certain level of test coverage or having no high-severity defects, are used to determine whether testing can be concluded. Testers also generate reports summarizing the testing activities, results, and any remaining risks or issues.

Test Closure Activities: After testing is completed, the final stage involves wrapping up the testing process. Test closure activities include analyzing lessons learned, archiving test artifacts, and preparing final test reports. The team may also conduct a retrospective to evaluate what worked well and what could be improved in future testing efforts.

2.4 Software Development Life Cycle Agile Methodology

Agile methodology, as a lightweight software development approach, emerged to address the limitations of traditional, heavy-weight development methodologies like the Waterfall model, which often struggled with high overheads, rigidity, and resistance to change. Introduced through the Agile Manifesto in 2001, Agile emphasizes flexibility, collaboration, and iterative progress to align software development with modern, dynamic business requirements (Alsaqqa et al. 2020).

Agile methodologies differ significantly from traditional approaches, as shown in Table 2.1. While traditional models like Waterfall are predictive and heavily reliant on upfront planning, Agile is adaptive, favouring short-term, iterative cycles. Agile projects

typically involve smaller teams, lower budgets, and focus on customer collaboration and minimal documentation. In contrast, traditional methods are suited for large-scale projects with fixed requirements and high budgets (Alsaqqa et al. 2020).

Table 2.1 Agile Development vs Traditional Development

Parameter	Traditional Methods	Agile Methods
Ease of Modification	Hard	Easy
Development Approach	Predictive	Adaptive
Development Orientation	Process Oriented	Customer Oriented
Project Size	Large	Small or Medium
Planning Scale	Long Term	Short Term
Management Style	Command and Control	Leadership and Collaboration [®]
Learning	Continuous Learning while Development	Learning is secondary to Development
Documentation	High	Low
Revenue	High	Moderate to Low
Number of Employees	Large	Small
Budget	High	Low
Number of Teams	Multiple	One
Team Size	Medium	Small

Agile cover various frameworks, including:

Scrum: A team-oriented framework that focuses on iterative cycles called "sprints," with roles such as Product Owner, Scrum Master, and Development Team. Scrum is widely adopted for its simplicity and focus on project management.

Extreme Programming (XP): Known for practices like pair programming, continuous integration, and test-driven development, XP emphasizes quality and responsiveness to changing requirements.

Feature-Driven Development (FDD): Centered on delivering high-quality software features incrementally, FDD is suitable for projects requiring adaptive changes.

2.4.1 Feature Driven Development Agile Methodology

Feature Driven Development (FDD), introduced by Jeff De Luca in 1997, is an agile methodology that structures software development around a prioritized list of features derived from project requirements (Alsaqqa et al. 2020). Renowned for its adaptability, FDD accommodates changes in requirements even during the later stages of development, making it a flexible approach to managing dynamic software projects.



Figure 2.1 FDD (Alsaqqa et al. 2020)

The primary objective of FDD is to deliver high-quality results through a structured yet iterative process. As illustrated in Figure 2.1, the FDD lifecycle comprises five sequential steps that guide the development process incrementally. These steps ensure systematic progress while maintaining flexibility to adapt to evolving project needs:

1. **Develop the Overall Model:** Stakeholders define the project's context and scope during this stage. Multiple models are created and evaluated, with the most suitable model selected to align with project requirements.
2. **Build the Feature List:** A comprehensive feature list is developed based on the requirements and the chosen model. This list is reviewed and validated by customers and subject matter experts before advancing to the next stage.
3. **Plan by Feature:** A high-level project plan is created based on the validated feature list. The sequence of development is determined by stakeholder priorities and feature interdependencies. This stage establishes timelines for each feature and assigns specific developers, referred to as "class owners," to ensure accountability.

4. **Design by Feature:** This iterative stage spans approximately two weeks per iteration. Chief programmers and class owners collaborate to create design packages, including sequence diagrams for each feature. These designs are reviewed and refined to ensure alignment with project goals.
5. **Build by Feature:** The final stage involves implementing the designs, writing code, and conducting rigorous testing procedures. Similar to the design phase, this stage follows an iterative approach, with completed features released incrementally. Once all features in a set are implemented, the process repeats for the next set of features.

2.5 LLMs

Large Language Models (LLMs) are advanced artificial intelligence systems designed to understand and generate human-like text by processing vast amounts of language data. These models utilize deep learning techniques, particularly transformer architectures, to capture the complexities of natural language, enabling them to perform a wide range of tasks such as text generation, summarization, and translation with high accuracy and fluency (Wu et al. 2023). The development of LLMs has significantly advanced natural language processing (NLP), providing tools that can comprehend context, infer meaning, and generate coherent responses (Islam and Moushi 2024).

Several Large Language Models (LLMs) are commercially available and widely utilized across industries. These models are developed by leading organizations and have distinctive features that cater to various applications in natural language processing. Some of the most commonly used LLMs include: ChatGPT-4 Developed by OpenAI, BERT (Bidirectional Encoder Representations from Transformers) created by Google, Claude developed by Anthropic, and LLaMA (Large Language Model Meta AI) released by Meta (formerly Facebook).

2.5.1 ChatGPT

ChatGPT, developed by OpenAI, is an application of the GPT series specifically designed for conversational interactions. It excels in understanding context, generating coherent responses, and providing detailed explanations or assistance across diverse topics. ChatGPT leverages reinforcement learning from human feedback (RLHF) to

fine-tune its ability to generate user-centric and safe responses, demonstrating performance comparable to human levels across a range of professional and academic benchmarks (Wu et al. 2023).

ChatGPT exists in multiple versions, such as GPT-3.5 and the latest GPT-4o, each with varying capabilities. The latest versions are generally more powerful, providing deeper context understanding and more accurate responses. On traditional benchmarks, GPT-4o outperform other LLMs in text processing, reasoning, and coding abilities while establishing new standards in multilingual support, audio processing, and visual capabilities (OpenAI 2024), as demonstrated in Figure 2.2.

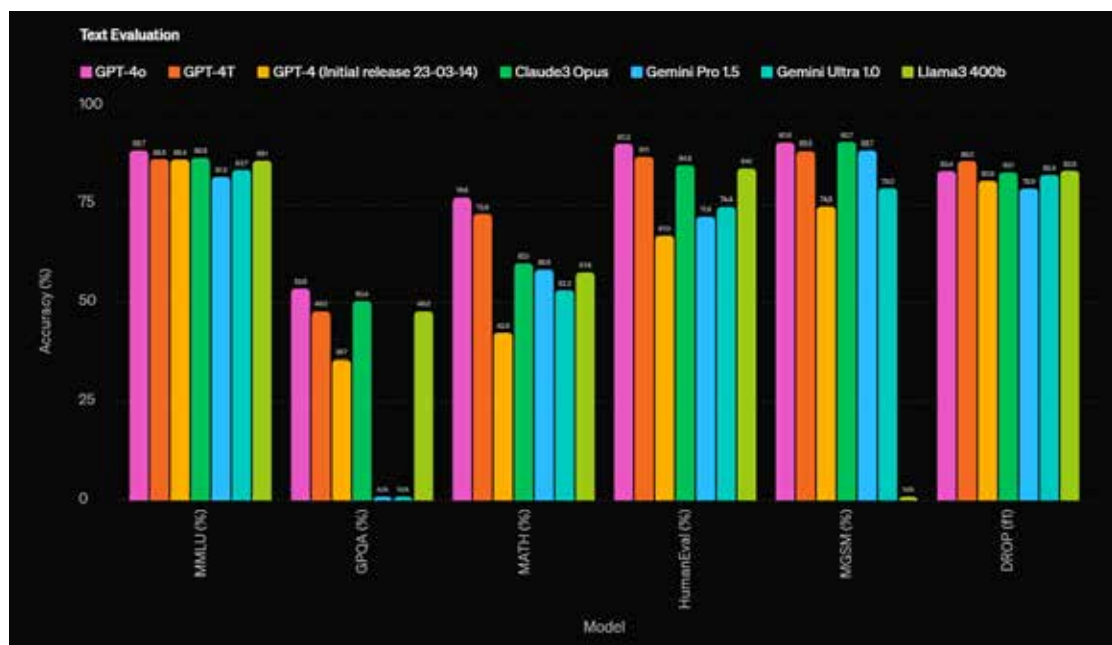


Figure 2.2 Text Evaluation Performance (OpenAI 2024)

On open-style questions evaluation benchmark (Myrzakhan et al. 2024) shown in table below, GPT-4o achieves the highest overall score. The findings emphasize GPT-4o's advancements in handling open-ended questions and its position as a leading model for reasoning, general knowledge, and domain-specific tasks, showcasing its state-of-the-art capabilities.

To effectively utilize ChatGPT, it is crucial to understand the core aspects of its functionality, including tokens, model versions, adaptability, and limitations.

Tokens and Context Length ChatGPT processes text in chunks called tokens, which can represent a word, part of a word, or punctuation. For example, the phrase "ChatGPT is a useful tool" consists of six tokens. Tokens are central to understanding the model's capabilities because it operates within a maximum token limit, which includes both the user's input and the model's response. GPT-4, for instance, supports up to 8,000 tokens, with some versions extending to 32,000 tokens. This limit means that longer conversations or documents may require concise inputs to ensure coherence and continuity in responses.

Memory and Context Limitations ChatGPT operates without memory between sessions, meaning it does not retain past interactions unless explicitly included in subsequent prompts. This stateless behavior requires users to provide sufficient context within each query, particularly for iterative or multi-step tasks. For instance, during software development, users need to consistently include code snippets or problem descriptions to maintain continuity in debugging or code-writing sessions.

Strengths ChatGPT excels at understanding and generating human-like language, making it invaluable for tasks ranging from natural language understanding to creative writing. Its ability to process complex requirements and translate them into actionable outputs is particularly beneficial in fields like software development. Developers often use ChatGPT for generating code snippets, debugging issues, and automating repetitive documentation tasks, significantly enhancing productivity and reducing time spent on mundane tasks. Additionally, ChatGPT's ability to generate creative solutions makes it a versatile tool for brainstorming and innovation.

Limitations Despite its strengths, ChatGPT has notable limitations. Its responses are not always accurate, as it generates outputs based on patterns in its training data rather than verified facts. Furthermore, the model may reflect biases present in its training data, potentially leading to skewed or inappropriate outputs. Another challenge is its static knowledge base, as ChatGPT only knows information available up to its training cutoff date (e.g., GPT-4's cutoff is 2023). For real-time or emerging knowledge, additional resources or human expertise are required.

2.6 Prompt Engineering

To fully utilize LLMs capabilities, it is essential to understand prompt engineering, which involves designing and refining prompts to elicit desired responses from the AI model. Prompt engineering is the process of crafting input prompts that effectively communicate the user's intent to the AI model, in order to improve the accuracy, relevance, and coherence of the generated responses.

2.6.1 Factors Influencing Prompt

Effective prompt engineering involves considering several factors (Ekin 2023) and design principles (Bsharat et al. 2023) that influence the quality and relevance of the generated output. These factors include:

User Intent: Understanding the user's objective and desired outcome is fundamental. A well-crafted prompt should align with the purpose of the interaction, whether it involves information retrieval, content generation, or problem-solving.

Model Understanding: Familiarity with the capabilities and limitations of the AI model is crucial. This knowledge enables the design of prompts that maximize the model's strengths while mitigating potential weaknesses.

Domain Specificity and Contextual Relevance: For specialized tasks, prompts should incorporate domain-specific terminology and provide relevant context. Including key details, background information, or situational descriptions helps the model produce responses anchored in the appropriate domain.

Task Alignment: Prompts should be closely tailored to the nature of the task. Using language and structure that clearly reflect the task requirements—such as phrasing prompts as questions, commands, or fill-in-the-blank statements—helps guide the model effectively.

Clarity and Specificity: Ambiguity in prompts can lead to suboptimal responses. Ensuring that instructions are clear and specific minimizes confusion and enhances the precision of the output.

Constraints: If specific requirements such as response length or format are necessary, these constraints should be explicitly defined within the prompt. This helps guide the model to adhere to desired output specifications.

Example Demonstrations: For complex tasks, including examples within the prompt can help illustrate the expected response format or type. This is particularly beneficial in "few-shot" or "zero-shot" learning scenarios, where examples clarify expectations.

Bias Avoidance: Prompts should be designed to minimize the activation of biases inherent in the model's training data. Neutral language and consideration of ethical implications are essential, especially for sensitive topics.

Incremental Prompting: For tasks requiring multiple steps, prompts can be structured incrementally to guide the model through each phase of the process. Iterative refinement based on the model's performance and feedback ensures better results.

2.6.2 Prompt Principle

Effective prompt engineering can be guided by a set of principles grouped into specific categories, as outlined on Table 2.2:

Table 2.2 Prompt Principles adopted from (Bsharat et al. 2023)

Category	Principle
Prompt Structure and Clarity	Design prompts with the intended audience in mind (e.g., assuming the audience is an expert in the field).
	Use affirmative directives, such as "do," and avoid negative language like "don't."
	Incorporate leading phrases, such as "think step by step," to guide the model's reasoning.
	Conclude prompts with output primers, where the desired response begins at the end of the prompt.
	Use delimiters and structured formatting, starting with tags like ###Instruction### and separating instructions, examples, or context with line breaks.
Specificity and Information	Apply example-driven prompting, including few-shot examples to guide the model.
	To enhance understanding or explain concepts clearly, use prompts like: "Explain [topic] in simple terms." "Explain as if I'm an 11-year-old."

Category	Principle
	“Write the [text] using simple language.”
	Ensure unbiased responses by explicitly requesting neutrality and the avoidance of stereotypes.
	Provide clear instructions to maintain consistency in language, such as: “Use the same language style as the provided [sample].”
	For continuing a text, use structures like: “I’m providing the beginning [text]: [Insert sample]. Continue with the same flow.”
	Explicitly define keywords, regulations, or hints necessary for content generation.
	For testing knowledge or understanding, prompt: “Teach me [topic] and include a test. Let me know if my answers are correct.”
User Interaction and Engagement	Encourage iterative clarification by having the model ask follow-up questions until it gathers enough details to provide accurate outputs.
	Use detailed requests to generate comprehensive text, for instance: “Write a detailed [essay] on [topic] with all necessary information.”
Content and Language Style	To revise text, request grammar and vocabulary improvements while preserving the original style, e.g.: “Revise this paragraph but keep its formal tone.”
	Use authoritative phrases like “Your task is” and “You MUST” to establish clear instructions.
	Assign a role to the language model.
	Avoid overly polite phrases and get straight to the point.
	Emphasize important words or phrases by repeating them within the prompt.
Complex Tasks and Coding Prompts	Break complex tasks into simpler, sequential steps in a conversational manner.
	For coding tasks involving multiple files, request scripts that automate file creation and integration, for example: “Generate a Python script to create multiple files based on the following [question].”
	Combine Chain-of-Thought (CoT) reasoning with few-shot examples to enhance problem-solving and task completion.

2.6.3 Prompting Technique

Below are prompting techniques commonly employed in generative AI systems. These methods optimize task performance through strategic prompt design and interaction. These techniques, whether used individually or in combination, exemplify the versatility of prompt engineering in enhancing generative AI performance across varied contexts and applications.

Few-Shot Prompting: Few-shot prompting involves providing the model with a few examples (exemplars) to demonstrate the desired task before requesting a solution. These examples act as guides, enabling the model to generate outputs aligned with the specified pattern or context. This technique is particularly effective for tasks requiring contextual learning without updating the model's parameters.

Zero-Shot Prompting: Zero-shot prompting does not use any exemplars. Instead, it relies on explicitly phrased instructions to guide the model in generating the desired output. This approach is suitable for general-purpose tasks and avoids the complexity of example selection while leveraging the model's inherent knowledge base.

Chain-of-Thought (CoT): prompting encourages the model to break down its reasoning process into sequential steps before delivering an answer. By explicitly structuring the logical path, CoT improves performance on complex tasks like mathematics, reasoning, and multi-step problem-solving.

Decomposition: divides a complex task into smaller, manageable sub-tasks. Each sub-task is addressed sequentially, either within a single prompt or through a series of prompts.

Assembling: combines outputs from multiple prompts or techniques to form a cohesive final answer. For example, ensembling strategies such as self-consistency aggregate diverse reasoning paths to identify the most reliable response.

Self-Criticism: prompts the model to evaluate its own output for errors or inconsistencies. It may involve iterative refinement, where the model critiques its response, offers feedback, and revises the answer accordingly.

2.7 Previous Studies

2.7.1 Prompt Pattern for Software Engineering

Prompting techniques have become a critical factor in maximizing the full potential of Large Language Models (LLMs) for software engineering tasks. These techniques, which involve crafting specific instructions to guide the model's outputs, have been

shown to significantly impact the quality and effectiveness of the results produced by LLMs.

Table 2.3 Previous Study on Prompt Pattern for Software Engineering

Paper	Objective	Methods	Contribution	Weakness
Improving ChatGPT Prompt for Code Generation (Liu et al. 2023a)	Optimize ChatGPT's prompts for better code generation.	Tested prompt engineering techniques with CodeXGlue benchmarks for text-to-code and code-to-code tasks.	Achieved significant performance improvements using chain-of-thought and multi-step optimized prompts..	High dependency on prompt specificity and limited generalization to complex scenarios.
A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT (White et al. 2023)	Create a catalog of reusable prompt patterns for software tasks.	Developed and documented 13 prompt patterns across different domains and use cases.	Provided a structured framework for reusable and scalable prompt designs.	Limited empirical validation across diverse software engineering scenarios.
ChatGPT Prompt Patterns for Improving Code Quality (White et al. 2024)	Design prompt patterns to enhance software artifact generation.	Applied prompt patterns to automate refactoring, quality assurance, and specification disambiguation.	Improved modularity and abstraction in generated artifacts.	Prompt complexity may reduce usability for non-expert users.
Improving Code Generation by Training with Natural Language Feedback (Chen et al. 2023)	Use natural language feedback during training to improve LLMs' performance.	Implemented Imitation Learning from Feedback (ILF) algorithm for iterative refinements.	Achieved 38% improvement in Pass@1 for Python code generation using human-written feedback.	Reliance on high-quality feedback and challenges in scaling for multi-bug scenarios.
Structured Chain-of-Thought Prompting for Code Generation (Li et al. 2023)	Enhance CoT prompting with program structures for better code generation.	Introduced SCoT prompting using structured reasoning and tested on HumanEval, MBPP, and MBCPP.	Outperformed standard CoT prompting in accuracy and developer preference.	Complexity in creating structured prompts and limited applicability across languages.
Test-Case-Driven Programming Understanding in LLMs (Tian et al. 2023b)	Improve code generation via test case-driven thought-eliciting and feedback-based prompting.	Devised μ FiX technique combining specification and feedback analysis to refine outputs iteratively.	Improved Pass@1 by 35.62% across six benchmarks for ChatGPT and DeepSeek-Coder.	Struggled with misaligned understanding in initial phases of complex tasks.

Table 2.3 shows that cataloging reusable prompt patterns has a significant contribution to this domain. The development of structured patterns for tasks such as requirements elicitation, quality improvement, and modular code generation has provided a scalable

framework for prompt engineering. These patterns, documented in forms similar to software patterns, allow users to address common challenges, such as disambiguation of requirements or adherence to coding standards, thereby enhancing the utility of LLMs in software development (White et al. 2023; White et al. 2024).

Another innovative approach involves natural language feedback loops, as seen in the Imitation Learning from Feedback (ILF) algorithm, which trains models using human-provided feedback during iterations. This method improves the learning process and addresses multi-step reasoning challenges, achieving a 38% relative improvement in Python code generation tasks (Chen et al. 2023).

These advancements in prompting are particularly relevant to this thesis. Effective prompting techniques could enhance LLM output by adopting structured and reusable prompting patterns. Through effective prompts, LLM capability as an assistant in software engineering tasks can be leveraged to its maximum.

2.7.2 LLMs in Requirement Engineering

Table 2.4 shows that LLMs can be used in requirement engineering to address challenges in elicitation, specification, and validation processes. LLMs, such as ChatGPT, have demonstrated remarkable capabilities in generating structured requirements, automating specification creation, and assessing requirement quality based on predefined attributes like abstraction, atomicity, and correctness (Ronanki et al. 2023b; Ronanki et al. 2023a). They are particularly adept at handling ambiguity through interactive dialogue, clarifying vague or poorly defined statements, and enabling faster alignment among stakeholders (Arora et al. 2024). Additionally, LLMs offer significant advantages in reducing the time and cost of RE tasks, allowing teams to produce outputs efficiently and with fewer resources (Marques et al. 2024; Rajbhoj et al. 2024). Their ability to integrate natural language understanding and in-context learning makes them highly suitable for generating coherent and contextually relevant requirements documentation, which enhances stakeholder collaboration and streamlines the requirements-gathering process (Fan et al. 2023; Marques et al. 2024).

Table 2.4 Previous Study on LLMs in Requirement Engineering

Paper	Objective	Methodology	Contribution	Weakness
ChatGPT as a Tool for User Story Quality (Ronanki et al. 2023b)	Assess ChatGPT's ability to evaluate user story quality.	Comparative study between manual, benchmark, and ChatGPT evaluations.	Demonstrated ChatGPT's potential in evaluating user stories with a best-of-three strategy.	Limited evaluation criteria and limited practitioner involvement.
Exploring the Efficacy of ChatGPT in Generating Requirements (Bencheikh and Höglund 2023)	Compare ChatGPT's requirements generation to human-written requirements.	Experimental study using freemium and premium ChatGPT versions, with feedback loops.	Highlighted ChatGPT's efficiency and role in supplementing human expertise.	Human-written requirements were more detailed and comprehensive.
Impact of Large Language Models on Generating Software Specifications (Xie et al. 2023)	Evaluate LLMs for generating software specifications.	Few-shot learning and prompt engineering with comparative analysis.	Identified advantages of LLMs in specification generation.	Ineffective prompts and lack of domain knowledge caused failures.
Investigating ChatGPT's Potential to Assist in Requirements Elicitation Processes (Ronanki et al. 2023a)	Explore ChatGPT's potential for assisting in requirements elicitation.	Interview-based survey comparing human and ChatGPT-generated requirements.	ChatGPT showed strong performance in abstraction, atomicity, and correctness of requirements.	Lack of domain-specific contextual understanding.
Advancing Requirements Engineering through Generative AI (Arora et al. 2024)	Assess the role of LLMs in advancing requirements engineering processes.	SWOT analysis and evaluation using real-world app scenarios.	Demonstrated potential for LLMs in RE processes like elicitation, specification, and validation.	Challenges in ambiguity management and reliance on human expertise for critical tasks.
Accelerating Software Development Using Generative AI (Rajbhoj et al. 2024)	Explore how Generative AI can accelerate the software development lifecycle (SDLC).	Systematic prompting approach validated with ChatGPT for small complex business applications.	Introduced a systematic prompting approach for SDLC and validated its practical implications.	Limited evaluation scope to small-scale projects and heavy reliance on effective prompt design.
Large Language Models for Software Engineering: Survey and Open Problems (Fan et al. 2023)	Survey the applications and challenges of LLMs in software engineering.	Comprehensive survey and literature review.	Mapped the capabilities of LLMs across the software engineering lifecycle	Lack of practical application data and over-reliance on theoretical assumptions.
Using ChatGPT in Software Requirements Engineering: A Comprehensive Review (Marques et al. 2024)	Review the applications, benefits, and challenges of ChatGPT in software requirements.	Systematic literature review focusing on elicitation, documentation, and validation.	Offered a detailed framework for leveraging ChatGPT in software requirements engineering.	Ethical concerns such as bias, inaccuracies, and lack of transparency in AI-generated requirements.

However, LLMs often lack deep domain-specific knowledge necessary for complex or niche scenarios, which can result in incomplete or inaccurate requirements (Ronanki et al. 2023a; Xie et al. 2023). The quality of their outputs heavily relies on well-crafted prompts, necessitating expertise in prompt engineering to achieve optimal results (Rajbhoj et al. 2024). Challenges such as contextual misinterpretation, ethical concerns, and stakeholder trust also limit their standalone applicability (Fan et al. 2023; Marques et al. 2024). While LLMs can generate requirements that are abstract, they may fail to provide the same level of detail and domain relevance as human experts (Bencheikh and Höglund 2023; Marques et al. 2024). Additionally, issues like hallucination—where the model generates plausible but incorrect outputs—remain a significant technical challenge in RE (Fan et al. 2023). Iterative engagement with users, as demonstrated in frameworks like AISD, can help mitigate these challenges by incorporating user feedback at each development stage.

Table 2.5 LLM Capability in Requirement Engineering

Capability	Description
Elicitation of Requirements	Rapid generation of structured requirements based on domain knowledge or stakeholder inputs
Automated Specification Generation	Creation of software specifications through prompt engineering and few-shot learning techniques
Quality Evaluation	Assessment of requirements against criteria like abstraction, atomicity, and correctness
Ambiguity Management	Resolving ambiguous statements through interactive dialogue
Speed and Cost Efficiency	Accelerating the RE process while reducing associated costs

Table 2.5 shows the capability of LLM in requirement engineering. Exploring LLM-assisted software development for the MBKM system at Swiss German University emphasizes their co-creation potential. By leveraging their strengths in elicitation, specification, and validation, the development process can be streamlined. However, the identified weaknesses must be addressed through integrating domain expertise, validating outputs, and refining prompt designs.

2.7.3 LLMs in Software Design

LLMs ability to process natural language inputs and generate meaningful outputs enables their application in tasks such as software modeling, architectural design, GUI prototyping, and specification synthesis, as shown in Table 2.6. LLMs, such as ChatGPT, have been effectively used to automate repetitive tasks, assisting with

generating software specifications, and producing visual artifacts like UML diagrams and GUI prototypes. For example, frameworks like AISD leverage LLMs for iterative design by engaging users throughout the design process, which improves efficiency and ensures alignment with user requirements (Zhang et al. 2024). Similarly, tools like RaWi employ LLMs to enable rapid GUI prototyping by retrieving reusable elements from large repositories based on natural language descriptions, significantly enhancing productivity for novice designers (Kolthoff et al. 2023).

Table 2.6 Previous Study on LLMs in Software Design

Paper	Objective	Methodology	Contribution	Weakness
Accelerating Software Development Using Generative AI (Rajbhoj et al. 2024)	Explore the use of ChatGPT in accelerating the software development lifecycle (SDLC).	Systematic prompting and meta-model concepts validated on small projects.	Proposed a prompting framework that accelerates SDLC stages.	Limited applicability to small-scale projects and heavy reliance on prompt design.
Experimenting a New Programming Practice with LLMs (Zhang et al. 2024)	Introduce and evaluate AISD for iterative LLM-driven software design and development.	Iterative design framework tested on novel benchmarks.	Demonstrated improved pass rates and efficiency in LLM-driven iterative design.	Difficulties in scaling to non-trivial, large-scale software systems.
Large Language Models for Software Engineering: Survey and Open Problems (Fan et al. 2023)	Review applications and challenges of LLMs in software engineering.	Comprehensive literature survey on LLMs in software engineering.	Identified gaps and challenges in LLM use for reliable and efficient software engineering artifacts.	High dependency on theoretical assumptions, lacking empirical validation.
On the Assessment of Generative AI in Modeling Tasks (Cámara et al. 2023)	Assess ChatGPT's capabilities in software modeling with UML and OCL.	Experimental analysis on ChatGPT's performance in generating UML diagrams.	Highlighted ChatGPT's strengths in simple syntax modeling and OCL constraints.	Poor scalability and inconsistency in handling complex modeling tasks.
Towards Human-Bot Collaborative Software Architecting with ChatGPT (Ahmad et al. 2023)	Investigate human-bot collaboration in software architecting with ChatGPT.	Case study involving novice architects using ChatGPT for service-based systems.	Demonstrated potential in bridging expertise gaps and improving architectural decision-making.	Limited insights into long-term architectural quality and impact.
Data-Driven Prototyping via Natural-Language-Based GUI Retrieval (Kolthoff et al. 2023)	Facilitate rapid GUI prototyping using NLP and data-driven GUI repositories.	Natural language-based retrieval of reusable GUI elements tested with a user study.	Improved prototyping productivity and usability for novice analysts	Restricted to GUI prototypes and limited adaptability for broader design tasks.
Large Language Models Based Automatic	Automate software specification synthesis using	Sequence-to-sequence learning with BERT for	Achieved a 21% improvement over state-of-the-art in	Requires extensive fine-tuning for domain-specific

Paper	Objective	Methodology	Contribution	Weakness
Synthesis of Software Specifications (Mandal et al. 2023)	large language models.	extracting software specifications.	specification synthesis.	text and multi-sentence dependencies.

However, LLMs also exhibit notable limitations in software design. Their performance is heavily reliant on good prompts and context-specific data, which can lead to inconsistencies or inaccuracies when dealing with complex or abstract design tasks. For instance, experiments with ChatGPT in software modeling revealed challenges in scalability and consistency, particularly when generating detailed UML diagrams or handling advanced modeling concepts (Cámara et al. 2023). Furthermore, LLMs are prone to "hallucination," where outputs may seem plausible but lack correctness, posing risks in tasks requiring precision, such as architectural decision-making (Ahmad et al. 2023; Fan et al. 2023)). The dependency on extensive fine-tuning for domain-specific tasks, as seen in specification synthesis using BERT, also highlights their limitations in adapting to varied design contexts without substantial customization (Mandal et al. 2023).

Table 2.7 LLM Capability in Software Design

Capability	Description
Accelerating Design Processes	Enhances the speed of prototyping and design through automation and retrieval techniques
Iterative Design Collaboration	Supports iterative engagement and refinement during the design process
Generating Visual Artifacts	Produces UML diagrams, GUI prototypes, and architectural artifacts based on natural language inputs
Automating Specification Synthesis	Extracts and synthesizes software specifications from textual data using advanced NLP techniques
Bridging Expertise Gaps	Assists novice designers and architects in creating high-quality designs with limited expertise

Table 2.7 shows the capability of LLM in software design. LLM capabilities and weaknesses provides an intriguing area of exploration. By leveraging their strengths in rapid prototyping, iterative design, and automation, LLMs can support the efficient development of the MBKM system at Swiss German University, particularly in addressing resource constraints and improving collaboration. At the same time, understanding and mitigating their limitations, such as through iterative human validation and integrating domain expertise, will be crucial to ensuring the reliability and quality of the designs generated.

2.7.4 LLMs in Software Development

Table 2.8 shows that LLMs have potential in software development in tasks such as code generation, debugging, program repair, and software design. Studies have highlighted the capabilities of LLMs to generate functional code and identify programming errors, achieving success rates of over 70% in certain tasks (Liu et al. 2023c; Sakib et al. 2023). LLMs can automate repetitive coding activities, assist in debugging, and even provide alternative solutions to complex programming challenges.

Table 2.8 Previous Study on LLMs in Software Development

Paper	Objective	Methodology	Contribution	Weakness
Extending the Frontier of ChatGPT: Code Generation and Debugging (Sakib et al. 2023)	Assess ChatGPT's programming capabilities.	Empirical evaluation on programming tasks.	Demonstrates 71.8% success rate on LeetCode tasks.	Struggles with debugging.
Is AI the Better Programming Partner? (Ma et al. 2023)	Compare human-AI with human-human pair programming.	Literature review and experimental studies.	Highlights AI's adaptability to varying expertise.	Limited measures for AI effectiveness.
No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT (Liu et al. 2023c)	Evaluate ChatGPT's code correctness, complexity, and security.	Experiments on diverse coding problems.	Addresses vulnerabilities and multi-round fixes.	Limited to the performance of ChatGPT at the time of work
Automatic Code Summarization via ChatGPT: How Far Are We? (Sun et al. 2023)	Assess ChatGPT's code summarization ability.	Comparative analysis with SOTA models.	Discusses open challenges in LLM-based summarization.	Inferior performance to dedicated models.
ChatGPT as a Software Development Tool (Hörnemalm 2023)	Examine ChatGPT's role in development tasks.	Observational and semi-structured interviews.	Identifies productivity and efficiency boosts.	Risk of over-reliance on AI.
ChatGPT is a Remarkable Tool—For Experts (Azaria et al. 2023)	Explore ChatGPT's utility across domains.	Case studies and theoretical exploration.	Advocates expert-guided strategic use.	Ethical and privacy concerns.
Demystifying GPT Self-Repair for Code Generation (Olausson et al. 2023)	Analyze GPT's self-repair capabilities.	Evaluation with APPS dataset.	Presented a new evaluation strategy.	Limitation on computational costs.
End-to-End Software Construction Using ChatGPT	Investigate ChatGPT in complete software projects.	Practical experiments with developer use cases.	Reveals effective prompt strategies.	Limited scalability for complex systems.

Paper	Objective	Methodology	Contribution	Weakness
(Monteiro et al. 2023)				
Refining ChatGPT-Generated Code (Liu et al. 2023b)	Improve ChatGPT's code generation quality.	Static analysis and error feedback loops.	Enhances quality by addressing maintainability.	Lacks detailed exploration of runtime issues.
Is ChatGPT the Ultimate Programming Assistant? (Tian et al. 2023a)	Evaluate ChatGPT's programming assistance capabilities.	Empirical study on code generation and repair tasks.	Insights on ChatGPT's reasoning and attention span.	Limited to ChatGPT 3.5

However, LLMs are not without limitations. They often struggle with debugging complex issues and understanding programming specifications, as seen in the limited performance of self-repair mechanisms (Olausson et al. 2023). The quality of LLM-generated code can also suffer from issues related to maintainability, runtime errors, and insufficient consideration of edge cases, as identified in studies such as "Refining ChatGPT-Generated Code" and "Towards Generating Functionally Correct Code Edits" (Liu et al. 2023b). Other than that, LLMs' effectiveness heavily relies on the quality of prompts and user inputs. Studies like "ChatGPT is a Remarkable Tool—For Experts" emphasize that LLMs perform best when guided by expert users, as non-experts may struggle to provide precise and effective prompts (Azaria et al. 2023).

Table 2.9 LLM Capability in Software Development

Capability	Description
Automated code generation and debugging	Generate functional code snippets and identify errors, achieving high success rates on programming tasks like LeetCode.
Program repair and iterative code refinement	Improve existing code by fixing bugs and refining functionality through iterative suggestions, including leveraging multi-round fixes.
Enhancing code quality through static analysis	Utilize static analysis tools and error feedback to address maintainability, style, and performance issues in generated code.
Code summarization and explanation	Provide natural language summaries and explanations of code, aiding comprehension and reusability.

The summary of LLM capability in software development is shown in Table 2.9. Relating these findings to the development of the MBKM system at Swiss German University, the capabilities of LLMs to generate and refine code based on prompts would potentially accelerating timelines and improving the system's quality. However, a structured approach to prompt engineering and the integration of human expertise is needed to ensure the generated solutions align with specific program requirements.

2.7.5 LLMs in Software Testing

Table 2.10 show that LLMs offers innovative solutions to automate labor-intensive tasks like unit test generation, vulnerability detection, and iterative test refinement. LLMs such as GPT-4 have demonstrated exceptional capabilities in detecting and fixing software vulnerabilities, outperforming traditional static code analyzers in identifying nuanced and evolving threats across various programming languages, while also providing viable fixes that significantly reduce vulnerabilities (Noever 2023). In automated unit test generation, LLMs have achieved commendable results creating human-like and meaningful test cases.

Table 2.10 Previous Study on LLMs in Software Testing

Paper	Objective	Methodology	Contribution	Weakness
Can Large Language Models Find and Fix Vulnerable Software? (Noever 2023)	Assess GPT-4's ability to detect and fix software vulnerabilities.	Comparative analysis with traditional static code analyzers on multiple programming languages.	Demonstrated higher vulnerability detection and fixing capability compared to traditional tools.	Limited scalability and lack of system-level vulnerability analysis.
Using Large Language Models to Generate JUnit Tests (Siddiq et al., 2023)	Evaluate LLMs' effectiveness in generating JUnit tests for Java.	Empirical study using multiple benchmarks to test correctness, coverage, and test smells.	Highlighted the potential for automation in test generation with good coverage for certain datasets.	Performance varied with dataset complexity and presence of test smells.
No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation (Yuan et al., 2023)	Investigate ChatGPT's capability in generating high-quality unit tests.	Quantitative analysis and user study on correctness, readability, and usability of generated tests.	Developed ChatTester for iterative refinement and improved test correctness.	Initial tests often had compilation errors and incorrect assertions.
An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation (Schäfer et al. 2023)	Evaluate TESTPILOT, an LLM-based tool, for JavaScript unit test generation.	Iterative test generation and adaptive re-prompting with failing tests.	Achieved higher coverage and non-trivial assertion rates than traditional tools.	Struggled with edge cases and generated tests were sometimes overly generic.
Experimenting a New Programming Practice with LLMs (Zhang et al. 2024)	Explore iterative test-driven development with LLMs.	Iterative framework integrating user feedback and LLM capabilities for system testing.	Enhanced efficiency and adaptability in iterative software testing.	Limited effectiveness on large-scale projects and complex system interactions.
Accelerating Software Development	Investigate systematic prompting in test	Systematic prompting approach validated	Highlighted improvements in coverage and	Results heavily reliant on well-crafted prompts and

Paper	Objective	Methodology	Contribution	Weakness
Using Generative AI: ChatGPT Case Study (Rajbhoj et al. 2024)	case generation using ChatGPT.	on small business applications.	efficiency with optimized prompts.	limited to small projects.
Large Language Models for Software Engineering: Survey and Open Problems (Fan et al. 2023)	Survey LLM applications and challenges in software engineering, including testing.	Comprehensive literature review identifying trends and gaps in LLM-based testing.	Provided insights on integrating automated testing techniques with LLMs.	Lacked empirical evaluation and practical implementations for complex testing scenarios.

Despite these promising advancements, the application of LLMs in software testing is not without limitations. A notable issue is the dependency on well-crafted prompts, as the quality of generated test cases often depends on the context provided to the AI (Rajbhoj et al. 2024). Compilation errors, incorrect assertions, and the presence of test smells such as redundant or empty tests are recurring issues in automated test generation, indicating the need for human oversight and iterative refinement to ensure test quality (Siddiq et al. 2023). Furthermore, LLMs face challenges in scaling their testing capabilities to large-scale projects with intricate system interactions, as observed in frameworks like AISD, which integrate LLMs with user-driven feedback to improve iterative development and testing.

Table 2.11 LLM Capability in Software Testing

Capability	Description
Vulnerability Detection and Fixing	Identifies and mitigates software vulnerabilities with higher accuracy than traditional tools
Automated Unit Test Generation	Generates meaningful and human-like test cases for various programming languages
Iterative Test Refinement	Refines and improves tests through adaptive re-prompting

Table 2.11 shows the LLM capability in Software Testing. By incorporating LLMs into the development of the MBKM system at Swiss German University, their capabilities in automated test generation and iterative refinement could streamline the testing process, reducing the time and effort required to validate system functionality. However, it is essential to address limitations through manual validation, leveraging domain expertise to refine outputs, and optimizing prompt designs to ensure relevance and accuracy.

2.8 Formulating Research Methodology

The research methodology in this thesis begins with a literature review to understand best practices in software engineering and the emerging capabilities of Large Language Models (LLMs) in this field. Table 2.12 summarizes the standard activities across each phase of the SDLC according to the SWEBOK guide.

Table 2.12 Software Development Lifecycle Activities

Phase	Activities (SWEBOK v3)
Requirements	• Define product and process requirements (functional, non-functional, constraints) • Identify requirements sources (stakeholders, goals, business rules, domain knowledge, operational context) • Apply elicitation techniques (interviews, scenarios, user stories, prototypes, observation, workshops) • Classify and prioritize requirements • Perform conceptual modeling (use cases, data flow, state models, object models) • Negotiate and resolve conflicting requirements • Perform formal analysis (where needed) • Produce requirements specifications (System Definition, SRS) • Validate requirements (reviews, prototyping, acceptance tests, model validation) • Manage requirements (traceability, change management, attributes, measurement)
Design	• Apply design fundamentals (abstraction, modularization, separation of concerns) • Address key design issues (concurrency, data persistence, error handling, interaction & presentation, security) • Define software architecture (architectural structures, viewpoints, styles, patterns, frameworks) • Make architecture design decisions and trade-offs • Perform detailed design of components (data structures, algorithms, interfaces) • Conduct UI design (principles, interaction modalities, localization) • Evaluate design against quality attributes (performance, reliability, maintainability, usability) • Document design using notations (UML, ERD, DFD, ADL, behavioral diagrams) • Apply design strategies (object-oriented, component-based, data-structure centered, function-oriented)
Construction (Implementation)	• Manage construction activities (planning, effort estimation, construction measurement) • Apply coding standards and guidelines • Select and use programming languages and environments • Write source code and perform unit-level construction testing • Apply techniques for minimizing

	complexity, anticipating change, reuse, and defensive programming • Integrate components (APIs, middleware, distributed systems, runtime configuration) • Apply construction-specific quality practices (assertions, exception handling, design by contract) • Use construction tools (IDEs, GUI builders, debuggers, unit testing frameworks, profilers)
Testing	• Define test objectives and scope • Plan and manage test processes • Apply test levels (unit, integration, system, acceptance) • Apply test techniques (input domain-based, code-based, fault-based, usage-based, model-based) • Design and execute test cases • Measure test coverage and effectiveness • Evaluate program under test (defects, reliability, quality attributes) • Evaluate test process itself (effectiveness, efficiency) • Use test tools (automation frameworks, test management tools, performance testing tools)

For the purpose of this study, Feature-Driven Development (FDD) is selected as the base SDLC methodology (Figure 2.3). FDD's emphasis on incremental development makes it an ideal framework for the MBKM system development process. By adopting FDD, the development process can be broken down into manageable feature sets, ensuring continuous progress while maintaining flexibility.



Figure 2.3 Feature-Drive Development

To integrate AI assistance into this methodology, each FDD phase is further decomposed into smaller tasks that are directly aligned with the specific capabilities of LLMs identified in the preceding section. This mapping is presented in Table 2.13, which links formal software engineering activities with potential LLM contributions. These mapped capabilities form the practical scope of AI involvement in the co-creation of the development of the MBKM system.

Table 2.13 Mapping Activities to LLM Capabilities

FDD Phase	SDLC Activity	LLM Capability
Develop Overall Model	Define requirements (functional/non-functional)	Generate initial requirement drafts, refine based on stakeholder prompts
	Identify requirements sources & goals	Summarize stakeholder input, infer hidden assumptions
	Apply elicitation techniques (scenarios, user stories)	Generate user stories, use cases, acceptance criteria
	Perform conceptual modeling	Draft UML diagrams (use case, class, sequence, activity)
	Negotiate & resolve conflicts	Suggest trade-offs, provide alternative requirement phrasings
	Produce requirement specifications (SRS)	Draft structured SRS documents
	Validate requirements (reviews, prototyping)	Generate mockups, simulate workflows, check for ambiguity
Build Feature List	Classify and prioritize requirements/features	Break down into feature lists, suggest prioritization
Plan by Feature	Estimate effort, assign features, define acceptance	Suggest task breakdown, effort estimation heuristics
Design by Feature	Apply design fundamentals (modularity, cohesion)	Recommend modular decomposition and refactoring
	Address design issues (security, persistence, error handling)	Suggest best practices, propose architectural safeguards
	Define software architecture	Generate architectural views, propose layered structures
	Perform detailed design (data structures, interfaces)	Generate class/interface stubs and pseudocode
	Conduct UI design	Suggest wireframes, text-based layout drafts
	Document design (notations)	Create UML diagrams, ER diagrams, sequence diagrams
Build by Feature	Manage construction activities	Suggest task breakdown, sequencing, developer assignments
	Apply coding standards & write source code	Generate code snippets following style guidelines

FDD Phase	SDLC Activity	LLM Capability
	Refactor, reuse, defensive programming	Improve existing code, recommend reuse patterns
	Integrate components (APIs, middleware)	Suggest API integration code snippets
	Construction testing (unit-level)	Generate unit test templates
	Define test objectives & scope	Suggest coverage goals and test strategy
	Design and execute test cases	Generate test cases from SRS or code logic
	Measure test coverage & evaluate defects	Propose metrics, identify missing cases
	Evaluate program under test	Simulate expected outcomes, highlight inconsistencies

Recognizing that effective AI assistance depends on precise communication, the author also establishes a framework for prompting, as prompts play a pivotal role in guiding each interaction with the LLM. The formulation of prompting techniques in this thesis is informed by existing literature and prior studies on prompting strategies (White et al. 2023; White et al. 2024) and is illustrated in Figure 2.4.



Figure 2.4 Proposed prompting framework based on previous study

Based on these foundations, the research methodology is structured around four main components: literature review for understanding of LLM integration into SDLC, a structured prompting framework, a ChatGPT-assisted Feature-Driven Development process, and an evaluation framework comparing traditional and AI-assisted development outcomes.