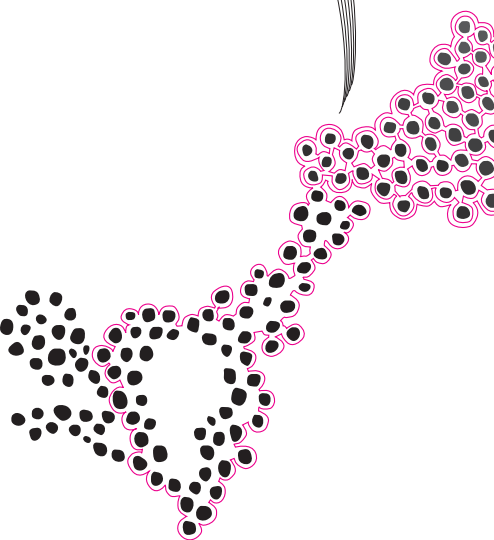




MSc Business Information Technology
Final Project

CLAIR: Generating On-Demand Low-Code Application Documentation through Knowledge Graph and LLM-based Multi-Agent System Integration

Tim Eichhorn



Supervisors:
Dr. L. Ferreira Pires
Dr. G. Sedrakyan

Enschede
February, 2025

Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

Contents

1	Introduction	1
1.1	Context	1
1.2	Problem Statement	3
1.3	Research and Scope	4
1.4	Research Questions	4
1.5	Approach	5
2	Low-Code & Documentation	8
2.1	Literature Review Approach	8
2.2	Low-Code	8
2.2.1	Definition and Components	9
2.2.2	Types of Low-Code Platforms	11
2.2.3	Benefits of Low-Code	12
2.2.4	Challenges of Low-Code	12
2.2.5	Conclusion	13
2.3	Documentation in Software Development	14
2.3.1	Costs and Benefits of Documentation	14
2.3.2	Practitioners' Perspective on Documentation	16
2.3.3	Documentation in Continuous Software Development	17
2.3.4	Documentation Quality Aspects	18
2.3.5	Towards Effective Documentation Practices	19
2.3.6	Conclusion	19
2.4	Low-Code Development Lifecycle	20
2.4.1	Design Phase	21
2.4.2	Development Phase	21
2.4.3	Testing Phase	22
2.4.4	Deployment Phase	23
2.4.5	Maintenance Phase	24
2.4.6	Conclusion	24
3	Case Study Design: Mendix	25
3.1	Mendix	25
3.2	Methodology	26
3.2.1	The Case Study Protocol	27
3.2.2	Conducting the Case Study	27
3.2.3	Analysing Case Study Evidence	28
3.2.4	Develop Conclusions, Recommendations, and Implications Based on the Evidence	29

3.3	Survey	30
3.3.1	Goals	30
3.3.2	Design	30
3.3.3	Data Collection	31
3.3.4	Data Analysis	31
4	Low-Code Developers' Perspective on Documentation	33
4.1	Respondents	33
4.2	Information Content (What)	34
4.3	Information Content (How)	37
4.4	Documentation Process & Tools	38
4.5	Design Phase Documentation	39
4.6	Development Phase Documentation	43
4.7	Maintenance Phase Documentation	46
4.8	Conclusion	48
5	Available Solutions	50
5.1	Automated Documentation Generation Techniques	50
5.1.1	Automatic Code Commenting and Summarisation	50
5.1.2	Context-Aware Documentation	51
5.2	Model-Driven Documentation Generation	51
5.3	Agile and Dynamic Documentation	51
5.4	Large Language Models and Software Documentation	52
5.5	Retrieval Augmented Generation	52
5.6	Conclusion	53
6	CLAIR: Connecting Low-Code and Artificial Intelligence for RAG	54
6.1	Requirements Specification	54
6.1.1	Purpose and Scope	54
6.1.2	Functional Requirements	55
6.1.3	Non-Functional Requirements	55
6.2	CLAIR Design	57
6.2.1	Knowledge Graph Database	58
6.2.2	LLM-based Multi-Agent System	62
6.3	Use Cases	65
6.3.1	On-Demand Documentation Generation	65
6.3.2	Annotation Generation	66
6.3.3	Debugging and Troubleshooting	66
6.3.4	New Logic Generation	66
6.3.5	Generating High-level Overview	67
7	CLAIR Validation	69
7.1	Validation Preparation	69
7.1.1	Objectives	69
7.1.2	Validation Questions	70
7.1.3	Participant Selection	70
7.1.4	Testing Plan	71
7.1.5	Data Collection and Analysis	72
7.2	Test Case 1 - Documentation Generation	72
7.3	Test Case 2 - Annotation Generation	75

7.4	Test Case 3 - Troubleshooting	77
7.5	Test Case 4 - New Logic	79
7.6	Test Case 5 - High-level Questions	81
7.7	Usability and User Experience	84
7.8	Discussion of the Results	86
	7.8.1 Requirements satisfaction	86
	7.8.2 Effect	88
	7.8.3 Trade-off	89
	7.8.4 Sensitivity	89
8	Final Remarks	90
8.1	Discussion	90
	8.1.1 Research Goal	90
	8.1.2 Research Questions	90
8.2	Main Contributions	94
	8.2.1 Theoretical Contributions	94
	8.2.2 Practical Contributions	95
	8.2.3 Broader Implications	96
8.3	Limitations	96
	8.3.1 Subjectivity and Potential Bias	96
	8.3.2 Limited Generalisability of Survey Findings	96
	8.3.3 Validation Constraints and Participants	96
	8.3.4 Reliance on Mendix	97
	8.3.5 Exclusion of Key Components	97
	8.3.6 LLM Model Choice	97
8.4	Future Work	98
	8.4.1 Usability Enhancements	98
	8.4.2 Cost and Sustainability Optimisation	98
	8.4.3 Integration with Deployment or CI/CD Pipelines	98
	8.4.4 Porting to Other Platforms and Outputs	98
	8.4.5 Specialisation of LLM-based Multi-Agent System (MAS)	98
	8.4.6 Validation Across Platforms and Companies	99
	8.4.7 Comparative Analysis of LLM Models	99
	8.4.8 Extending Knowledge in the Graph Database	99
	8.4.9 Deploying and Expanding CLAIR's Role	99
	8.4.10 Creation of a Validated Documentation Dataset	99
8.5	Conclusion	100
A	Extra results survey	110
B	Knowledge Graph Schema Details	115
C	Testing Scenarios	118
D	Goal Question Metric Process & Survey Questions	119
	D.1 Quality of Generated Documentation	119
	D.2 Usability & User Experience	120

List of Figures

1.1	The design cycle for this thesis, based on the cycle presented by Wieringa [95]	6
2.1	Features of Low-Code platforms (Copied from [12])	10
2.2	Common Architecture of Low-Code tools. (Copied from [39])	10
2.3	A meta-model for documentation development, usage-and cost process (copied from [99]).	15
2.4	A meta-model for documentation benefit (copied from [99]).	15
2.5	Possible relations between challenges and practices. The green box indicates a positive effect on the contribution to better documentation (copied from [84]).	17
2.6	A meta-model for documentation quality (copied from [99]).	18
2.7	The Low-Code Development Lifecycle.	20
3.1	Overview of the Mendix runtime architecture	26
3.2	Design of the survey used in our study (authors own adaptation of [1]).	32
4.1	Importance of documentation issues to Low-Code practitioners, according to the results of the survey.	35
4.2	Documentation process and tool issue results of the survey.	39
4.3	Design Phase Documentation usage results	41
4.4	Design Phase Documentation quality results	42
4.5	Development Phase Documentation usage results	44
4.6	Development Phase Documentation quality results	45
4.7	Maintenance Phase Documentation usage and quality results	47
5.1	Comparison between LLM, RAG, and GraphRAG (copied from [63]).	53
6.1	CLAIR architecture	57
6.2	Knowledge Graph schema for storing Mendix model elements	59
6.3	Refined data pipeline (including the extra JSON reduction step)	60
6.4	Transformation process: (a) Input JSON file and (b) Reduced JSON file.	60
6.5	The LLM-based Multi-Agent System Design in Flowise	64
6.6	CLAIR Use Cases	65
6.7	Example of On-Demand Documentation Generation with CLAIR	68
7.1	Participants distribution in terms of years of experience with Mendix.	71
7.2	Test Case 1 - Quantitative results	73
7.3	Test Case 2 - Quantitative results	75
7.4	Test Case 3 - Quantitative results	77
7.5	Test Case 4 - Quantitative results	80

7.6 Test Case 5 - Quantitative Results 82
7.7 Usability and user experience results 85

List of Tables

2.1	Design phase documentation	22
2.2	Development phase documentation	22
2.3	Test phase documentation	23
2.4	Deployment phase documentation	23
2.5	Maintenance phase documentation	24
4.1	Results question: What type of documentation would you use more if they were more correct, complete, up-to-date, findable, readable and usable? . . .	48
4.2	Information Content (What & How)	49
4.3	Documentation Processes & Tools	49
6.1	Functional requirements for CLAIR	55
6.2	Quality requirements for the generated documentation	56
6.3	Non-functional requirements for CLAIR	56
7.1	Average results of the quality requirements for the generated documentation	87
A.1	Reasons for not using Business Case	110
A.2	Reasons for not using Business Process Documentation	110
A.3	Reasons for not using Requirements Documentation	111
A.4	Reasons for not using Roadmap & Release Plan	111
A.5	Reasons for not using Architecture Design Documentation	111
A.6	Reasons for not using Mockups and UI Documentation	112
A.7	Reasons for not using Data Model Documentation	112
A.8	Reasons for not using Application Logic Documentation	112
A.9	Reasons for not using Source Model Comments	113
A.10	Reasons for not using Component Documentation	113
A.11	Reasons for not using Dependencies Documentation	113
A.12	Reasons for not using Maintenance Documentation	113
A.13	Reasons for not using Service Level Agreements	114
A.14	Reasons for not using Issue Tracking Logs	114
A.15	Reasons for not using User Manuals	114

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
BPMN	Business Process Model Notation
CSD	Continuous Software Development
CI/CD	Continuous Integration Continuous Development
CLAIR	Connecting Low-Code and Artificial Intelligence for RAG
DSM	Design Science Methodology [95]
GPT	Generative Pre-trained Transformer
GQM	Goal Question Metric
GUI	Graphical User Interface
IDE	Integrated Development Environment
IR	Information Retrieval
IT	Information Technology
LCSD	Low-Code Software Development
LDCP	Low-Code Development Platform
LLM	Large Language Model
MAS	Multi-Agent System
MBD	Model-Based Development
OD3	On-demand Developer Documentation
RAG	Retrieval-Augmented Generation
RNN	Recurrent Neural Network
RQ	Research Question
SDK	Software Development Kit
SLA	Service Level Agreement
SQL	Structured Query Language
TAM	Technology Acceptance Model
TAR	Technical Action Research [95]
UML	Unified Modeling Language

Abstract

Low-Code has revolutionised software development by enabling rapid application creation with minimal coding effort. However, as Low-Code applications scale, challenges related to documentation, maintainability, and technical debt become increasingly prevalent. Inadequate documentation impedes collaboration, maintenance, troubleshooting, and knowledge retention, particularly in agile development environments where documentation is often disregarded. This thesis introduces CLAIR (Connecting Low-Code and Artificial Intelligence for RAG), an AI-driven documentation assistant that leverages a knowledge graph and a LLM-based Multi Agent System to generate on-demand, context-aware documentation for Low-Code applications. The tool is validated using the Mendix platform, and the study employs a Design Science Methodology to design, develop, and validate the proposed solution.

A comprehensive literature review explores challenges in Low-Code documentation, emphasising issues such as fragmented knowledge, poor traceability, and the impact of missing documentation. A survey and case study at CAPE Groep, an Low-Code consultancy firm, further highlight the documentation needs of Low-Code developers and business analysts. To address these issues, CLAIR integrates knowledge graphs to structure and store Low-Code application data, enabling efficient querying and multi-hop reasoning. Additionally, a Multi-Agent LLM System dynamically generates and enhances documentation based on application data and user queries.

CLAIR automates documentation generation across various phases of the Low-Code Development Lifecycle, including the design, development, and maintenance phases. Key features include automated extraction of domain models, microflows, and dependencies, generation of high-level summaries and technical details, and support for troubleshooting. The system enhances maintainability, knowledge retention, and team collaboration by ensuring up-to-date, structured, and queryable on-demand documentation.

Validation was conducted through expert evaluations and a series of test cases using Technical Action Research, demonstrating CLAIR's ability to generate accurate, usable, and context-aware documentation. Findings indicate that automated documentation significantly reduces the time and effort needed to create high-quality documentation. This documentation leads to reduced cognitive load, technical debt, and maintenance effort, making it a valuable asset for Low-Code development teams.

This research contributes to the fields of Low-Code development, automated documentation, and AI-driven knowledge management, proposing an innovative approach that combines knowledge graphs and LLMs to enhance documentation processes. By bridging the gap between Low-Code application development and AI-driven automation, CLAIR sets a foundation for future advancements in intelligent Low-Code documentation and maintainability solutions.

Keywords: Low-Code, Automated Documentation, On-Demand Documentation, Knowledge Graphs, LLM-based Multi-Agent Systems, Mendix, Maintainability, AI-driven Documentation.

Chapter 1

Introduction

This chapter introduces the research topic and helps creating an understanding of the knowledge required for the thesis. Section 1.1 contextualises the study by exploring the rapid evolution of software development and the unique challenges posed by Low-Code platforms, particularly regarding documentation. Section 1.2 defines the problem statement. In Section 1.3, the research scope and object are introduced. Section 1.4 outlines the research questions. Finally Section 1.5, details how the Design Science Methodology [95] will guide the development and validation of the proposed solution and describes the structure of the thesis.

1.1 Context

The rapid evolution of software development practices has brought significant attention to the processes and tools needed to create efficient, scalable, maintainable, and affordable applications [24, 53, 60]. Simultaneously, Low-Code platforms have impacted the way applications are designed, developed, and tested, enabling users with minimal coding knowledge to rapidly create and deploy functional applications [47]. These platforms visually abstract a layer of complexity from traditional coding, making it accessible to a broader range of users [30]. This democratisation of application development offers substantial advantages, including faster development cycles and a reduced need for specialised coding skills. However, as Low-Code applications scale in size and functionality, they face unique challenges distinct from similar challenges in traditional environments [47, 64], particularly for managing growing complexity [41]. Low-Code applications, initially simple and easy to manage, can become unwieldy as new features are added and as they are tightly integrated with other systems. This growth can lead to increased complexity, affecting maintainability, scalability, and performance [41], which can be difficult to manage and mitigate. The ease of use that initially attracts organisations and developers to Low-Code platforms can inadvertently lead to increased technical debt [21], as the underlying complexity becomes more difficult to manage without traditional coding methods, practices, rigorous architectural oversight and supportive tools [41].

In the realm of software engineering, documentation is a foundational aspect that supports maintainability and scalability throughout the entire lifecycle of software development [1, 14, 26, 99]. It encompasses all forms of records related to a system, including technical specifications, architecture diagrams, user guides, and maintenance procedures. Effective documentation plays a key role in ensuring that both developers and stakeholders have

a clear understanding of the system’s design, functionality, and evolution [68]. It not only facilitates onboarding and collaboration among team members but also provides a shared knowledge base that supports decision-making and development continuity [26, 99]. Therefore effective high-quality documentation plays a crucial role in managing the complexity of applications [1, 26, 66, 68, 99]. As systems grow, maintaining clear and up-to-date documentation helps developers navigate the intricacies of both technical and business aspects, reducing the risk of hidden complexity [26].

Low-Code platforms, while efficient in the short term, require continuous improvement processes to handle the complexity that arises as applications scale [67]. To this end, agile methods are commonly used in software development due to their emphasis on flexibility, rapid iteration, and responsiveness to change [15, 21]. Low-Code platforms enable faster development cycles by allowing users to build applications with minimal coding, aligning naturally with agile methods’ focus on delivering small, incremental improvements [3]. Furthermore, many platforms provide built-in support and collaboration tools based on agile principles [21]. However, research has shown that developers find team work, communication, and collaboration in Low-Code challenging [6, 17]. Therefore, while agile methods facilitate speed and adaptability, they can also introduce complexity into the Low-Code development processes. Particularly because the agile manifesto itself values “*working software over comprehensive documentation*” and states that “*The most efficient and effective method of conveying information to and within a development team is face-to-face conversation*” [10]. However, research indicated that practitioners see problems with this agile principle [76]. Practitioners consider documentation valuable, yet they often find that there is insufficient documentation available in their projects [93]. The focus on minimal documentation and constant changes contributes to challenges in maintaining long-term knowledge retention, system cohesion, and scalability [15]. As a result, balancing agile method’s strengths with effective documentation management strategies is crucial for the development of sustainable Low-Code applications.

For practitioners, the trade-off between reducing development time through agile practices and ensuring adequate documentation is a key factor in managing complexity [1, 15]. Low-Code platforms, by their nature, often rely on visual development environments, which can obscure the need for detailed documentation [6]. However, as these systems scale, the lack of documentation can hinder future development and maintenance efforts [48]. This is further illustrated by the difficulty in locating information about assets within Low-Code development platforms [17]. Ensuring that adequate documentation is maintained, especially in Low-Code environments where agile methods are supported and actively promoted, is essential for managing complexity and ensuring the long-term viability of Low-Code applications [14]. However, in the context of Low-Code applications, documentation is often neglected both due to the abstract nature of development [3] and the frequent use of agile methodologies [15, 84, 93]. Maintaining useful and up-to-date documentation helps developers and citizen developers navigate the complexity that emerge as applications grow in size and functionality [26, 66, 68]. It ensures that system components, dependencies, and design decisions are transparent, making it easier to manage maintainability, reduce technical debt, and ensure the sustainability of Low-Code solutions over time [26, 66, 99].

Although there has been significant academic attention to the importance of documentation in software development [1, 82, 26, 99, 93], the field has not yet fully matured [68]. Currently most of the research is focused on identifying the benefits [14], issues [2], quality requirements [66, 80, 99], and future research directions [68] of documentation, with only select few studies aiming to develop artifacts to support creation, maintenance, and

sharing of high-quality documentation [31, 92]. In the context of Low-Code application development, to the best of our knowledge, no previous effort to develop an artifact to support the creation of high-quality documentation has been published.

1.2 Problem Statement

Low-Code development platforms enable rapid application development. However, as applications grow in size and complexity, maintaining up-to-date and high-quality documentation becomes increasingly challenging [14]. The combination of visual programming and the involvement of citizen developers often results in outdated or incomplete documentation [6], which leads to communication breakdowns [11], lesser quality designs [14], and hampers future maintenance [14, 15] and increases the cognitive load of an application [60].

Currently, documentation and other system knowledge is often of low quality with inadequate information and dispersed across various teams, systems, and tools [1]. Without a mature consolidated system that supports the creation and management of documentation, organisations face a growing risk of technical debt and accidental complexity [53]. Additionally, without accurate and up-to-date documentation, identifying and reducing technical debt and areas of accidental complexity in the system becomes difficult, leading to increased maintenance costs and decreased system agility [14]. Consolidating this key information in a single environment is crucial to ensure long-term knowledge is stored, available, and distributed, which is a crucial factor of software sustainability [89].

Traditional documentation processes are often costly, time-consuming, and prone to human error [31], making it difficult for organisations to maintain the high-quality documentation necessary for effective communication and maintainability. Furthermore, in many agile methods, which are actively encouraged in Low-Code application development, minimal documentation is part of the process, further exacerbating the issue [7, 15]. Furthermore, existing documentation is often fragmented across different systems and formats, which makes it difficult for decision-makers to have a complete view of their Low-Code application portfolios and are therefore often unaware of the lack of documented system knowledge of the applications within their portfolio. Overall, changing documentation practices without disrupting existing workflows becomes a crucial challenge [7].

Another significant challenge in Low-Code application development, and software development in general, is the unpredictability of documentation needs [68]. It is difficult to foresee exactly which aspects of the system require detailed documentation and which will be inherently understood by developers and stakeholders. This makes traditional documentation approaches inefficient, as they often lead to either over-documentation or critical gaps [1]. An adaptive approach is particularly beneficial in environments with mixed development expertise [68], such as Low-Code involving citizen developers, where the need for explanatory content varies significantly. To this end, automated on-demand documentation is expected to ensure that the right information is available when needed, thus reducing cognitive load and supporting both effective collaboration and maintainability without overburdening developers with exhaustive, static documentation [68].

1.3 Research and Scope

Based on our problem statement, it follows that improving the quality, availability, and comprehensiveness of documentation in Low-Code environments is crucial to addressing challenges such as technical debt, maintainability, knowledge fragmentation, and overall system sustainability. Furthermore, understanding how to improve documentation to better share knowledge could provide substantial benefits for Low-Code development teams, ensuring that accurate, up-to-date information is readily available for both technical and business users, which enhances collaboration. To this end, automated on-demand documentation is expected to significantly help stakeholders develop in dynamic visual development environments while minimising cognitive complexity.

The main objective of this thesis therefore is to improve the documentation process and quality for Low-Code applications. The enhanced documentation should provide valuable insights into application components, processes, and dependencies to improve system understandability, maintainability, sustainability, and overall knowledge retention.

The scope of this thesis is limited to Low-Code applications. Our focus lies in leveraging the technical details of these applications and their interrelationships to generate documentation. To illustrate and validate these objectives, we will conduct a case study using Mendix¹, a leading Low-Code development platform [90], which is representative of many major Low-Code platforms [25].

1.4 Research Questions

Our main research question is the following:

"How can an automated documentation assistant enhance the quality and process of documentation for Low-Code applications?"

Aligning with the methodological approach of Wieringa our sub-research questions are designed as knowledge questions, which require gathering and analysing empirical data about the world [95]. We decompose the main question into two principal sub-questions. Each sub-question is further refined into more specific queries that systematically build the knowledge required to propose and validate a solution. This stepwise approach ensures that we first capture and describe the complexity of the problem (descriptive knowledge questions [95]) and then explore how to address it (explanatory and design-oriented knowledge questions [95]). Below are the sub-questions and a brief rationale for how each contributes to answering the main research question.

This research question has been decomposed in the following sub-questions:

1. What are the key documentation challenges in Low-Code application development?
 - (a) What are the main challenges in documentation in Software development in general?
 - (b) How do the unique characteristics of Low-Code impact documentation issues?
 - (c) What are the implications of fragmented, inconsistent, in-adequate documentation for Low-Code applications?

¹<https://www.mendix.com/>

- (d) What are the documentation needs and challenges during each phase of the Low-Code development lifecycle?
2. To what extent can automated documentation be applied in Low-Code development environments, specifically in the context of complex Mendix applications?
 - (a) What are the specific documentation needs for different stakeholders in Low-Code?
 - (b) What are currently available solutions for automated documentation?
 - (c) What are the design requirements for our automated documentation assistant?

The first sub-question group is primarily descriptive. It gathers empirical insights into the broad and specific challenges that Low-Code practitioners face in documenting their applications. Understanding these issues lays the foundation for defining what “enhanced” documentation quality and process should look like. Thus, these questions identify the problem landscape that any automated solution must tackle. Building upon the insights from the first group of sub-questions, sub-question 2 is more explanatory and design-oriented. It examines the distinct documentation requirements of various Low-Code stakeholders, existing automated documentation tools or techniques, and how these insights translate into specific design requirements for an automated documentation assistant. By investigating these areas, sub-question 2 provides the necessary knowledge to propose a concrete solution, thereby relating back to the main research question on how an assistant can enhance documentation quality and processes in Low-Code applications. Overall, the knowledge gained from sub-question 1 contextualises the specific shortcomings and complexities that must be addressed, while sub-question 2 demonstrates how automated documentation can meet those challenges. Together, they culminate in a comprehensive response to our main research question, informing the design, implementation, and validation of an automated documentation assistant that improves Low-Code documentation practices.

1.5 Approach

To answer our main research question we developed an automated documentation assistant for Low-Code applications. To this end we employed the Design Science Methodology (DSM) by Wieringa [95]. This methodology allows for the systematic design and investigation of artifacts within their application contexts. In line with the DSM, the design problem at the core of this thesis is as follows:

Improve	<i>the maintainability, understandability, sustainability, development efficiency, and quality of complex Low-Code applications</i>
by designing	<i>a documentation assistant that knows the structure of a Low-Code application</i>
that	<i>supports the automated generation of on-demand, precise, and up-to-date documentation, offering context-aware insights from both technical and business perspectives</i>
in order to	<i>help Low-Code platform developers, citizen developers, and business analysts efficiently reduce technical debt, troubleshoot, develop, scale, and maintain their Low-Code applications while fostering knowledge retention and collaboration.</i>

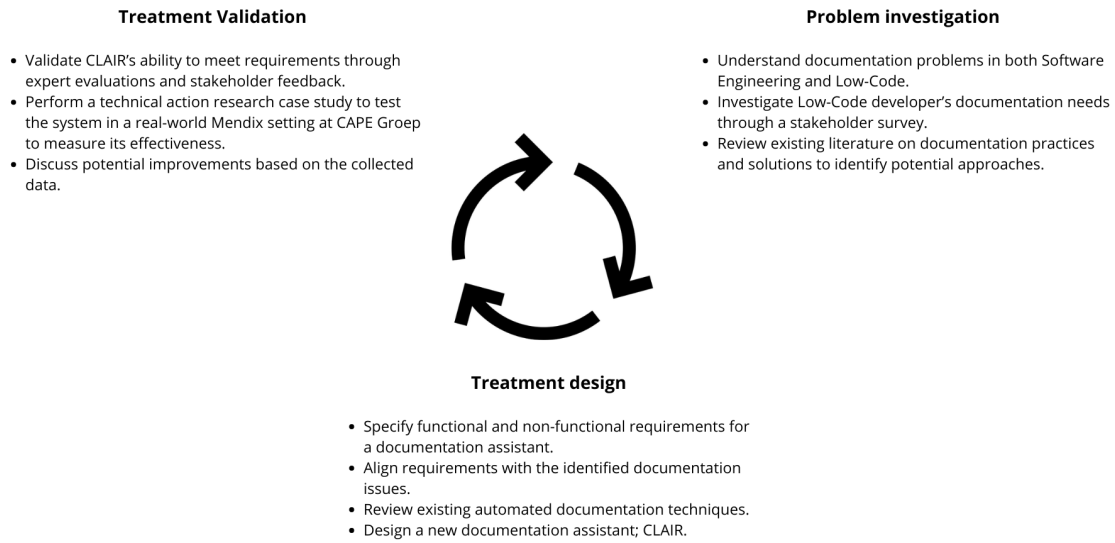


FIGURE 1.1: The design cycle for this thesis, based on the cycle presented by Wieringa [95]

We aim to improve the maintainability, understandability, sustainability, development efficiency, and quality of complex Low-Code applications by enhancing the documentation process and its quality. Research has consistently demonstrated a direct relationship between documentation quality and these aspects, this is further discussed in Chapter 2. Therefore, our validation focuses on assessing the quality of the resulting documentation and documentation process, rather than directly measuring aspects like maintainability and efficiency.

Mapping of DSM phases to the structure of the thesis

Problem Investigation (Chapters 2, 3 and 4)

- Focused on identifying documentation challenges in Low-Code environments, particularly Mendix.
- Combined insights from:
 - Literature Review: Synthesised knowledge on documentation challenges in Low-Code and traditional software development.
 - Stakeholder Survey: Conducted within CAPE Groep to understand current practices, challenges, and requirements for documentation in Mendix.
 - Aimed to establish a foundation for designing an artifact addressing the documentation issues faced in Low-Code.

Treatment Design (Chapters 5 and 6)

- A review of currently available solutions, assessing their ability to satisfy the challenges and needs determined in the problem investigation.
- Specify both the functional and non-functional requirements for the automated documentation assistant.
- Design and develop an automated documentation assistant for Low-Code applications, specifically Mendix.
- Key design objectives:
 - Generate accurate, on-demand, and context-aware documentation for various users.
 - Enhance maintainability, development efficiency, and support collaboration.
- Iterative design process integrated feedback from problem investigation to ensure alignment with stakeholder needs.

Treatment Validation (Chapter 7)

- Conducted expert evaluations and Technical Action Research (TAR) [95] to assess the functionality, usability, and impact of the artifact.
- Validation goals:
 - Ensure the generated documentation meets quality requirements.
 - Identify areas of improvement related to usability and user experience.
- Evaluated feedback to define potential improvements to the design and align it more with user needs.

Treatment Implementation

- Implementation is outside the scope of this thesis, as per Wieringa's Design Science Methodology [95].
- Future work is discussed in Section 8.4

Chapter 2

Low-Code & Documentation

This chapter provides the literature review we performed on Low-Code development platforms, the challenges of maintaining documentation, and the potential solutions to these issues according to literature. The review synthesises academic research on documentation in software development and documentation practices within Low-Code environments and identifies the potential of automation in improving the documentation process and quality for Low-Code applications.

2.1 Literature Review Approach

This research adopted an exploratory literature review, to investigate questions related to Low-Code platforms and documentation, as it is suitable for exploring broad and complex questions where little prior research exists [18]. Our exploratory literature review examined foundational literature on Low-Code platforms and the role of documentation in both software development and Low-Code. This review establishes a broad understanding of Low-Code architecture and common challenges faced by organisations in maintaining accurate, up-to-date documentation. It highlights key studies on documentation challenges and the impacts on technical debt and cognitive complexity.

In contrast to a systematic literature review, this approach does not aim to provide a conclusive answer or solution to a research question, it rather explores the available information on the topic [18]. Since this type of review does not require a specific structure, the research was guided by the following steps: formulation of research questions, database searches (using the concepts outlined in the research questions), identification of relevant themes and concepts, and analysis of the results. The repositories searched consisted of Scopus, IEEE Xplore, ISI Web of Science and ACM Digital Library. Furthermore, for relevant papers their references and citations were also investigated to further add to the corpus of literature examined in this thesis. A paper was deemed relevant if it provided insights that were directly applicable to the challenges of documentation.

2.2 Low-Code

Low-Code Development Platforms (LDCPs) are environments designed to enhance software development productivity by combining minimal source code with interactive graphical interfaces, enabling rapid application creation [12]. They aim to bridge the gap between business requirements and technical implementation, empowering non-programmers, or

"citizen developers" ¹, to create and adapt applications with minimal coding [5, 12, 25, 71]. The goal is to improve business IT alignment and enable organisations to adapt quickly to changing requirements [3, 12]. Additionally, LDCPs promise several benefits over traditional coding practices, including increased productivity, reduced development costs, and simplified application maintenance [12, 64]. However, despite these benefits, LDCPs also introduce unique challenges related to scalability [5, 69], maintainability [6, 64], and technical debt [21, 41], especially as applications grow in complexity. While Low-Code is intended to reduce the development burden, it often requires traditional coding for complex functionality, which can lead to maintenance difficulties and increased technical debt.

2.2.1 Definition and Components

Despite the wide spread adoption, the academic literature still lacks a clear, common understanding of Low-Code [13, 20, 25, 39, 47].

Tisi et al. [85] define LCDPs as cloud-based software development tools offered via a Platform-as-a-Service model that enable users to create fully functional applications using interactive graphical user interfaces, visual diagrams, and declarative languages. Other publications assign the term "No-Code Development Platform" to this concept [51], however [25] consider this term to be a marketing positioning statement and should be considered as a component of LCDPs.

Al Alamin et al. [3] introduce the notion of "Low-Code Software Development" (LCSD) as an emerging paradigm that merges minimal source code with a graphic interface to promote rapid application development. This correlates with [86], who state that Low-Code platforms simplify and accelerate application development by offering visual interfaces and drag-and-drop functionality, replacing the need for complex programming languages and traditional software development environments. They employ automatic code generation and transformation techniques, using the Model-Based Development (MBD) approach to automatically create code from high-level abstract models.

LDCPs integrate various classical development components into a single environment, differing from traditional software development infrastructures by incorporating most tools and components necessary for specific software development projects [12], streamlining the development process and reducing the complexity of software projects [39]. Additionally, Low-Code platforms often provide pre-built templates to simplify the development process, reducing accidental complexity, and offering cloud-based environments for application life-cycle management [20].

Overall, despite the lack of a commonly agreed concept of Low-Code, we can conclude that based on the combination of the provided conceptualisations, LCDPs aim at streamlining and simplifying the software development process, making it more accessible to various types of developers by incorporating visual tools, automatic code generation, and integrated environments.

These LCDPs consist of various components and features, some more common than others across various platforms [12]. Figure 2.1 shows an overview of features found in ten representative Low-Code platforms analysed by [12]. Furthermore, Kirchhof et al. [39] find that when examining Low-Code tools, it is evident that many share a similar structure, which is depicted in Figure 2.2. These tools typically require developers to model

¹Citizen developers are business users with little to no coding experience who build applications. <https://www.mendix.com/glossary/citizen-developer/>

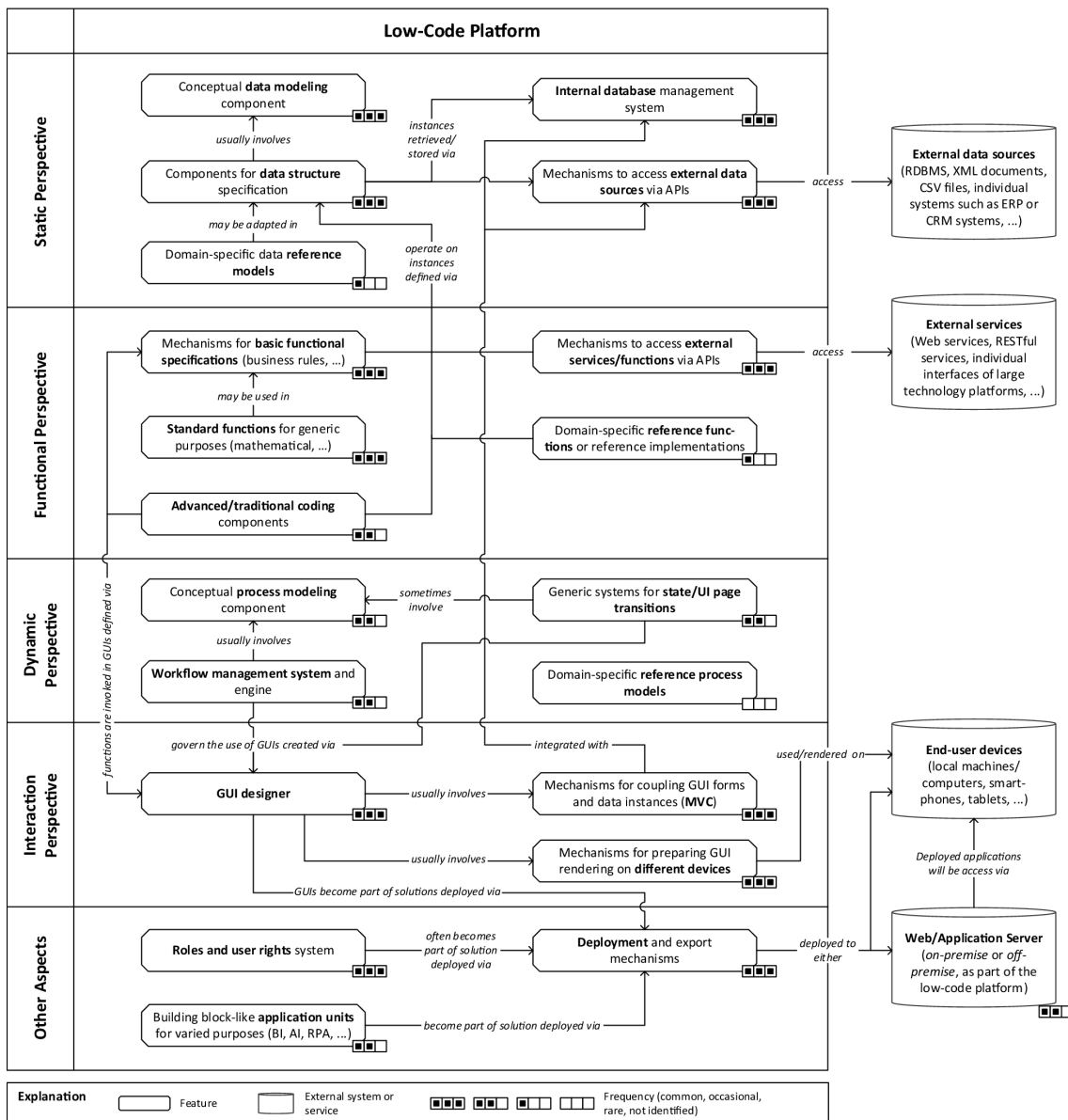


FIGURE 2.1: Features of Low-Code platforms (Copied from [12])

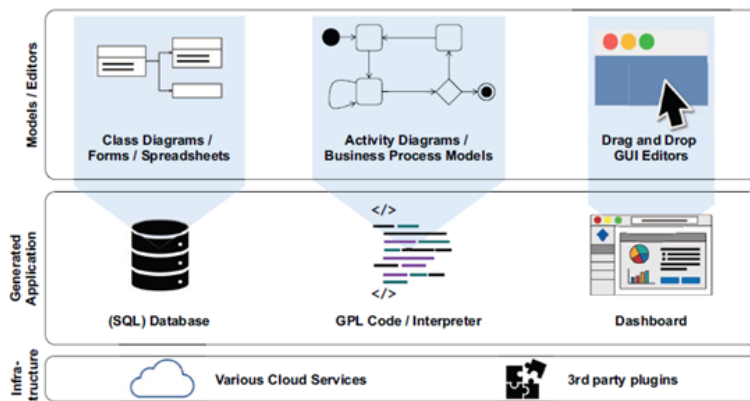


FIGURE 2.2: Common Architecture of Low-Code tools. (Copied from [39])

three aspects of an application: data structures, behaviour, and GUI. Data is modelled using class diagrams, forms, or spreadsheets. Behaviour is usually modelled with activity diagrams or business process models, while GUIs are designed using visual drag-and-drop editors. However, there are deviations, such as UMLP’s use of a textual GUI description language. Technically, three types of artifacts are common: SQL databases generated from data models, application behaviour managed through code or model interpreters, and front-end definitions dependent on the target platform, which can often be extended or replaced with handwritten code.

2.2.2 Types of Low-Code Platforms

There has been extensive research into comparing various Low-Code Platforms [25, 39, 71, 86]. Despite this research, limited attention was given to grouping or classifying these different platforms. To this end, Frank et al. [25] propose a classification system that distinguishes between Basic Data Management Platforms, Workflow Management Systems, Extended GUI-Centric and Data-Centric IDEs, and Multi-Use Platforms.

Basic Data Management Platforms: these platforms primarily focus on handling and organising data. They provide users with tools to create, manage, and manipulate databases through a user-friendly interface, without needing extensive coding knowledge. An example is Quickbase², which offers functionalities like relational data modelling and pre-built templates to support business information management efficiently.

Workflow Management Systems: this category includes platforms designed to automate and streamline business processes. They offer capabilities to design, execute, and monitor workflows, making it easier for organisations to manage complex processes. Bonita Studio³ is a notable example, providing tools for process modelling, task automation, and performance tracking. These platforms often feature drag-and-drop interfaces and support for integrating with various enterprise systems.

Extended GUI-Centric and Data-Centric IDEs: platforms in this category extend traditional Integrated Development Environments (IDEs) by incorporating graphical user interfaces (GUIs) and data-centric capabilities. They aim to simplify application development by allowing users to visually design interfaces and data models. Mendix⁴ and OutSystems⁵ are prominent examples, offering comprehensive tools for creating web and mobile applications with a strong emphasis on GUI and data integration. Furthermore, these platforms support extensive customisation and integration with external services through APIs.

Multi-Use Platforms: these versatile platforms are designed for a broad range of business applications, encompassing features from the other categories. They provide a unified environment for application configuration, integration, and development. Platforms like Microsoft PowerApps⁶ fall into this category, enabling users to build a variety of applications that can integrate seamlessly with other business systems and services. They support both citizen developers and professional developers by offering a mix of no-code and high-code tools, enhancing the flexibility and scalability of development processes.

²<https://www.quickbase.com/>

³<https://www.bonitasoft.com/>

⁴<https://www.mendix.com/>

⁵<https://www.outsystems.com/>

⁶<https://www.microsoft.com/nl-nl/power-platform/products/power-apps>

2.2.3 Benefits of Low-Code

According to literature, [3, 12, 25, 39, 64], Low-Code platforms promise various benefits such as increased productivity, reduced development costs, and improved ability of organisations to adapt to rapidly changing requirements.

- **Reduced Development Time:** Low-Code platforms significantly shorten development cycles [47, 86] by automating code generation and often providing reusable components [64], resulting in improved return on investments in software development projects [5, 48]. Furthermore, for many Low-Code platforms the learning curve is quite low and one can start modelling very fast [47]. Therefore they allow rapid prototyping and quick response to market demands [3].
- **Lower Deployment Effort:** these platforms simplify deployment processes [3], reducing the need for extensive configuration and setup [64]. Some LCDPs even provide ‘one-step deployment’ [6]. Furthermore, many LCDP inherently support the development of cross-platform web and mobile applications [64], making it easier to deploy apps for mobile devices [47, 86].
- **Easier Maintenance:** the visual nature of Low-Code development makes it easier to update and maintain applications [5, 64], as changes can be made directly through the platform’s interface [3].
- **Democratisation of software development:** LCDPs can potentially empower citizen developers to create applications without deep technical knowledge [3, 5, 6, 12, 13, 20, 25, 39, 64, 69, 71]. This democratisation of software creation enables quicker turnaround times for application development, fosters innovation by enabling domain experts to design solutions tailored to specific business needs [3, 69], and reduces the burden on IT departments [64]. Additionally, by leveraging citizen developers, organisations can potentially alleviate the shortage of professional developers in the IT sector [6, 69], ensuring that projects can still progress despite the demand for software growing faster than the number of technical staff [5, 47].

2.2.4 Challenges of Low-Code

Despite their advantages, Low-Code platforms face several challenges, as Low-Code applications grow in size and complexity, maintaining high performance and low maintainability becomes difficult [3, 5, 38, 41, 64]

- **Low-Code often turns into high-code:** Low-Code may lack the flexibility needed for highly sophisticated applications, often requiring traditional coding to achieve specific functionalities [40, 41]. Because of this, development processes increase in difficulty, resulting in the need for more experienced programmers [3, 25, 40, 41].
- **Difficulty of maintenance and debugging:** Low-Code promises to reduce maintenance effort and costs, however this only seems to be true for simple applications, while more complex ones actually introduce debugging [69] and maintenance issues [6]. Low-Code developers face challenges regarding bugs in Low-Code platforms [47] and difficulties in using different application maintenance features provided by the platform [3]. Furthermore, LCDPs also receive updates, which further complicates the maintenance of Low-Code applications as developers also find it difficult to update versions of LCDPs [3]. Low-Code applications that were created must be rolled out, maintained, and further developed. The maintainability and scalability of these

applications is a quality criterion that is hardly considered by non-developers. In retrospect, this can result in great effort and costs in IT to rebuild an application.

- **Technical Debt:** over time, Low-Code applications can accumulate technical debt, making them harder to maintain and evolve [3]. Due to the nature of Low-Code and their development platforms, best practices from high-code applications are difficult to directly apply [41]. Furthermore, there is a risk of creating additional technical debt since citizen developers may produce poorly structured applications without proper oversight [39].
- **Poor customisation:** customisation in LCDPs often leads to decreased maintainability and performance, especially for complex applications [6, 38]. When extensive custom code or functionality is needed to meet specific requirements, it can result in "spaghetti" code, making the application difficult to maintain and prone to issues with reliability and usability [6]. Such customisations can be more straightforward in traditional development environments, where well-structured libraries like JavaScript or .Net are used, avoiding the bulk and complexity that Low-Code customisation introduces [6]. The impact of customisation on complexity is amplified when development teams change during the life-time of an application [8].
- **Scalability:** there are some concerns regarding the scalability of Low-Code applications. Tisi et al. [85] highlight limitations, noting that LCDPs are primarily designed to support small applications and are not yet suitable for large-scale projects or mission-critical enterprise applications. Furthermore, [71] identify scalability challenges through a technical survey and benchmarking applications on various Low-Code platforms, in which developers mention the low scalability of these platforms as a reason for their reluctance to use them.
- **Documentation:** Low-code applications often suffer from inadequate documentation [64, 41, 33]. The lack of adequate documentation significantly affects the maintainability and quality of Low-Code projects [32, 41]. Studies have shown that citizen developers, key users of LCDPs, often struggle to find relevant explanations and resources [6], which impacts their ability to create effective solutions [40, 3]. This struggle is exacerbated by the scarcity of well-structured materials and practical examples [71]. As a result, organisations adopting LCDPs must address these documentation challenges to ensure that the benefits of Low-Code development are fully realised, without compromising the quality and maintainability of the applications [32]. Effective documentation is crucial for ensuring the quality and maintainability of Low-Code applications, yet current practices often fall short, impacting the ability of both citizen and professional developers to efficiently leverage these platforms.

2.2.5 Conclusion

Overall, Low-Code development represents a paradigm shift in software development, enabling rapid application creation with minimal coding and fostering greater business-IT alignment. While these platforms offer substantial benefits in terms of productivity, agility, and ease of use, they also present unique challenges related to maintainability, scalability, customisation, and documentation. Understanding the classification, components, and benefits of Low-Code platforms, provides a foundation for addressing the documentation and maintainability issues inherent in large Low-Code applications. Although Low-Code research is spreading, it still lags behind the state-of-the-art software design, development, and maintenance research [12].

2.3 Documentation in Software Development

The term documentation in the context of software development relates to any written, visual, verbal artifact or activity that transfers knowledge between stakeholders, related to the software product [93]. The word documentation stems from the etymological meaning for [83]:

1. teaching (Latin: docere),
2. pointing out, or
3. instructing with evidence and authority.

Documentation is a fundamental element of software development, providing essential support throughout the software lifecycle for understanding, maintaining, and evolving systems [26, 99]. Despite its critical role, software documentation is often perceived as a secondary activity and post-development burden rather than an integral part of the development process [1, 68, 84, 99]. Numerous studies have pointed the out the issues with inadequate documentation in Low-Code [64, 41, 33, 6, 3, 40, 71, 32]. However, to our knowledge, no solution has yet addressed this problem. Therefore, it is both reasonable and necessary to explore literature related to documentation in traditional software development to look for inspiration. This section explores the costs, benefits, and quality issues associated with software documentation, drawing insights from studies to unveil the issues of documentation and the potential value of solving these issues. These insights serve as the basis for a balanced approach to improving documentation practices in Low-Code.

2.3.1 Costs and Benefits of Documentation

Zhi et al. [99] provides a systematic overview of the research on software documentation cost, benefit, and quality. The study reveals that documentation incurs significant costs, accounting for up to 11% of the overall software project expenditure. Such costs are associated with both pre-maintenance (e.g., requirements, design, testing) and maintenance tasks (corrective, adaptive, perfective, and preventive [44]). These costs imply the need for effective utilisation of documentation throughout the software lifecycle to justify the investment needed to create them. Their meta-model for documentation development, usage-and cost process is presented in Figure 2.3.

According to the study, documentation benefits span several domains, including development aid, maintenance support, and management decision-making as show in Figure 2.4. High-quality documentation reduces development and maintenance effort, particularly in aiding software comprehension, architecture, and code. However, maintaining such quality requires continuous effort, and many projects face challenges like inadequate content and outdated information, impacting perceived benefit [99].

To quantify documentation benefits, metrics such as reduction in effort and perceived importance are employed. Reduction in effort refers to how documentation saves time during maintenance tasks, while perceived importance reflects developers' recognition of its value [99]. These metrics suggest that the cost of documentation should be evaluated against its potential for reducing future effort.

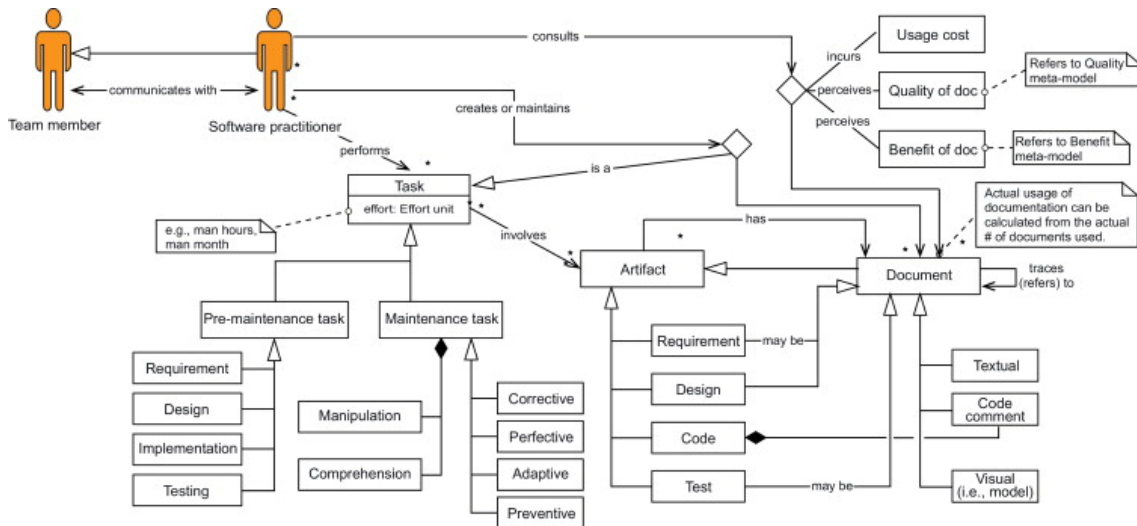


FIGURE 2.3: A meta-model for documentation development, usage-and cost process (copied from [99]).

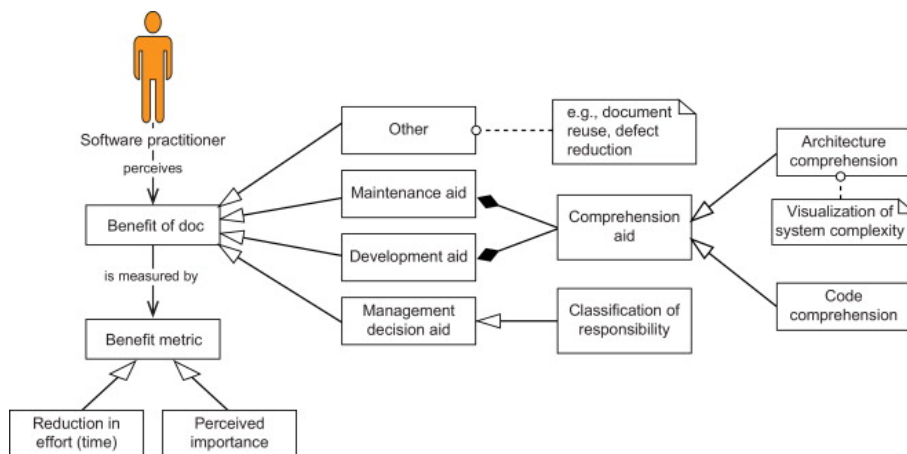


FIGURE 2.4: A meta-model for documentation benefit (copied from [99]).

2.3.2 Practitioners' Perspective on Documentation

Aghajani et al. [1] provide valuable insights into the documentation issues perceived by practitioners and the solutions they employ. One notable finding is the widespread neglect of documentation during corrective maintenance, with issues such as lack of traceability, untrustworthiness, and incompleteness among the most critical barriers to effective maintenance [1]. In a similar research, Garousi et al. [26], investigate the value, degree of usage, and usefulness of technical documentation in an industrial setting. The study reveals that technical documentation is used most frequently as an information source during development and less frequently during maintenance. In contrast, source code comments are often the preferred information source for maintenance tasks, highlighting the importance of code comments in aiding software comprehension [26]. In line with this, Aghajani et al. [1] highlighted that while code comments are particularly useful for tasks like debugging and program comprehension, they are often insufficient or missing due to time constraints or understaffing.

Next to this, Aghajani et al. [1] present evidence that practitioners readability, accuracy, consistency, and understandability deem the most important quality attributes for documentation. Yet, in practice, developers often face challenges like insufficient time, inadequate content, and inconsistency between code and documentation [1]. Garousi et al. [26] emphasise that the usefulness of documentation is influenced by its up-to-dateness, accuracy, and preciseness. Their results show that these attributes have the highest impact on how effectively documentation supports development and maintenance tasks. The study also highlights that there is no significant difference in the usage of different types of documentation during development versus maintenance. This finding suggests that ensuring consistent quality across all documentation types is crucial for maximising their utility throughout the software lifecycle [26].

Furthermore, Garousi et al. [26] underscore the challenges practitioners face in determining the appropriate amount and depth of documentation. Developing too much or too little documentation can lead to inefficiencies, with overly extensive documents reducing cost-effectiveness. This finding supports the need for balancing documentation efforts with the expected benefits, ensuring that documentation remains a valuable asset rather than an overhead. Tools that automate code comment generation or documentation could alleviate some of these issues, enhancing the quality of documentation in practical settings [26, 1, 68].

Similarly Robillard et al. [68] advocate for a shift towards automated, on-demand developer documentation, which they refer to as OD3, to address the traditional inefficiencies of manual documentation practices. Although developer documentation, such as source code comments, API references, and design documents, plays a vital role in software maintenance, its high cost and lack of immediate return on investment often relegates it to a lower priority [68]. OD3 suggests to mitigate this issue by leveraging automated tools to generate documentation based on developer requests, enhancing its relevance and reducing the manual burden. To this end, the authors identify three primary challenges to realising OD3: information inference, forming document requests, and document generation. Advances in natural language processing and feature location techniques can potentially bridge the gap between high-level features and source code, enhancing the semantic understanding of how documentation elements interrelate [68]. Moreover, presenting complex information coherently is crucial for improving the usability of documentation, especially when responding to intricate developer questions.

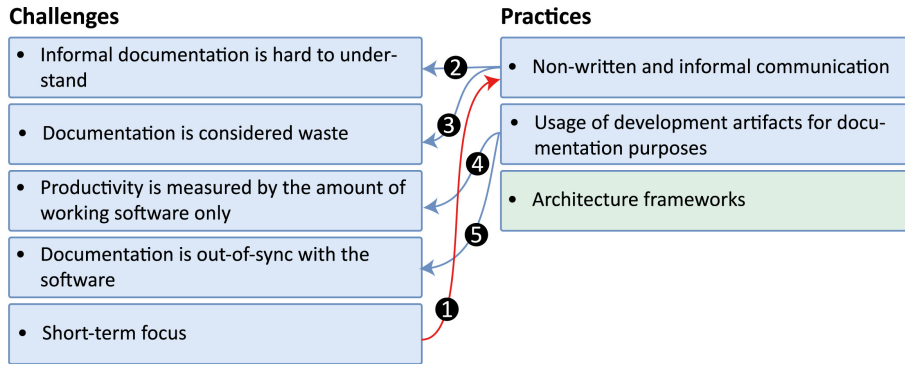


FIGURE 2.5: Possible relations between challenges and practices. The green box indicates a positive effect on the contribution to better documentation (copied from [84]).

2.3.3 Documentation in Continuous Software Development

Theunissen et al. [84] explores the challenges, practices, and tools related to documentation in Continuous Software Development (CSD), which encompasses Agile, Lean, and DevOps methodologies. This is due to the principles and values of these various CSD approaches; documentation is considered as waste when it does not contribute to the end product (Lean). Working software is valued over comprehensive documentation (agile value [10]). Face-to-face communication is more effective in conveying information (agile principle [10]). Documentation as infrastructure-as-code (DevOps). However, the study reveals that poor documentation is a persistent issue in CSD, hindering knowledge transfer, complicating maintenance, and introducing a steep learning curve for new team members [84].

The challenges identified include informal documentation that is difficult to understand, documentation being considered waste, and documentation often being out-of-sync with the software due to the emphasis on short-term focus and working software over comprehensive documentation. The practices in CSD often involve non-written and informal communication, and using development artifacts as a form of documentation. However, the lack of formal, comprehensive documentation leads to difficulties in knowledge retention, especially when team members change or when new developers join a project. Theunissen et al. [84] also note that documentation is often scattered across multiple tools with no central repository, making it challenging for stakeholders to find a single source of truth. The possible relations between challenges and practices presented in the paper are shown in Figure 2.5.

To mitigate these challenges, the authors [84] recommend adopting practices such as using executable documentation, leveraging modern tools to retrieve and transform information into documentation, and practising minimal upfront documentation combined with detailed design for knowledge transfer afterward. The study also highlights the importance of architecture frameworks in supporting effective documentation in CSD environments. Furthermore, it underscores the need for knowledge-preserving documentation practices that can stand alone and provide clarity and continuity over time.

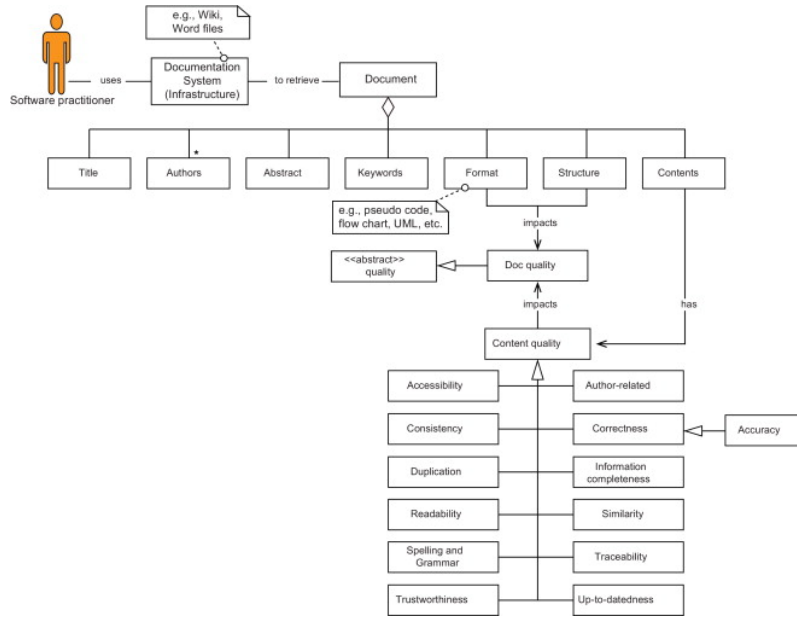


FIGURE 2.6: A meta-model for documentation quality (copied from [99]).

2.3.4 Documentation Quality Aspects

The study by Zhi et al. [99] presents a meta-model, shown in Figure 2.6 that incorporates several quality aspects for documentation, including completeness, accuracy, readability, accessibility, and traceability. A well-structured document is easier to use, and attributes like content quality, document format, and structure directly impact perceived quality. This meta-model provides a structured approach to evaluating the quality of documentation, which is crucial for improving its usability and effectiveness.

Garousi et al. [26] further emphasise that document quality attributes such as up-to-dateness, relevance, accuracy, and completeness are critical for ensuring that documentation meets the needs of developers. The gap between the perceived and expected quality of documentation, particularly regarding its up-to-dateness, was identified as a major concern. Addressing this gap is essential for improving the overall utility of technical documentation.

Robillard et al. [68] extend the discussion on documentation quality by emphasising the need for integrating traceability, feature location, and information presentation. By ensuring that documentation is dynamically linked to code and that changes are reflected seamlessly, the challenges of outdated information and inconsistency can be mitigated. Presenting information in a user-friendly manner, possibly through hierarchical, on-demand documents, can also address the complexity often associated with traditional documentation.

Aghanjani et al. [1] emphasise the importance of attributes like clarity and accessibility. Documentation that is easy to find and understand significantly improves productivity, especially during maintenance. Practitioners also expressed the need for better tools that integrate into development environments to enhance the automation of documentation tasks, which is essential given the time pressures commonly faced in software projects.

Finally, the study by Tang and Nadi [80] introduces a systematic approach for assessing documentation quality using an automatic evaluation tool for Java, JavaScript, and Python

libraries. The study identifies several quality attributes such as completeness, readability, ease of use, and up-to-dateness in line with the other papers. Tang and Nadi emphasise that documentation quality is crucial for effective software usage, with metrics like text readability and code readability being particularly significant for developers. Poorly written or outdated documentation can deter developers from adopting a library, highlighting the need for actionable quality metrics and automation to maintain high standards. This aligns with the emphasis on providing well-structured, accessible, and up-to-date documentation to support developer productivity and library adoption.

2.3.5 Towards Effective Documentation Practices

Addressing the challenges associated with software documentation requires a multifaceted approach that considers both technical and organisational factors. The studies discussed in this chapter highlight several strategies to improve documentation quality and effectiveness:

- **Automated Documentation Tools:** Robillard et al. [68] propose leveraging advances in natural language processing to automate the creation of developer documentation, reducing the burden on developers and ensuring consistency.
- **Emphasis on Quality Attributes:** Quality metrics such as completeness, readability, and accuracy should be central to documentation practices [26, 80, 99]. Tools that evaluate these attributes can help maintain high-quality documentation throughout the software lifecycle [80].
- **Integration into Development Processes:** Practitioners' feedback indicates the need for integrating documentation activities into the core development workflow, rather than treating it as an afterthought [1, 26, 84]. Ensuring that documentation is (automatically) updated alongside code changes can prevent issues of inconsistency and obsolescence [68].
- **On-Demand and Modular Documentation:** The concept of on-demand documentation [68] presents an opportunity to cater to specific information needs of developers, making documentation more accessible and useful. Providing modular, context-specific documentation can significantly enhance developer productivity.
- **Knowledge-Preserving Documentation in CSD:** Theunissen et al. [84] emphasise the need for practices that ensure knowledge preservation in CSD environments, such as using architecture frameworks and leveraging tools to centralise and synchronise information.

2.3.6 Conclusion

Software documentation plays a vital role in the success of software projects, aiding in system comprehension, maintenance, and decision-making. However, it is often plagued by high costs, insufficient quality, and a lack of proper integration into the development lifecycle. The studies reviewed in this section indicate that a shift towards automated, on-demand, and quality-focused documentation can enhance its value while mitigating many of the current challenges. Emphasising the importance of documentation quality attributes, integrating documentation into development practices, and leveraging advanced tools for automation are crucial steps toward improving the overall effectiveness of software documentation.

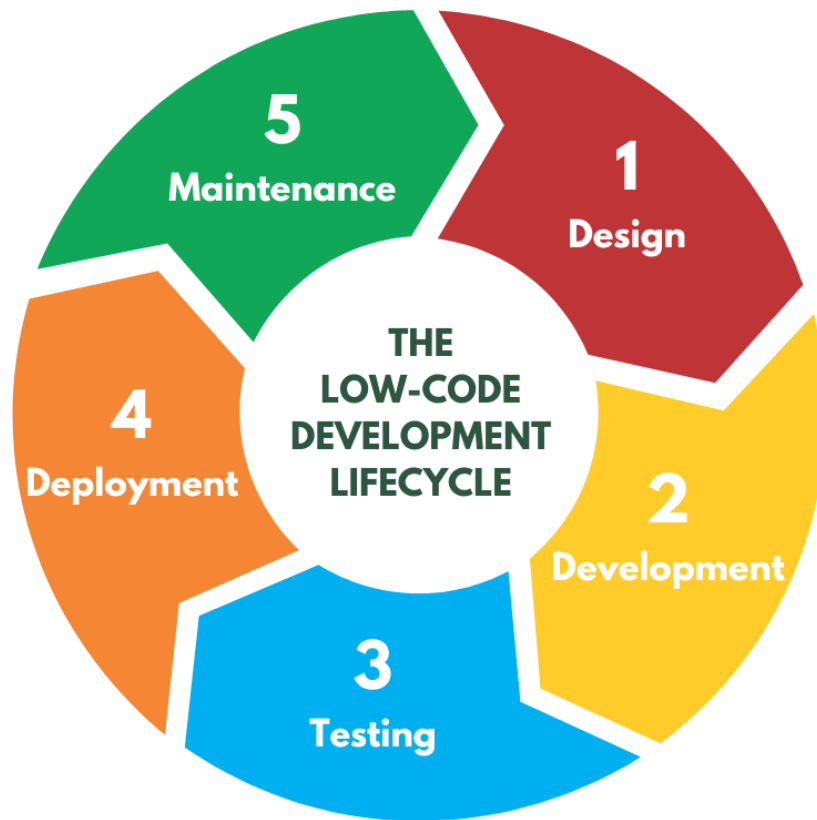


FIGURE 2.7: The Low-Code Development Lifecycle.

2.4 Low-Code Development Lifecycle

This section examines the documentation issues inherent in each phase of the Low-Code Development Lifecycle, presented in Figure 2.7. By providing a comprehensive analysis of each phase, we address how deficiencies in documentation can contribute to various issues throughout the lifecycle of Low-Code applications. The Low-Code Development Lifecycle presented is based on the lifecycle presented in [3], however we decided to split the testing and deployment phase due to the difference in information need between both phases. Furthermore, this split is in line with the software development lifecycle promoted by Mendix⁷ and other papers discussing the Low-Code Application Development Lifecycle such as [72].

A thorough discussion of the documentation requirements and associated complexities for each development phase is necessary because each phase has distinct activities and information needs [26], and the absence of documentation in earlier phases often propagates complexities to subsequent stages. By breaking down the lifecycle into specific phases and analysing their individual documentation demands, we highlight the diverse impacts of inadequate documentation on the overall system. Furthermore, this approach allows the reader to understand the scope and purpose of our research, emphasising the significance of effective documentation practices in enhancing the understandability, maintainability, and quality of Low-Code applications.

⁷<https://www.mendix.com/blog/agile-software-development-lifecycle-stages/>

Documentation Concerns

Each phase of the Low-Code Development Lifecycle presents distinct documentation requirements and challenges, resulting in potential issues such as increased maintenance effort, reduced development speed and understandability, and limited flexibility [26, 99, 2]. Different types of documents are produced at each phase, which are subsequently used in later phases to ensure continuity and cohesion across the development process [26]. For example, requirements documentation and design documents are foundational "sources of truth" for developers to build upon, and source code comments are a major source of information during maintenance and debugging. Due to the iterative nature of the Low-Code development lifecycle, documentation should be continuously updated based on feedback from subsequent phases. In this section, all of the development phases are discussed and their associated issues due to poor documentation. Additionally, for each phase, a table with commonly found document types and their descriptions is presented.

2.4.1 Design Phase

During this initial phase, teams define the application's expected features, non-functional requirements, and technical specifications. Initial designs for domain models, user interfaces, and system architecture are created. As stated the table 2.1 key documents in this phase include Business Case, Business Process Documentation, Requirements Documentation, Architecture Design Documentation, and Mock-ups/UI Documentation.

Issues: Missing or incomplete documentation of stakeholder needs, platform capabilities, or previous projects can introduce significant complexities. For example, difficulties arise when prior design decisions are not accessible, or when requirements appear incompatible with the Low-Code platform despite existing custom components. Unrealistic planning and budgetary promises can also force developers into shortcuts, scaling down time dedicated to ensuring future scalability.

Potential Impact: Inadequate documentation leads to misunderstandings between developers and stakeholders. Without thorough documentation, such as Architecture Design Documentation, architectural choices may be poorly documented, resulting in models that are difficult to modify. Insufficient Business Process and Requirements Documentation may also misalign development with business needs, leading to technical debt. Short-term, poorly documented solutions often lack the vision for sustainable application design.

2.4.2 Development Phase

During the development phase, teams use the previously created documents and create and use Data Model Documentation, Application Logic Documentation, Source Model Comments, Component Documentation, and Dependencies Documentation to build the application by implementing data models, application logic, and other components. In this phase, both citizen developers and experienced developers collaborate to create the application.

Issues: Missing or inadequate documentation can lead to ambiguity, redundancy, and inconsistencies in model and flow implementation. For instance, lack of awareness of existing microflows may result in redundant flows, adding technical debt. Citizen developers without access to best practices might create inefficient or unscalable components. Furthermore, documentation gaps hinder component reuse, with developers relying on manual processes or the Mendix marketplace to find existing components.

TABLE 2.1: Design phase documentation

Document type	Description
<i>Business Case</i> [78, 93]	Describes the justification for the project and its expected business value.
<i>Business Process Documentation</i> [78, 7]	High-level process flows and details about how business operations will integrate into the system workflows, guiding both development and testing.
<i>Requirements Documentation</i> [1, 26, 29, 78, 84, 93, 99]	Describes stakeholder needs, business requirements, and functional specifications shared or developed together with external stakeholders.
<i>Roadmap & Release Plan</i> [7, 93]	High-level planning document showing major project milestones and timelines and specifics on what features or modules will be delivered in each release.
<i>(Architecture) Design Documentation</i> [1, 26, 29, 78, 84, 99]	Provides system architecture, design models, and technical details including infrastructure design and high-level system architecture.
<i>Mock-ups and UI Documentation</i> [1, 7, 78, 93]	Visual representations of design concepts, often shared with clients for feedback. These mock-ups are not only for feedback but also for aligning developers and testers with the intended user interface design.

Potential Impact: Poor documentation leads to increased technical debt, inconsistent practices, and a fragmented application landscape. The lack of documentation also affects collaboration and onboarding, particularly for new developers. New developers face a steep learning curve due to the lack of documented logic and rationale behind design decisions, further complicating future development. The absence of documentation on reusable components results in duplicated efforts and inconsistent implementations.

TABLE 2.2: Development phase documentation

Document type	Description
<i>Data Model Documentation</i> [1, 7, 26, 93]	Describes business entities and their relationships within the system.
<i>Application Logic Documentation</i> [1, 7]	This documentation captures the logic flows, processes, and decision points within the Low-Code application. It includes details about workflows, user actions, automated tasks, and integrations with external systems.
<i>Source Model Comments</i> [1, 26, 78, 84, 93, 99, 7]	Model files containing in-line comments and explanations to help developers understand the system’s logic and structure.
<i>Component Documentation</i> [1, 7, 78, 99]	Provides details about internal reusable components or custom modules built by the team. In the case of an externally developed component or custom module, a reference to this documentation.
<i>Dependencies Documentation</i> [7, 78, 84]	Lists all dependencies, including external libraries, internal system dependencies, and third-party integrations, ensuring proper version control and compatibility throughout the lifecycle.

2.4.3 Testing Phase

During the testing phase, teams verify whether the application works according to the defined functional and non-functional requirements. Test cases are created, and the application is tested for potential issues.

Issues: Missing or outdated Test Case Documentation and Application Logic Documentation makes it difficult for testers to fully understand system behaviour and create comprehensive test scenarios. Additionally, without accurate Component Documentation and Dependencies Documentation, testers may misinterpret the context and purpose of each component, leading to gaps in test coverage where critical bugs could go unnoticed.

Potential Impact: Gaps in Test Coverage Reports and Defect Reports result in incomplete test coverage, leaving critical issues undetected. Moreover, unclear relationships between system components due to missing documentation increase testing complexity, often leading to extended timelines and potentially high-risk defects in production. Finally, due to the lower quality of testing Quality Assurance Documentation becomes more difficult or even inaccurate.

TABLE 2.3: Test phase documentation

Document type	Description
<i>Test Case Documentation</i> [1, 7, 78, 84, 93, 99]	Internal documentation of test cases, scenarios, and expected outcomes based on requirements, ensuring comprehensive coverage across the system.
<i>Automated Testing Scripts</i> [38, 84]	Scripts used within the development team for running automated tests across the system.
<i>Test Coverage Reports</i> [93]	Internal reports detailing the extent of the code covered by tests, ensuring comprehensive testing.
<i>Defect Reports</i> [27]	Captures internal documentation of bugs, root causes, and potential fixes.
<i>Quality Assurance Documentation</i> [1, 93]	Reports that validate compliance with non-functional requirements (e.g., performance, security), shared with external auditors or clients to verify system quality.

2.4.4 Deployment Phase

During deployment, the system is moved from development to production, and Deployment Guides and CI/CD Pipeline Documentation ensure smooth deployment across environments. Furthermore, the Release Notes ensure that both developers and end users are aware of the new features, improvements, known issues and potential deprecations.

Issues: Lack of detailed deployment documentation results in inconsistent configurations across environments. Missing records of deployment steps, dependencies, and environment settings can cause deployment failures that are difficult to troubleshoot. Undocumented platform-specific constraints in Low-Code environments may also lead to unexpected deployment issues.

Potential Impact: Deployment complexity is exacerbated by frequent manual interventions, inconsistent settings, and errors, leading to instability in production. Missing documentation of successful deployments complicates troubleshooting and delays release cycles, increasing operational risk.

TABLE 2.4: Deployment phase documentation

Document type	Description
<i>Deployment Guides</i> [1, 78, 93]	Step-by-step deployment instructions shared with external stakeholders or system administrators responsible for production deployment.
<i>CI/CD Pipeline Documentation</i> [1, 84]	Documentation of automated build, test, and deployment processes. Includes scripts and workflows used internally for continuous integration and deployment.
<i>Release Notes</i> [1, 93]	Documents summarising features, improvements, and known issues for users or clients.

2.4.5 Maintenance Phase

In this phase, teams manage ongoing modifications, updates, and feature additions. Key documents such as Maintenance Documentation, Versioning Logs, and Issue Tracking Logs are crucial for maintaining system performance and adapting to evolving business needs.

Issues: Missing or outdated documentation makes it challenging to understand the architecture and the decisions made during earlier phases of development. Without clear documentation of microflows, domain models, and custom components, developers may find it difficult to assess the impact of changes and the interactions between system components. Furthermore, platform updates could lead to old features or dependencies requiring adaptation or replacement. Without proper documentation of both the purpose, implementation, and requirements for the logic components of an application this becomes very complex.

Potential Impact: Lack of proper documentation results in slower development cycles during maintenance, as developers spend extra time deciphering existing components. This increases the likelihood of introducing bugs or causing unintended side effects when making changes. Additionally, lack of documentation leads to increased time spent troubleshooting when performance issues arise. The inability to easily track previous modifications, and their reasoning, makes evolving the system a challenging process, and it becomes increasingly difficult to ensure consistency and maintain scalability, ultimately leading to increased technical debt, reduced maintainability and difficult evolvability.

TABLE 2.5: Maintenance phase documentation

Document type	Description
<i>Maintenance Documentation</i> [26, 93]	Comprehensive internal documentation detailing how to operate, update, and troubleshoot the system. This includes system dependencies, known issues, and ongoing maintenance logs.
<i>Versioning Logs</i> [1, 99]	Internal tracking of system updates, changes, and bug fixes.
<i>Service-Level Agreements (SLA)</i> [93]	Agreements between the development team and external clients outlining response times, system availability, and support obligations.
<i>Issue Tracking Logs</i> [84]	Shared logs for tracking and reporting on externally reported issues or requests for system updates.
<i>User Manuals</i> [1, 7, 26, 78, 99]	Comprehensive guides provided to end-users, explaining how to use the system, its features, and troubleshooting common issues.

2.4.6 Conclusion

High-quality and available documentation is crucial across all phases of the Low-Code Development Lifecycle. Inadequate or missing documentation significantly exacerbates technical debt, complicates maintenance, slows down onboarding, and limits evolvability, ultimately jeopardising the long-term success of Low-Code applications. By ensuring comprehensive and continuously updated documentation, organisations can mitigate these issues, leading to more maintainable, evolvable, sustainable and high-quality Low-Code solutions.

Chapter 3

Case Study Design: Mendix

This chapter discusses the case study. Section 3.1 provides an overview of the Mendix Low-Code platform, which is the target Low-Code platform for our artifact. Section 3.2 discusses the case study methodology, detailing the structured processes used to gather insights and validate the research in a practical, real-world context. Section 3.3 highlights the goals and design of the stakeholder survey, which was distributed to capture Low-Code developers' perceptions of documentation and identifying key opportunities for automation in documentation practices.

3.1 Mendix

Mendix has been recognised by analysts like Gartner, IBM, and SAP as a leading Low-Code development platform. The platform is designed to achieve rapid application development [54], while focusing on the collaboration between business and IT to improve business logic [90]. The platform is well-suited for democratising application development by allowing users with minimal coding expertise to build robust applications. However, as these applications scale, managing complexity becomes crucial.

The Mendix platform comprises several key components that provide a comprehensive development environment. These components support the entire lifecycle of application development, from collaboration and requirement management in the Developer Portal, to visual development using Mendix Studio and Studio Pro, and execution via the cloud-native Mendix Runtime. Mendix Studio Pro caters to different user needs, with Studio Pro allowing "citizen developers"¹ to design interfaces using drag-and-drop features, while also providing advanced functionality for professional developers. Key elements of Mendix applications include domain models, which define entities, attributes, and relationships, microflows and nanoflows for automating business logic, and a Graphical User Interface (GUI) editor that enables intuitive, code-light interface creation. Microflows handle server-side processes, whereas nanoflows execute quick, client-side operations, both of which reduce coding effort by providing visual models inspired by Business Process Model and Notation (BPMN). An overview of the Mendix runtime architecture is presented in Figure 3.1.

¹Citizen developers are business users with little to no coding experience who build applications. <https://www.mendix.com/glossary/citizen-developer/>

²Copied from: <https://www.mendix.com/evaluation-guide/enterprise-capabilities/architecture/runtime-architecture/>

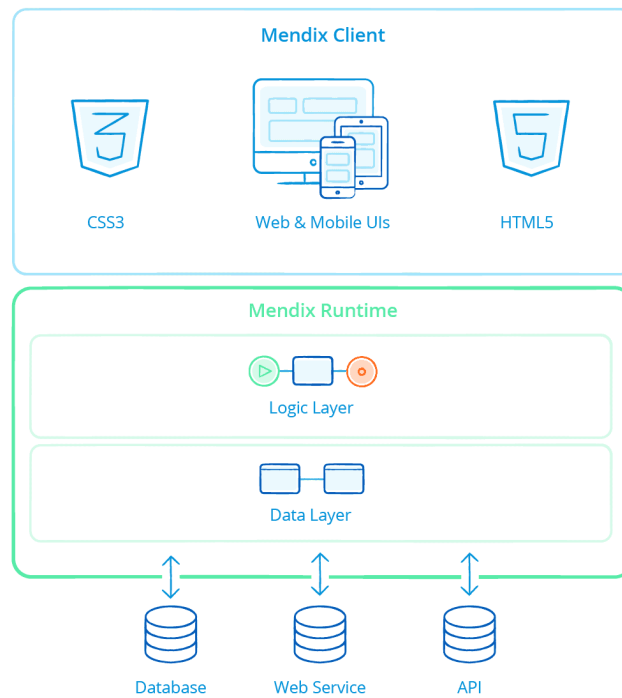


FIGURE 3.1: Overview of the Mendix runtime architecture²

3.2 Methodology

This section provides an overview of the case study methodology employed and the stages of the case study process. A case study was selected as the primary research method due to its suitability for investigating real-world phenomena in depth [97]. The case study conducted in this thesis is structured according to the four-stages applied by Tellis [81]:

1. Design the case study protocol.
 - (a) Determine the required skills
 - (b) Develop and review the protocol
2. Conducting the case study.
 - (a) Prepare for data collection
 - (b) Distribute questionnaire
 - (c) Conduct interviews
3. Analysing evidence.
 - (a) Analysis strategy
4. Develop conclusions, recommendations, and implications based on the evidence.

3.2.1 The Case Study Protocol

The first stage in designing the case study for this research involves developing a protocol to ensure reliability and focus. This protocol comprises two key components: determining the required skills, and developing and reviewing the protocol. The researcher obtained a knowledge foundation in Mendix application development and Low-Code environments by conducting an extensive literature review prior to this thesis. Furthermore, regular consultation with academic supervisors, company supervisors, and industry experts ensured the applied approaches remained grounded and aligned with research goals. Furthermore, Yin [97] emphasised the importance of creating a structured protocol to enhance the reliability of case study research. The protocol underwent reviews with both academic and company supervisors as well as other stakeholders. During these reviews the clarity and relevance of the survey questions were discussed, as well as the feasibility of the planned data collection methods. The finalised protocol ensured that the study captured all stakeholders' perspectives on the challenges and opportunities of documentation in Low-Code. The elements of the protocol are:

- **Case Study Overview:** The objective of our research is to address challenges in Low-Code application documentation by designing an automated assistant. The case study will serve as practical investigation into the problems found in literature. By validating the literary findings with practitioner's insights we ensure our project is aligned with both academic and industry needs.
- **Field Procedures:** Access to the production-level Mendix applications and relevant stakeholders at a representative company, CAPE Groep, was obtained. Data collection involves a survey and semi-structured interviews, supported by literature review and observations of real-world development practices.
- **Case Study Questions:** The research questions presented in Section 1.4.1 ("What are the key documentation challenges in Low-Code application development?" and "To what extent can automated documentation be applied in Low-Code development environments?") guide data collection and analysis.

3.2.2 Conducting the Case Study

The steps undertaken to conduct the case study focus on preparation for data collection, survey distribution, and interviews. These activities support triangulation of evidence, systematic data collection, and reliability and validity throughout the research.

The following steps were taken in the preparation phase:

1. Defining Data Sources

Although surveys are not explicitly reported as a primary source of evidence [97], their use is widely accepted in Software Engineering case studies due to their effectiveness in collecting structured and scalable data across a diverse sample of respondents [52]. Therefore the three primary data sources for this case study are:

- **Documentation:** Project reports, application models, and design artifacts within both the Mendix platform and CAPE Groep.
- **Surveys:** Used to gather broad insights into documentation challenges and automation opportunities in Low-Code application development.
- **Interviews:** Semi-structured interviews with Low-Code developers.

2. Pretesting Instruments

Both the survey and interview protocols underwent pretesting. Initial drafts were reviewed by domain experts, including academic supervisors and industry practitioners. Pretesting and iterative refinement of the instruments enhance its reliability and validity [81].

3. Ethical Considerations

Ethical guidelines were strictly adhered to, including voluntary participation, informed consent, anonymity, and data protection. Participants were provided with an overview of the research objectives and the intended use of their contributions.

Distribution of the Questionnaire

Surveys serve as a vital tool for collecting broad insights from a diverse group of stakeholders. Yin [97] recommends that questionnaires be distributed systematically, followed by reminders to ensure a high response rate. The survey was designed to gather initial insights into documentation challenges and opportunities for automation in Low-Code application development:

- **Survey Design:** Questions were structured to balance qualitative and quantitative data collection, incorporating open-ended questions for nuanced feedback or overlooked scenarios. Furthermore, partial rank-order questions [19] were included to prioritise features and challenges.
- **Survey Platform:** Qualtrics³ was chosen as the survey platform because it allows users to build and distribute surveys, collect responses, and analyse response data, all from within the same platform.
- **Target Audience:** The survey was distributed to a representative sample of stakeholders, including Low-Code developers, business analysts, and project managers at CAPE Groep. This ensured representation of both technical and business perspectives.
- **Enhancing Participation:** To maximise participation, the survey was accompanied by a clear introductory statement outlining its purpose and importance. A reminder email was sent one week after the initial distribution. Furthermore, potential respondents were reminded to fill in the survey by approaching them.

3.2.3 Analysing Case Study Evidence

The analysis focuses on examining, categorising, and synthesising evidence to address the research questions. Various techniques were applied to ensure that findings are both valid and meaningful, balancing qualitative insights with quantitative rigour [81]. Furthermore, in exploratory case studies where statistical robustness may not always be achievable, the approaches introduced by Miles and Huberman [55] are particularly valuable [81]. Miles and Huberman [55] propose alternative methods for qualitative analysis, such as creating data arrays, frequency tabulations, and visual displays to uncover patterns and relationships. The analysis in our study primarily relied on pattern-matching and explanation-building techniques:

- **Pattern-matching:** Empirical data from surveys and interviews were compared against the challenges and opportunities identified in the literature review (Chapter 2). This

³<https://www.qualtrics.com/>

helped validate the relevance of the research questions and ensured alignment with real-world complexities in Low-Code documentation practices.

- **Explanation-building:** Insights were synthesised iteratively to refine understanding of how documentation automation could address identified challenges. This process involved continuous feedback from stakeholders, ensuring that findings remained grounded in practical realities.

In addition, the study employed thematic analysis to group findings into categories aligned with the project’s objectives, some of the categories are:

- Documentation challenges in Low-Code application development.
- Opportunities for documentation automation in Low-Code application development
- Key features required for an automated documentation assistant.

Principles of Data Collection

To ensure robust data collection, multiple sources of data were utilised, including surveys, interviews, and literature analysis. This triangulation enhanced the reliability and validity of the findings by providing diverse perspectives on the research problem. A structured case study database was also created, where all collected data, such as survey responses and interview transcripts, were systematically organised into folders on the researcher’s laptop. This repository served as a central reference for analysis and reporting, ensuring accessibility and consistency throughout the study.

Furthermore, Survey data, including partial rank-order and Likert-scale responses, were analysed using descriptive statistics. For example, partial rankings of documentation challenges provided quantitative evidence of their perceived importance and frequency. Furthermore, the analysis prioritised the most impactful insights, focusing on key documentation challenges and automation opportunities in Low-Code environments.

3.2.4 Develop Conclusions, Recommendations, and Implications Based on the Evidence

In any case study, the reporting phase is critical, as it bridges the gap between the research and its practical application [81]. This phase ensures that findings are accessible to stakeholders and actionable in real-world contexts. A well-designed research may lose its impact if the results are poorly communicated [97]. Therefore, the results have been presented clearly, avoiding excessive technical jargon, to ensure that both technical and non-technical audiences within the Low-Code development ecosystem can easily grasp the study’s conclusions and implications. This ensured that developers, business analysts, and decision-makers could derive value from the findings. The conclusion, recommendations, and implications of the case study are presented in Chapter 4.

3.3 Survey

This section outlines the goals of this specific survey and details the survey’s design, implementation, and analysis processes.

3.3.1 Goals

To address the design problem for an automated documentation assistant, it is crucial to first understand the current practices, challenges, and needs [95] related to documentation in Low-Code application development. The survey was conducted to:

1. Capture stakeholder perceptions of existing documentation practices.
2. Identify pain points and inefficiencies in documentation workflows.
3. Determine desired features for an automated documentation assistant.

A survey was chosen as a means to gather a wide range of perspectives from stakeholders across technical and business domains, such as Low-Code developers, business analysts, and project managers. By combining this input with findings from the literature review, the research ensures a comprehensive understanding of the documentation landscape. Surveys allow participants to respond at their convenience and enable the collection of both qualitative and quantitative data [52]. The survey in this study included a mix of open-ended and multiple-choice to maximise engagement and ensure the reliability of the responses. Furthermore, partial-ordering questions [19] were included to identify the most critical issues faced by respondents. This approach involves asking respondents to rank a limited subset of items from a larger list, ensuring cognitive and emotional feasibility [19]. Specifically, participants in this study were asked to select their top 5 issues from three lists of 26 options, 14 options, and 10 options, respectively. By narrowing the selection to a manageable subset, the method strikes a balance between depth and usability, avoiding the challenges of ranking an extensive list while still generating meaningful data [19]. This design supports clearer prioritisation and reduces the likelihood of ties, yielding insights into the relative importance of documentation issues.

3.3.2 Design

The survey was designed by building upon prior research conducted in high-code environments, specifically referencing the study by Aghajani et al. [1]. The design of our survey is shown in Figure 3.2. In the survey by Aghajani et al. they assess practitioner’s perspective on different aspects of software documentation, such as what information content is presented, how this information is presented, the process of creating documentation, and finally the tools used during this process. Aghajani et al. [1] provide a structured approach to evaluating software documentation issues and related tasks, which served as a basis for adapting the survey to the Low-Code domain. However, given the intrinsic differences between traditional and Low-Code development paradigms, modifications were necessary to ensure relevance and applicability. Key changes included translating high-code-specific elements into Low-Code equivalents. For example, questions addressing source code comments were transformed to focus on model annotations, aligning with the visual and model-driven nature of Low-Code platforms. Additionally, questions irrelevant to Low-Code development, such as those centred on extensive high-code debugging tools, were ignored.

Furthermore, in contrast to the study done by Aghajani et al. [1] we opted for asking respondents to indicate their current usage and opinion of each of the documentation types for the design phase, development phase, and maintenance phase previously discussed in Section 2.4. For each document type, respondents indicated how often they used them using a Likert scale (1=Never, 5=Always), if they did not use them, we inquired the reason why. On the other hand, if they did use the documents we requested them to assess the quality of the documents based on 6 quality aspects (using a Likert scale from 1 to 5); correctness, completeness, up-to-dateness, usefulness, readability, and findability. We excluded the testing phase documents to keep the survey in a reasonable size and we excluded the deployment phase documents as these are least relevant in Low-Code environments where deployment is mostly handled by the platforms themselves [6]. Finally, the survey was streamlined to shorten the time required for completion, addressing concerns that lengthy surveys could reduce respondent engagement and completion rates [52]. This adjustment was particularly critical for engaging a diverse pool of participants, including consultants, team leads, and management who often face time constraints.

Our survey retained the core goals of understanding documentation challenges and priorities but presented them in a way that reflects the Low-Code context. We aimed to ensure that the insights generated would be both transferable and actionable for improving documentation practices in Low-Code platforms. The questions of the distributed survey are available in our repository [23].

3.3.3 Data Collection

The survey was created and distributed using Qualtrics. To ensure confidentiality, no personally identifiable information beyond participant roles and years of experience was collected, and responses were anonymised for analysis. Before distributing the survey, a pretesting phase was conducted to refine the questions. The survey link was distributed via email, accompanied by an introductory statement explaining the purpose and importance of the research. A reminder email was sent one week later to encourage participation.

3.3.4 Data Analysis

A systematic approach was used to analyse the raw data obtained with the survey:

- **Categorisation:** Responses were grouped into three themes:
 1. Perceived importance of documentation.
 2. Current challenges in Low-Code documentation.
 3. Need and features for an automated documentation assistant.
- **Quantitative Analysis:** Responses to multiple-choice and rank-order questions were analysed using descriptive statistics, such as frequency distributions.
- **Qualitative Analysis:** Open-ended responses were analysed to uncover recurring patterns and unique perspectives.

These insights guided the refinement of the system’s design objectives which are discussed in Chapter 6.

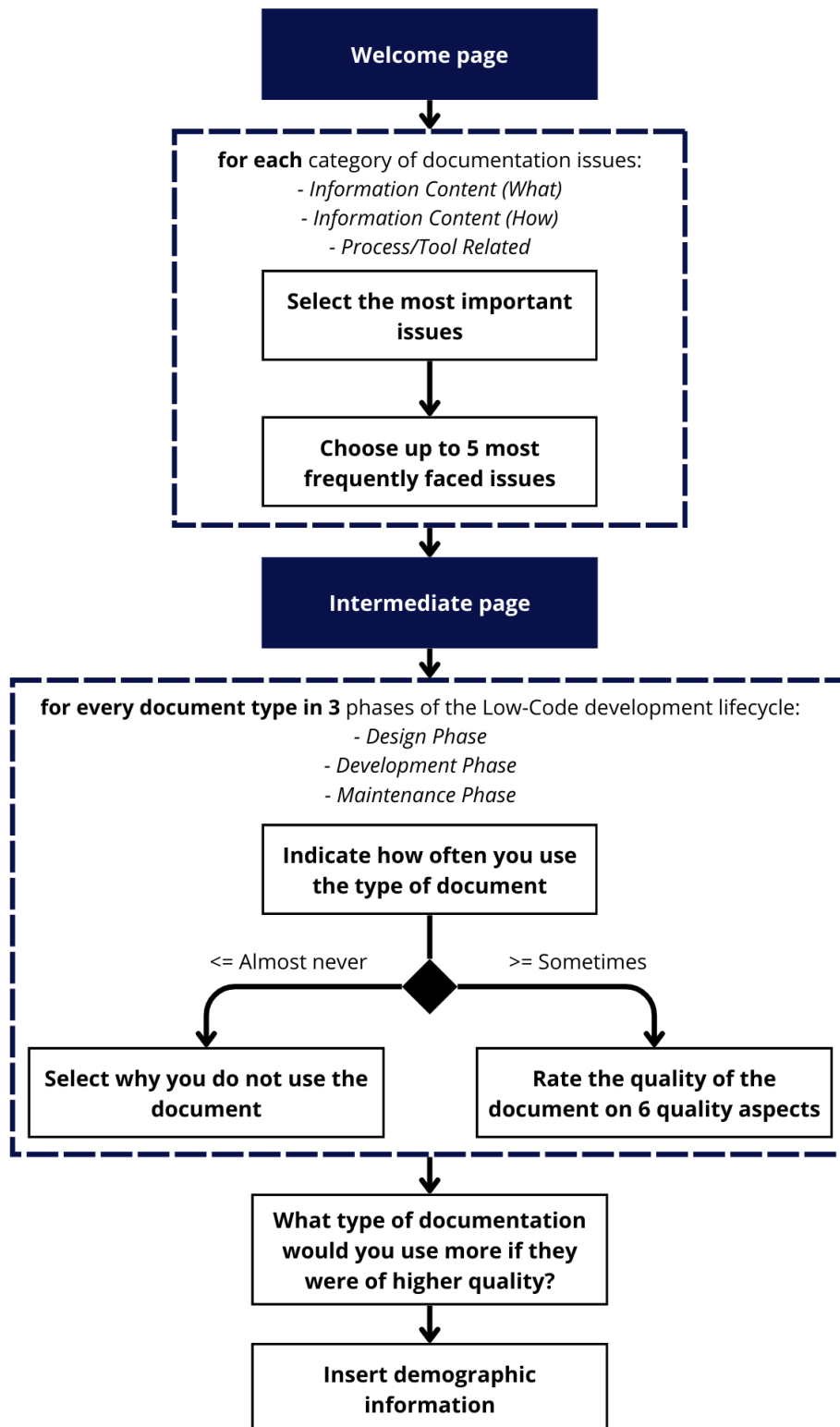


FIGURE 3.2: Design of the survey used in our study (authors own adaptation of [1]).

Chapter 4

Low-Code Developers' Perspective on Documentation

This chapter presents the analysis and findings from the survey conducted during the research on the documentation challenges and needs in Low-Code. The survey design is previously discussed in Section 3.3.2. The results provide insights into the challenges, opportunities, and stakeholder priorities related to Mendix application documentation, guiding the development of the automated documentation assistant.

4.1 Respondents

The survey received a total of 24 responses, representing a diverse set of roles within the organisation, which provided valuable insights into documentation practices and challenges. The largest group of respondents were consultants¹ (7), followed by Low-Code application developers and team leads, each contributing 5 responses. Customer support professionals provided 4 responses, while management contributed 2, and 1 response came from a developer. This range of roles ensured a comprehensive perspective, capturing input from individuals directly involved in development, those managing teams, and others supporting or overseeing processes. This diversity highlights the varied documentation needs across technical, operational, and managerial levels.

In addition to gathering insights from diverse roles, the survey also captured the respondents with different years of experience. The majority of participants (11) reported having 3-5 years of experience, followed by 6 respondents with 5-10 years of experience, and 5 with less than 3 years. Only 2 respondents had over 10 years of experience. This distribution indicates a respondent pool with a solid mix of early-career professionals and experienced practitioners, ensuring that the feedback reflects both fresh perspectives and seasoned insights into Low-Code application development and documentation processes.

¹At CAPE Groep, consultants also develop Low-Code applications.

4.2 Information Content (What)

First, we discuss the findings from the survey responses related to Low-Code documentation issues under the Information Content (What) category, highlighting key practitioner concerns regarding Correctness, Completeness, and Up-to-dateness [1]. Furthermore, we also link our findings to the results from the literature discussed in Chapter 2. Figure 4.1 summarises the responses collected for the first part of the survey.

Logic Behaviour Clarifications as a Central Concern

*Logic behaviour clarifications*² emerged as a top priority, with practitioners identifying inaccuracies, omissions, and outdated descriptions as the most critical issues across all dimensions. Incorrect logic descriptions were flagged by 92% of respondents as a major issue, while 79% noted missing logic explanations as a key gap. Furthermore, 83% of participants pointed to outdated logic behaviour documentation as a persistent challenge. These findings confirm the importance of accuracy, comprehensiveness, and synchronisation in documentation. Garousi et al. [26] emphasise similar challenges, noting that up-to-dateness and accuracy directly influence documentation’s effectiveness in supporting maintenance tasks.

Challenges with Annotations and User Documentation

Annotations spanning microflows, nanoflows, and domain models presented a recurring challenge, with inaccuracies flagged by 58% of respondents and omissions by 58% as well. *Missing developer guidelines* and *user documentation* were similarly pervasive, noted by 50% and 54% of practitioners, respectively. The significance of documentation in close proximity to the source has been repeatedly emphasised in the literature [99, 1, 26]. Additionally, Garousi et al. [26] emphasise that while technical documentation is a primary information source during development, it is used less frequently during maintenance, where developers often rely on source code comments for debugging and comprehension tasks. However, Aghajani et al. [1] highlight that such comments are often insufficient or missing due to time constraints or understaffing, and our findings suggest that this is also the case in Low-Code.

Keeping Documentation Up-to-Date in Agile Environments

Outdated documentation was consistently identified as a significant issue in Low-Code. Practitioners highlighted the lack of synchronisation between application updates and their documentation, particularly regarding *new features* (75%), *outdated references* (46%), *missing documentation for a new release* (42%) and *outdated examples* (29%). This aligns with Theunissen et al. [82]’s findings on the difficulties of maintaining documentation in Continuous Software Development, where iterative updates often render documentation obsolete. Strategies like automated synchronisation can mitigate these challenges by ensuring real-time updates [82, 68].

Integration of Technical Artifacts with Business Processes

Another significant finding is the importance of aligning technical documentation with business context. Practitioners emphasised that errors (54%) or omissions (42%) in *mapping microflows, nanoflows, and domain models to their corresponding business processes*

²the Low-Code translation of "code behaviour clarifications" [1]

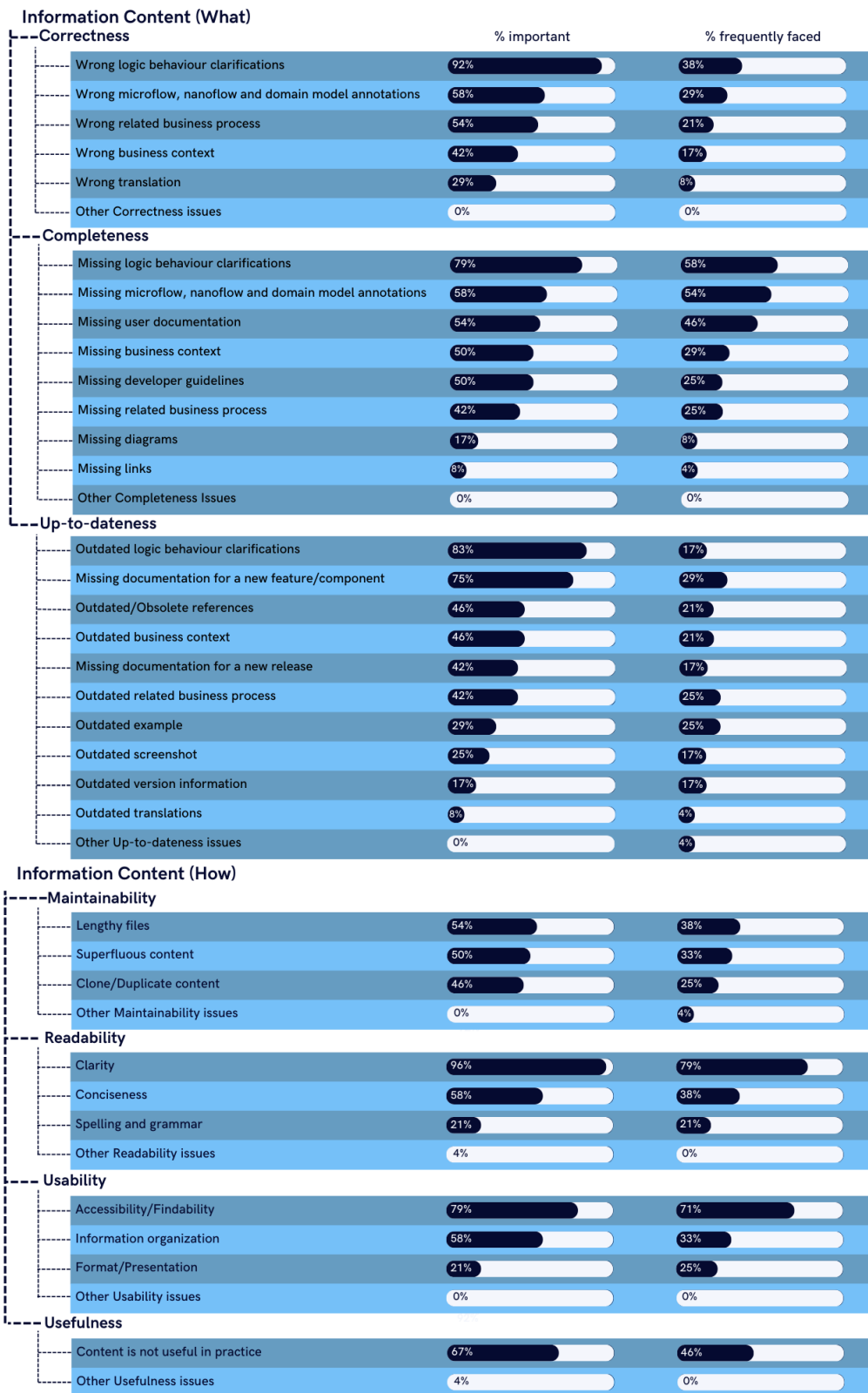


FIGURE 4.1: Importance of documentation issues to Low-Code practitioners, according to the results of the survey.

often lead to misaligned expectations, hindering collaboration. Furthermore, *missing or incorrect business context* documentation was highlighted by 50% and 42% of respondents, respectively, as critical gaps. *Outdated business context and references* were similarly problematic, with 46% citing them as an important documentation issue. This suggests the need for documentation that integrates technical artifacts with organisational objectives to facilitate support to both technical and non-technical stakeholders. These findings resonate with Theunissen et al. [84], who stress the importance of connecting technical and business documentation to enhance organisational memory and communication. The lack of such integration creates misalignments and complicates stakeholder collaboration, a challenge also identified by Zhi et al. [99], who emphasise the role of documentation in bridging development and management objectives.

Visual aids

Visual aids, such as diagrams and screenshots, were less frequently mentioned in comparison to the findings by Aghajani et al. [1]. *Missing diagrams* and *outdated visuals* were only deemed important by 17% and 25% of the respondents, respectively. This can be attributed to the visual nature of Low-Code development, therefore reducing the need to create additional visualisations.

Frequency of Issues Faced

The survey results reveal that the challenges in Low-Code documentation regarding correctness, completeness, and up-to-dateness are not only perceived as important, but also frequently faced by practitioners. Completeness emerged as the most frequently occurring category of issue, with *missing logic behaviour clarifications* (58%) and *missing microflow, nanoflow, and domain model annotations* (54%) dominating the concerns. Additionally, 46% of respondents highlighted *missing user documentation*, emphasising the essential role of documentation to both internal and external stakeholders [26].

Among correctness issues, *wrong logic behaviour clarifications* (38%) and *wrong microflow, nanoflow, and domain model annotations* (29%) were the most frequently cited problems, reflecting practitioners' frequent struggles with understanding the behaviour and interconnections of system components. This aligns with Aghajani et al.'s findings that insufficient accuracy and consistency in documentation hinder software comprehension, particularly for debugging and maintenance [1].

Up-to-dateness issues were similarly prominent, with 29% of the respondents citing *missing documentation for new features/components* and 25% of the respondents pointing to *outdated examples* and *outdated related business processes*. These challenges reflect the difficulties practitioners face in synchronising documentation with iterative development cycles [82].

One key observation here is that although correctness and up-to-dateness issues occur significantly less according to the respondents, one has to keep in mind that in order for documentation to be wrong or outdated, it needs to exist. Taking this into account reveals the actual magnitude of the issue. Many of the documentation required for the efficient and sustainable development and maintenance of Low-Code applications is missing, and in case documentation is available, it is frequently inaccurate or outdated.

4.3 Information Content (How)

Apart from the correctness, completeness, and up-to-dateness, the survey also investigated issues related to maintainability, readability, usability, and usefulness of Low-Code documentation. These dimensions were identified as critical for ensuring documentation effectively supports developers in maintaining, understanding, and using applications [1]. The results are linked to key insights from the literature.

Maintainability of Documentation

54% of the respondents identified *lengthy files* as a significant maintainability issue, followed by *superfluous content* (50%) and *cloned or duplicate content* (46%). These issues reflect inefficiencies that can complicate updating and maintaining documentation, leading to higher technical debt [2]. Garousi et al. [26] emphasise that usefulness of documentation diminishes when it is overly extensive and/or contains redundant information, recommending concise and targeted documentation to optimise maintainability. Furthermore, Robillard et al. [68] advocate for improved automated documentation tools to eliminate redundancy.

Readability of Documentation

Readability emerged as a top concern, with 96% of the respondents prioritising *clarity* and 58% emphasising *conciseness*. *Spelling and grammar* were noted as secondary concerns by 21% of the participants. These findings underscore the importance of clear and concise writing in facilitating developer comprehension, and align with literature. Aghajani et al. highlight clarity and understandability as critical documentation attributes, particularly for facilitating debugging and program comprehension [1]. Similarly, Zhi et al. emphasise that improving readability reduces the cognitive load on developers and enhances the overall effectiveness of documentation [99]. Furthermore, one of the respondents highlighted an interesting readability issue not yet discussed in the gathered literature, namely that the author of documentation incorrectly assumes the reader has the necessary foreknowledge, therefore making the information ineffective.

Usability of Documentation

Accessibility and findability were the most commonly cited usability issues, noted by 79% of the respondents, while 58% highlighted *information organisation* as a key gap. Issues related to *format and presentation* were mentioned by only 21% of the respondents. These findings align with Robillard et al.'s [68] emphasis on well-structured, modular, on-demand and accessible documentation to support efficient navigation and information retrieval. Theunissen et al. [82] further highlight the importance of centralising documentation to ensure stakeholders have access to a single source of truth.

Usefulness of Documentation

The survey reveals that 67% of the respondents found documentation *content not useful in practice*, reflecting a gap between the documentation provided and the developers' needs. One respondent pointed out that this was due to the current need for combining many small bits of information from various sources, rendering them not useful when separate. These finding aligns with studies by Garousi et al. [26], who emphasise that documentation must be relevant and practical to be effective, suggesting a focus on integrating documentation

into development workflows. Aghajani et al. [1] further highlight that the perceived usefulness of documentation is closely tied to its accuracy, up-to-dateness, and alignment with developers' tasks.

Frequency of Issues Faced

Practitioners frequently encounter several key documentation challenges that significantly impact their workflows. *Clarity* was overwhelmingly identified as the most critical issue, with 79% of respondents highlighting its importance, followed by *accessibility/findability* (71%), underscoring the necessity for clear and well-organised documentation. *Lengthy files* (38%) and *superfluous content* (33%) were noted as common maintainability issues, complicating efforts to keep documentation concise and relevant. Furthermore, nearly half of the respondents (46%) found that documentation *content was often not useful in practice*, reflecting a gap between the intended and actual utility of documentation. Issues like *clone/duplicate content* (25%), *spelling and grammar* (21%), and *format/presentation* (25%) were less frequently reported but still indicate areas for improvement. These results align with literature, emphasising the need for concise, clear, and accessible documentation to enhance usability and maintainability, ultimately improving its effectiveness in supporting development tasks.

4.4 Documentation Process & Tools

This section discusses our survey findings regarding issues with documentation processes and tools in Low-Code environments, which are summarised in Figure 4.2. The responses highlight critical challenges such as time constraints, organisation, and a lack of available tools.

Time Constraints as a Dominant Process Issue

A *lack of time to write documentation* was identified as both the most pressing and frequently faced issue by 75% and 83% of the respondents, respectively, emphasising the tension between rapid development cycles of Low-Code and maintaining comprehensive documentation. This aligns with findings from Theunissen et al. [82], who noticed that documentation is often deprioritised in Continuous Software Development due to time pressures and the focus on delivering functional software, thereby exacerbating documentation gaps [93].

Limited Automation and Tooling Deficiencies

A *lack of or poor automation in documentation tools* emerged as the most significant tooling-related issue, noted by 79% of the respondents. This finding underscores the need for automated documentation solutions, as advocated by Robillard et al. [68], who emphasise that automation reduces the burden on developers and ensures up-to-date, consistent documentation. Additionally, 42% of the participants reported *missing features in their current tools*, indicating a gap between practitioner needs and available features. These deficiencies highlight the importance of developing flexible, feature-rich tools tailored to the specific demands of Low-Code environments.

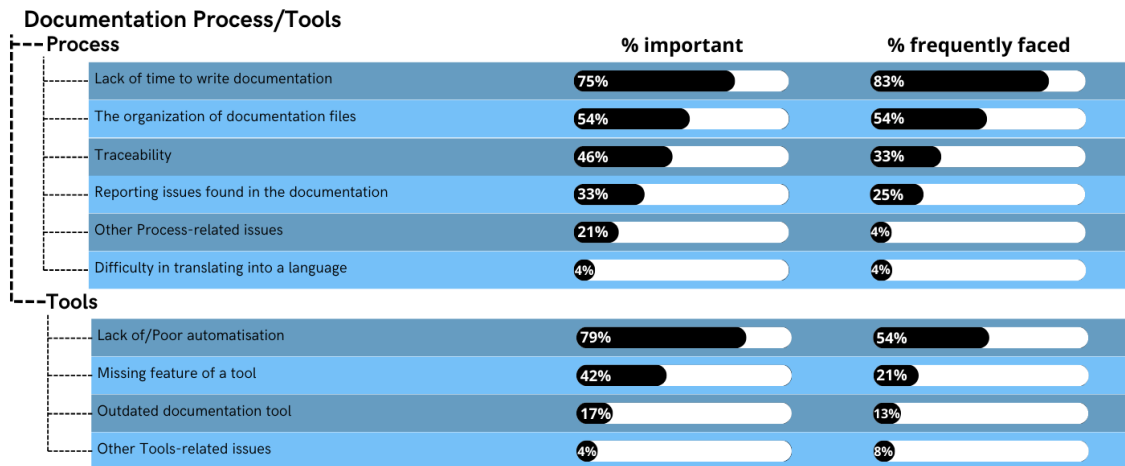


FIGURE 4.2: Documentation process and tool issue results of the survey.

Organisational Challenges and Traceability

Organisational issues, including the *structure and accessibility of documentation files*, were highlighted by 54% of the respondents, while 46% pointed to challenges with *traceability*. These concerns align with Aghajani et al.’s findings that poor organisation and lack of traceability hinder the effective use of documentation during maintenance and debugging tasks [1]. Addressing these issues through structured file systems and enhanced traceability mechanisms is essential for improving documentation usability across the software lifecycle [92].

4.5 Design Phase Documentation

This section analyses survey responses regarding the usage, quality, and reasons for not using various document types in the design phase of Low-Code development, previously discussed in Section 2.4.1. The results highlight discrepancies in document availability, quality concerns, and practitioners’ perceived value. Figure 4.3 presents an overview of the results regarding usage. Furthermore, Figure 4.4 shows an overview of the quality results. Finally, Appendix A, gives the results regarding reasons for non-usage.

The survey results for the design phase reveal challenges in the usage, availability, and quality of key documentation types, reflecting gaps that directly impact alignment, collaboration, and development efficiency. Requirements Documentation and Business Case emerged as the most utilised, with 59% and 50% of the respondents using them most of the time or always, respectively. However, these documents face persistent issues, particularly in up-to-dateness, rated as low as 2.6 for business case documents and 3.1 for requirements documentation. Missing or incomplete requirements were a recurring barrier, with 67% of the respondents who indicated that they do not use these documents (denoted as non-users from here on) reporting that these documents either do not exist or cannot be found, underscoring traceability concerns. Similarly, while Business Process Documentation was valued for its usefulness (4.1) and readability (3.8), its low up-to-dateness (2.5) reduces its effectiveness, confirming challenges found in iterative environments where documentation get out of sync with changes [82].

In contrast, Roadmap & Release Plans and Mockups and UI Documentation exhibited

particularly low usage, with 46% and 67% of the respondents, respectively, stating that they rarely or never use them. For Roadmaps, 55% of non-users indicated that these documents simply do not exist, reflecting a gap in planning documentation during the design phase. Mockups and UI documentation, while rated highly for usefulness (4.3) and readability (4.3) by those who use them, are hindered by availability issues 44% of non-users reported that such documents are not created at all. These findings suggest that visual and strategic design artifacts, which could enhance stakeholder alignment and development clarity, are often disregarded or overlooked. The data overall highlights a recurring theme of incomplete, unavailable, or outdated design-phase documentation, which could lead to misalignment and rework.

Furthermore, the results underscore the critical role of improving traceability, adopting automation to maintain document synchronisation, and fostering a structured approach to document creation. Addressing these challenges would enable design-phase documentation to better support decision-making, reduce ambiguity, and provide a stronger foundation for subsequent phases of Low-Code development.



FIGURE 4.3: Design Phase Documentation usage results

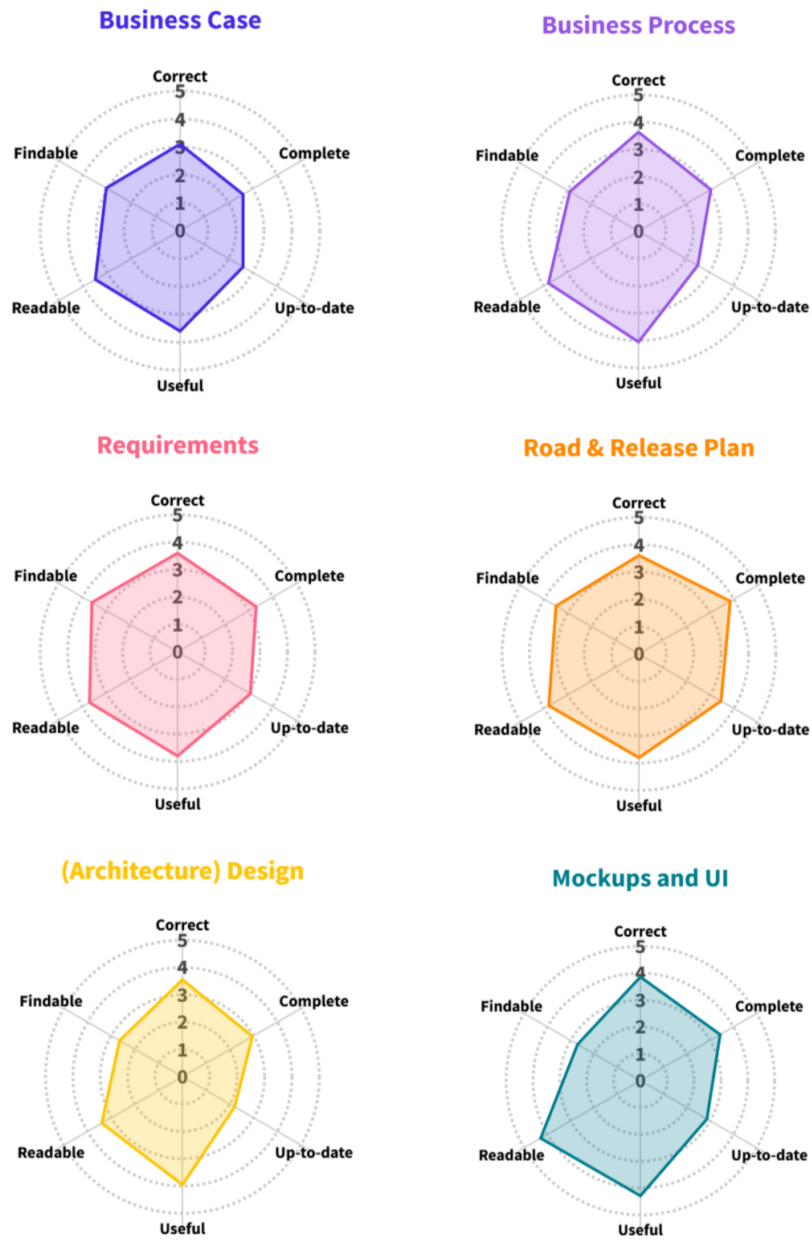


FIGURE 4.4: Design Phase Documentation quality results

4.6 Development Phase Documentation

This section analyses the survey responses regarding usage, quality, and reasons for not using various document types in the development phase of Low-Code development, previously discussed in Section 2.4.2. The results highlight discrepancies in document availability, quality concerns, and practitioners' perceived value. Figure 4.5 presents an overview of the results regarding usage. Figure 4.6 shows an overview of the quality results. Appendix A gives the results regarding reasons for non-usage.

The results for the development phase reveal underutilisation of core documentation types, with persistent challenges surrounding availability, completeness, and up-to-dateness. Data Model Documentation and Application Logic Documentation are the least utilised, with 50% or more of the respondents reporting they never or almost never use them. For Data Model Documentation, 42% of the respondents noted its absence as the primary barrier, while for Application Logic Documentation, 53% of the respondents reported similar gaps. This unavailability is particularly concerning, as both document types are foundational for understanding data relationships and application logic, critical components in Low-Code development workflows. When used, their quality ratings reveal further concerns: completeness (2.5 for Data Models, 3.3 for Microflows) and up-to-dateness (2.7 and 2.8, respectively) were rated as low, reflecting ongoing struggles in maintaining documentation synchronisation, which is a challenge consistent with literature on iterative development environments [82, 93, 29].

Source Model Comments (annotations) saw higher usage, with 42% of the respondents using them sometimes and 29% most of the time, indicating their perceived value in providing guidance near the source material. However, 40% of non-users cited issues such as incompleteness, outdated content, or non-existence, highlighting the need for better management of this lightweight documentation type. Despite these issues, respondents rated annotations positively for usefulness (3.9), readability (4.1), and findability (4.26), suggesting they are well-received when available and up-to-date.

Component Documentation and Dependencies Documentation similarly showed mixed usage patterns. 29% of the respondents reported never using Component Documentation, with 45% attributing this to its absence, while Dependencies Documentation was largely unused by 71% of the respondents due to availability concerns. Both document types play a critical role in managing reusable components and external dependencies, which are central to Low-Code development's scalability. For those who use them, Dependencies Documentation performed relatively better, with correctness (3.7) and completeness (3.4) receiving favourable scores. Component Documentation, however, exhibited more quality concerns, with up-to-dateness (2.7) remaining a key issue, indicating a gap in maintaining alignment with evolving application components.

Overall, the results for the development phase point to recurring issues of documentation availability and quality, particularly for critical artifacts like Application Logic and Data Model Documentation. These findings underscore the need for structured documentation practices, improved traceability, and synchronisation tools to reduce effort and ensure documentation remains accurate and accessible throughout the development process. Without addressing these gaps, development-phase documentation run the risk becoming a bottleneck, increasing complexity and technical debt while reducing collaboration and maintainability.

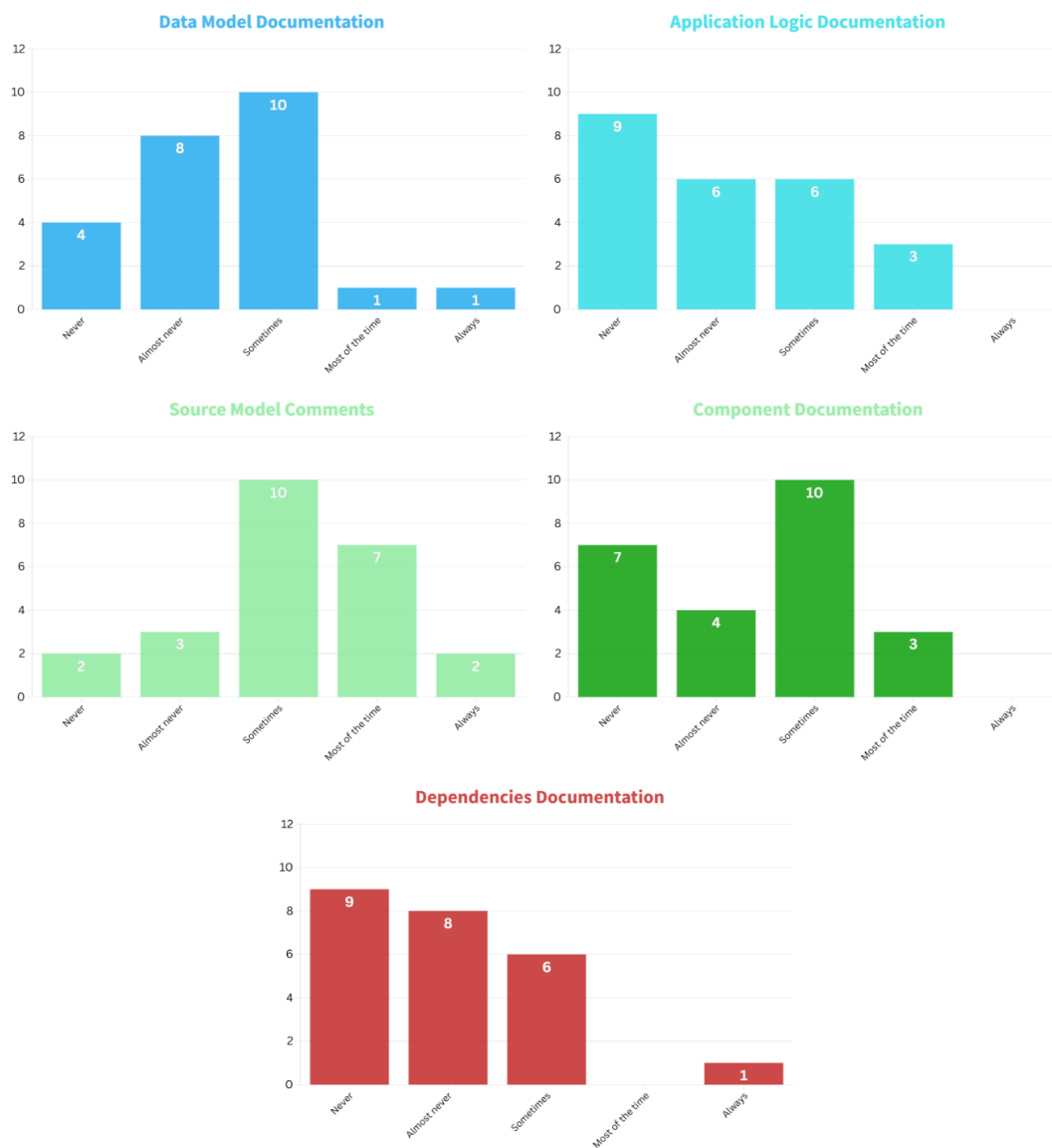
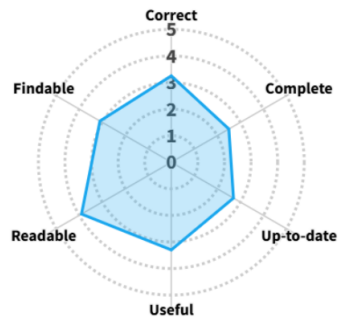
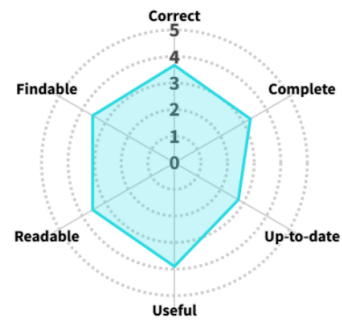


FIGURE 4.5: Development Phase Documentation usage results

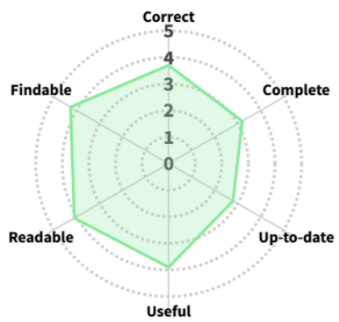
Data Model Documentation



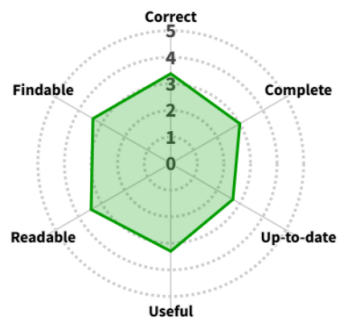
Application Logic Documentation



Source Model Comments



Component Documentation



Dependencies Documentation

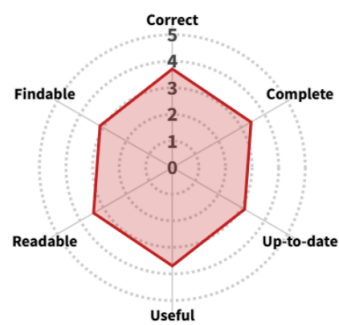


FIGURE 4.6: Development Phase Documentation quality results

4.7 Maintenance Phase Documentation

This section analyses the survey responses regarding usage, quality, and reasons for not using various document types in the maintenance phase of Low-Code development, previously discussed in Section 2.4.5. The results highlight discrepancies in document availability, quality concerns, and practitioners' perceived value. Figure 4.7 shows an overview of the results regarding both the usage and quality. Appendix A provides the results regarding reasons for non-usage.

The results for the maintenance phase highlight widespread underutilisation and mixed perceptions of documentation quality. Maintenance Documentation and User Manuals emerged as the least utilised document types, with 37% and 50% of the respondents, respectively, reporting they never or almost never use them. For Maintenance Documentation, 33% of the respondents indicated that it does not exist, while 22% mentioned lack of value as a reason for not using it. Despite being essential for ongoing updates and troubleshooting, the quality of Maintenance Documentation was rated as poor, particularly for completeness (2.5) and up-to-dateness (2.3). Similarly, User Manuals suffer from availability issues, with 42% of the non-users stating they do not exist and 33% reporting they are inaccessible. This aligns with their low usage frequency, as only 17% of the respondents reported using them most of the time or always. Although rated highly for readability (3.9), User Manuals struggle with up-to-dateness (2.8).

Service-Level Agreements (SLAs) and Issue Tracking Logs were moderately used, with 21% and 29% of the respondents using them most of the time, respectively. For SLAs, 42% of the non-users cited a lack of perceived value, highlighting potential misalignment between documentation intent and user needs. Despite this, SLAs received relatively positive ratings for correctness (3.5) and completeness (3.6), suggesting they are valuable when used. Notably, the quality of Issue Tracking Logs was perceived favourably in terms of usefulness (3.5) and findability (4.0), indicating their importance in identifying and resolving issues efficiently.

Overall, the results highlight gaps in availability and up-to-dateness across maintenance-phase documents, particularly for foundational artifacts like Maintenance Documentation and User Manuals. Poor availability and perceived low value of these documents hinder knowledge transfer, troubleshooting, and system evolution, leading to inefficiencies and increased maintenance costs. Addressing these challenges through improved traceability, automation, and structured updates could significantly enhance the role of maintenance documentation in supporting long-term Low-Code application sustainability.

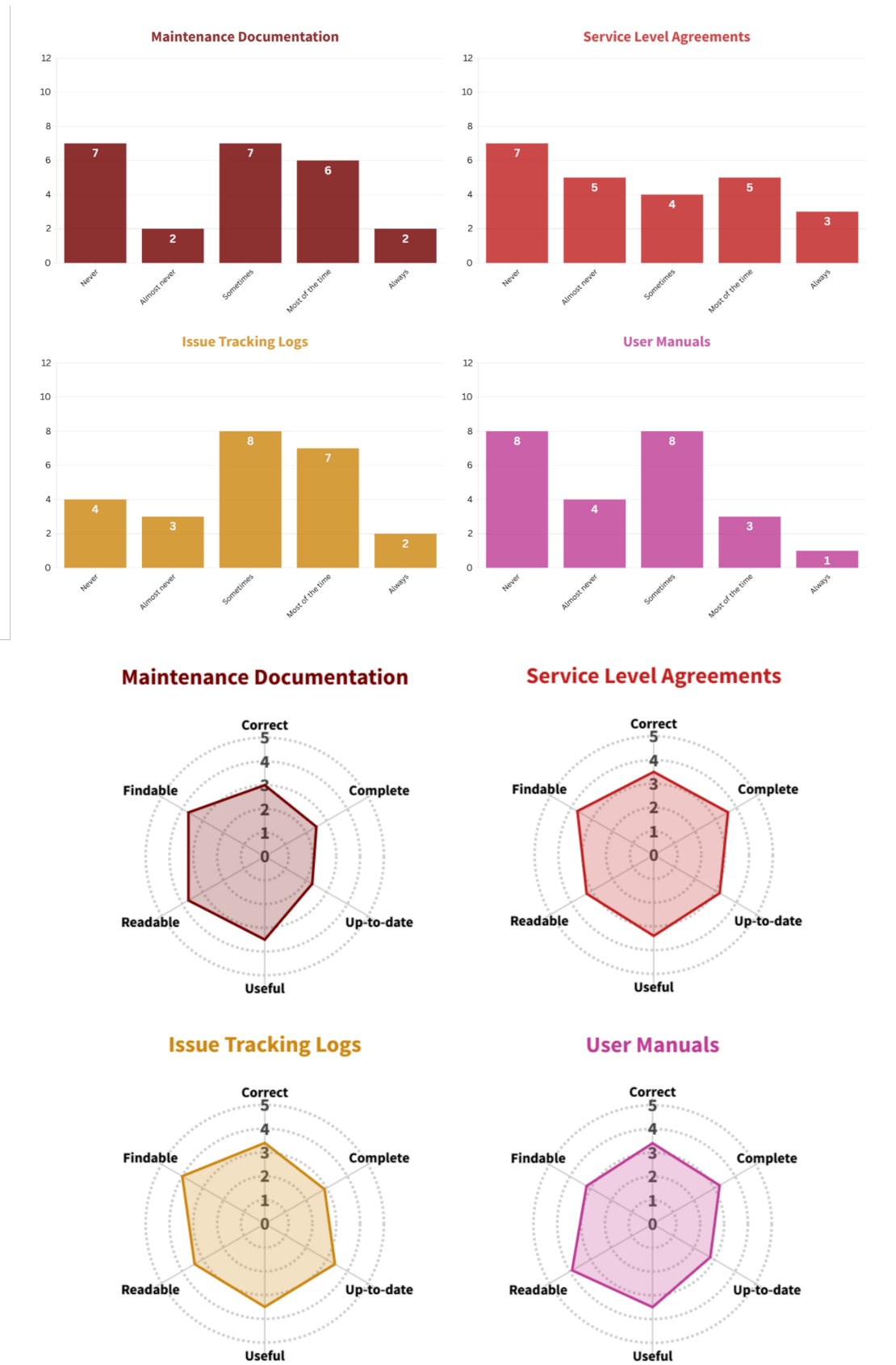


FIGURE 4.7: Maintenance Phase Documentation usage and quality results

4.8 Conclusion

This section concludes the chapter by discussing the last survey question and summarising our findings and their paired conclusions in Table 4.2 and Table 4.3. The final survey question sought to identify the types of documentation that respondents would use more frequently if they were improved in terms of correctness, completeness, up-to-dateness, findability, readability, and usability. The results to this question are presented in Table 4.1. The responses revealed that (Architecture) Design Documentation (63%), Business Process Documentation (58%), and Business Case Documentation (54%) were the most desired types, highlighting their foundational role in framing other documentation types. Respondents emphasised that inaccuracies or incompleteness in these high-level documents often cascade into other forms of documentation, diminishing their utility. Respondents also highlighted significant desire to increase use of Maintenance Documentation (50%) and Application Logic (Microflow) Documentation (46%), and User Manuals (42%). These types are critical for understanding ongoing system operations and application logic, especially in dynamic Low-Code environments. However, as the results in Section 4.2 showed they are often outdated or inaccessible, reducing their utility for developers and stakeholders.

Additional comments revealed challenges in the current state of documentation practices in line with the findings presented in literature. Some respondents pointed to challenges related to outdated documents stored in inaccessible locations, the lack of proactive documentation updates, and the absence of centralised repositories. This reflects a broader concern about the usability and discoverability of documentation in Low-Code environments. Documentation types such as Roadmaps and Release Plans (21%), Service-Level Agreements (17%), and Data Model Documentation (13%) received lower prioritisation. This suggests that while these documents are valuable, their perceived relevance may vary depending on specific roles or contexts.

Overall, the survey underscores a need to address documentation gaps, ensuring their accessibility, and promoting regular updates to keep pace with evolving Low-Code applications. These insights align closely with literature findings and the broader objectives of this research to enhance the quality and availability of documentation in Low-Code environments.

TABLE 4.1: Results question: What type of documentation would you use more if they were more correct, complete, up-to-date, findable, readable and usable?

Documentation Type	Percentage		
(Architecture) Design Documentation	63%	Component documentation	25%
Business Process Documentation	58%	Dependencies Documentation	25%
Business Case	54%	Roadmap & Release Plan	21%
Maintenance Documentation	50%	Mockups and UI documentation	21%
Application Logic (Microflow) Documentation	46%	Service-Level Agreements (SLA)	17%
User Manuals	42%	Issue Tracking Logs (Tickets)	17%
Requirements Documentation	29%	Data Model Documentation	13%
Source Model Comments (Annotations)	29%		

TABLE 4.2: Information Content (What & How)

Finding	Conclusion
Missing or outdated logic behaviour documentation and annotations significantly hinder comprehension and debugging efforts.	Correctness, completeness, and synchronisation of documentation are critical. Integrated source annotations can reduce debugging challenges.
Misalignments between technical artifacts and business processes exacerbate these issues	Aligning technical documentation with business objectives could enhance collaboration.
Agile iterations frequently lead to outdated documentation. Practitioners also noted fragmented and redundant documentation as barriers to maintainability and usability	Automated synchronisation strategies ensuring up-to-date documentation are essential for maintaining relevance and reducing technical debt in rapidly evolving Low-Code environments.
Clarity, accessibility, and concise content were prioritised, but fragmented information and incorrect assumptions about user knowledge reduced practical utility.	Clear, centralised, and modular documentation could support diverse stakeholder needs. Integration into workflows and alignment with developer tasks enhances usability and practical utility.

TABLE 4.3: Documentation Processes & Tools

Finding	Conclusion
Time pressures in fast-paced Low-Code development cycles often make documentation become disregarded, leading to incomplete or outdated content.	Automated tools and processes are vital to ensure documentation keeps pace with development and does not fall behind.
Existing tools lack essential features or automation, resulting in inconsistent documentation practices.	There is a critical need for feature-rich, automated tools tailored to Low-Code platforms. Such tools can reduce the burden on developers while ensuring synchronisation with iterative updates.
Poorly organised documentation files and limited traceability mechanisms complicate maintenance and debugging tasks.	Structured file systems and robust traceability mechanisms could improve usability and collaboration, facilitating faster problem resolution and better long-term knowledge retention.

Chapter 5

Available Solutions

This chapter analyses the current state of research and practical tools related to automated documentation generation, highlighting their various techniques, benefits, and limitations. In line with the research methodology proposed by Wieringa [95], this short review identifies gaps and opportunities that we addressed with our proposed documentation assistant.

5.1 Automated Documentation Generation Techniques

Automated documentation generation has been a prominent theme in Software Engineering [56], particularly due to the increasing complexity of software systems and the burden of maintaining comprehensive documentation manually. Existing studies propose diverse techniques, ranging from algorithm-driven to model-driven methods, as well as modern AI integrations that address multiple aspects of documentation challenges. However, many of these solutions often struggle with scalability, flexibility, context-awareness, or alignment with evolving development practices.

5.1.1 Automatic Code Commenting and Summarisation

A substantial body of work focuses on generating automatic comments and summaries directly from source code [75, 56, 57]. Moreno and Marcus [56] categorise these automated software summarisation strategies into text-to-text, code-to-text, code-to-code, and mixed-artifact summarisation, highlighting the various goals and approaches of different summarisation tools. Various algorithms, particularly those based on information retrieval (IR), deep learning, and hybrid methods, have been investigated [75]. While IR-based techniques can be efficient, their reliance on static templates limits their flexibility. Meanwhile, Deep Learning models, such as recurrent neural networks (RNNs) and transformers, generate higher-quality comments by integrating both structural and lexical code features [45]. However, the scalability of these methods and the lack of standardised datasets remain major obstacles to broader adoption [75]. Additionally, many of these strategies focus primarily on syntactic or structural code insights, overlooking the broader application goals and nuances. Furthermore, many existing approaches emphasise code summarisation rather than generating contextually rich or purpose-driven descriptions [50]. Therefore, Song et al. argue for a need of a customised and intelligent automatic generation system that meets various scenarios [75].

5.1.2 Context-Aware Documentation

Addressing context-awareness is another key focus within documentation generation. Tools such as McBurney and McMillan’s [50] context-aware summariser incorporate method dependencies and interactions, offering deeper insights into the “why” and “how” of code operation. Similarly, LAMBADDOC [4] showcases the usefulness of documenting Java lambda expressions by combining metadata and natural language generation templates to provide more comprehensive information. While these strategies improve code comprehension, their applicability is often limited by factors such as poor scalability, tight coupling with specific programming languages, and limited output flexibility.

5.2 Model-Driven Documentation Generation

Low-Code documentation generation is closely related to model-driven documentation, which leverages structured templates and mappings to create various artifacts such as requirements, design specifications, and system overviews directly from underlying software models. This approach reduces manual overhead and promotes consistency in environments that employ Model-Driven Development. Wang et al. [94] present a model-driven documentation generator integrated into the SmartOSEK IDE [98], aiming to bridge the gap between software models and the corresponding documentation. Their tool ensures consistency across artifacts by using a hierarchical documentation model, wherein each node corresponds to key sections, such as requirements, preliminary design, and detailed design specifications. A dedicated mapping mechanism links elements from system models (e.g., UML diagrams) to these documentation nodes, allowing both textual and graphical data to be incorporated. Other frameworks, such as those proposed by Henzgen and Strey [31], further extend the scope of model-driven documentation by aggregating UML, BPMN, and GSN model information into ISO-compliant documents. Although these model-driven systems underscore the potential of dynamic templates and metamodel mappings, their effectiveness hinges on model completeness and up-to-dateness, frequently requiring manual oversight to address inconsistencies [31]. Moreover, these model-driven solutions rely on predefined outputs, limiting flexibility, and responsiveness in rapidly changing projects or user requirements [68].

5.3 Agile and Dynamic Documentation

Agile development methodologies, which are characterised by iterative and incremental practices, often disregard documentation in favour of frequent releases. To address this challenge, Voigt et al. [91] introduce SprintDoc, which is a tool that integrates documentation activities into Agile workflows by linking task management systems (e.g., Jira) with version-controlled documentation repositories. Likewise, Silva et al. [74] propose methods like Dynamic Documentation Generation and Automated Documentation Testing to curtail technical debt and improve customer-facing documentation. While these tools promote traceability and adaptability, they currently lack real-time contextual updates and do not fully utilise advanced natural language generation techniques to deliver rich documentation outputs.

5.4 Large Language Models and Software Documentation

In recent years, Large Language Models (LLMs) have shown considerable promise for automating software documentation [37, 58, 77]. Tools like Codex excel at generating context-aware descriptions across multiple programming languages [37]. By synthesising code-level data into human-readable text, Codex significantly reduces manual effort. However, several issues remain, including the tendency toward verbose outputs, limited domain specificity, hallucination, and reliance on proprietary large-scale infrastructures that can pose data privacy risks [37]. Other notable work includes Naimi et al. [58], which utilises LLMs to transform UML use case diagrams into documentation. While this method demonstrates the advantages of combining structured inputs with generative models, it relies heavily on prompt engineering and has limited adaptability to evolving development requirements. Distilled GPT models [77] present an alternative solution, offering lightweight yet effective code summarisation capabilities that preserve data custody and often achieve similar results as larger models like GPT-3.5. Despite their strengths, current LLM-based solutions face constraints such as high computational overhead, dependence on proprietary datasets, hallucination, limited customisation and lack of flexibility for specialised needs.

5.5 Retrieval Augmented Generation

A key limitation of LLMs lies in their reliance on pre-trained knowledge, which can lead to incomplete, outdated, or overly general responses, often referred to as “hallucinations” [9], when queried about domain-specific details [63]. Retrieval-Augmented Generation (RAG) addresses this challenge by coupling LLMs with external knowledge sources, allowing them to query and incorporate relevant information in real time [9]. This approach significantly improves accuracy, adaptability, and context-awareness, making it highly valuable for software development tasks [73]. In this area, Parvez et al. [62] propose a framework named REDCODER, a retrieval-augmented generation solution specifically tailored to code generation and summarisation. REDCODER extends state-of-the-art dense retrieval methods to search for relevant code or corresponding natural language descriptions, thereby bridging the gap between unimodal data (solely code or language) and bimodal instances (paired code-description). Their results demonstrate that supplementing code-generation and summarisation models with retrieved content leads to more comprehensive and accurate outputs [62]. However, retrieval methods common in RAG systems often struggle with complex reasoning for query formulation [46] and the handling of intricate and complex structures and relationships [63], leading to incomplete retrieval and responses [65].

Aiming to solve these issues, graph-based RAG methodologies have emerged as a compelling paradigm for enhancing the interaction between LLMs and external knowledge [22, 63]. GraphRAG [22, 63] leverages the structural relationships between entities to enhance retrieval precision and depth, effectively capturing relational knowledge and enabling more accurate, context-aware responses [63]. A comparison between Direct LLM, RAG, and GraphRAG can be seen in Figure 5.1. As the figure shows, when responding to user queries, LLMs may provide shallow or insufficiently specific answers. RAG partially mitigates this by retrieving relevant textual information, but its effectiveness is limited by the length of the text and the flexibility of natural language in expressing entity relationships, making it challenging to highlight “influence” or indirect relations central to the query [63]. In contrast, GraphRAG methods use explicit representations of entities and their relationships enabling precise answers. Building on this, some attempts have been made to apply this concept to software repositories. Liu et al. [46] introduce CODEXGRAPH,

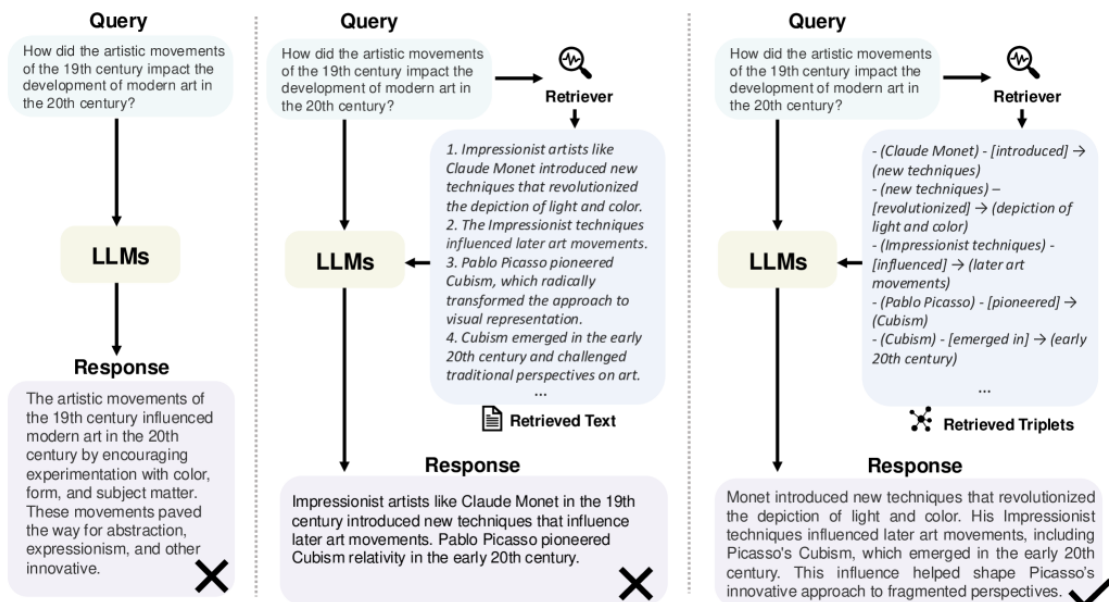


FIGURE 5.1: Comparison between LLM, RAG, and GraphRAG (copied from [63]).

which couples LLM agents with a graph database interface. This architecture encodes code elements and relationships as nodes and edges within a unified graph schema. Through static analysis, CODEXGRAPH supports precise queries and multi-hop reasoning, making it particularly effective for debugging, code navigation, and unit testing [46]. In contrast, RepoGraph [61] introduces line-level granularity into code representations, strengthening the traceability of code dependencies and execution flows. By relying on sub-graph retrieval algorithms, RepoGraph delivers localised yet comprehensive code segment analysis. However, it still depends on predetermined retrieval algorithms, limiting its capacity to adapt to less structured or evolving repository configurations. Both CODEXGRAPH and RepoGraph underscore the potential of graph-based representations for addressing challenges such as scalability and contextual relevance at the repository level.

5.6 Conclusion

Overall, this chapter reports on a range of automated documentation generation approaches identifying strengths and persistent challenges. Traditional IR-based and Deep Learning methods excel at producing code-level summaries but often have poor scalability, lack contextual awareness, and domain adaptability. Model-driven approaches leverage structured templates and metamodels to create consistent artifacts, yet they remain dependent on comprehensive and up-to-date models. Agile and dynamic documentation solutions bring documentation closer to fast-paced development cycles but fall short in delivering real-time, context-rich outputs. Recently, LLM-based systems have shown strong potential for generating more natural and expressive documentation, although issues such as verbosity, hallucination, and reliance on proprietary infrastructures pose ongoing hurdles. RAG and GraphRAG further extend the capabilities of LLMs by incorporating external knowledge sources to deliver improved accuracy, multi-hop reasoning, and deeper context. Overall, while existing tools demonstrate valuable progress, gaps remain in addressing scalability, customisability, real-time adaptability, and the specific needs of Low-Code.

Chapter 6

CLAIR: Connecting Low-Code and Artificial Intelligence for RAG

The primary goal of this research is to improve the documentation quality and process for Low-Code applications. The enhanced documentation should provide valuable insights into application components, processes, and dependencies to improve system maintainability, understandability, sustainability and overall knowledge retention. To this end, we propose an AI-driven assistant, which we named CLAIR that integrates knowledge graphs and LLM agents for retrieval-augmented generation¹ to automate and support on-demand documentation for Low-Code applications. This chapter outlines the design of CLAIR, starting with the specification of the requirements in Section 6.1, after which in Section 6.2 the overall design is introduced, and ending with Section 6.3 which introduces the use cases of CLAIR.

6.1 Requirements Specification

This section defines the purpose and scope of the artifact we developed in this project, CLAIR, as well as its functional and non-functional requirements. The requirements are derived through a combination of literature review, empirical research, and stakeholder analysis in line with DSM [95].

6.1.1 Purpose and Scope

The purpose of CLAIR is to address documentation gaps in Low-Code applications by dynamically generating on-demand documentation. This aims to improve the documentation quality and process specifically addressing the challenges and needs identified in Chapter 4. CLAIR focuses on the Mendix Low-Code platform and agile development environments, where documentation often lags due to rapid delivery cycles.

¹Retrieval-Augmented Generation (RAG) is a method that combines external data retrieval with text generation to improve the accuracy and relevance of outputs, addressing the limitations of LLMs in handling domain-specific queries [9].

Scope

For the current implementation, the focus is limited to data and application logic elements within Mendix applications. Specifically, CLAIR extracts and processes components such as modules, domain models, and microflows, which form the foundation of application logic and data relationships. At this stage, UI-related elements, such as pages, and custom code components, including Java actions, scheduled events, and other advanced customisations, are excluded. This focused approach ensures a manageable and scalable implementation, while addressing the critical components of data and application logic that have significant impact on maintainability and knowledge retention.

6.1.2 Functional Requirements

To achieve the intended goals, CLAIR must satisfy the functional requirements shown in Table 6.1. These requirements are based on the combination of the literary findings in Chapter 2, the results of the survey presented in Chapter 4, and the identified gaps in the current solution, discussed in Chapter 5.

TABLE 6.1: Functional requirements for CLAIR

ID	Requirement	Description
FR1	Knowledge Extraction	Extract components like domain models, microflows, and their relationships from Mendix applications.
FR2	Store Retrievable Knowledge	Store extracted knowledge of components in a graph repository, retrievable by the documentation assistant.
FR3	On-Demand Documentation Generation	Automatically generate dynamic context-aware documentation based on user queries.
FR4	Chat Interface	Provide a chat interface for intuitive user interaction
FR5	Multi-Purpose Functionality	The system must be able to provide various types of documentation, discussed in Section 2.4 to support different purposes across the development lifecycle, including design, development, and maintenance.

6.1.3 Non-Functional Requirements

Non-functional requirements concern the quality of CLAIR’s implementation and the quality of the documentation it generates. To address this dual focus, we define two distinct sets of requirements. Table 6.2 specifies the quality criteria for the generated documentation, ensuring it meets stakeholder needs identified in Chapter 4. Table 6.3, outlines the non-functional requirements concerning the usability of CLAIR itself.

TABLE 6.2: Quality requirements for the generated documentation

ID	Requirement	Description
QR1	Correctness	The generated documentation must accurately reflect the actual Mendix application components, ensuring no errors or discrepancies.
QR2	Completeness	All relevant components, including entities, microflows, and their relationships, must be fully included in the generated documentation.
QR3	Relevancy	The assistant should focus on providing only the necessary and context-specific information based on the user's query, avoiding redundancy or irrelevant details.
QR4	Understandability	The generated content must be clear, concise, and easy to understand for both technical and non-technical stakeholders.
QR5	Readability	The documentation must be well-structured, logically organised, and formatted to facilitate quick navigation and comprehension. ²
QR6	Usefulness	The documentation must serve its intended purpose, aiding stakeholders in understanding, maintaining, and improving Low-Code applications.
QR7	Up-to-dateness	The documentation must reflect the latest state of the Low-Code application, ensuring it remains current after changes or updates.

TABLE 6.3: Non-functional requirements for CLAIR

ID	Requirement	Description
NFR1	Usability	The system must be easy to use, ensuring users can interact with the tool intuitively.
NFR2	Information Availability	The system must be able to provide the requested information on-demand without delays, less than 30 seconds.
NFR3	Helpfulness	The responses and generated documentation must be helpful in addressing users' needs and goals.
NFR4	Accuracy	The system must provide accurate and reliable information that aligns with user queries.
NFR5	Time Efficiency	The system must save users time by automating documentation generation and reducing manual effort.
NFR6	Scalability	The system must be able to handle large and complex Mendix applications without significant slowdowns.
NFR7	Workflow Integration	The system must integrate seamlessly into users' workflows, encouraging frequent and practical use.
NFR8	Flexibility	The system must be able to handle diverse queries, providing tailored responses based on the specific information requested.

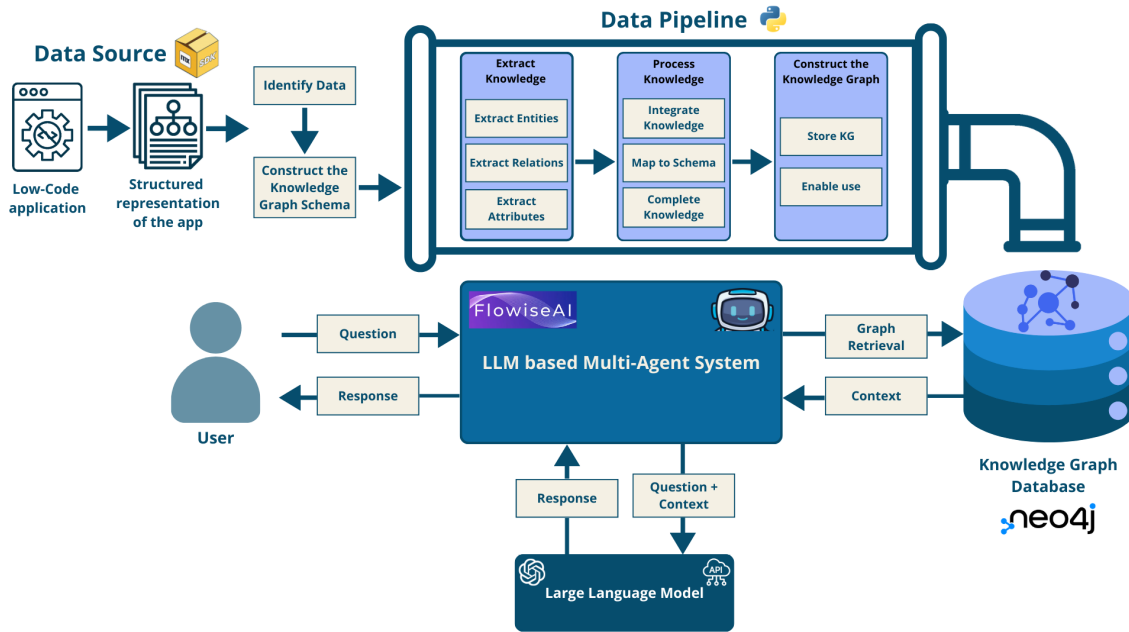


FIGURE 6.1: CLAIR architecture

Wieringa emphasises that requirements must be justified using contribution arguments. These arguments predict how an artifact should contribute to stakeholder goals [95].

Contribution Argument:

If CLAIR generates high-quality, up-to-date, on-demand documentation, *and assuming* Mendix developers rely on visual tools and agile practices which results in inadequate documentation, *then* CLAIR will improve maintainability, reduce technical debt, improve collaboration, and enhance application sustainability.

6.2 CLAIR Design

CLAIR is an artifact designed to automate and enhance documentation for Low-Code applications, particularly Mendix. The system bridges Mendix applications and Large Language Models (LLMs) through a knowledge graph interface, chosen for its ability to provide a structured, queryable representation of complex interrelated application data. By leveraging a data pipeline, it transforms structured data extracted from Mendix applications into a knowledge graph database, serving as a central knowledge repository that supports efficient and dynamic information retrieval. The architecture, illustrated in Figure 6.1, also includes an LLM-based Multi-Agent System (MAS) [28] to process user queries. This MAS was implemented for its capability to interpret natural language queries, generate graph queries, and retrieve relevant insights from the knowledge graph, providing accurate, context-aware responses tailored to users’ needs. The modular and scalable design ensures flexibility and adaptability, allowing users to access up-to-date, dynamic, and context-aware documentation on-demand. This section details the rationale behind the selection and integration of these components, as well as the processes of constructing the knowledge graph, the data pipeline, the knowledge graph database, and the LLM-based Multi-Agent System.

6.2.1 Knowledge Graph Database

To ensure a consistent and structured representation of the application data, we adopted the approach for knowledge graph development as outlined by Tamašauskaitė and Groth [79]. This method involves the sequential steps: identifying data, constructing the schema, extracting and processing knowledge, and integrating the data into a graph structure.

Data Source

As discussed in Section 3.1, Mendix applications consist of various interconnected components, including domain models, microflows, entities, and attributes. All of these components can be designed in Mendix’s visual development environment called Mendix Studio Pro. Built applications are stored in Mendix specific file types with a .mpr and .mpk extension. To build a knowledge graph, we need to be able to extract the components in a systematic manner and transform them to a structured representation. The Mendix Model Software Development Kit³ (SDK) allows programmatic access to the structure of Mendix models, by providing an API that enables developers to read, analyse, and interact with Mendix application models. The SDK supports JavaScript/TypeScript and allows access to all Mendix model elements, including domain models, microflows, pages, and custom widgets. Furthermore, the SDK provides a function that returns the structure of an element in an application as plain JSON. Given that we need a structured textual representation for our design we used the SDK to create a script that retrieves JSON representations of following components:

- **Domain Models**
 - Extracted entities, their attributes, and data types.
 - Captured relationships between entities (associations and generalisations).
- **Microflows and Nanoflows**
 - Extracted the logic within each microflow, including actions, decision splits, loops, and more.
 - Captured dependencies between microflows and other entities or modules.
- **Modules**
 - Extracted module and element hierarchies.

Schema Design

Using the Mendix SDK, we ensured a systematic and reliable extraction process, providing the foundation for constructing a knowledge graph. To construct the knowledge graph schema, we combined the components extracted from the Mendix application with the publicly available Mendix Metamodel⁴. This approach ensured that the schema not only reflected the specific structure of the application but also adhered to the general Mendix model structure and relationships defined by the platform. By following this method, we adhered to a top-down knowledge graph construction process [79]. The top-down approach leverages an existing domain metamodel or structured dataset, both of which are available, to guide the development of the knowledge graph schema [79]. The schema specifies the

³<https://docs.mendix.com/apidocs-mxsdk/mxsdk/sdk-intro/>

⁴<https://docs.mendix.com/apidocs-mxsdk/mxsdk/mendix-metamodel/>

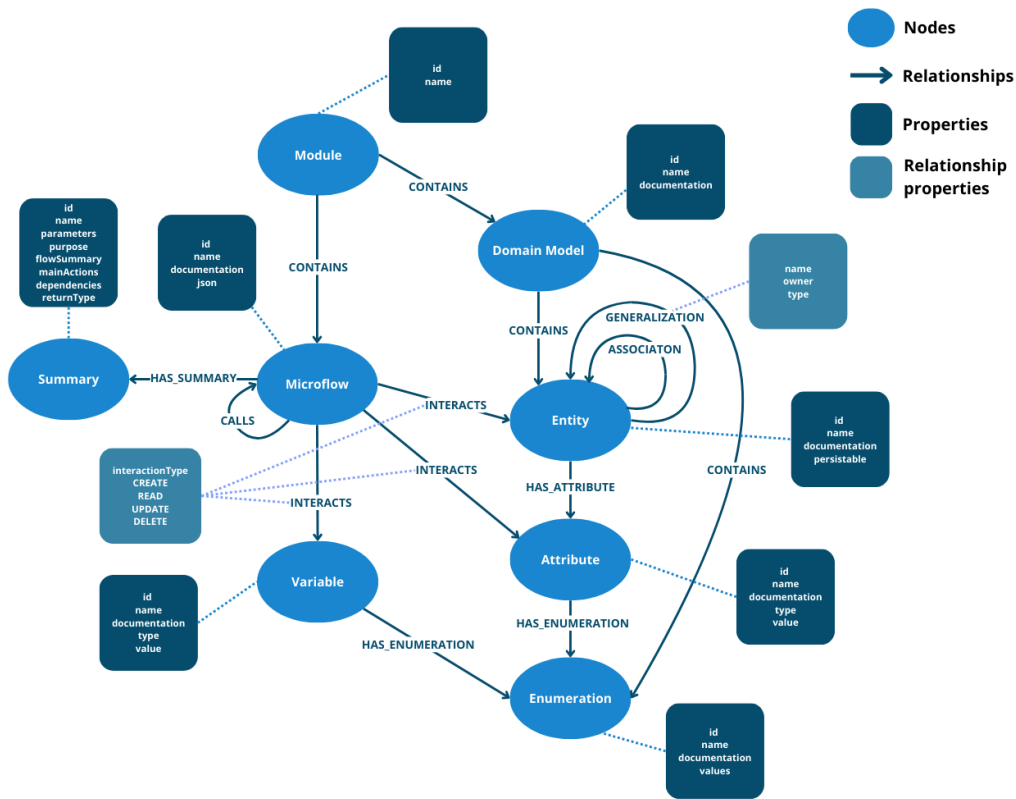


FIGURE 6.2: Knowledge Graph schema for storing Mendix model elements

categories of nodes, edges, and their properties, which dictate how the graphs are structured and stored within the graph database [79]. The constructed knowledge graph schema can be seen in Figure 6.2. The details of the schema can be found in Appendix B.

Data Pipeline

The second step to construct knowledge graphs is to extract and process knowledge from the data [79]. To this end, we have setup a data pipeline that extracts, enhances, and processes the desired knowledge from the JSON representations extracted using the Mendix SDK, Python, and a LLM API. For this section we zoom in on the data pipeline component in Figure 6.1. Figure 6.3 gives an overview of this pipeline, which includes an additional data preparation step related to reducing the size of the JSON files, each step of the pipeline is discussed below.

1) Reduce JSON File Size

JSON files generated from the Mendix Model SDK for microflows are verbose, containing keys and values required for Mendix Studio Pro or background processes. These elements, while essential for the Studio Pro development environment to be able to interact with models, are irrelevant for the logic and structure of the resulting application, and therefore for our AI Assistant. To address this, the extracted JSON files undergo a cleaning process where information only needed for Studio Pro is removed ensuring that the final files are concise and focused. Figure 6.4 illustrates the file reduction by showing an original and reduced file. This reduction serves multiple purposes:

- Improved Speed: Smaller file sizes enable faster data processing and querying.

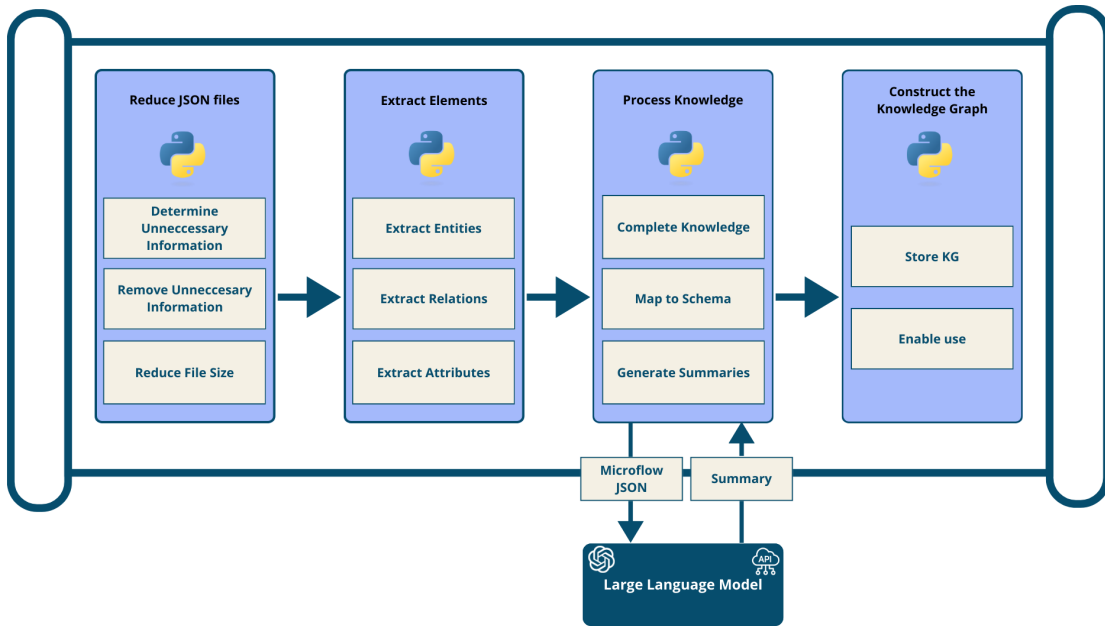


FIGURE 6.3: Refined data pipeline (including the extra JSON reduction step)



FIGURE 6.4: Transformation process: (a) Input JSON file and (b) Reduced JSON file.

- **Enhanced Accuracy:** By removing irrelevant data, we improve the clarity of the information presented to the LLM.
- **Reduced Token Costs:** Smaller JSON files reduce the amount of data sent to the LLM API, reducing costs associated with token usage.

2) Extract Entities, Relationships, and Attributes

Key elements from the JSON are extracted to form the abstract representation of the application structure within the knowledge graph. This involves processing modules, domain models, microflows, nanoflows, entities and attributes to capture their information and relationships. This extraction ensures that the knowledge graph reflects the full structure and functionality of the Mendix application, providing a comprehensive view of its components.

3) Enhance Data with Microflow Summaries

Apart from extracting and processing the information related to applications components inherent in the JSON files we also generate summary representations of JSON parts related to microflows. We do this to address the limitations of LLM context windows⁵ and optimise token usage. The high-level summaries of microflows are generated by an LLM and added to the knowledge graph as summary nodes. These summaries serve as condensed representations of microflows, enabling efficient querying for broad questions while preserving detailed information for deeper exploration. This is done by a Python script that calls an LLM API, generating structured summaries for each microflow. In our design, we use GPT-4o, which is a model developed by OpenAI⁶ and is among the top-performing models as of January 2025⁷. Furthermore, the model excels in reasoning, Mathematics, and code as well as showing good results in similar settings [46, 61, 34]. We prompt the API with both instructions and a JSON representation of a microflow. We do this for each of the microflows in an application. By using structured outputs⁸, we ensure consistency, with each summary node containing:

- **Flow Summary:** a high-level textual overview of the microflow.
- **Purpose:** the primary objective of the microflow.
- **Main Actions:** key actions performed within the flow.
- **Parameters:** inputs and outputs involved in the flow.
- **Dependencies:** external elements that the microflow interacts with, capturing relationships and impacts.
- **Return Type:** the type of output or result produced by the microflow.

4) Automatically Construct the Knowledge Graph

The final step involves populating a graph database with the extracted data, represented in the JSON files. Python scripts utilising the Neo4j library⁹ are used to automate this process, ensuring that the knowledge graph is accurately and efficiently constructed. Queries are automatically generated to add extracted entities, attributes, relationships, and summaries from the JSON files to the graph database. Duplicate entities and relationships are

⁵The GPT-4o model has a limit of 128K tokens on its context window [22]. When prompting the API with a larger input, it will crash.

⁶<https://openai.com/>

⁷<https://huggingface.co/spaces/lmarena-ai/chatbot-arena-leaderboard>

⁸<https://platform.openai.com/docs/guides/structured-outputs>

⁹<https://pypi.org/project/neo4j/>

identified and handled to prevent redundancies. Furthermore, after the initial data ingestion, the scripts re-check the graph to verify the existence and validity of all nodes and relationships, ensuring a robust and consistent structure. This automated pipeline provides a seamless and scalable way to construct and maintain the knowledge graph.

Knowledge Graph Database

Our knowledge graph database is built using Neo4j¹⁰, a leading graph database platform designed to store and manage highly connected data. Neo4j is commonly used in research [45, 61] and offers several benefits, including scalability, flexibility in modelling relationships, and optimised performance for graph-based queries, making it well-suited for representing the intricate dependencies and interactions within applications. Furthermore, Neo4j uses the Cypher query language¹¹, a powerful, declarative language specifically designed for querying and manipulating graph data. This is another great benefit because research has demonstrated that state-of-the-art LLM models have a high accuracy in generating correct Cypher queries to interact with graph databases, especially OpenAI’s GPT model [34]. By selecting a database provider that supports Cypher, we ensure that our LLM MAS can effectively retrieve the correct information to answer user questions. This synergy between Neo4j and advanced LLM capabilities not only enhances the system’s ability to deliver precise and contextually relevant responses but also lays the foundation for the integration with CLAIR’s AI-driven component.

6.2.2 LLM-based Multi-Agent System

The LLM-based Multi-Agent System (MAS) [28] represents a core component of the CLAIR architecture, enabling advanced, automated interactions between users and the knowledge graph database. Leveraging Flowise¹², an open-source low-code platform, this system enables us to integrate specialised agents designed to execute distinct roles within a structured workflow. Flowise provides a simplified framework for orchestrating LLM-powered processes, reducing implementation complexity while maintaining robust functionality. Figure 6.5 shows an overview of the LLM-based MAS in Flowise.

MAS Concept and Structure

An agent in the MAS context is a system that employs LLMs to determine the control flow of applications. These agents can be categorised into Single Agents, which perform a single task, and Multi-Agents, which collaborate to address complex tasks [35]. When the context becomes too complex for a single agent to track, dividing the application into smaller, independent agents significantly improves their accuracy and performance [35, 46, 34]. In a MAS the Supervisor Agent serves as the central coordinator, managing communication and task distribution among Worker Agents. It ensures the overall efficiency of the workflow by interpreting agent outputs and directing subsequent steps. This hierarchical structure facilitates collaboration and dynamic task allocation, crucial for handling complex, multi-step processes [35]. Furthermore, LLM-based MAS leverage natural language processing for agent communication, where each agent specialises in specific tasks while interacting with other agents through natural language, significantly enhancing system flexibility [43] and efficiency [96].

¹⁰<https://neo4j.com/>

¹¹<https://neo4j.com/docs/getting-started/cypher/>

¹²<https://flowiseai.com/>

Design Considerations

Key considerations in the MAS design include:

- **Efficient Communication Protocols:** Ensuring seamless agent interactions to maintain workflow integrity [35].
- **Data Transformation Strategies:** Minimising information loss during data preprocessing and transformation while optimising storage and token cost [43].
- **Scalable Architecture:** Supporting dynamic system expansion and real-time processing [35].

Furthermore, we applied iterative enhancement of prompts through automated generation and optimisation. Engineering appropriate prompts for LLMs is a challenging task that demands significant resources and requires expert human input [16]. However, recent advancement in LLMs have made them very suitable for prompt engineering and optimisation [16], and we applied this technique to iteratively generate and enhance the prompts for each agent in the MAS.

Agents and Workflow

The MAS consists of five agents, each fulfilling a critical role in the query-response pipeline:

1. **The Supervisor Agent** orchestrates the entire workflow. It initiates the query process by delegating tasks to other agents and consolidates their outputs into a cohesive response. As the central coordinator, the Supervisor ensures consistency and optimises the system’s performance by monitoring agent interactions and resolving bottlenecks.
2. **The Information Assessor Agent** evaluates user input to identify the specific information required. Based on the user’s query it generates a concise list of data points essential for answering the query, ensuring efficient subsequent processing.
3. **The Cypher Query Generator Agent** translates the Information Assessor’s output into precise Cypher queries compatible with the Neo4j database. This agent leverages the Neo4j graph schema in a zero-shot prompt¹³ to ensure queries are syntactically correct and aligned with the user’s intent.
4. **The Query Executor Agent** interacts with the Neo4j database through a custom built JavaScript tool, executing Cypher queries and using the retrieved data to answers the user’s query.
5. **The Critic Agent** validates the completeness and accuracy of the Query Executor’s output. If the response is incomplete or ambiguous, the Critic initiates a refinement cycle by providing specific feedback to the Supervisor. If the response meets standards, it finalises and delivers the output to the user.

¹³zero-shot prompting refers to the technique of guiding LLMs to perform specific tasks or reasoning processes using only a task description or template, without providing any task-specific examples [42].

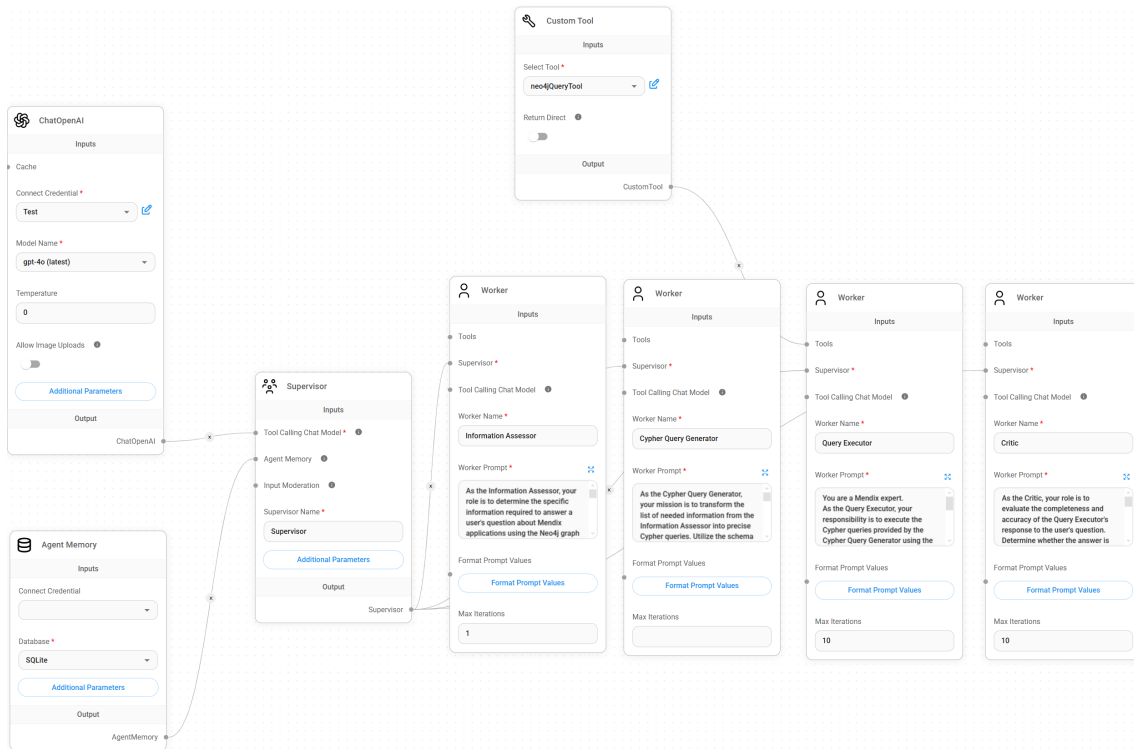


FIGURE 6.5: The LLM-based Multi-Agent System Design in Flowise

Benefits

The LLM-based MAS provides several benefits:

- **Specialisation and Collaboration:** Role-specific agents optimise task execution while collaborative interactions enhance overall efficiency [96].
- **Scalability:** The modular design allows additional agents to be integrated as needed, supporting new functionalities.
- **Improved Context-Aware Query Resolution:** Iterative validation by the Critic ensures comprehensive, accurate responses [46]. Furthermore the Critic ensures that responses contextually relevant, enhancing user satisfaction [35].

By integrating natural language-based interaction, role specialisation, and iterative validation, the MAS forms a flexible, responsive foundation capable of addressing multiple different complex user queries.

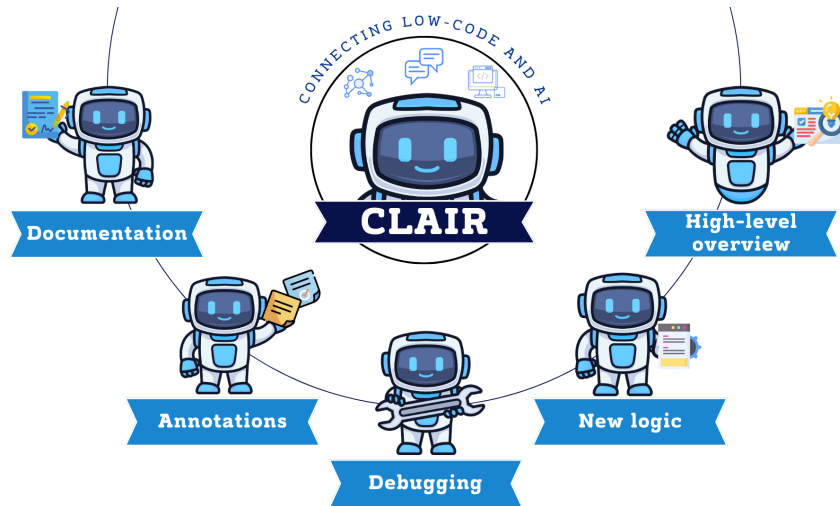


FIGURE 6.6: CLAIR Use Cases

6.3 Use Cases

Although the initial goal and design of CLAIR were aimed at automatically generating documentation, due to LLM’s adaptability and flexibility we have created a system that is capable of serving more purposes than documentation alone. These additional use cases were inspired by research into integrating LLMs and code repositories, such as RED-CODER’s retrieval-augmented code generation and summarisation framework [62], RE-POGRAPH’s repository-level navigation [61], and CodexGraph’s bridging of LLMs with code databases [46]. Together, these studies highlight how advanced graph-based systems can facilitate not only technical insight but also knowledge retrieval and collaborative development. CLAIR leverages these advancements to potentially address challenges across Low-Code development, enhancing Mendix documentation, debugging, and process optimisation. This section presents the practical implementation cases of CLAIR. Through use cases such as on-demand documentation generation, annotation generation, debugging and troubleshooting, new logic generation, and generating high-level overview, the potential capabilities of the system are discussed, demonstrating its ability to handle complex tasks through collaboration between LLM agents. Figure 6.6 visualises the different potential use cases identified.

6.3.1 On-Demand Documentation Generation

The MAS can generate comprehensive and context-aware documentation for Mendix applications on demand. By leveraging the structured knowledge stored in the Neo4j database and the natural language capabilities of the agents, users can retrieve technical documentation detailing the structure and logic of microflows, domain models, modules, and their intricate relationships. This feature ensures up-to-date and accurate documentation tailored to user queries, significantly reducing manual documentation efforts. Additionally, this greatly improves the findability of documentation, as users can directly request the required information instead of searching for it manually. This eliminates the need, for example, for a central wiki with a search engine, and enables the generation of documentation customised to user needs. The system can also include extra explanations or additional information as required by users, addressing a major challenge in traditional documentation where content is often either too detailed, minimal, or irrelevant, from the perspective of

the reader [1]. An example of how a request for documentation looks like is presented in Figure 6.7. This example includes all of the messages sent by the various agents, but in the final implementation this should be hidden for the end-user to improve usability.

6.3.2 Annotation Generation

Apart from generating technical documentation, CLAIR is also capable of generating annotations, which are akin to comments in traditional code environments. Documentation situated close to the source code, or models in Low-Code, has repeatedly been emphasised in literature as critical factor for maintainability and collaboration. Our findings in Chapter 4 highlight that such annotations are frequently needed but are often missing, outdated, or inaccurate. Given that CLAIR has access to the detailed logic of microflows users can request an annotation for a specific action or piece of logic within a microflow. This enables developers to generate more up-to-date, accurate and clear annotations with a consistent structure throughout their project.

6.3.3 Debugging and Troubleshooting

Inspired by the application of LLM’s and code repository for debugging in other papers [46, 61]. We foresee that the system can potentially aid developers in identifying and resolving issues within Mendix applications by:

- Supporting stack traces as input and dissecting them to identify the root cause of the error.
- Allowing users to query specific microflows or components for potential causes of errors.
- Identifying dependencies and their potential impact on the application.
- Generating actionable insights, such as suggested fixes or additional data required for resolution. Therefore enhancing debugging efficiency by providing targeted and relevant information.

6.3.4 New Logic Generation

Next to debugging, based on the flexibility of LLM-based MAS [35], CLAIR should also be able to generate textual descriptions of new microflow logic based on user’s requirements. By utilising existing components within the application, the system ensures that the generated logic is specific and applicable to the current project. This feature is particularly useful for:

- Supporting new Mendix developers in learning the platform.
- Assisting experienced developers with inspiration, validation, or support during the creation of complex logic.
- Minimising the cognitive complexity of understanding and implementing intricate new logic. Research indicates that combining visual representations (such as models) with textual explanations significantly enhances comprehension and retention [49], enabling developers to grasp new complex workflows more effectively.

6.3.5 Generating High-level Overview

Finally, CLAIR addresses the challenge of global sensemaking questions in large-scale applications by employing concepts inspired by the Graph RAG approach outlined by Edge et al. [22]. The concept relates to generating summaries of various hierarchical levels in a graph database in order to generate partial responses which are once again summarised into a final response. In our implementation, this relates to generating high-level descriptions of what modules do and support. These responses are generated based on the previously generated microflow summaries. The hierarchical nature of this summarisation process ensures that the system can handle broad, global sensemaking questions that exceed the context window limitations of traditional LLMs [22]. By structuring data into graph communities and generating both community-level and global summaries, CLAIR achieves a balance between detail and scalability. This capability is valuable for developers, managers, and business analysts, enabling them to dissect and understand the application's architecture in the context of its supported business processes, therefore aiming to support better decision-making and alignment of technical implementations with organisational goals.



FIGURE 6.7: Example of On-Demand Documentation Generation with CLAIR

Chapter 7

CLAIR Validation

This chapter discusses the third step of the DSM according to Wieringa [95] namely, the validation phase. In Section 7.1 the procedure for validating CLAIR is introduced. In terms of the participants selection, the testing setup, the data collection and analysis methodology. Sections 7.2, till 7.6 present the results for each test case. Finally, Section 7.8 discusses our results, relating them to the requirements, goals and validation questions.

7.1 Validation Preparation

This section outlines the methodology and setup we applied for testing and validating CLAIR with Mendix developers. The testing process was designed to evaluate CLAIR's effectiveness, usability, and adaptability in real-world Low-Code development scenarios. By engaging Mendix developers, we aimed to gather actionable insights and feedback to refine and improve the system. The approach aligns with the principles of expert opinion and Technical Action Research (TAR) [95], ensuring both systematic validation and practical applicability.

7.1.1 Objectives

The primary objectives of the testing and validation phase have been:

- Effectiveness evaluation: assess the system's ability to generate accurate, context-aware, and relevant documentation.
- Usability assessment: determine how intuitive and user-friendly CLAIR is for Mendix developers.
- Adaptability measurement: evaluate how well CLAIR addresses diverse developer needs, including debugging, troubleshooting, and logic generation, as discussed in Section 6.3.
- Feedback collection: gather qualitative and quantitative feedback to identify areas for enhancement.

These objectives reflect the iterative cycles emphasised in both DSM and TAR, where stakeholder feedback and measurable outcomes are used to continuously refine the artifact [95].

7.1.2 Validation Questions

To validate the treatment, CLAIR was evaluated in accordance with the guidelines outlined in the DSM [95]. Specifically, Wieringa suggests a series of questions for artifact validation:

- **Effect** (*Artefact X Context = Effects*)
 1. How does CLAIR perform when generating documentation for Mendix applications?
 2. How does CLAIR help improve the documentation in Mendix applications?
 3. How does CLAIR respond to varied developer queries, such as generating microflow documentation or clarifying debugging issues?
 4. To what extent can CLAIR be integrated with existing applications?
- **Trade-off** (*Alternative Artefact X Context = Effects*)
 1. What alternative tools or approaches exist for similar applications?
 2. How does CLAIR compare to these alternatives in terms of usability and effectiveness?
- **Sensitivity** (*Artefact X Alternative Context = Effects*)
 1. What changes occur in CLAIR's usability and effectiveness if applied to larger or smaller Mendix applications?
 2. How can CLAIR be adapted for contexts outside Mendix applications, such as other Low-Code platforms?
 3. What assumptions about Mendix does CLAIR's design depend on, and how do these assumptions affect its adaptability to other contexts?
- **Requirements Satisfaction** - do effects satisfy requirements?
 1. Do CLAIR's capabilities meet the functional requirements outlined in Table 6.1?
 2. Do CLAIR's generated responses match the quality requirements defined in Table 6.2?
 3. Does CLAIR's performance fulfil the non-functional requirements outlined in Table 6.3?

7.1.3 Participant Selection

The participants group consisted of Mendix developers with varying levels of expertise, from junior to senior practitioners. We looked for 10 participants, who were recruited at CAPE Groep's development and support teams, leveraging their diverse expertise in Mendix applications, to gather both quantitative and qualitative insights. This group size and diversity ensured a comprehensive evaluation of CLAIR's capabilities across different skill levels. Furthermore, this inclusive participants selection aligns with TAR's principle of engaging diverse stakeholders to validate practical utility in real-world settings. The distribution of the participants and their expertise can be seen in Figure 7.1.

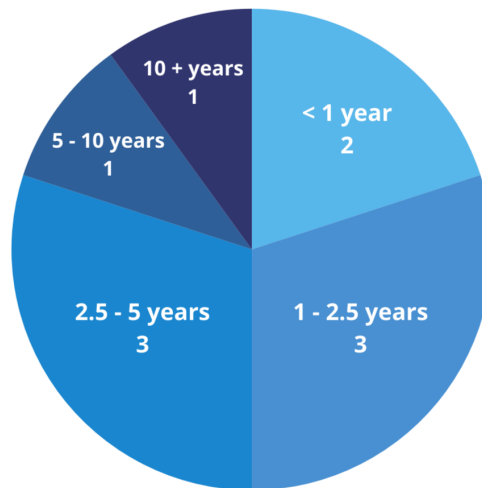


FIGURE 7.1: Participants distribution in terms of years of experience with Mendix.

7.1.4 Testing Plan

To properly test CLAIR, a testing plan to evaluate CLAIR’s performance through structured scenarios with both qualitative and quantitative data collection methods has been designed. This plan consisted of defining various testing scenarios, preparing instructions, and setting up the testing environment.

The testing environment includes the following components:

- **CLAIR Setup:** a fully operational instance of CLAIR integrated with a database populated with real-world Mendix application data chosen by the participant. This means that each participant could choose their own application.
- **Testing Platform:** a Flowise web interface allowing developers to interact with CLAIR, submit queries, and receive generated documentation.
- **Introduction:** an explanation and walkthrough to guide participants in using CLAIR, along with printed out instructions including tips for query formulation to address any unclarities. Furthermore, participants are informed of the current limitations of CLAIR (e.g., no UI elements).

During the testing sessions, participants are instructed to complete five different test scenarios designed to evaluate the different capabilities of CLAIR discussed in Section 6.3:

1. Generate Documentation for a Microflow.
2. Generate Annotations for a Microflow Component.
3. Troubleshoot Issues.
4. Generate New Microflow Logic.
5. High-Level Overview Queries.

A complete description of the testing scenarios can be found in Appendix C.

We provided participants with realistic scenarios and a realistic testing environment, ensuring alignment with TAR’s focus on solving real-world problems while allowing empirical validation of CLAIR’s features [95]. The introduction and structured guidance reflects our focus on usability and empirical data collection, while ensuring the participants are properly equipped to interact effectively with the system.

7.1.5 Data Collection and Analysis

Data collection for the CLAIR validation involved:

- **Interviews:** semi-structured interviews during the testing sessions, to gather qualitative feedback on both user experience and CLAIR’s effectiveness.
- **Post-Scenario Surveys:** following each test scenario, each participant answered 6 questions designed to assess various quality aspects based on the requirements specified in Table 6.2. The participants could add to each answer with comments to clarify their answer or suggest improvements.
- **Post-Test Survey:** participants responded to statements using a Likert scale (1 = Strongly Disagree, 5 = Strongly Agree) to assess usability and satisfaction.

The evaluation metrics used in the surveys were determined using the Goal Question Metric method (GQM) [87]. An overview of the GQM process along with the list of questions for both surveys can be found in the Appendix D.

Data analysis involved:

- Statistical evaluation of quantitative metrics.
- Analysis of qualitative feedback to identify strengths and areas for improvement.
- Comparative analysis of performance across different developer skill levels.

7.2 Test Case 1 - Documentation Generation

This test case focused on evaluating CLAIR’s ability to generate documentation for microflows in Mendix applications. Participants were instructed to generate documentation for three microflows, each with increasing size and complexity. The results are visualised in Figure 7.2.

Correctness

CLAIR demonstrated a high degree of accuracy in generating documentation, with most participants agreeing that the outputs were technically correct and aligned with the provided input. However, occasional issues arose, such as minor errors in interpreting meaning from naming conventions or misrepresenting relationships between entities. While these errors were not frequent, they highlighted a need for refining the system’s dependence on naming conventions. This suggests CLAIR could benefit from additional logic to validate inferred relationships and names, or the injections of semantics to reduce misinterpretation. Inconsistencies in user-defined names could also be an indication of improper naming conventions applied by developers, thereby highlighting an area of improvement for the application.

Test Case 1 - Documentation Generation

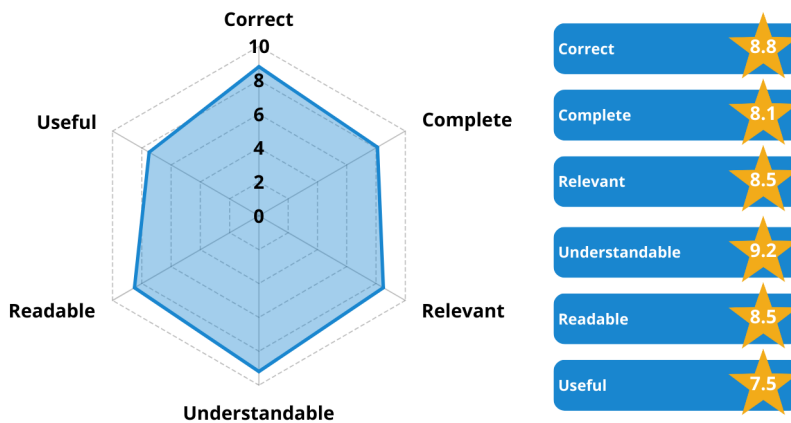


FIGURE 7.2: Test Case 1 - Quantitative results

Completeness

While CLAIR's outputs were generally comprehensive, providing detailed responses, including multi-layered documentation, participants noted that the system sometimes lacked high-level context, focusing more on technical details ("what" happens) in the microflow rather than providing deeper insights into the rationale ("why" something happens). It was pointed out that this could potentially be improved by including business context into the system. Apart from this, participants expressed that the combination of high-level summaries and detailed information was really helpful. However, for larger and more complex microflows, the generated documentation occasionally missed critical interactions with other components.

Relevance

The system scored well in terms of relevance, with participants finding that the generated documentation typically aligned with their queries and provided meaningful information. Various participants praised CLAIR's ability to provide purpose and explain relationships between microflows, especially if they tailored their queries. However, feedback suggested that the system could benefit from prioritising essential details over exhaustive coverage, which sometimes diluted the relevance of the output. Furthermore, some participants remarked that while the generated documentation was relevant, much of the information could be retrieved directly from the microflow. However, this comment was mainly given by the more experienced developers, indicating a difference in perception of ease of understanding complex logic.

Understandability

This was the strongest metric for this test scenario, as participants praised the clarity of the generated documentation. The use of consistent terminology and alignment with Mendix's model structure contributed to its high understandability. Suggestions for improvement

included organising information chronologically to enhance logical flow and ensuring that the most critical details are presented first.

Readability

Participants found the documentation easy to read, with bullet points and structured outputs contributing to its readability. However, the ordering of information could be improved, particularly for complex microflows where chronological ordering of sub-microflows and actions within the flow was suggested to improve logical flow and readability further.

Usefulness

Although usefulness received the lowest score among the metrics, a 7.5 for a prototype system is an encouraging result, especially given the complexity and experimental nature of CLAIR. Participants consistently highlighted the tool's value for onboarding new developers and team members, making it a significant asset for simplifying complex applications or understanding unfamiliar models. CLAIR's functionality for breaking down the purpose and inner workings of large and complex microflows was especially appreciated, with participants recognising its capacity to clarify intricate workflows. Furthermore, the "purpose" section in the documentation stood out as a key feature, offering valuable insights into the rationale and logic behind microflows. Participants noted that this section should remain constant over time for a microflow, serving as a stable reference point for making changes in the future. Experienced developers, who are already familiar with the application, found CLAIR less impactful. However, this was also expected, given that for the test scenarios the participants were very familiar with the application, therefore reducing the need for documentation in general.

Key Insights

- CLAIR performs well for generating technically correct documentation, but further development is needed to eliminate a few edge-case inaccuracies.
- While CLAIR provides granular detail, it could enhance completeness by integrating application-wide context and offering explanations for design choices, although participants did indicate this might be difficult to infer based solely on application structure.
- CLAIR's relevancy could be optimised by balancing the depth of information with the specific focus of the query.
- CLAIR excels in delivering clear and comprehensible documentation, but improved information hierarchy and structure would make the documentation even more user-friendly.
- CLAIR is highly effective as a tool documentation creation and for onboarding, especially for new developers or those tackling complex workflows. However, its utility could be broadened to better serve experienced developers and business stakeholders by incorporating additional features, such as business context and coverage of elements beyond the current scope, such as REST services and Java actions.

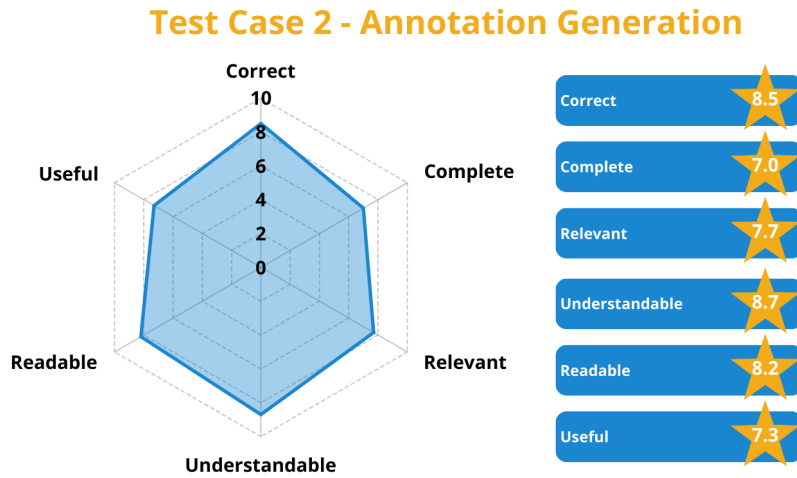


FIGURE 7.3: Test Case 2 - Quantitative results

7.3 Test Case 2 - Annotation Generation

This test case focused on CLAIR’s ability to generate annotations for complex parts of microflows in Mendix applications. Participants were instructed to generate annotations for three different components of a microflow, such as splits, loops or complex database retrieves. The results are visualised in Figure 7.3.

Correctness

CLAIR delivered generally accurate annotations, with participants praising its ability to generate valid responses and, in some cases, include examples to clarify the annotations. The tool’s reliance on naming conventions was both an asset and a limitation. When naming conventions were clear and consistent, CLAIR produced accurate outputs. However, deviations in naming logic occasionally led to incorrect assumptions. Furthermore, some annotations were deemed overly generic and missed the specific nuances of the microflow logic or its interactions with preceding and succeeding microflows.

Completeness

Completeness emerged as the weakest metric for this test case. Participants appreciated when annotations went beyond the immediate logic of the split and incorporated broader application-level insights, although this was inconsistent. While CLAIR excelled at describing "what happens" in a microflow, it often failed to address the "why" behind actions, which participants deemed crucial for effective annotations. Missing context from the broader application or interactions with other components further impacted the perceived completeness. However, CLAIR identified key activities and provided a solid foundation for annotations in almost all cases. Finally, some participants indicated that responses were quite verbose and lacked some precision, which could reduce their usability as quick, targeted explanations.

Relevance

Participants found CLAIR’s annotations relevant to their queries but occasionally too generic or shallow in detail. While its focus on specific components, such as e.g. split conditions, was effective, the lack of deeper insights reduced the overall relevance of the output. Furthermore, relevance often depended on the quality of the query and the user’s ability to provide sufficient context and guidance. Additionally, some more experienced participants felt that CLAIR’s annotations did not significantly add value beyond what they could write themselves, although this was also not the purpose of the use case. CLAIR is supposed to help developers write annotations, not replace their annotations.

Understandability

Once again understandability was a strong point, with participants praising the clarity and straightforward nature of CLAIR’s annotations. Clear phrasing and alignment with Mendix terminology contributed to the system’s high performance in this metric. This metric received no specific criticisms, highlighting CLAIR’s success in producing clear and comprehensible outputs.

Readability

Participants appreciated CLAIR’s structured outputs, such as grouping information into logical sections. However, they noted that the annotation itself was sometimes buried under additional context or explanations, requiring users to sift through the response to find the key information. Therefore, responses could be reordered to prioritise the annotation itself, ensuring it is immediately visible and accessible.

Usefulness

The usefulness of CLAIR’s annotation generation was recognised, particularly for developers who dislike writing annotations. However, experienced developers noted that the generated annotations sometimes lacked the depth or specificity needed for real-world application, indicating their preference to write their own annotations. Although, our findings in Section 4.2 indicate that annotations are currently more often than not missing, outdated, or wrong. Furthermore, various participants indicated that this feature would be greatly enhanced if it was integrated with the development environment, thereby allowing annotations to be generated on-demand during development. This would allow CLAIR to provide a template annotation to which the developer can add some context and or reasoning for the particular design choice.

Key Insights

- CLAIR reliably generates correct annotations, but accuracy can be improved by addressing assumptions based on naming conventions and enhancing its ability to consider broader contextual nuances.
- CLAIR’s completeness could be enhanced by emphasising the rationale behind actions, and maintaining brevity.
- CLAIR can improve relevancy by tailoring annotations more closely to the user’s query and providing greater or lesser depth where necessary.

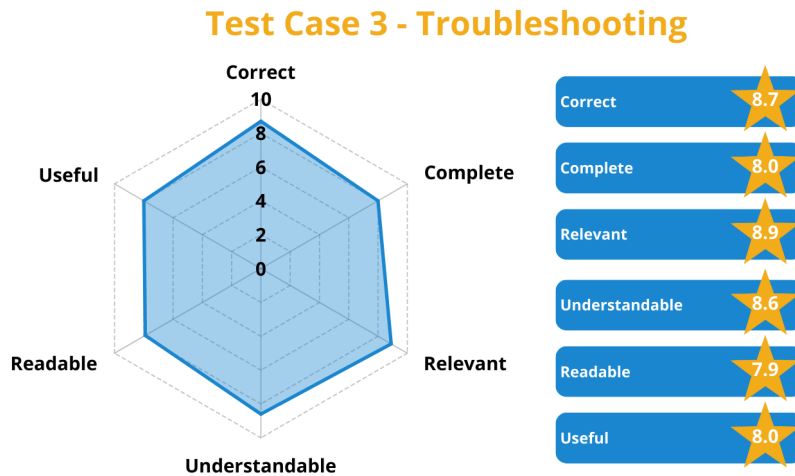


FIGURE 7.4: Test Case 3 - Quantitative results

- CLAIR’s annotations are clear and easy to understand, but focusing on brevity and avoiding verbosity would further enhance their clarity.
- CLAIR should prioritise presenting annotations first, followed by any supporting details, to improve readability.
- CLAIR’s annotations are valuable for reducing repetitive tasks, but integration with the development environment would enhance its utility.

7.4 Test Case 3 - Troubleshooting

This test case evaluated CLAIR’s ability to assist Mendix developers in identifying and resolving issues within applications by providing guidance based on known errors or stack traces. The results are visualised in Figure 7.4.

Correctness

CLAIR demonstrated a high level of accuracy in diagnosing issues and suggesting potential causes. The system consistently generated accurate suggestions based on the error descriptions or stack traces provided. Participants appreciated its ability to identify problematic attributes, pinpoint involved entities, and even anticipate potential future issues based on the provided input. Occasionally, CLAIR included excessive and less relevant information in its responses, which blurred the clarity of the most accurate suggestions. Despite this CLAIR demonstrates great potential for correctly identifying root causes and suggesting solutions.

Completeness

Participants found CLAIR’s responses to be generally complete, offering thorough insights into the potential causes of issues. The system provided a comprehensive breakdown of

potential causes, often exceeding participant expectations in identifying multiple aspects of an issue. However, several participants remarked that CLAIR provided too much information, making it difficult to focus on the most relevant parts. Focusing on likely causes rather than providing exhaustive details could improve usability. Furthermore, responses sometimes required follow-up queries to retrieve context, such as related microflows, which could have been included in the initial response. However, this does conflict with the statements of too much information. Therefore, there is a need to balance here or provide an option to users to hide or see more information.

Relevance

Relevance was one of the highest-scoring metrics in this test case. Participants highlighted CLAIR's ability to focus on the problem described, making its outputs both applicable and actionable. This result was even produced without any instructions by merely copy pasting the stack trace into the chat interface. Participants especially appreciated the tool's ability to prioritise potential causes of issues, ordering them by likelihood. This feature was noted as a significant advantage. Sometimes participants indicated that the tool's suggestions were only partially directly related to the microflow and a bit generic, however, they also pointed out that this could indicate that the problem is not occurring where they requested CLAIR to check.

Understandability

Participants appreciated CLAIR's clear and concise language, which made its responses easy to understand. Furthermore, CLAIR effectively highlighted problem areas in a manner that was easy to comprehend. However, minor adjustments to phrasing could improve the intuitiveness of the troubleshooting advice. A suggestion made by a participant was to frame the troubleshooting advice as a hypotheses, as e.g., "CLAIR thinks this issue may be caused by...", as that would improve the clarity of the suggestions.

Readability

Readability was the lowest-scoring metric for this test case. Although still performing well and participants appreciated the structured responses, some found the information overwhelming or difficult to navigate. Multiple participants felt the most critical information, such as conclusions, was buried under unnecessary details. To this end, participants suggested starting with a clear summary of the recommended solution, followed by supporting details.

Usefulness

Participants recognised CLAIR's potential as a valuable troubleshooting tool, particularly for newer developers or those unfamiliar with the application. Many participants found CLAIR great as a starting point for investigating issues. Some indicated that they currently often use a general LLM, such as ChatGPT, to decipher their stack traces, however given that those do not know the structure of the application, their responses are often too generic. CLAIR gives guidance till the attribute level and even provides working solutions. Additionally, capabilities such as copying stack traces directly into the system for guidance were seen as highly valuable. Another suggestion made during the testing sessions was that it would be very valuable if CLAIR included references to external resources, such as documentation, community knowledge, or Internet searches supplement its guidance.

Overall, the system's ability to guide debugging processes while also suggesting potential solutions was deemed highly useful.

Key Insights

- CLAIR performs well in troubleshooting tasks, but eliminating irrelevant information would enhance the perception of correctness and reliability.
- CLAIR should aim to deliver comprehensive responses in fewer iterations, reducing the need for follow-up queries to refine or complete the information.
- CLAIR excels in generating relevant output for troubleshooting, but could be further improved presenting prioritised solutions first and limiting verbosity in its output.
- CLAIR is a valuable tool for troubleshooting but could be enhanced with additional context and external resource integration to better support developers of all experience levels.

7.5 Test Case 4 - New Logic

This test case evaluated CLAIR's ability to generate detailed descriptions for new microflows based on user-provided requirements. The results are visualised in Figure 7.5

Correctness

Correctness received the lowest score among the metrics for this test case, with participants identifying some technical inaccuracies in CLAIR's outputs. The system generally adhered to the user-provided requirements, and demonstrated a good understanding of user requirements, translating them into microflow logic effectively in many cases. Furthermore, participants appreciated the system's ability to propose innovative logic, even when some steps required refinement. However, some approaches proposed by CLAIR were sometimes technically correct but misaligned with standard best practices. One participant noted: "If you implement this, it will work, however it does not align with Mendix and CAPE best practices, it would be great to enhance the system with this information". On top of this, in some occasions the system suggested unsupported actions, such as grouping and using maps or dictionaries, which are not feasible in Mendix.

Completeness

Completeness was one of the stronger aspects of CLAIR's performance in this test case. CLAIR generated detailed and thorough descriptions, often providing sufficient information to construct a working microflow. Particularly, the system excelled in describing simpler microflows, delivering outputs that were actionable and complete. However, for more complex microflows, some participants noted gaps in details, such as missing XPath¹ constraints in retrieve actions. Furthermore, suggestions to improve completeness included better handling of parameters and integrating checks and error handling directly into the generated microflow logic, as currently the system sometimes adds it as suggestion.

¹Mendix XPath is one of the Mendix query languages designed to retrieve data. XPath uses path expressions to select data of Mendix objects and their attributes or associations. (<https://docs.mendix.com/refguide/xpath/>)

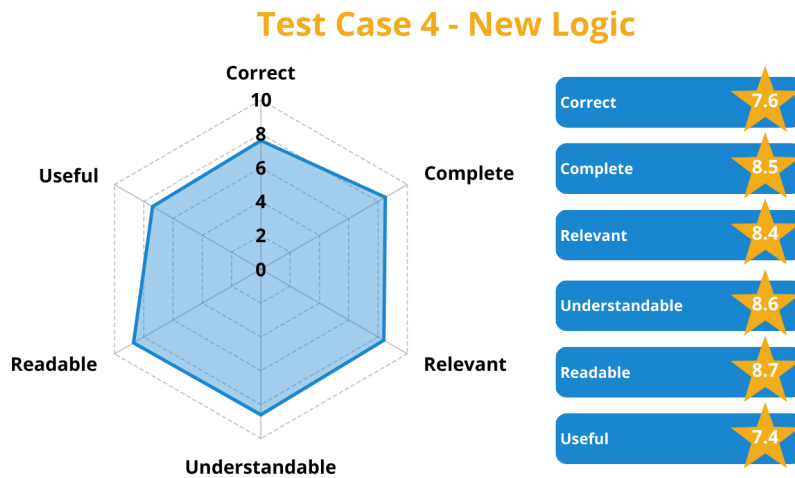


FIGURE 7.5: Test Case 4 - Quantitative results

Relevance

CLAIR demonstrated strong contextual awareness and relevancy, particularly when provided with clear, straightforward prompts. However, there were opportunities to improve the alignment of outputs with user expectations. CLAIR effectively interpreted user requirements and generated relevant responses, making it accessible even with minimal prompting. Furthermore, participants appreciated its adaptability to different levels of detail in prompts. However, some participants felt the need for iterative refinements to improve the relevance of outputs, particularly for complex requirements. Some participants suggested better query guidance or predefined templates to help users frame their requests more effectively and receive outputs closely aligned with their expectations.

Understandability

Participants appreciated the clarity of CLAIR’s logic descriptions, which were easy to follow and understand. The structured language used in the outputs made them accessible to developers of varying skill levels, although minor adjustments to phrasing, such as emphasising the reasoning behind certain actions, could further improve understandability.

Readability

CLAIR’s structured outputs, including step-by-step descriptions and logical grouping, were highlighted as key contributors to readability. Participants noted that CLAIR’s consistent organised formatting enhanced comprehension and reduced the effort required to understand the outputs. However, some participants suggested improvements in the ordering of steps, ensuring that the sequence aligns more with the logical flow of the microflow.

Usefulness

Usefulness received a relatively lower score, reflecting feedback from participants that CLAIR's value varies depending on the user's experience level and the complexity of the task. Overall, CLAIR was viewed as a valuable tool for new developers, offering inspiration and guidance in constructing microflows. Furthermore, participants highlighted its potential as a "co-developer" or "sparring partner" for sparking ideas and validating logic. In contrast, some participants noted that experienced developers might find limited value in CLAIR for simpler microflows, as they could generate similar logic independently in quicker fashion. Also, right now the participants had to put in quite extensive information related to the requirements. Some indicated that if one is able to write such a detailed prompt than you should be able to build the microflow yourself quite easily. However, if the prompt could be written in a more 'business process' like fashion then this would be of great value. Multiple participants suggested an integration possibility with user stories.

Key Insights

- While the tool generally performed well, reducing technical inaccuracies and refining its understanding of Mendix-specific constraints would improve correctness.
- CLAIR's relevance is strong for straightforward use cases, but could be improved for complex queries through better prompt handling and user guidance.
- CLAIR's readability and understandability is strong, with room for minor adjustments in step ordering to improve logical flow.
- CLAIR is a useful tool for supporting new developers but could better cater to experienced developers through advanced features and best-practice integration.
- Participants suggested incorporating more "business process" like prompts or linking the tool to user stories to enhance its applicability.

7.6 Test Case 5 - High-level Questions

This test case assessed CLAIR's ability to handle high-level queries and global sensemaking questions, providing overviews of modules and their purposes. The results are visualised in Figure 7.6.

Correctness

CLAIR demonstrated a strong ability to correctly identify module functionalities and their relationships. The tool effectively extracted meaningful insights from module components and provided an overview of their purpose, which participants found helpful for understanding unfamiliar modules. However, some participants observed that CLAIR occasionally fabricated details based on microflow or module names, which undermined trust in its responses. Next to this, the system sometimes exhibited "tunnel vision" by focusing too narrowly on individual modules, missing critical interactions with other modules.

Completeness

Participants appreciated CLAIR's ability to provide comprehensive module overviews, with some feedback noting that its descriptions involved insights that are difficult to infer from

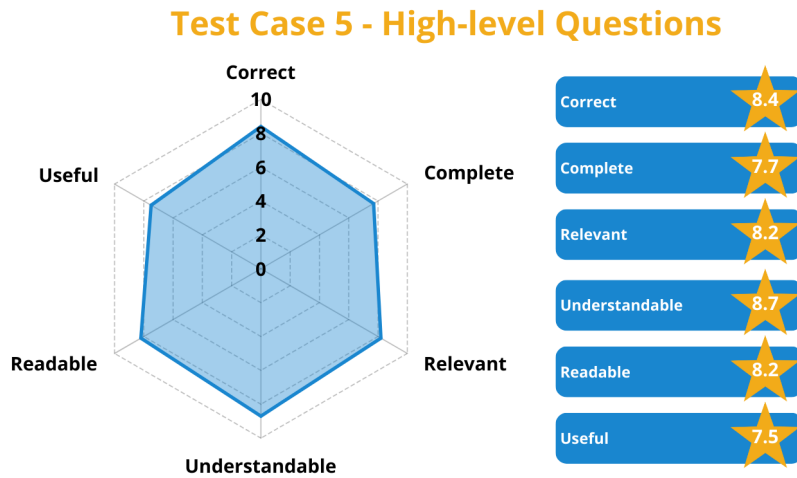


FIGURE 7.6: Test Case 5 - Quantitative Results

the module structure. Furthermore, the system offered detailed insights into module functionality, helping participants get a clear picture of its core actions, although some participants found CLAIR’s responses too detailed for a high-level overview, suggesting it should focus on key concepts and avoid diving too deeply into specifics. Particularly, missing details about inter-module interactions and overarching application context were highlighted as gaps in completeness.

Relevance

CLAIR’s outputs were largely relevant to the queries, providing meaningful information about modules and their purposes. However, there was room to improve alignment with user expectations. CLAIR provided relevant first impressions of modules, particularly for users unfamiliar with their purpose or structure. Therefore, the tool’s ability to extract and summarise module functionality was celebrated as a valuable initial exploration tool in a new application. Despite this, some participants noted that the system sometimes generated overly verbose or too detailed responses, detracting from their relevance. As a result, users suggested providing more guided prompts or predefined templates to improve alignment with specific needs.

Understandability

Understandability was the highest-rated metric of this test case, reflecting CLAIR’s ability to produce clear and concise explanations of modules. Particularly, participants consistently highlighted the clarity of CLAIR’s language and descriptions, making outputs accessible even for less experienced developers. However, some participants noted that sometimes the summarisation was too brief, thereby requiring the reader to have some prior knowledge before fully comprehending the module’s purpose.

Readability

CLAIR's structured responses were generally well-received, but some participants identified opportunities to enhance the flow and organisation of information. Particularly, the use of bullet points and clear logical groupings of module functionalities improved user navigation through responses. However, some participants found the sequence of information less intuitive, suggesting reordering to prioritise the most critical insights.

Usefulness

CLAIR was highly valued for providing quick overviews of modules, particularly for onboarding new team members or understanding unfamiliar applications. Additionally, participants appreciated its ability to uncover details about modules whose names provided little indication of their purpose. However, experienced developers who are already familiar with the application found the tool less useful, as the generated descriptions added limited value beyond what they could infer from domain models. Furthermore, missing a bit of business context was a recurring theme, with participants emphasising the importance of aligning technical details with broader application goals. Therefore, in this case CLAIR's utility lies primarily in onboarding and initial exploration, but enhancing its alignment with business context and inter-module relationships could make it more broadly useful.

Key Insights

- CLAIR demonstrated a strong ability to correctly summarise module functionalities and their relationships.
- Responses could be improved by focusing on high-level insights and cross-module relationships while avoiding unnecessary technical detail.
- CLAIR's outputs could benefit from improved guidance and response filtering to ensure outputs remain focused and meaningful.
- CLAIR's outputs were easy to understand, with minor refinement such as optimising the logical flow of responses and simplifying language needed for broader audiences.
- CLAIR's usefulness is strongest for onboarding and high-level orientation but requires additional contextual integration to appeal to experienced developers and business users.

7.7 Usability and User Experience

The usability and user experience evaluation of CLAIR involved participant responses to several statements on a Likert scale, assessing various aspects of the system’s performance, user-friendliness, and its perceived value in a real-world Mendix development workflow. In this way, the evaluation aligned with the core principles of the extended Technology Acceptance Model (TAM) [88], which is the most frequently used theory to assess user acceptance of AI technologies [36]. The aggregated scores, along with participant comments, provide insights into CLAIR’s strengths and areas for improvement in terms of usability and practicality. Figure 7.7 summarises the results.

Overall Usability

Participants found CLAIR relatively easy to use (3.8) and learn (4.0), reflecting a well-structured interface and straightforward interaction model. The onboarding guidance and structured responses helped new users become familiar with the tool. However, the lower score for ease of use compared to ease of learning suggests that while the system is easier to understand with some guidance, some friction points remain in usage, such as crafting effective prompts or navigating complex outputs. Participants appreciated the chat-based interface but some suggested the addition of predefined query templates to streamline interactions and reduce user effort when crafting effective prompts.

Information Accuracy and Relevance

CLAIR performed well in providing helpful (4.4) and relevant (3.8) responses, with participants particularly praising the clarity and structure of its outputs. However, slightly lower scores for accuracy (3.6) indicate occasional issues, such as fabricated details, reliance on naming conventions, or incomplete context integration. While participants highlighted its helpfulness, ensuring precise and context-aware outputs remains critical for building trust and reliability, which is essential for AI adoption [36].

Efficiency and Organisational Impact

The system demonstrated clear time-saving potential (4.2), with participants noting its ability to generate structured documentation and annotations that would otherwise require significant effort. However, the lower scores for improving organisation (3.7) and traceability (3.6) highlight a need for enhanced features, such as integration with existing documentation workflows or Mendix Studio Pro. However, overall CLAIR’s potential to streamline documentation workflows was a significant strength, with participants seeing value in using it to standardise and organise documentation.

Adoption

Participants were generally positive about incorporating CLAIR into their workflows, especially for onboarding, troubleshooting, and generating documentation for complex applications. Furthermore, the system’s appeal as a co-developer tool for brainstorming or validating ideas was also highlighted. However, some participants noted its limited utility for experienced developers, who may already have in-depth knowledge of their applications, although in this scenario, the deprecation of CLAIR’s value can be attributed to experienced developers not needing any documentation when working with familiar applications.



FIGURE 7.7: Usability and user experience results

7.8 Discussion of the Results

This section discusses the outcomes of the tests presented in Sections 7.2 till 7.6, comparing them against the requirements outlined in Section 6.1 and the validation questions presented in Section 7.1.2. The analysis evaluates whether CLAIR meets its intended objectives and satisfies both functional and non-functional requirements.

7.8.1 Requirements satisfaction

Functional Requirements

Table 6.1 shows 5 functional requirements that CLAIR needed to satisfy in order to achieve the intended purpose. Due to our design choices, we have successfully implemented each of the requirements:

- **FR1** Knowledge Extraction: by leveraging the Mendix Model SDK, CLAIR effectively and accurately extracts domain models, microflows, and their relationships from Mendix applications. Our testing sessions did not reveal any missing or incorrectly extracted data. Therefore this requirements is fully met.
- **FR2** Store Retrievable Knowledge: using the data pipeline, the system efficiently processes these components and represents them in the knowledge graph. By leveraging Cypher queries, the system accurately extracted the requested information. This effective extraction of knowledge in the Neo4j graph database was successfully validated during the tests, ensuring reliable retrieval.
- **FR3** On-Demand Documentation Generation: CLAIR dynamically generates documentation tailored to specific user queries, with high relevance and accuracy confirmed during user testing.
- **FR4** Chat Interface: the built-in Flowise chat interface facilitated smooth interaction with users, which was further improved by locally storing chat history, therefore enabling follow-up questions.
- **FR5** Multi-Purpose Functionality: as extensively discussed in Section 6.3, CLAIR is able to handle a diverse range of queries for various purposes. As the test results show, CLAIR effectively supports multiple documentation types, addressing varied needs across the Low-Code development lifecycle.

Quality Requirements

One of the key factors of a successful validation of CLAIR was to assess the quality of the output generated by the tool. Therefore we specified a list of quality requirements and we assessed them for each test case separately. Overall the results are very promising. In Table 7.1, we present the requirements with the average result across the 5 test scenarios. The average scores ranged from 7.5 to 8.7, thereby we can conclude that each of the quality requirements are fully met. However, concerning up-to-dateness, we currently have to manually update the database, which we did to prepare for testing. Therefore the data and the generated documentation was up-to-date, however to ensure this holds in the future a coupling must be made with a CI/CD or deployment pipeline.

TABLE 7.1: Average results of the quality requirements for the generated documentation

ID	Requirement	Average Result
QR1	Correctness	8.4
QR2	Completeness	7.9
QR3	Relevancy	8.3
QR4	Understandability	8.7
QR5	Readability	8.3
QR6	Usefulness	7.5
QR7	Up-to-dateness	Inherent to the design.

Non-Functional Requirements

Table 6.3 shows 8 non-functional requirements that CLAIR needed to satisfy in order to facilitate a smooth user experience. Most of these requirements were measured during the post-test surveys, and the results are shown in Figure 7.7.

- **NFR1** Usability: as the results from the testing indicate, most users found CLAIR easy to use. Furthermore, they indicated that learning how to use it was also a smooth experience. However, some participants suggested usability improvements, such as incorporating template queries and adding options to hide or show specific details in the outputs. Overall, this requirement has been met.
- **NFR2** Information availability: users indicated that the system is able to provide information on request without any issues. Some minor delays in some responses were observed, however this could be primarily due to the testing setup involving locally running instances of Flowise and Neo4j. This requirement is deemed met, but further testing in a fully deployed environment is recommended for additional validation.
- **NFR3** Helpfulness: the responses generated by CLAIR were rated as helpful, with this quality aspect scoring the highest score. Users appreciated the contextually relevant and accurate information provided during their interactions with the tool. This feedback indicates that the system meets its goal of being helpful and supportive in addressing documentation needs.
- **NFR4** Accuracy: the system’s ability to accurately provide requested information was rated with a mean score of 3.6. While most users found the responses accurate, there were occasional discrepancies or cases where the generated information required clarification. Therefore, this requirement is partially met, where further fine-tuning of the knowledge graph and query handling mechanisms are expected improve this aspect.
- **NFR5** Time Efficiency: CLAIR was reported to save time in writing documentation. Participants emphasised that automation reduced their manual effort significantly, meeting this requirement effectively.

- **NFR6 Scalability:** this requirement was not directly measured during the user tests but was indirectly observed through performance variations across different query types and application sizes. The smallest application (2,151 nodes and 4,322 relationships) and the largest application (8,396 nodes and 16,467 relationships) in the graph database both performed at similar speeds, demonstrating the efficiency and capability of graph databases for managing and querying large datasets.
- **NFR7 Workflow integration:** most users indicated that CLAIR integrates well into their workflows. However, further integration could be achieved by incorporating CLAIR directly into the development environment.
- **NFR8 Flexibility:** CLAIR’s ability to respond to diverse queries and adapt to different documentation needs was demonstrated during the various testing scenarios. While most users were satisfied with the flexibility, further enhancements, such as more tailored outputs for each use case, could strengthen this capability.

7.8.2 Effect

What performance does CLAIR achieve with generating documentation for Mendix applications?

CLAIR demonstrated strong capability in generating high-quality documentation for Mendix applications. The system processes components effectively, with accuracy confirmed during validation. The system demonstrated consistent performance across different application sizes, with no significant delays or resource issues observed even for the largest applications tested. This highlights the efficiency of CLAIR’s graph database architecture. Overall, CLAIR’s performance is effective for its intended use cases, consistently demonstrating efficiency across different application sizes, with no significant delays or resource issues observed even for the largest applications tested.

How does CLAIR help improve the documentation in Mendix applications?

By automating the documentation process, CLAIR significantly reduces manual effort, ensuring timely, accurate and up-to-date documentation at all times. The ability to provide tailored, context-aware documentation makes it easier for developers and business analysts alike to understand and maintain applications. Furthermore, participants highlighted its capability to enhance traceability and organisation, which are critical in continuous development environments [82].

How does CLAIR respond to varied developer queries, such as generating microflow documentation or debugging issues?

CLAIR’s integration with a Multi-Agent LLM System enables effective handling of diverse queries. It generates precise microflow documentation, supports debugging, and provides high-level overviews of modules. User feedback indicated that CLAIR adapts to varied query scenarios, consistently delivering relevant and actionable information.

To what extent can CLAIR be integrated with existing applications?

CLAIR integrates seamlessly with Mendix applications through its use of the Mendix Model SDK. CLAIR’s robust integration capabilities make it well-suited for direct deployment and use without requiring further adaptation for most use cases.

7.8.3 Trade-off

What alternative tools or approaches exist for similar functionalities?

Chapter 5 extensively discussed alternative tools and approaches, including model-driven documentation generation, traditional documentation generation systems, and other LLM based solutions. While many alternatives provide specific features like code smell detection or static documentation, CLAIR uniquely combines knowledge graphs and LLM integration to offer dynamic context-aware outputs.

How does CLAIR compare to these alternatives in terms of usability and effectiveness?

CLAIR's usability scores and participants intention to usage indicate that it performs competitively compared to alternatives. Its ability to provide tailored documentation and handle diverse queries gives it an edge in adaptability. Furthermore, CLAIR's focus on dynamic, context-aware documentation provides distinct value, even though some traditional documentation tools might excel in niche areas such as code commenting. These tools serve complementary purposes rather than being direct competitors.

7.8.4 Sensitivity

What changes occur in CLAIR's usability and effectiveness if applied to larger or smaller Mendix applications?

For smaller applications, CLAIR operates with minimal latency and high efficiency. For larger applications, CLAIR maintained consistent usability and responsiveness, showcasing its ability to handle varying complexity without noticeable performance degradation.

How can CLAIR be adapted for contexts outside Mendix applications, such as other Low-Code platforms?

What assumptions about Mendix does CLAIR's design depend on, and how do these assumptions affect its adaptability to other contexts?

CLAIR's architecture, based on graph databases and LLM integration, is inherently adaptable to other Low-Code platforms. Adjustments to the extraction pipeline would be required to align with other platforms. For example, porting the system to Microsoft PowerApps or OutSystems requires a method to be setup to translate these applications into a structured format, such as JSON. Next to this, CLAIR assumes a specific Mendix meta-model, including domain models, microflows, and their relationships. These assumptions simplify its integration with Mendix but may limit its applicability to platforms with different structures. Adapting CLAIR to other contexts would involve reconfiguring its knowledge graph schema and ensuring compatibility with different platform architectures. Furthermore, CLAIR leverages Mendix's long-standing presence and extensive online documentation. This documentation enhances the knowledge inherent in LLM models, enabling them to reason effectively over Mendix applications. Therefore, porting the system to other platforms would rely on the availability and quality of documentation, potentially reducing the system's reasoning capabilities if adapted for those contexts. Nevertheless, the flexibility inherent in the design highlights CLAIR's potential for being applied to a broader range of Low-Code applications, while acknowledging the effort required for adaptation especially for platforms with less-documented ecosystems.

Chapter 8

Final Remarks

This chapter consolidates the key findings and implications of this research. Section 8.1 discusses the research questions and objectives, interpreting the findings based on the literature review, survey results, system design, and validation. Section 8.2 outlines the theoretical and practical contributions of this research, highlighting its impact on both academia and industry. Section 8.3 addresses the limitations of the study, including methodological considerations, platform dependencies, and areas for improvement. Section 8.4 presents directions for further research, exploring potential enhancements to CLAIR’s functionality, scalability, and adaptability. Finally, Section 8.5 provides the conclusion.

8.1 Discussion

8.1.1 Research Goal

The primary objective of this research was to improve the documentation process and quality for Low-Code applications. The enhanced documentation should lead to improved system understandability, maintainability, sustainability, and knowledge retention. This goal was realised through the development of CLAIR, which has been able to address these goals by automating the generation of context-aware, up-to-date documentation, which helps address the broader challenges of identified in our literature review and survey.

8.1.2 Research Questions

RQ1. What are the key documentation challenges in Low-Code application development?

a) What are the main challenges in documentation in Software development in general?

Software documentation is widely acknowledged as essential for understanding, maintaining, and evolving systems. However, challenges remain, resulting in incomplete, inconsistent, or outdated documents. These issues stem from several factors, including the high costs of creation and maintenance, the perception of documentation as a post-development burden rather than an integral part of the development lifecycle, and difficulties in ensuring key quality attributes such as readability, accuracy, and up-to-dateness. Moreover, practical constraints, such as time limitations and reluctance to write documentation, further widen the gap between documentation needs and actual practices, leading to missing or low-quality content. These general software development challenges set an important baseline for understanding documentation challenges in Low-Code.

(b) How do the unique characteristics of Low-Code impact documentation issues?

Building upon the general challenges, Low-Code’s rapid iteration cycles that often lack robust documentation practices, its visual modelling approaches, and mixed technical and non-technical user base exacerbate documentation problems. Furthermore, unique challenges in Low-Code environments include inadequate traceability between visual elements and their underlying logic, a lack of standardised documentation, and the high cognitive load required to navigate interconnected components. Additionally, documentation is frequently scattered across tools, making it challenging for stakeholders to locate a single source of truth. Overall, the documentation issues are extrapolated in Low-Code, while currently available approaches to help mitigate these issues are not tailored around the specific characteristics of Low-Code.

(c) What are the implications of fragmented, inconsistent, in-adequate documentation for Low-Code applications?

The consequences of these heightened challenges become more evident in Low-Code contexts. Fragmented, inconsistent, or inadequate documentation has significant implications for Low-Code applications, particularly in terms of maintainability, collaboration, and scalability. Without high-quality documentation, developers face increased challenges in understanding application logic, resulting in longer debugging times and higher maintenance costs, creating barriers to effective knowledge sharing among teams, consequently this hinders the onboarding of new developers. Additionally, as Low-Code applications grow in complexity, fragmented documentation becomes a bottleneck for scaling development efforts. Overall, inadequate documentation diminishes the long-term sustainability of Low-Code projects, highlighting the critical need for Low-Code specific solutions.

(d) What are the documentation needs and challenges during each phase of the Low-Code development lifecycle?

As discussed in Section 2.4, the documentation needs and challenges across the Low-Code development lifecycle vary, reflecting the unique demands of each phase. In the design phase, clear and comprehensive documentation is essential to translate business requirements into technical designs. However, documentation at this stage often lacks sufficient detail, creating ambiguities that impact subsequent phases. During development, iterative changes in visual models and rapid development frequently result in documentation that is out-of-sync with the application, making it difficult for developers to trace dependencies. In the testing phase, the lack of detailed documentation hampers validation efforts, reducing the efficiency of quality assurance processes. Deployment introduces its own challenges, as deployment documentation is often neglected in Low-Code, potentially leading to errors and inefficiencies. Finally, the maintenance phase suffers the most from inadequate or outdated documentation, as maintaining complex applications without a clear understanding of their structure and dependencies becomes increasingly difficult over time. Identifying these lifecycle-specific challenges confirms the need for context-specific, real-time documentation to ensure that critical information remains current and accessible to diverse stakeholder needs.

Together, the answers to the sub-questions of RQ1 establish a clear understanding of the underlying and context-specific challenges that automated documentation in Low-Code environments must address. These findings not only justify the need for a specialised automated documentation system but also outline the areas, such as on-demand generation, workflow integration, and up-to-dateness, where automation has the greatest potential impact.

RQ2. To what extent can automated documentation be applied in Low-Code development environments, specifically in the context of complex Mendix applications?

(a) What are the specific documentation needs for different stakeholders in Low-Code?

Having established the overarching documentation challenges, we focused on the diverse range of stakeholders involved in Low-Code development, including developers, citizen developers, and business analysts and their needs. The results of the survey, presented in Chapter 4, show that the needs vary significantly across stakeholders. Developers require detailed technical documentation that provides insights into application logic, dependencies, and implementation specifics to support debugging and development tasks. For citizen developers, documentation must be accessible and easy to understand, with a focus on explaining the functionality of components and processes without requiring deep technical knowledge. Business analysts, in contrast, need documentation that links technical structures to business processes, enabling them to validate requirements and monitor alignment with organisational goals. Additionally, technical support teams require documentation that offers quick access to troubleshooting guides, error descriptions, and solutions to common issues. The challenge lies in providing tailored documentation to these diverse stakeholders while ensuring consistency and accuracy across all outputs. Therefore, these diverse documentation requirements motivate the need for an adaptive, context-sensitive solution.

(b) What are currently available solutions for automated documentation?

In Chapter 5, we discussed existing solutions, for automated documentation, including static tools and more dynamic tools that leverage machine learning or natural language processing to enhance documentation processes. These tools are useful but lack the dynamic and context-aware capabilities required for generating accurate comprehensive documentation. Model-driven approaches offer structured templates and metamodels for creating consistent documentation artifacts. These approaches are particularly effective in ensuring uniformity and clarity. However, they rely heavily on comprehensive and up-to-date models, which can be challenging to maintain in fast-paced development environments like Low-Code. Agile and dynamic documentation solutions align better with rapid iteration cycles by embedding documentation into development workflows. Furthermore, recently, systems powered by LLMs have shown strong potential for producing more natural and expressive documentation. These systems excel at handling diverse documentation queries and generating tailored outputs. However, they bring up challenges, since verbosity, hallucination, and dependence on proprietary infrastructures remain significant obstacles. Furthermore, emerging techniques like Retrieval-Augmented Generation (RAG) and GraphRAG are promising alternative since they demonstrate the value of integrating external knowledge sources with LLMs for more accurate, real-time documentation insights. Despite the advancements in automated documentation, gaps remain in addressing the scalability, customisability, and real-time adaptability required for Low-Code platforms.

(c) What are the design requirements for our automated documentation assistant?

By synthesising the unique demands of Low-Code development and the limitations of existing automated documentation tools we defined key design requirements for CLAIR. As stated in Section 6.1, the system should support knowledge extraction from Low-Code applications to provide a comprehensive understanding of the application structure. It should also enable the storage and retrieval of this knowledge in a graph-based repository, ensuring scalability and fast access to information. Further, CLAIR should support on-demand documentation generation, offering context-aware insights tailored to diverse stakeholder

queries, including both technical and non-technical users. A user-friendly chat interface is essential for enabling intuitive interaction with the system, allowing stakeholders to access relevant documentation without a steep learning curve. Furthermore, the assistant should provide multi-purpose functionality, generating high-quality documentation that supports various phases of the development lifecycle. By adhering to these requirements, CLAIR addresses the specific needs of Low-Code environments, ensuring it delivers value to all stakeholders while improving documentation quality and process efficiency.

These requirements represent the fundamental capabilities an automated assistant must fulfil to overcome the recognised challenges, thereby positioning an automated documentation assistant as a response to RQ2 and linking back to the overarching goal of enhancing documentation quality and processes in Low-Code.

"How can an automated documentation assistant enhance the quality and process of documentation for Low-Code applications?"

This research has shown that an automated documentation assistant can improve the quality and process of documentation for Low-Code applications by systematically addressing key documentation challenges identified in literature discussed in Chapter 2. Furthermore, the results of the survey in Chapter 4 show that effective documentation in Low-Code environments is often hindered by issues such as fragmentation, inconsistency, and outdated information. Automating documentation mitigates these challenges by ensuring comprehensive, real-time updates and by structuring documentation in a way that is accessible, dynamic, and tailored to the diverse needs of stakeholders.

First, automated documentation improves quality by reducing human errors, inconsistencies, and knowledge fragmentation. The survey results revealed that 79% of respondents found existing documentation tools insufficient, citing missing automation features and poor traceability. Addressing these concerns, automated assistants can leverage structured data extraction to systematically capture key artifacts ensuring all relevant components are documented in a standardised and complete manner. This eliminates the risk of missing or outdated documentation, a common problem in agile and iterative development environments.

Second, our findings in Chapter 7 show that by delivering real-time, context-aware insights, an automated assistant empowers various stakeholders to access precisely the information they need. Rather than providing static, one-size-fits-all documentation, these systems can dynamically generate content tailored to specific queries or tasks, improving the clarity and relevance of the documentation. Such flexibility helps organisations adapt their documentation strategy to different phases of the development lifecycle, from initial design through maintenance and support.

Third, seamless integration with the Low-Code development environment ensures that documentation is closely aligned with ongoing development activities. Both literature and survey findings highlighted the difficulty of maintaining up-to-date documentation in agile workflows. Automated assistants can either continuously track changes in application components or update documentation on demand, considerably reducing the gap between the actual system state and what is recorded in the documentation. This approach allows teams to maintain accurate, current information without imposing excessive manual upkeep, thus combating the pervasive issue of outdated documentation.

Additionally, an automated assistant contributes to long-term knowledge retention and maintainability. With a single source of truth that is both comprehensive and easily accessible, development teams can focus on strategic tasks, such as scaling and innovation. As the findings of our validation show, new team members particularly benefit from improved onboarding resources. Furthermore, by structuring documentation in a knowledge graph, developers can navigate complex dependencies and system relationships more effectively.

Overall, for Low-Code environments requiring an automated documentation assistant, this research provides actionable guidelines; achieve structured application data by leveraging metamodel-based extraction techniques, ensure context-aware dynamic documentation generation, integrate the assistant within the development workflow, and finally leverage the inherent flexibility of an LLM-based Multi-Agent System to provide diverse responses tailored to diverse stakeholder needs.

In this research, these principles were operationalised through the development and validation of CLAIR, an automated documentation assistant tailored to the Mendix Low-Code platform. CLAIR leverages the Mendix metamodel to extract key application components and their interrelationships. By integrating a knowledge graph with an LLM-based Multi-Agent System, CLAIR automates documentation generation and provides real-time contextual insights. The prototype was evaluated using Technical Action Research and expert evaluations, presented in Chapter 7, measuring its ability to generate documentation for various use cases. The validation process demonstrated that CLAIR improved documentation completeness, saved users time while writing documentation and overall provided helpful responses to stakeholders with diverse expertise levels. In doing so, CLAIR not only demonstrates the feasibility of automated documentation in Low-Code environments but also highlights its significant benefits: enhanced documentation completeness, ensured up-to-dateness, decreased time constraints, and increased usefulness of documentation. Through structured data extraction, interactive querying, and seamless integration, CLAIR effectively improves the quality and process of Low-Code documentation, addressing key challenges in maintainability, technical debt, sustainability, and system scalability.

8.2 Main Contributions

This research provides contributions to both academia and practice in the domain of Low-Code development and automated documentation. These contributions are outlined below.

8.2.1 Theoretical Contributions

Highlighting the Low-Code Documentation Challenges

To our knowledge, this is the first research to focus specifically on the importance and challenges of documentation in Low-Code environments, underscoring its critical role in reducing technical debt and improving collaboration. This research identified key documentation challenges in software development, and related these challenges to the specific characteristics of Low-Code platforms. By synthesising insights from existing literature and practice, it highlights gaps such as reliance on rapid iteration cycles and visual development environments that often lack robust documentation practices, fragmented documentation, and inadequate traceability, offering a clear roadmap for future research on documentation in Low-Code environments.

Low-Code Developers’ Perspectives on Documentation Issues

Through the survey and case study, this research provides a structured and empirical understanding of how Low-Code developers perceive documentation challenges, particularly in areas such as knowledge retention, maintainability, and accessibility. This comprehensive survey inquired participants across a range of documentation topics such as importance of problems faces, frequency of these issues, their perspective on the current documentation processes and tools, and finally their opinions on the criticality and quality of various document types needed throughout the Low-Code development lifecycle. By identifying the gaps between documentation needs and current practices, this research builds a foundation for addressing the unique characteristics of Low-Code environments, which often diverge from traditional software development practices. These insights contribute to the theoretical understanding of documentation in the lifecycle of Low-Code applications.

Survey of Current Solutions

This research provided an analysis of existing automated documentation generation tools, assessing their strengths and weaknesses while identifying the specific gaps they fail to address in Low-Code platforms. The evaluation identifies key limitations, such as inadequate contextual awareness, limited scalability, and a lack of customisation for the unique requirements of Low-Code environments. These insights not only underline the need for tailored solutions but also positions our proposed solution (CLAIR) as a step forward in addressing these challenges effectively.

8.2.2 Practical Contributions

Innovative State-of-the-Art Solution

The design and implementation of CLAIR provides a practical, automated solution to enhance the documentation process for Low-Code applications, particularly Mendix applications. CLAIR addresses issues such as fragmented, out-dated documentation, and manual effort by offering context-aware, query-based insights tailored to diverse stakeholder needs. To our knowledge, CLAIR is the first solution to combine Low-Code, knowledge graphs, and LLM agents into a unified approach for automated documentation, supporting an innovative and novel approach for addressing documentation challenges in Low-Code environments.

Improved Documentation for Low-Code Platforms

CLAIR enhances the quality, accessibility, and usability of documentation in Low-Code environments. By supporting multiple documentation types (e.g., technical guides, troubleshooting resources), it improves knowledge retention and collaboration among developers, citizen developers, and technical support teams. By automating the extraction and generation of documentation, CLAIR helps reduce technical debt and supports scalable development practices. Its architecture ensures that even large and complex Mendix applications can be documented systematically, improving maintainability and troubleshooting efficiency.

Adaptable Solution Architecture

The solution architecture of CLAIR is not only adaptable to other Low-Code platforms beyond Mendix but also to traditional software development environments. Its graph-based

design and reliance on semi-structured data ensure flexibility for a wide range of applications. Furthermore, the integration of LLM-based Multi-Agent reasoning enhances the system's adaptability, enabling potential applications in domains outside software development, provided the input data is (semi-)structured and relational. Overall, CLAIR's architecture demonstrates a structured approach to augmenting an LLM-based Multi-Agent System with external, interrelated knowledge. This design opens up opportunities for future research, with significant potential to explore its capabilities and applications across various domains.

8.2.3 Broader Implications

Industrial Relevance

CLAIR demonstrates the potential for automated documentation tools to be adopted in real-world Mendix projects, reducing the time and effort required for documentation while improving collaboration and maintainability. The Technical Action Research conducted during this study ensured that CLAIR was tested on a near-implementation level, validating its practical functionality and robustness in realistic settings. This approach demonstrates CLAIR's readiness to be quickly deployed in production environments, making it a valuable tool for organisations seeking to optimise their development workflows. Additionally, its adaptable architecture extends its relevance beyond Mendix, offering opportunities for broader adoption across diverse Low-Code platforms.

8.3 Limitations

8.3.1 Subjectivity and Potential Bias

The study, including the survey and case study, was conducted through the lens of a single researcher. While rigorous guidelines were followed to minimise bias and ensure objectivity, there remains the potential for subjective interpretation or omission of certain insights. The singular perspective introduces a level of bias that could be addressed in future research by involving multiple researchers or peer-reviewed analyses.

8.3.2 Limited Generalisability of Survey Findings

The initial survey conducted during this research was limited to one platform (Mendix) and distributed within a single organisation (CAPE Groep). This approach ensured a clear understanding of process and tool-related issues specific to this context. However, this narrow focus limits the generalisability of the findings to other organisation and Low-Code platforms. Broader distribution across multiple organisations and Low-Code platforms could potentially provide different results and insights.

8.3.3 Validation Constraints and Participants

The validation of CLAIR was conducted in a testing environment with a limited sample size of 10 participants, who were known to the researcher before the validation. While efforts were made to include participants with diverse roles, background, and experience levels within CAPE Groep, this familiarity and limited diversity could introduce bias into the results. Additionally, the testing environment and the limited scope of use cases may not fully capture the variability in challenges and scenarios encountered in broader industry contexts or other organisations. Furthermore, due to time constraints and a lack of

a validated data set, we did not employ a data-driven systematic analysis of the outputs generated by CLAIR, therefore, we have no quantitative evidence that validates the consistency of the generated documentation. However, the consistency of the outputs was qualitatively validated by the author of this thesis by examining the outputs across the different testing sessions.

8.3.4 Reliance on Mendix

First, the current implementation of CLAIR relies on the Mendix metamodel structure, which ensures that the knowledge graph nodes and their relationships accurately reflect the platform architecture. While this alignment optimises CLAIR for Mendix, it limits the tool’s direct applicability to other Low-Code platforms that may use less structured or different metamodels. Adapting CLAIR for such platforms would require modifications to the knowledge graph schema and the data extraction process to accommodate other platform architectures.

Second, CLAIR leverages the extensive knowledge embedded in state-of-the-art LLMs about Mendix, which benefits from the platform’s long-standing presence as a major player in the Low-Code industry. The availability of comprehensive online documentation, tutorials, and community discussions on Mendix enables LLMs to reason effectively within this domain. However, this benefit may not extend to other Low-Code platforms, particularly newer or less-documented ones, of which the LLM’s internal knowledge may be more limited. This discrepancy underscores the importance of evaluating the system’s performance across different platforms and adapting it for contexts where LLM reasoning might require supplementary training or additional external resources.

8.3.5 Exclusion of Key Components

To maintain feasibility and focus, several key elements of Mendix applications, such as UI components, custom Java actions, and external service calls, were excluded from the current implementation. While this streamlined approach allowed us to focus on core logic, incorporating these elements in the future could introduce additional complexity. This may negatively impact the performance and accuracy of CLAIR, particularly during the retrieval and contextualisation of relevant knowledge from the graph. Therefore, the scalability of CLAIR to fully incorporate all aspects of Mendix applications remains an open question.

8.3.6 LLM Model Choice

The solution uses the GPT-4 model due to its demonstrated capability of accurately generating Cypher queries, which is a critical aspect of CLAIR’s functionality. However, the performance of LLMs can vary significantly depending on their usage [59], and this research did not explore the impact of alternative models on CLAIR’s overall performance. Furthermore, due to time and budget constraints, we currently opted for an untrained general purpose LLM. However, research has shown that training and optimising models for a specific purpose potentially increases their accuracy, speed, and reduces token cost [70]. Therefore, a limitation of our research is that we have not investigated how, other possibly locally deployed open-source models trained for the specific task of generating Low-Code documentation perform.

8.4 Future Work

8.4.1 Usability Enhancements

Based on the validation results, improving the usability of CLAIR is a key area for future development. This can include integrating predefined query templates to guide users in framing their queries more effectively, and embedding CLAIR directly into development environments. Such enhancements would streamline user interactions and ensure a more intuitive experience for both technical and non-technical stakeholders.

8.4.2 Cost and Sustainability Optimisation

While on-demand documentation generation provides significant benefits, such as flexibility to user's needs, the token cost and energy consumption associated with LLMs pose challenges for long-term sustainability. Future work could investigate hybrid approaches, such as storing frequently requested documents in a database. This would allow CLAIR to retrieve pre-generated outputs for common queries, reducing the need for repeated LLM processing and minimising both the cost and environmental impact of the system.

8.4.3 Integration with Deployment or CI/CD Pipelines

One of the limitations of CLAIR's current implementation is the static nature of its knowledge graph updates, requiring manual intervention to synchronise with application changes. Future research could explore integrating CLAIR with deployment or CI/CD pipelines to automate this process. To this end, efficient approaches could be developed to detect changes in the application structure and update only the relevant portions of the graph database as well as reprocess the affected summary nodes. This would ensure that documentation remains aligned with the application's current state while avoiding unnecessary processing.

8.4.4 Porting to Other Platforms and Outputs

Porting CLAIR to other Low-Code platforms, such as OutSystems or Microsoft PowerApps, remains an important direction for future research. This would involve tailoring the knowledge graph schema and data extraction pipelines to accommodate the different platform-specific structures. Additionally, CLAIR's outputs could be extended beyond text to include visual representations, such as automatically generated diagrams or models, providing richer and more versatile documentation formats.

8.4.5 Specialisation of LLM-based Multi-Agent System (MAS)

Currently, CLAIR employs a single Multi-Agent System to handle all queries and use cases. Future iterations could explore the development of specialised MAS for specific purposes, such as technical documentation, or troubleshooting. By doing this, we could more strictly determine the steps taken and output generated by each system, making them more aligned with user needs for each use case. A central conversational AI could then delegate queries to the appropriate MAS, potentially improving accuracy, speed, and reliability while optimising the retrieval and response generation process.

8.4.6 Validation Across Platforms and Companies

The documentation issues identified in this research were based on surveys and case studies conducted within a single organisation (CAPE Groep) and a single platform (Mendix). Future studies should validate these findings across different Low-Code platforms and organisations to assess whether the issues identified are universal or platform/organisation-specific. This would improve the generalisability of the results and provide broader insights into Low-Code documentation challenges.

8.4.7 Comparative Analysis of LLM Models

Exploring the performance of various LLM models within CLAIR’s architecture is another promising area for research. A comparative analysis of proprietary and open-source models, including performance and cost trade-offs, could inform the design of future systems. Furthermore, deploying and training locally hosted open-source models could also provide interesting opportunities for future research.

8.4.8 Extending Knowledge in the Graph Database

Currently, CLAIR focuses on core Mendix components, such as domain models and microflows. Future research could investigate how to incorporate additional elements, such as UI components and custom Java actions into the graph database. Capturing these elements would require schema changes and careful consideration of their impact on performance. Furthermore, currently the inputs for CLAIR are the application structure and LLM-generated summaries. Future iterations could enrich the graph database with additional knowledge sources, such as ontologies, business documentation, user feedback, or design decision records. This could improve CLAIR’s ability to link technical structures to business contexts, providing potentially more comprehensive and actionable insights.

8.4.9 Deploying and Expanding CLAIR’s Role

Deploying CLAIR in real-world development environments and gathering longitudinal feedback from users is essential to evaluate its long-term impact, usability, and effectiveness. By observing how teams integrate CLAIR into their workflows over time, we could gain valuable insights. Additionally, the inherent flexibility of the LLM-based MAS can be explored to extend CLAIR’s use cases beyond documentation. For instance, CLAIR’ could potentially be extended to actively assist developers during the application development process by, for example, offering intelligent recommendations for reusable components or suggesting design optimisations. Such enhancements would transform CLAIR from a documentation assistant into a comprehensive development support tool, further amplifying its value for Low-Code development teams.

8.4.10 Creation of a Validated Documentation Dataset

The development of a validated documentation dataset can be a crucial step toward systematically improving and evaluating CLAIR’s outputs. Such a dataset would contain high-quality, thoroughly reviewed documentation samples, providing a benchmark for measuring CLAIR’s outputs. By comparing CLAIR-generated documentation against these validated samples, researchers could obtain quantitative metrics on clarity, accuracy, and completeness using data-driven analysis, in contrast to our reliance on expert validation.

This dataset could also serve as training data for enhanced LLM fine-tuning or reinforcement learning approaches, allowing CLAIR to iteratively improve its output with feedback. Building and curating this dataset would require collaboration with domain experts and standardised evaluation criteria, laying the groundwork for more rigorous, data-driven validation of automated Low-Code documentation solutions.

8.5 Conclusion

This research set out to explore how automated documentation can enhance the quality and efficiency of documentation processes in Low-Code application development. The investigation was driven by the observation that documentation remains a persistent challenge in software development, while these issues are extrapolated in Low-Code environments, particularly as applications scale in complexity. The study systematically examined the key documentation issues faced by Low-Code developers, the extent to which automation can address these challenges, and the design requirements for an effective automated documentation assistant. This investigation culminated in the design and development of CLAIR, a novel documentation assistant that leverages knowledge graphs and an LLM-based Multi-Agent System to generate on-demand context-aware documentation.

While CLAIR has demonstrated its effectiveness in automating documentation for Mendix applications, its broader implications extend beyond the specific case study presented in this research. By structuring application knowledge in a graph-based repository and integrating LLM-based reasoning, CLAIR provides a scalable and adaptable approach to documentation generation that can be refined and expanded for different Low-Code platforms. The validation results showed that automated documentation can save time writing documentation, improve maintainability, support onboarding, and even assist during debugging, making it a viable solution for addressing documentation challenges in dynamic development environments.

However, as with any novel approach, limitations remain. CLAIR’s reliance on Mendix’s metamodel structure restricts its immediate applicability to other Low-Code platforms without adaptation. The exclusion of certain elements such as UI components and external service calls represents an area for future improvement. Additionally, LLM accuracy, cost, and sustainability constraints introduce challenges that must be addressed for long-term viability. Future research should explore optimising CLAIR’s architecture for different platforms, integrating it into CI/CD pipelines, and enhancing its retrieval-augmented generation capabilities. Moreover, the potential of CLAIR extends beyond documentation, by improving its Multi-Agent System, it could evolve into an intelligent development assistant, aiding in debugging, system analysis, and decision-making.

In conclusion, this research highlights the critical role of documentation in Low-Code development and demonstrates that by effectively structuring AI-driven automation, documentation process and quality can be significantly improved. CLAIR represents a step forward in bridging the gap between human expertise and automated knowledge retrieval, paving the way for more intelligent, adaptive, and scalable documentation solutions. By integrating structured data extraction, interactive querying, and context-aware information retrieval, CLAIR not only advances the state of Low-Code documentation but also sets the foundation for future innovations in AI-assisted Low-Code development.

Bibliography

- [1] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C. Shepherd. Software documentation: The practitioners' perspective. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 590–601. ACM, 6 2020. doi:[10.1145/3377811.3380405](https://doi.org/10.1145/3377811.3380405).
- [2] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Marquez, Mario Linares-Vasquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1199–1210. IEEE, 5 2019. doi:[10.1109/ICSE.2019.00122](https://doi.org/10.1109/ICSE.2019.00122).
- [3] Md Abdullah Al Alamin, Sanjay Malakar, Gias Uddin, Sadia Afroz, Tameem Bin Haider, and Anindya Iqbal. An empirical study of developer discussions on low-code software development challenges. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 46–57, 2021. doi:[10.1109/MSR52588.2021.00018](https://doi.org/10.1109/MSR52588.2021.00018).
- [4] Anwar Alqaimi, Patanamon Thongtanunam, and Christoph Treude. Automatically generating documentation for lambda expressions in java. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 310–320. IEEE, 5 2019. doi:[10.1109/MSR.2019.00057](https://doi.org/10.1109/MSR.2019.00057).
- [5] Hana A. Alsaadi, Dhefah T. Radain, Maysoon M. Alzahrani, Wahj F. Alshammari, Dimah Alahmadi, and Bahjat Fakieh. Factors that affect the utilization of low-code development platforms: survey study. *Revista Română de Informatică și Automatică*, 31:123–140, 9 2021. doi:[10.33436/v31i3y202110](https://doi.org/10.33436/v31i3y202110).
- [6] Zaher Alyousef. Challenges development teams face in low-code development process. Master's thesis, Open University, 11 2021. URL: <https://research.ou.nl/en/studentTheses/challenges-development-teams-face-in-low-code-development-process>.
- [7] Scott Ambler. *Agile Modeling: Effective Practice for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc, 2002. doi:<https://dl.acm.org/doi/10.5555/863226>.
- [8] Vard Antinyan, Mirosław Staron, and Anna Sandberg. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering*, 22:3057–3087, 12 2017. doi:[10.1007/s10664-017-9508-2](https://doi.org/10.1007/s10664-017-9508-2).

- [9] Muhammad Arslan, Hussam Ghanem, Saba Munawar, and Christophe Cruz. A survey on rag with llms. *Procedia Computer Science*, 246:3781–3790, 2024. 28th International Conference on Knowledge Based and Intelligent information and Engineering Systems (KES 2024). URL: <https://www.sciencedirect.com/science/article/pii/S1877050924021860>, doi:10.1016/j.procs.2024.09.178.
- [10] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. The agile manifesto, 2001. Agile Alliance. URL: <http://agilemanifesto.org/>.
- [11] Michel Benaroch and Kalle Lyytinen. How much does software complexity matter for maintenance productivity? the link between team instability and diversity. *IEEE Transactions on Software Engineering*, 49:2459–2475, 4 2023. doi:10.1109/TSE.2022.3222119.
- [12] Alexander C. Bock and Ulrich Frank. Low-code platform. *Business & Information Systems Engineering*, 63:733–740, 12 2021. doi:10.1007/s12599-021-00726-8.
- [13] Alessio Bucaioni, Antonio Cicchetti, and Federico Ciccozzi. Modelling in low-code development: a multi-vocal systematic review. *Software and Systems Modeling*, 21:1959–1981, 10 2022. doi:10.1007/s10270-021-00964-0.
- [14] Vikas Chomal and Jatinderkumar Saini. Significance of software documentation in software development process. *International Journal of Engineering Innovation & Research*, 3:410–416, 2014. URL: https://ijeir.org/administrator/components/com_jresearch/files/publications/IJEIR_1009_Final.pdf.
- [15] M. Coram and S. Bohner. The impact of agile methods on software project management. In *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pages 363–370. IEEE, 4 2005. doi:10.1109/ECBS.2005.68.
- [16] Wendi Cui, Jiaxin Zhang, Zhuohang Li, Hao Sun, Damien Lopez, Kamalika Das, Bradley Malin, and Sricharan Kumar. Phaseevo: Towards unified in-context prompt optimization for large language models, 2024. URL: <https://arxiv.org/abs/2402.11347>, arXiv:2402.11347.
- [17] Daniel Dahlberg. Developer experience of a low-code platform: an exploratory study. Master’s thesis, Umeå University, 2020.
- [18] Punyashlok Dash. Analysis of literature review in cases of exploratory research. *SSRN Electronic Journal*, 2019. doi:10.2139/ssrn.3555628.
- [19] Frederico A. de Carvalho and Valdecy Faria Leite. Attribute importance in service quality: an empirical test of the pbz conjecture in brazil. *International Journal of Service Industry Management*, 10:487–504, 12 1999. doi:10.1108/09564239910289021.
- [20] Davide Di Ruscio, Dimitris Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, and Manuel Wimmer. Low-code development and model-driven engineering: Two sides of the same coin? *Software and Systems Modeling*, 21:437–446, 2022. doi:10.1007/s10270-021-00970-2.

- [21] Renato Domingues, Miguel Reis, Miguel Araújo, Marcelo Marinho, and Mário J. Silva. Tracking technical debt in agile low code developments. In *Anais do XXVII Congresso Ibero-Americano em Engenharia de Software (CIbSE 2024)*, pages 226–240. Sociedade Brasileira de Computação, 5 2024. doi:10.5753/cibse.2024.28450.
- [22] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization, 2024. URL: <https://arxiv.org/abs/2404.16130>, arXiv:2404.16130.
- [23] Tim Eichhorn, 2025. URL: <https://github.com/TS-Eichhorn/Low-Code-Developers-Perspective-on-Documentation---Survey-Questions>.
- [24] Carlos Fernandez-Sanchez, Juan Garbajosa, Carlos Vidal, and Agustin Yague. An analysis of techniques and methods for technical debt management: A reflection from the architecture perspective. In *2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics*, pages 22–28. IEEE, 5 2015. doi:10.1109/SAM.2015.11.
- [25] Ulrich Frank, Pierre Maier, and Bock Alexander. Low code platforms: Promises, concepts and prospects: A comparative study of ten systems, 12 2021. URL: <https://doi.org/10.17185/dupublico/47018>, doi:10.17185/dupublico/75244.
- [26] Golar Garousi, Vahid Garousi-Yusifoglu, Guenther Ruhe, Junji Zhi, Mahmoud Mousavi, and Brian Smith. Usage and usefulness of technical software documentation: An industrial case study. *Information and Software Technology*, 57:664–682, 1 2015. doi:10.1016/j.infsof.2014.08.003.
- [27] Vahid Garousi, Ebru Göçmen Ergezer, and Kadir Herkioglu. Usage, usefulness and quality of defect reports. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–6. ACM, 6 2016. doi:10.1145/2915970.2916009.
- [28] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges, 2024. URL: <https://arxiv.org/abs/2402.01680>, arXiv:2402.01680.
- [29] Basit Habib, Rohaida Romli, and Malina Zulkiffi. Identifying components existing in agile software development for achieving “light but sufficient” documentation. *Journal of Engineering and Applied Science*, 70:75, 12 2023. doi:10.1186/s44147-023-00245-1.
- [30] Jens Heidrich, Michael Kläs, Andreas Morgenstern, Pablo Oliveira Antonino, Adam Trendowicz, Jochen Quante, and Thomas Grundler. From complexity measurement to holistic quality evaluation for automotive software development, 10 2021. doi:10.48550/arXiv.2110.14301.
- [31] Arne Henzgen and Lukas Strey. Model-driven approach for automatic model information aggregation in structured documents. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 403–413. IEEE, 10 2023. doi:10.1109/MODELS-C59198.2023.00072.

- [32] Matteus Herinksson. Exploring the use of low-code software development in the automotive industry. Master's thesis, Linköping University, 10 2023. URL: <https://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-201559>.
- [33] Marvin Heuer, Christian Kurtz, and Tilo Böhmann. Towards a governance of low-code development platforms using the example of microsoft powerplatform in a multinational company. In *Proceedings of the 55th Hawaii International Conference on System Sciences*, 1 2022. doi:10.24251/HICSS.2022.831.
- [34] Markus Hornsteiner, Michael Kreussel, Christoph Steindl, Fabian Ebner, Philip Empl, and Stefan Schönig. Real-time text-to-cypher query generation with large language models for graph databases. *Future Internet*, 16:438, 11 2024. doi:10.3390/fi16120438.
- [35] Cheonsu Jeong. Beyond text: Implementing multimodal large language model-powered multi-agent systems using a no-code platform, 2025. URL: <https://arxiv.org/abs/2501.00750>, arXiv:2501.00750.
- [36] Sage Kelly, Sherrie-Anne Kaye, and Oscar Oviedo-Trespalacios. What factors contribute to the acceptance of artificial intelligence? a systematic review. *Telematics and Informatics*, 77:101925, 2023. URL: <https://www.sciencedirect.com/science/article/pii/S0736585322001587>, doi:10.1016/j.tele.2022.101925.
- [37] Junaed Younus Khan and Gias Uddin. Automatic code documentation generation using gpt-3. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6. ACM, 10 2022. doi:10.1145/3551349.3559548.
- [38] Rohit Khankhoje. Beyond coding: A comprehensive study of low-code, no-code and traditional automation. *Journal of Artificial Intelligence & Cloud Computing*, pages 1–5, 12 2022. doi:10.47363/JAICC/2022(1)148.
- [39] Jörg Christian Kirchhof, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Navigating the low-code landscape: A comparison of development platforms. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 854–862. IEEE, 10 2023. doi:10.1109/MODELS-C59198.2023.00135.
- [40] Sebastian Käss, Susanne Strahringer, and Markus Westner. A multiple mini case study on the adoption of low code development platforms in work systems. *IEEE Access*, 11:118762–118786, 2023. doi:10.1109/ACCESS.2023.3325092.
- [41] Timothy C. Lethbridge. Low-code is often high-code, so we must design low-code platforms to enable proper software engineering. In *Leveraging Applications of Formal Methods, Verification and Validation*, pages 202–212, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-89159-6_14.
- [42] Yinheng Li. A practical survey on zero-shot prompt design for in-context learning. In *Proceedings of the Conference Recent Advances in Natural Language Processing - Large Language Models for Natural Language Processings*, page 641–647. INCOMA Ltd., Shoumen, BULGARIA, 2023. URL: http://dx.doi.org/10.26615/978-954-452-092-2_069, doi:10.26615/978-954-452-092-2_069.
- [43] Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, Yun Wang, Linjun Shou, Ming Gong, and Nan Duan.

- Taskmatrix.ai: Completing tasks by connecting foundation models with millions of apis, 2023. URL: <https://arxiv.org/abs/2303.16434>, arXiv:2303.16434.
- [44] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21:466–471, 6 1978. doi:10.1145/359511.359522.
- [45] Shangqing Liu, Yu Chen, Xiaofei Xie, Jingkai Siow, and Yang Liu. Retrieval-augmented generation for code summarization via hybrid gnn, 2021. URL: <https://arxiv.org/abs/2006.05405>, arXiv:2006.05405.
- [46] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Shieh, and Wenmeng Zhou. Codexgraph: Bridging large language models and code repositories via code graph databases, 2024. URL: <https://arxiv.org/abs/2408.03910>, arXiv:2408.03910.
- [47] Yajing Luo, Peng Liang, Chong Wang, Mojtaba Shahin, and Jing Zhan. Characteristics and challenges of low-code development: The practitioners’ perspective. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. ACM, 10 2021. doi:10.1145/3475716.3475782.
- [48] Eder Martinez and Louis Pfister. Benefits and limitations of using low-code development to support digitalization in the construction industry. *Automation in Construction*, 152:104909, 8 2023. doi:10.1016/j.autcon.2023.104909.
- [49] Richard E. Mayer. *Multimedia Learning*. Cambridge University Press, 3 edition, 2020. doi:10.1017/9781316941355.
- [50] Paul W McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 279–290. Association for Computing Machinery, 2014. doi:10.1145/2597008.2597149.
- [51] Joe McKendrick. The rise of the empowered citizen developer, 11 2017. URL: <https://www.dbta.com/DBTA-Downloads/ResearchReports/THE-RISE-OF-THE-EMPOWERED-CITIZEN-DEVELOPER-7575.pdf>.
- [52] Jorge Melegati and Xiaofeng Wang. Case survey studies in software engineering research. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12. ACM, 10 2020. doi:10.1145/3382494.3410683.
- [53] Tom Mens. Research trends in structural software complexity. *CoRR*, 8 2016. doi:10.48550/arXiv.1608.01533.
- [54] Lionel Mew and Daniela Field. A case study on using the mendix low code platform to support a project management course. In *Proceedings of the EDSIG Conference*. ISCAP (Information Systems & Computing Academic Professionals), 2018. URL: <https://iscap.us/proceedings/2018/pdf/4621.pdf>.
- [55] M.B. Miles and A.M. Huberman. *Qualitative Data Analysis: A Sourcebook of New Methods*. SAGE Publications, 1984. URL: <https://books.google.nl/books?id=5AFHAAAAMAAJ>.

- [56] Laura Moreno and Andrian Marcus. Automatic software summarization: the state of the art. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 511–512. IEEE, 5 2017. doi:[10.1109/ICSE-C.2017.169](https://doi.org/10.1109/ICSE-C.2017.169).
- [57] Michael Moser and Josef Pichler. eknows: Platform for multi-language reverse engineering and documentation generation. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 559–568. IEEE, 9 2021. doi:[10.1109/ICSME52107.2021.00057](https://doi.org/10.1109/ICSME52107.2021.00057).
- [58] Lahbib Naimi, El Mahi Bouziane, Abdeslam Jakimi, Rachid Saadane, and Abdellah Chehri. Automating software documentation: Employing llms for precise use case description. *Procedia Computer Science*, 246:1346–1354, 2024. doi:[10.1016/j.procs.2024.09.568](https://doi.org/10.1016/j.procs.2024.09.568).
- [59] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models, 2024. URL: <https://arxiv.org/abs/2307.06435>, arXiv:2307.06435.
- [60] John Ousterhout. *A Philosophy of Software Design*. Yaknyam Press, 1 edition, 11 2021. doi:<https://dl.acm.org/doi/10.5555/3288797>.
- [61] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. Repograph: Enhancing ai software engineering with repository-level code graph, 2024. URL: <https://arxiv.org/abs/2410.14684>, arXiv:2410.14684.
- [62] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Retrieval augmented code generation and summarization, 2021. URL: <https://arxiv.org/abs/2108.11601>, arXiv:2108.11601.
- [63] Boci Peng, Yun Zhu, Yongchao Liu, Xiaohe Bo, Haizhou Shi, Chuntao Hong, Yan Zhang, and Siliang Tang. Graph retrieval-augmented generation: A survey, 2024. URL: <https://arxiv.org/abs/2408.08921>, arXiv:2408.08921.
- [64] Benny Petersson and William Evans. How does low-code development correspond with best practice in software development? Master’s thesis, Malmö University, 6 2023. URL: <https://www.diva-portal.org/smash/get/diva2:1766201/FULLTEXT02.pdf>.
- [65] Huy N. Phan, Hoang N. Phan, Tien N. Nguyen, and Nghi D. Q. Bui. Repohyper: Search-expand-refine on semantic graphs for repository-level code completion, 2024. URL: <https://arxiv.org/abs/2403.06095>, arXiv:2403.06095.
- [66] Reinhold Plosch, Andreas Dautovic, and Matthias Saft. The value of software documentation quality. In *2014 14th International Conference on Quality Software*, pages 333–342. IEEE, 10 2014. doi:[10.1109/QSIC.2014.22](https://doi.org/10.1109/QSIC.2014.22).
- [67] Hengameh Rezaei, Filippa Ebersjo, Kristian Sandahl, and Mirosław Staron. Identifying and managing complex modules in executable software design models-empirical assessment of a large telecom software product. In *2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*, pages 243–251. IEEE, 10 2014. doi:[10.1109/IWSM.Mensura.2014.27](https://doi.org/10.1109/IWSM.Mensura.2014.27).

- [68] Martin P. Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Charro, Neil Ernst, Marco Aurelio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vasquez, Gail C. Murphy, Laura Moreno, David Shepherd, and Edmund Wong. On-demand developer documentation. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 479–483. IEEE, 9 2017. doi:10.1109/ICSME.2017.17.
- [69] Karlis Rokis and Marite Kirikova. Challenges of low-code/no-code software development: A literature review. In *Perspectives in Business Informatics Research*, pages 3–17. Springer International Publishing, 2022. doi:10.1007/978-3-031-16947-2_1.
- [70] Zhyar Rzgar K. Rostam, Sándor Szénási, and Gábor Kertész. Achieving peak performance for large language models: A systematic review. *IEEE Access*, 12:96017–96050, 2024. doi:10.1109/ACCESS.2024.3424945.
- [71] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 171–178. IEEE, 8 2020. doi:10.1109/SEAA51224.2020.00036.
- [72] Gayane Sedrakyan, Maria Eugenia Iacob, and Jos van Hillegersberg. Towards lowdevsecops framework for low-code development: Integrating process-oriented recommendations for security risk management. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24)*. ACM, 9 2024. doi:10.1145/3652620.3688335.
- [73] Jiho Shin, Reem Aleithan, Hadi Hemmati, and Song Wang. Retrieval-augmented test generation: How far are we?, 2024. URL: <https://arxiv.org/abs/2409.12682>, arXiv:2409.12682.
- [74] Lakmal Silva, Michael Unterkalmsteiner, and Krzysztof Wnuk. Towards identifying and minimizing customer-facing documentation debt. In *2023 ACM/IEEE International Conference on Technical Debt (TechDebt)*, pages 72–81. IEEE, 5 2023. doi:10.1109/TechDebt59074.2023.00015.
- [75] Xiaotao Song, Hailong Sun, Xu Wang, and Jiafei Yan. A survey of automatic generation of source code comments: Algorithms and techniques. *IEEE Access*, 7:111411–111428, 2019. URL: <http://dx.doi.org/10.1109/ACCESS.2019.2931579>, doi:10.1109/access.2019.2931579.
- [76] Christoph Johann Stettina and Werner Heijstek. Necessary and neglected? an empirical study of internal documentation in agile software development teams. In *Proceedings of the 29th ACM International Conference on Design of Communication, SIGDOC '11*, page 159–166, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2038476.2038509.
- [77] Chia-Yi Su and Collin McMillan. Distilled gpt for source code summarization. *Automated Software Engineering*, 31:22, 5 2024. doi:10.1007/s10515-024-00421-4.
- [78] Shams Tabrez and Islam Jan. Documentation and agile methodology. Master’s thesis, Uppsala University, 12 2013. URL: <https://www.diva-portal.org/smash/get/diva2:678784/FULLTEXT01.pdf>.

- [79] Gytė Tamašauskaitė and Paul Groth. Defining a knowledge graph development process through a systematic review. *ACM Transactions on Software Engineering and Methodology*, 32:1–40, 1 2023. doi:10.1145/3522586.
- [80] H Tang and S Nadi. Evaluating software documentation quality. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 67–78, 2023. doi:10.1109/MSR59073.2023.00023.
- [81] Winston Tellis. Application of a case study methodology. *The Qualitative Report*, 9 1997. doi:10.46743/2160-3715/1997.2015.
- [82] Theo Theunissen. Identifying conditions for effective communication with just enough documentation in continuous software development. In *CAiSE (Doctoral Consortium)*, pages 11–20, 2020. URL: <https://ceur-ws.org/Vol-2613/paper2.pdf>.
- [83] Theo Theunissen, Stijn Hoppenbrouwers, and Sietse Overbeek. In continuous software development, tools are the message for documentation. In *Proceedings of the 23rd International Conference on Enterprise Information Systems*, pages 153–164. SCITEPRESS - Science and Technology Publications, 4 2021. doi:10.5220/0010367901530164.
- [84] Theo Theunissen, Uwe van Heesch, and Paris Aygeriou. A mapping study on documentation in continuous software development. *Information and Software Technology*, 142:106733, 2 2022. doi:10.1016/j.infsof.2021.106733.
- [85] Massimo Tisi, Jean-Marie Mottu, Dimitrios S Kolovos, Juan de Lara, Esther Guerra, Davide Di Ruscio, Alfonso Pierantonio, and Manuel Wimmer. Lowcomote: Training the next generation of experts in scalable low-code engineering platforms. In *International Conference on Software Technologies: Applications and Foundations*, 2019. URL: <https://api.semanticscholar.org/CorpusID:196613731>.
- [86] Burak Uyank and Ahmet Sayar. Analysis and comparison of automatic code generation and transformation techniques on low-code platforms. In *2023 5th International Conference on Software Engineering and Development (ICSED)*, pages 17–27. ACM, 10 2023. doi:10.1145/3637792.3637795.
- [87] Rini van Solingen (Revision), Vic Basili (Original article, 1994 ed.), Gianluigi Caldiera (Original article, 1994 ed.), and H. Dieter Rombach (Original article, 1994 ed.). *Goal Question Metric (GQM) Approach*. John Wiley & Sons, Ltd, 2002. doi:10.1002/0471028959.sof142.
- [88] Viswanath Venkatesh and Fred D. Davis. A theoretical extension of the technology acceptance model: Four longitudinal field studies. *Management Science*, 46(2):186–204, 2000. arXiv:<https://doi.org/10.1287/mnsc.46.2.186.11926>, doi:10.1287/mnsc.46.2.186.11926.
- [89] Colin C. Venters, Rafael Capilla, Stefanie Betz, Birgit Penzenstadler, Tom Crick, Steve Crouch, Elisa Yumi Nakagawa, Christoph Becker, and Carlos Carrillo. Software sustainability: Research and practice from a software architecture viewpoint. *Journal of Systems and Software*, 138:174–188, 4 2018. doi:10.1016/j.jss.2017.12.026.
- [90] Paul Vincent, Kimihiko Lijima, Mark Driver, Jason Wong, and Yefim Natis. Magic quadrant for enterprise low-code application platforms, 8 2019.

- [91] Stefan Voigt, Detlef Hüttemann, Andreas Gohr, and Michael Große. Agile documentation tool concept. In *Developments and Advances in Intelligent Systems and Applications*, pages 67–79. Springer International Publishing, 2018. doi:10.1007/978-3-319-58965-7_5.
- [92] Stefan Voigt, Susanne Kaufmann, and Christina Maischak. Successful interorganizational collaboration through structured wikis: : A case study from a german knowledge transfer project. In *2022 International Conference on Advanced Enterprise Information System (AEIS)*, pages 127–137. IEEE, 12 2022. doi:10.1109/AEIS59450.2022.00025.
- [93] Gerard Wagenaar, Sietse Overbeek, Garm Lucassen, Sjaak Brinkkemper, and Kurt Schneider. Working software over comprehensive documentation – rationales of agile teams for artefacts usage. *Journal of Software Engineering Research and Development*, 6:7, 12 2018. doi:10.1186/s40411-018-0051-7.
- [94] Chao Wang, Hong Li, Zhigang Gao, Min Yao, and Yuhao Yang. An automatic documentation generator based on model-driven techniques. In *2010 2nd International Conference on Computer Engineering and Technology*, volume 4, pages V4–175–V4–179, 2010. doi:10.1109/ICCET.2010.5485654.
- [95] Roel J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-662-43839-8.
- [96] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryan W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation, 2023. URL: <https://arxiv.org/abs/2308.08155>, arXiv:2308.08155.
- [97] Robert K. Yin. *Case Study Research and Applications*. Sage Inc, 6 edition, 1 2017.
- [98] Minde Zhao, Zhaohui Wu, Guoqing Yang, Lei Wang, and Wei Chen. Smartosek: A real-time operating system for automotive electronics. In *Embedded Software and Systems*, pages 437–442, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/11535409_63.
- [99] Junji Zhi, Vahid Garousi-Yusifoglu, Bo Sun, Golara Garousi, Shawn Shahnewaz, and Guenther Ruhe. Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software*, 99:175–198, 1 2015. doi:10.1016/j.jss.2014.09.042.

Appendix A

Extra results survey

TABLE A.1: Reasons for not using Business Case

Why do you not use a Business Case? (multiple answers allowed)	Count (N=4)
<i>I don't find it valuable</i>	1
<i>It is incomplete or incorrect</i>	1
<i>It is outdated</i>	1
<i>I can't find it</i>	1
<i>I can't understand it</i>	1
<i>Other</i>	1: "Not used yet"

TABLE A.2: Reasons for not using Business Process Documentation

Why do you not use Business Process Documentation? (multiple answers allowed)	Count (N=5)
<i>It does not exist</i>	1
<i>It is unavailable for me</i>	1
<i>I can't find it</i>	1
<i>Other</i>	3: "The customer doesn't see the value so it does not get implemented.", "Personally i like to make my own", "Often not relevant"

TABLE A.3: Reasons for not using Requirements Documentation

Why do you not use Requirements Documentation? (multiple answers allowed)	Count (N=6)
<i>It does not exist</i>	4
<i>It is unavailable for me</i>	1
<i>It is incomplete or incorrect</i>	1
<i>It is outdated</i>	1
<i>I can't find it</i>	4
<i>Other</i>	1: "not relevant for me"

TABLE A.4: Reasons for not using Roadmap & Release Plan

Why do you not use Roadmap & Release Plan? (multiple answers allowed)	Count (N=11)
<i>I don't find it valuable</i>	2
<i>It does not exist</i>	6
<i>It is unavailable for me</i>	5
<i>It is incomplete or incorrect</i>	2
<i>It is outdated</i>	4
<i>I can't find it</i>	1

TABLE A.5: Reasons for not using Architecture Design Documentation

Why do you not use (Architecture) Design Documentation? (multiple answers allowed)	Count (N=4)
<i>I don't find it valuable</i>	1
<i>It does not exist</i>	1
<i>It is unavailable for me</i>	1
<i>It is outdated</i>	1
<i>Other</i>	2: "not yet used", ""

TABLE A.6: Reasons for not using Mockups and UI Documentation

Why do you not use Mockups and UI Documentation? (multiple answers allowed)	Count (N=16)
<i>I don't find it valuable</i>	3
<i>It does not exist</i>	7
<i>It is unavailable for me</i>	4
<i>It is incomplete or incorrect</i>	1
<i>It is outdated</i>	3
<i>Other</i>	5: "The customer doesn't see the value so it does not get implemented", "not used yet", "We don't have the skills to create these documents", "Not relevant", "Not relevant for me"

TABLE A.7: Reasons for not using Data Model Documentation

Why do you not use Data Model Documentation? (multiple answers allowed)	Count (N=12)
<i>It does not exist</i>	5
<i>It is unavailable for me</i>	3
<i>It is incomplete or incorrect</i>	1
<i>It is outdated</i>	3
<i>I can't find it</i>	3
<i>I can't understand it</i>	1
<i>Other</i>	1: "not relevant for me"

TABLE A.8: Reasons for not using Application Logic Documentation

Why do you not use Application Logic Documentation? (multiple answers allowed)	Count (N=15)
<i>I don't find it valuable</i>	2
<i>It does not exist</i>	8
<i>It is incomplete or incorrect</i>	2
<i>It is outdated</i>	2
<i>I can't find it</i>	4
<i>I can't understand it</i>	2
<i>Other</i>	2: "flow should be easy enough you do not need it", "Instead use annotations, they make the documentation findable where it is needed"

TABLE A.9: Reasons for not using Source Model Comments

Why do you not use Source Model Comments? (multiple answers allowed)	Count (N=5)
<i>I don't find it valuable</i>	2
<i>It does not exist</i>	2
<i>It is incomplete or incorrect</i>	2
<i>It is outdated</i>	2
<i>I can't understand it</i>	1

TABLE A.10: Reasons for not using Component Documentation

Why do you not use Component Documentation? (multiple answers allowed)	Count (N=15)
<i>I don't find it valuable</i>	2
<i>It does not exist</i>	5
<i>It is unavailable for me</i>	2
<i>It is incomplete or incorrect</i>	1
<i>It is outdated</i>	2
<i>I can't find it</i>	2
<i>I can't understand it</i>	1
<i>Other</i>	1: "not relevant for me"

TABLE A.11: Reasons for not using Dependencies Documentation

Why do you not use Dependencies Documentation? (multiple answers allowed)	Count (N=17)
<i>I don't find it valuable</i>	4
<i>It does not exist</i>	6
<i>It is unavailable for me</i>	3
<i>It is incomplete or incorrect</i>	2
<i>It is outdated</i>	4
<i>I can't find it</i>	1
<i>I can't understand it</i>	1
<i>Other</i>	3: "not relevant for me", "Not my role", ""

TABLE A.12: Reasons for not using Maintenance Documentation

Why do you not use Maintenance Documentation? (multiple answers allowed)	Count (N=9)
<i>I don't find it valuable</i>	2
<i>It does not exist</i>	3
<i>It is unavailable for me</i>	1
<i>It is incomplete or incorrect</i>	1
<i>It is outdated</i>	1
<i>I can't find it</i>	1
<i>Other</i>	2: "not used yet", ""

TABLE A.13: Reasons for not using Service Level Agreements

Why do you not use Service Level Agreements? (multiple answers allowed)	Count (N=12)
<i>I don't find it valuable</i>	5
<i>It is unavailable for me</i>	2
<i>I can't find it</i>	1
<i>Other</i>	5: "Useful for support, not for me", "I don't need to deal with SLA's, I leave that to support", "I do not perform outside work hours services for clients", "Don't have to use it for now.", "I've mostly been involved in project that aren't live yet recently, so it has not been relevant for a while"

TABLE A.14: Reasons for not using Issue Tracking Logs

Why do you not use Issue Tracking Logs (multiple answers allowed)	Count (N=7)
<i>I don't find it valuable</i>	1
<i>It does not exist</i>	1
<i>It is unavailable for me</i>	1
<i>It is outdated</i>	1
<i>Other</i>	4: "My projects don't use tickets", "Clients communicate issues directly and fix them via devops user story's", "Don't have to use it for now", ""

TABLE A.15: Reasons for not using User Manuals

Why do you not use User Manuals (multiple answers allowed)	Count (N=12)
<i>I don't find it valuable</i>	2
<i>It does not exist</i>	5
<i>It is unavailable for me</i>	4
<i>It is incomplete or incorrect</i>	1
<i>It is outdated</i>	3
<i>I can't find it</i>	1
<i>Other</i>	1: "Don't have to use it for now"

Appendix B

Knowledge Graph Schema Details

Nodes

- Module:
 - id (String): Unique identifier of the module
 - name (String): Name of the module
- Folder:
 - name (String): Name of the folder
 - subfolders (String[]): Array of subfolders
- DomainModel:
 - id (String): Unique identifier of the domain model
 - name (String): Name of the domain model
 - documentation (String): Description or additional information about the domain model manually added by developers.
- Microflow:
 - id (String): Unique identifier of the microflow
 - name (String): Name of the microflow
 - documentation (String): Description or additional information about the microflow manually added by developers.
 - json (String): JSON representation of the microflow, capturing actions, activities, flow, and other details.
- Entity:
 - id (String): Unique identifier of the entity
 - name (String): Name of the entity
 - documentation (String): Description or additional information about the entity manually added by developers.
 - persistable (String): Indicates whether the entity is persisted to the database.

- Attribute:
 - id (String): Unique identifier of the attribute
 - name (String): Name of the attribute
 - documentation (String): Description or additional information about the attribute manually added by developers.
 - type (String): Data type of the attribute (e.g., String, Integer, Enumeration).
 - value (String): Default value of the attribute, if applicable.
- Enumeration:
 - id (String): Unique identifier of the enumeration
 - name (String): Name of the enumeration
 - documentation (String): Description or additional information about the enumeration manually added by developers.
 - values (String[]): Array of possible values for the enumeration.

Relationships

- CONTAINS:
 - Source: Module or Folder or DomainModel
 - Target: Folder or Microflow or DomainModel or Entity or Enumeration
- INTERACTS:
 - Source: Microflow
 - Target: Entity or Attribute
 - Properties:
 - * interactionType (String): Type of interaction between the microflow and the target (e.g., "create", "read", "update", "delete").
- CALLS:
 - Source: Microflow
 - Target: Microflow
- GENERALIZATION:
 - Source: Entity
 - Target: Entity
- ASSOCIATED_WITH:
 - Source: Entity
 - Target: Entity
 - Properties:
 - * name (String): Name of the association

- * type (String): Type of association, either "Reference" or "ReferenceSet"
- * owner (String): Specifies the multiplicity, either "both" (1-to-1) or "default" (1-to-*)

- HAS_ATTRIBUTE:

- Source: Entity
- Target: Attribute

- HAS_ENUMERATION:

- Source: Attribute
- Target: Enumeration

Appendix C

Testing Scenarios

During the testing sessions participants complete five test scenarios designed to evaluate different capabilities of CLAIR discussed in Section 6.3.

1. Generate Documentation for a Microflow
 - Choose three microflows of varying complexity.
 - Query CLAIR to generate detailed documentation.
 - Assess the documentation’s accuracy, completeness, and clarity.
2. Generate Annotations for a Microflow Component
 - Identify three complex parts of a microflow (e.g., splits, loops).
 - Request annotations for these components.
 - Evaluate the quality of the generated annotations.
3. Troubleshoot Issues
 - Provide CLAIR with stack traces or descriptions of two known issues.
 - Assess the system’s ability to guide problem resolution and suggest fixes.
4. Generate New Microflow Logic
 - Provide CLAIR with requirements for a new microflow.
 - Assess the generated description for completeness and applicability to the specified requirements.
5. High-Level Overview Queries
 - Query CLAIR for high-level summaries of three modules.
 - Evaluate the generated overviews for accuracy, relevance, and usefulness in understanding module purposes.

Appendix D

Goal Question Metric Process & Survey Questions

This appendix provides an overview of the Goal Question Metric (GQM) [87] process used in this research to evaluate CLAIR’s performance. The GQM method was employed to systematically assess the effectiveness, usability, and quality of CLAIR’s generated documentation based on predefined evaluation criteria. The structured approach of GQM ensures that the collected data aligns with the research objectives and allows for a comprehensive analysis of CLAIR’s capabilities.

The GQM methodology, introduced by Basili et al. [87], is a structured framework for defining and evaluating software quality. It follows a hierarchical approach where:

1. **Goals** define the high-level objectives of the evaluation.
2. **Questions** are formulated to assess whether these goals are met.
3. **Metrics** are established to quantify the responses to these questions.

For this study, the GQM framework was applied to two main areas:

D.1 Quality of Generated Documentation

Goal

Assess the effectiveness of CLAIR in generating high-quality documentation for Low-Code applications.

Questions

1. How correct is the documentation generated by CLAIR?
2. How complete is the generated documentation?
3. How relevant is the generated documentation to the user’s needs?
4. How understandable is the generated documentation ?
5. How readable is the documentation in terms of formatting and clarity?
6. How useful is the generated documentation?

Metrics

- Correctness: participant ratings on a scale from 1 (Very Poor) to 10 (Excellent).
- Completeness: participant ratings on a scale from 1 (Very Poor) to 10 (Excellent).
- Relevancy: participant ratings on a scale from 1 (Very Poor) to 10 (Excellent).
- Understandability: participant ratings on a scale from 1 (Very Poor) to 10 (Excellent).
- Readability: participant ratings on a scale from 1 (Very Poor) to 10 (Excellent).
- Usefulness: participant ratings on a scale from 1 (Very Poor) to 10 (Excellent).

Additionally participants were encouraged to provide qualitative comments to elaborate on their scores, highlighting any missing details, inconsistencies, or areas for improvement.

D.2 Usability & User Experience

Goal

Evaluate the usability and user experience of CLAIR and assessing how well it integrates into Low-Code development workflows.

Questions

For this component, we opted to provide statements to the participants, to which they responded with their level of agreement on a Likert scale.

1. "The system was easy to use."
2. "The system is able provides the requested information"
3. "The responses/documentation were helpful."
4. "The system accurately provides the required information"
5. "The system would save me time writing documentation."
6. "The system improves the organisation of documentation"
7. "The system was easy to learn how to use."
8. "The system improves traceability of the documentation"
9. "I would use this tool in my workflow."

Metrics

- Ease of Use: Likert scale from 1 (Strongly Disagree) to 5 (Strongly Agree).
- Information Retrieval: Likert scale from 1 (Strongly Disagree) to 5 (Strongly Agree).
- Helpfulness: Likert scale from 1 (Strongly Disagree) to 5 (Strongly Agree).
- Accuracy: Likert scale from 1 (Strongly Disagree) to 5 (Strongly Agree).
- Efficiency: Likert scale from 1 (Strongly Disagree) to 5 (Strongly Agree).

- Organisation: Likert scale from 1 (Strongly Disagree) to 5 (Strongly Agree).
- Learnability: Likert scale from 1 (Strongly Disagree) to 5 (Strongly Agree).
- Traceability: Likert scale from 1 (Strongly Disagree) to 5 (Strongly Agree).
- Intend to Use: Likert scale from 1 (Strongly Disagree) to 5 (Strongly Agree).