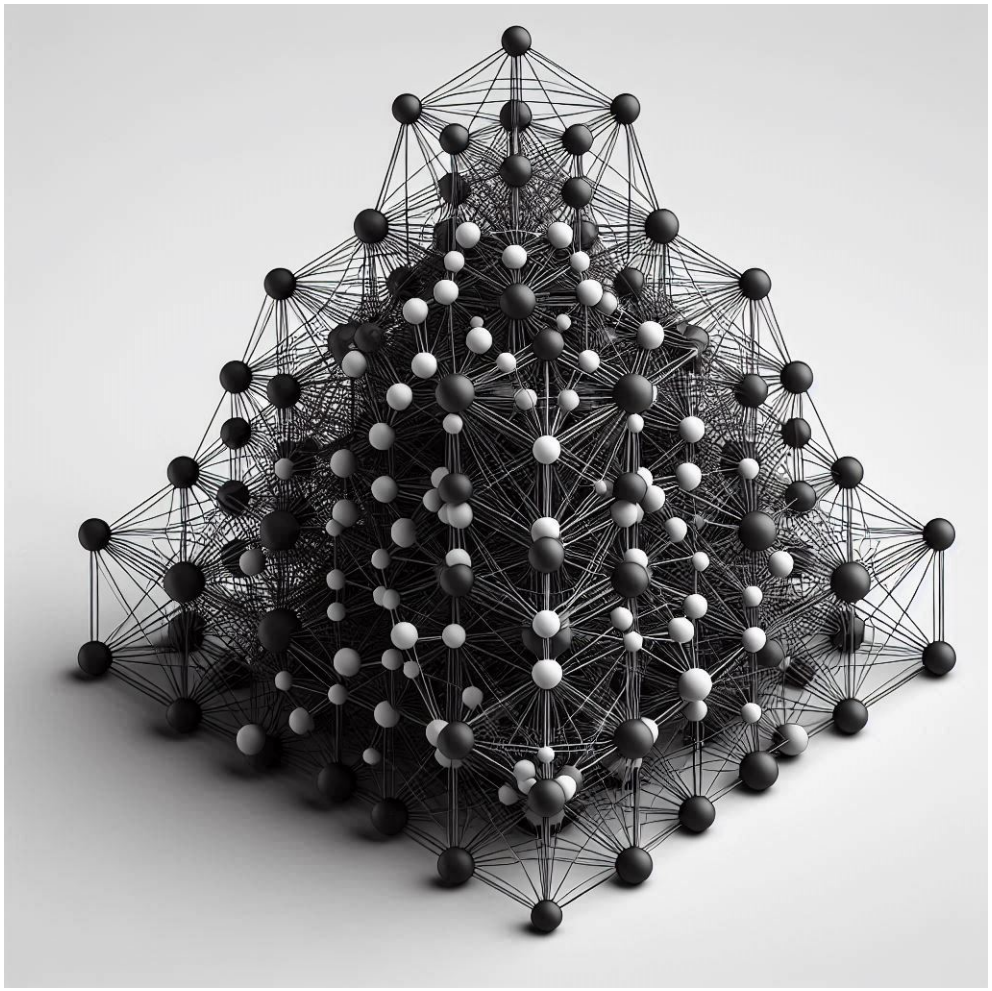
Discovering Common Anti-patterns Present in Low-Code using Multi-Layered Graph-Based Pattern Mining

Unveiling Pitfalls Navigating the Low-Code Landscape



Wessel Oosterbroek

Discovering Common Anti-patterns Present in Low-Code using Multi-Layered Graph-Based Pattern Mining

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Wessel Oosterbroek
born in Oegstgeest, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: The author generated this image in part with DALL-E 3, OpenAI's large-scale image-generation model. Upon generating the draft image, the author reviewed, edited, and revised the image to their own liking and takes ultimate responsibility for the content of this publication.

Discovering Common Anti-patterns Present in Low-Code using Multi-Layered Graph-Based Pattern Mining

Author: Wessel Oosterbroek
Student id: 4961544

Abstract

In recent years Low-Code has seen a surge in popularity amongst companies to speed up their workflows. Yet, scientific work on Low-Code is still in its infancy. We set out to investigate the presence of anti-patterns within Low-Code applications. Given the typically less technically inclined nature of Low-Code developers, as well as the specific use cases of Low-Code in general, we expect that these anti-patterns differ from traditional programming languages. We apply a graph-based methodology to mine edit patterns across real-world commit data supplied to us by Mendix, one of the leading platforms in the Low-Code space. Additionally, we discuss the lack of current guidelines in the Low-Code field. While we are able to find common edit patterns using our approach, linking them to anti-patterns remains difficult in practice. We do establish that Low-Code in Mendix might lack reuse-ability and that the Low-Code often revolves around a few distinct tasks. However, there is a current lack of quality data available to properly assess the development practices of Low-Code developers and anti-patterns, increasing the availability of high-quality data is essential for further research in this area.

Thesis Committee:

Chair:	Prof. dr. A Zaidman, Faculty EEMCS, TU Delft
Committee Member:	Dr. S Chakraborty, Faculty EEMCS, TU Delft
Company Supervisor:	Maurits Elzinga, Mendix
Company Supervisor:	Robbert Jan Grootjans, Mendix
University Supervisor:	Prof. dr. A Zaidman, Faculty EEMCS, TU Delft

Preface

As I conclude what is likely my final project at TU Delft, I would like to extend my gratitude to Andy Zaidman and my daily supervisors, Maurits Elzinga and Robbert Jan Grootjans, for their continuous support, invaluable insights, and many points of feedback, even when I seemed at an impasse. I am also deeply thankful to the Modelling and Services team at Mendix, from whom I received significant assistance regarding Mendix and enjoyed the many book club sessions with.

This is likely my last project as a student, and while I encountered several challenges, from difficulties with data processing to struggles with maintaining motivation. Despite these hurdles, I am proud of the research I have completed.

Wessel Oosterbroek
Delft, the Netherlands
October 15, 2024

Contents

List of Figures

Chapter 1

Introduction

Fueled by the ever-increasing complexity of modern-day software, attention to code quality, structure, and software testing plays a major role in today's Software Development Cycle [15]. To guide developers in this process, significant strides have been made in supporting their workflow, whether we consider Static Analysis Tools, advancements in Software Testing, or the development of best practices within the field of Software Engineering.

At the basis of all this progress is research that investigates and identifies issues with code quality, code structure, or the presence of bugs in software projects, i.e., deficiencies. Across the mainstream programming languages and technologies, one can discern the notable progress made regarding available tools. From the ongoing research on automatically generated test cases [2], to the perpetual refinement of Static Analysis Tools [8].

In recent years, there has been a surge in the adoption of so-called *Low-Code Development Platforms* (*LCDP*) [5, 13, 32]. While existing literature generally agrees on which platforms fall under that term [5, 13, 20, 35, 36], a precise consensus on the definition of Low-Code remains elusive. The term is believed to have originated from a Forrester report published in 2014, where it is defined as "a platform that significantly reduces the amount of manual coding required to develop an application" [35, 36]. Subsequent interpretations expand upon this notion, incorporating the facilitation of domain experts' participation in the development stages of an application into the definition [20]. It is not uncommon to see the term Model-Driven Engineering (MDE), closely related to Low-Code, being incorporated into the definition as well [5, 9, 13]. At the same time, LCDPs are employed in a wide variety of use cases, from manufacturing, production, robotics, and automation systems to application development, which all have their different nuances [16, 23, 27].

One of the most notable impacts of LCDPs is their ability to accelerate development cycles. By substantially reducing the amount of manual coding required, these platforms enable rapid prototyping, iteration, and deployment of applications [5, 23, 38, 40]. This results in a faster development pace, improving the time-to-market and enhancing the responsiveness of software development teams.

A key concept in the field of Low-Code is the term *Citizen Developer*, which refers to domain experts with limited technical knowledge, who represent the primary user group of LCDPs [42]. The desire to make the development process more inclusive for this group is driven by two primary benefits:

1. Domain experts contribute valuable domain expertise, which can be lacking among developers. In other words, LCDPs attempt to narrow the gap between business and development teams [4, 42].

2. Citizen Developers contributing to development helps to mitigate the current scarcity of skilled developers [4, 26].

Despite its apparent value, given the growing number of businesses turning towards it as a tool to speed up development [32, 35], Low-Code remains a domain that has seen limited interest from the scientific community. Several open questions and challenges persist, particularly in the areas of testing, code quality, and supporting developers in building applications with LCDPs [17, 20, 38].

Khoram et al. detail the current challenges and opportunities in Low-Code testing, underlining the necessity of accommodating Citizen Developers by creating a test environment with reduced technical requirements [20]. However, the absence of a generalized testing framework for Low-Code applications leads most LCDPs to rely on third-party tools that do not match the needs of Citizen Developers, which are driven by their lack of technical knowledge [20, 42]. Khoram et al. mark the need for further research in this domain to enhance academic knowledge of the Low-Code field, as well as to get an understanding of how to develop tools that better support citizen developers working with LCDPs in testing their applications [20].

We observe a similar pattern when considering code quality in LCDPs. While existing literature acknowledges the current challenges and underscores the importance of adhering to software engineering standards in Low-Code applications [20, 38], the lack of research on actionable guidelines or tools for developers that focus on Low-Code is noteworthy. Especially when considering the rapid growth of the Low-Code community in recent years [5, 13, 32].

Given that LCDPs are widely recognized for their ability to accelerate application deployment compared to traditional development solutions [5, 23, 24], one would expect the emergence of comprehensive guidelines and tools to maintain code quality in such an environment. Yet, these resources remain scarce, leaving developers with limited support for maintaining proper development standards during development.

Additionally, while Citizen Developers can be valuable contributors [4], they, like any software developer, are prone to making mistakes. Their non-technical background may exacerbate challenges related to maintaining code quality, testing, and adhering to other software engineering practices [20].

One of the leading platforms in the Low-Code space, Mendix[1] encounters these challenges. While the platform offers a Unit Testing module[2], anecdotal evidence has shown it lacks usability and developer-friendliness, limiting development speed. This contradicts the observation that ease of development should be one of the primary considerations, given that we are working with a platform made for fast-paced development. Some third-party solutions offer more robust support for testing Mendix applications, such as Menditect[3], but these are commercial proprietary tools. Moreover, although some guidelines concerning best practices exist, a comprehensive overview of anti-patterns commonly present within Low-Code applications is absent.

In summary, the lack of research into tools and guidelines supporting code quality for LCDP developers manifests itself in several challenges:

1. The absence of generalised testing frameworks designed for LCDPs [17, 20].

2. Limited technical knowledge among Citizen Developers, hindering their familiarity with software development standards [20].

3. The low time-to-market mentality of LCDPs could comprise the code quality of applications made with LCDPs.

[1] https://www.mendix.com/
[2] https://marketplace.mendix.com/link/component/390
[3] https://menditect.com/

As a first step towards developing better tools with a solid basis, there needs to be an understanding of what type of issues developers face when developing applications with LCDPs. To the best of our knowledge, no studies currently exist that investigate this problem.

One way to identify the introduction of bugs or other issues related to code quality is through anti-patterns. Anti-patterns are common development patterns within a software program that have negative consequences; there are many types of such anti-patterns related to code quality, performance, security, design principles, and bugs [3, 37]. Hence, a large amount of research has been conducted to detect and help avoid the usage of anti-patterns during development [7, 1].

To narrow the scope of this thesis, we will concentrate on anti-patterns that indicate either the presence of bugs or issues with code quality. The definition of anti-patterns is usually described more broadly, encompassing other aspects as well [3, 37]. For this purpose, in the scope of this thesis, we define anti-patterns as follows:

Anti-pattern: A frequently occurring pattern that indicates the presence of a functional bug or a structural problem with code quality.

In this thesis, we are concerned with how anti-pattern detection could be applied to currently existing Low-Code platforms, for which we are interested in the difference between LCDPs and traditional programming languages in two important aspects:

1. **Empowering Citizen Developers:** LCDPs enable non-technical domain experts to actively participate in the application development process.

2. **High-Level of Abstraction:** These platforms often employ a model-based approach and focus on development through a Graphical User Interface (GUI).

Khorram et al. define LCDPs as ”A Low-Code Development Platform (LCDP) is a software on the cloud whose target clients are non-programmers aimed at building applications without having IT knowledge.”, while this definition does encompass Citizen Developers it does not include any notion on how LCDPs exactly cater to them, nor is being hosted in the cloud a crucial element for our purposes [20]. Gomes et al. state that ”The surgency of terms like ’low-code’ or ’no-code’ are usually used when referring to platforms, applications, or products with a high-level programming abstraction, that are intended for end-user development (sometimes also called Citizen development) through Model-Driven Engineering (MDE) principles and that aim to serve as a tool for resolving prevailing challenges or for meeting new requirements” [13]. While this definition seems to describe Low Code Development platforms, it does not, however, explicitly mention them. Additionally, both definitions do not mention a GUI or extend their definition of non-programmers to be domain experts.

We propose the following definition, drawing inspiration from [13] and [20], but adapted to emphasize the two aforementioned points:

A Low-Code Development Platform (LCDP): A software, whose target clients are non-technical domain experts, allowing building applications through a high level of abstraction, typically via GUIs and model-based approaches. Its primary aim is to address prevalent challenges or fulfill new requirements.

For the detection of anti-patterns, we will focus on the Low-Code component of LCDPs, hence, we define Low-Code specifically as well.

Low-Code: A form of code that can be presented in a GUI, usually in the form of nodes, e.g. blocks. That allows users, with limited technical expertise, to understand, modify, and contribute to said code.

We propose an in-depth study of the type of code quality issues through the analysis of anti-patterns and their rate of presence within applications built with LCDPs, including functional and maintainability defects or other structural problems. From this point onward, we will refer to those defects as anti-patterns. Compared to traditional programming languages, LCDPs are often more domain-specific in nature and have limited capabilities [34]. For this reason, compounded by the limited technical expertise of Citizen Developers, we postulate that anti-patterns in applications built with LCDPs may differ in nature compared to software built with more traditional programming languages.

To provide a foundation for further studies, and on which to develop better tooling for LCDPs, we aim to provide an in-depth analysis using real-world data of Low-Code applications built with Mendix. Mendix collects anonymized edit data of all applications it hosts to which we have been provided access to provide insights and improve the developer experience. Therefore, we define the goal of this thesis as follows:

Thesis Goal: Leverage Mendix's Low-Code application data to analyse the prevalence of common (anti-)patterns, to attempt to discover anti-patterns that impact code quality and indicate bugs. Additionally, provide an overview of existing company guidelines for designing Low-Code applications to accommodate this analysis. With the end goal of providing a foundation for further studies and on which to develop better tooling for LCDPs.

Having this goal in mind, we formulate the following main research question:

Main RQ: Can we detect common anti-patterns in the historical edit data of Low-Code Applications available through Mendix?

We address this main research question through answering the following three sub-questions:

RQ #1: Through a manual inspection of Mendix guidelines, what types of anti-patterns exist and are commonly found in applications developed with the Mendix Low-Code platform?

As a first step, we summarise and discuss the current Mendix guidelines on code quality to give an overview of their use in practice and to provide a baseline of currently available knowledge.

RQ #2: Using statistical analysis on the data available at Mendix, how can we identify anti-patterns that indicate the presence of a bug or a design problem present in Low-Code applications?

Next, we outline our approach for mining edit patterns from the provided data and apply it to gain insight into the prevalent anti-patterns in Mendix applications.

RQ #3: With what frequency does each discovered anti-pattern appear within the dataset?

We aim to determine the rate of presence of each identified anti-pattern within the dataset, establishing a baseline understanding of their prevalence in Mendix applications, and by extension Low-Code in general.

Consequently, given these research questions, the contribution of this thesis consists of three parts:

Contribution #1: Based on existing methods to perform graph-based change pattern mining, we propose a novel multi-layered approach that enables discovering patterns over a longer time span. It is capable of detecting patterns that span multiple revisions, where the current state-of-the-art is unable to do so [19, 29, 30].

Contribution #2: We implement and apply our approach to the historical edit data of 13 Mendix projects. We mine frequently occurring edit patterns present in these 13 projects.

Contribution #3: We provide a thorough evaluation of the discovered patterns and address the challenges and limitations we faced when applying our approach to the historical edit data of 13 Mendix projects. Which informs future research directions and tool development, ensuring better methodologies for mining patterns in similar datasets.

The next chapters will address the research questions outlined in this introduction. Chapter 2 provides the necessary background context and precise definition of the problem, while Chapters 3, 4, and 5 will cover our approach and analyse the results. In Chapter 6 we cover related works in the field of repository mining. Finally, a conclusion is given in Chapter 7.

Chapter 2

Background

To provide the reader with the necessary background, this chapter introduces the Mendix Low-Code platform and the data we have been provided with. First, we outline the general structure of Mendix applications and provide an overview of key components. Specifically, we highlight Microflows as they are the core of Low-Code development in Mendix and the focus of this Thesis. For more in-depth background information on how to build applications with Mendix, we refer the reader to the Mendix documentation[1]. Next, we provide an overview of the limitations of the data provided by Mendix in the second part of this chapter.

2.1 Mendix Background

As we state in Chapter 1, Mendix is a Low-Code platform established in 2005 [24]. Positioned as the top Low-Code platform in the 2023 Gartner Magic Quadrant for Enterprise Low-Code Application Platforms, Mendix is one of the major players in the Low-Code space [12]. Mendix aims to accelerate development processes for its users while simultaneously facilitating the participation of Citizen Developers in the development cycle of an application [24]. Mendix achieves this goal through a predominantly GUI-based IDE, enabling the construction of Mendix applications with minimal traditional code. Although it is possible to integrate functionality written in Java into a Mendix application, it allows users to create applications entirely through Low-Code.

A Mendix application is built using various components serving many different functions. However, this introduction is mostly limited in scope to those components that are most important to understand in the context of this thesis: The IDE itself, Pages, and Microflows, respectively.

2.1.1 Studio Pro

Mendix applications are built in Studio Pro, an IDE purpose-built for Low-Code development. Figure 2.1 showcases the Studio Pro IDE interface, which provides developers with the tools necessary to build Mendix applications.

In Studio Pro, the interface's left side displays the application's structure under development, presenting a hierarchical overview of all its components. This includes essential elements such as Pages and Microflows, as well as general settings and the Domain Model, which allows the user to define the application's database structure.

The central portion of the IDE interface shows the currently selected component, whether it be a Page, Microflow, or the Domain Model. This view allows developers to visualise and interact with the selected component, making it straightforward to introduce changes as needed.

[1] https://docs.mendix.com/

On the right-hand side of the interface, developers can access the properties of the currently selected component, along with a toolbox containing pre-built elements and widgets. These pre-configured components, such as buttons with specific actions, streamline the development process by providing ready-made solutions for common functionalities and intend to reduce the need for manual work.

There is much more to Studio Pro as an IDE, the goal here, however, is to illustrate that it is a GUI-focused development tool that focuses on providing functionality to quickly create prototypes or full applications, such as through pre-configured elements and widgets.

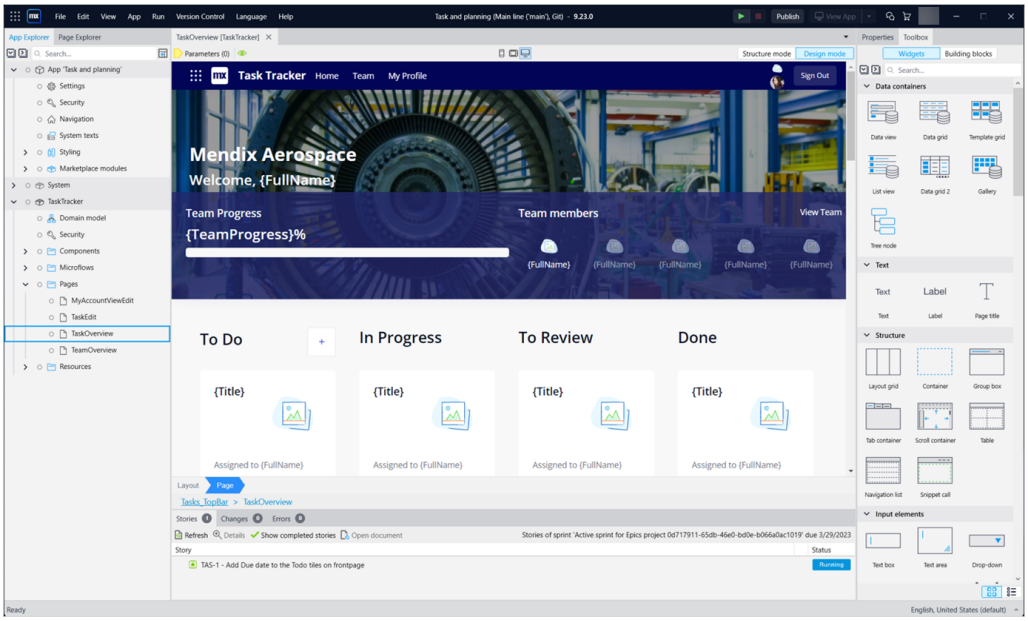


Figure 2.1: A Mendix Project opened in Studio Pro 10.

2.1.2 Pages

Pages, as the name implies, serve as a user interface through which the user interacts with a Mendix application, e.g. web pages in a browser-based application. They can be constructed with pre-built components, like layouts, list views, and buttons, accessible in Studio Pro. These components enable developers to customise the appearance and behaviour of the application's interface.

Pages can interact with Microflows, which contain the application's logic. In the next section, we will discuss Microflows in more depth and show how this interaction allows developers to implement complex functionality within an application.

2.1.3 Microflows

Microflows are an important part of the Mendix platform, allowing developers to define custom logic. While, in general, not as versatile as other programming languages they allow Citizen Developers to express their application's logic visually.

Within the context of a Mendix application, Microflows can triggered to run in a variety of ways, such as being called by another Microflow, a user interacting with a page, or scheduled events. This grants developers precise control over the execution of their logic.

To give an idea of how Microflows look in practice, it is essential to define their fundamental components. A Microflow represents a visual model of a process, designed to automate specific logic. Each Microflow consists of several key elements, which collectively dictate its functionality. These components are outlined as follows:

1. A Start Point and one or several End Points.
2. Any input data to be provided to the Microflow when executing it.
3. Arrows or sequence flows, that dictate the flow within the Microflow.
4. Nodes, or blocks, that perform certain actions.
5. Specific nodes that split the application's flow, such as loops and conditional nodes.

As the name indicates, a Start Point indicates the point where the execution of a Microflow starts; There is always one Start Point indicated by a green dot, a Start Point always has one outgoing flow, depicted by a black arrow, to the node that will be executed first. Similarly, an End Point indicates the end point of a Microflow. As an example, Figure 2.2 displays an empty Microflow, consisting of solely a Start and End Point.

Figure 2.2: An empty Microflow consisting of a Start and End Point.

The flow between the different nodes in a Microflow is dictated by Sequence Flows. A Sequence Flow is visually represented as a black arrow starting in an origin node and ending in a destination node. When we finish the execution of the origin node we follow the outgoing Sequence Flow and continue with the execution of the destination node. Figure 2.2 shows a sequence flow originating from the Start Point going directly to the End Point of the Microflow.

In an endpoint, it is possible to return a variable or object to the caller. Additionally, Microflows can have one or more input parameters that can be used during execution of the Microflow. Figure 2.3 shows the same Microflow we have seen in Figure 2.2, but it now includes a User object that is given as input variable to the Microflow and also returned in the End Point.

Figure 2.3: An empty Microflow consisting of a Start and End Point with a User object as input parameter and return variable.

A Microflow can contain an arbitrary number of nodes, which can perform a wide range of actions, from data retrieval to applying a transformation on the input data. Usually, a node

has one flow going in and one going out, which points to the next node that has to be executed after completion. However, several nodes can split the flow into more than one outgoing flow, such as conditional nodes which are similar to if-statements, and loops. Eventually, every flow path must be either merged into another one or end in an endpoint.

In figure 2.4, one can see an example of a Microflow. This particular Microflow is responsible for changing the password of a given user account. As input data, the Account Password Data object is provided, which contains the password to which the user would like to change their password. To ensure it is changed correctly, the user has been asked to enter it twice, and thus the object contains two password inputs.

Execution of the Microflow starts in the green dot, representing the Start Point, from which we enter a node that retrieves the user account from the application's database. Next, we compare the two passwords in a conditional node. If the two passwords provided as input are not equal, we follow the flow with the 'false'-tag and an error message is displayed. Otherwise, we follow the flow with the 'true'-tag, and the password is saved, followed by an End Point.

If an additional feature was needed, such as displaying a message when the password is saved, one could easily extend the behaviour of this Microflow by adding a new node between the node that saves the user's password and the End Point of the 'true' branch.

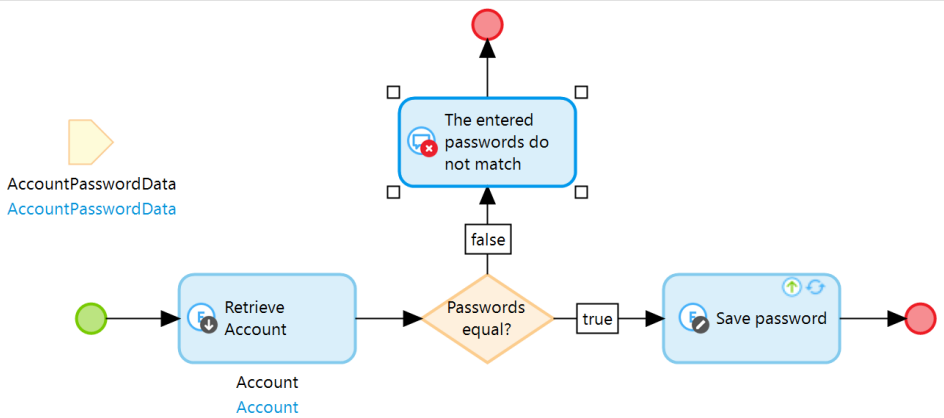


Figure 2.4: Microflow that changes a user's password to a given one.

If we try to place Microflows in the framework of traditional Object-Oriented Programming languages, they fall somewhere in between classes and functions. They often contain more functionality than what normally would be considered a function in a traditional language. On the other hand, Microflows are not comparable to objects or classes and are focused on providing one functionally, characteristics that lean closer to a function than a class.

As we note earlier, these Microflows are crafted through this visual interface and allow Citizen Developers to perform a wide range of operations, including invoking other Microflows, calling third-party services, and transforming data. Now that we have introduced Microflows on a high level, we will dive deeper into the data Mendix provided in the next sections.

2.2 Data Overview

As we state in our research question, see Chapter 1, we aim to find Low-Code edit patterns present in Low-Code applications, that are indicative of anti-patterns. Hence, we are interested in how Mendix projects, hereafter also referred to as models, evolve over time.

To motivate our approach, and to help navigate this thesis, we introduce in more detail the problem we consider in this thesis. The data we have been granted access to has several

constraints when compared to more complete datasets, such as the full commit history of a project, which related works often use for an analysis of this kind. In this Chapter, we first provide an overview of the type of data Mendix gathers. Next, we discuss the limitations that this poses upon the analysis we would like to perform, specifically in comparison to having a full commit history available. Third, we define the problem we address in this thesis, incorporating the data limitations present.

2.2.1 Mendix's Data Collection

Mendix gathers data on applications it hosts in its cloud as part of its data collection efforts. This includes in-house applications built by Mendix itself and applications built by third parties. While enterprise options to host applications on-premise exist, a majority of apps are hosted by Mendix.

Mendix collects this data with the sole purpose of improving its products and services. Moreover, all data collected is anonymized before storage, as such it does not contain any personal or confidential information. Yet, to maintain customer privacy, we are not allowed to share any models used for our analysis in their entirety. While we do describe the characteristics of such models from a zoomed-out, the eventual output of this thesis consists only of short edit patterns commonly found in Microflows. These are not traceable back to any specific project or customer.

Apart from generic metadata such as project size, last-edit date, and Studio Pro Version, Mendix collects information on what changes are made to an application over its lifespan. This can be compared to having a git commit history of the application, with a few key differences:

1. Information regarding changes made to a specific application is stored at a frequency of once per day.
2. Commit Messages, authors, bug reports, or other artifacts are not tracked.
3. Only the main branch is considered when checking for changes, if any change is made on another branch it will not be registered by Mendix until it is merged back into the main branch.
4. A full backup of the main branch of a model is made if any changes to that main branch are made on a given day.
5. Any collected data is anonymized before storage to remove any personal data in the project.
6. The collection of data on changes made to an application is not guaranteed, i.e. data loss can occur due to, for example, server outage.

While this approach of data collection is generally of sufficient quality to facilitate product analysis, e.g. to track the usage of certain modules within Mendix over time, it makes tracking changes between revisions significantly more complex, as we will clarify in the next section.

2.2.2 Data Limitations

Keeping track of changes made to models once per day by making full backups results in several challenges. Some are more logistical in nature, such as handling the vast amounts of data generated, while others are related to the frequency of data collection.

However, the greatest challenge comes from the lack of granularity, caused by two main reasons: Some key data points present in regular commits are missing from the collected

data, namely the commit message, author, and timestamp. Moreover, tracking frequency is limited to once per day and to one branch, resulting in multiple commits made on the same day being identical to one single commit from our point of view.

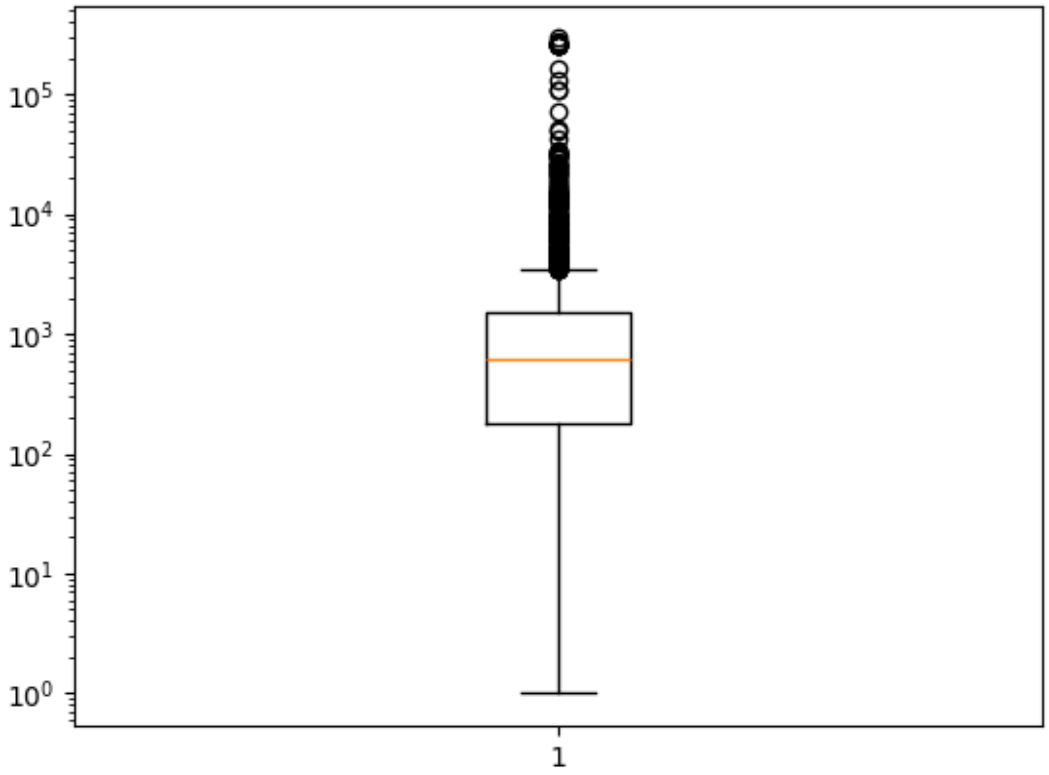


Figure 2.5: The number of edits made to Mendix projects within all revisions available for analysis on a Logarithmic Scale.

Existing state-of-the-art approaches often use commit messages or additional information alongside the changes made in a commit to discover edit patterns. Thus the absence of this data limits us in the approaches that can be applied. Additionally, the low tracking frequency and tracking only the main branch results in large edit transactions, with outliers that contain over 100.000 edits within a single commit. Figure 2.5 shows the distribution of revision sizes of the Mendix projects we use in our analysis. The presence of these large edit transactions complicates the extraction of small, frequently used, edit patterns.

To summarise the problem we address in this thesis, given the limitations of the data we have been provided, we provide the following problem definition:

Problem Definition: Detect anti-patterns through repository mining, solely through the edits taken place between different revisions, in the absence of any external data. Additionally, our approach should be able to handle commits of varying sizes, including larger ones.

2.2.3 Data Schema

When a user edits a Mendix model in Studio Pro, it is saved to disk in a custom file format: an MPR file. This file contains the full model, necessary libraries, and additional editor-related data.

However, the data Mendix collects on how a model changes over time is stored differently. Each project revision is stored in a tabular format in which each component of the model and its properties are present. With a component, sometimes called an object, we refer to all components in a Mendix model, from the nodes within a Microflow to the buttons on a page. Objects can have a parent object associated with them, e.g. a Microflow is also stored as an object, which parents all components within it.

We show an example of this format in Figure 2.6, where a part of a Mendix model is visible. The table contains multiple rows for each unique object, with the unique row identifier being the combination of the objectid and name columns. The objectid is a unique identifier generated for each object within a project. The name column contains the name of a property, e.g. a sequence flow has an origin object and a destination object as properties. For each property belonging to an object a separate row is present, thus a single object has many rows associated with it. The value column contains the value of the specific property in question. Additionally, there is an object type column to indicate the type of the object and a parentid column to indicate the parent object of an object.

projectid	revisio	unitid	objectid	parentid	objecttype	name	value	idpath	typepath	ispointer
af68b827-(	5	014ae	014ae60d-6b1b-4f25-8a8b-e9de0b01008e	014ae60d-	Microflows$Microflow	name	Expense_Approve	/014ae60d	/Microflow:	FALSE
af68b827-(	5	014ae	014ae60d-6b1b-4f25-8a8b-e9de0b01008e	014ae60d-	Microflows$Microflow	flows	[6a5febfb-8e24-4e2c-a315-7cec1	/014ae60d	/Microflow:	FALSE
af68b827-(	5	014ae	6a5febfb-8e24-4e2c-a315-7cec1e4b0baa	014ae60d-	Microflows$SequenceFlow	origin	e3cdb3aa-d076-4913-98ec-2662	/014ae60d	/Microflow:	TRUE
af68b827-(	5	014ae	6a5febfb-8e24-4e2c-a315-7cec1e4b0baa	014ae60d-	Microflows$SequenceFlow	caseValue	331ed851-a5b0-4740-9679-558a	/014ae60d	/Microflow:	FALSE
af68b827-(	5	014ae	331ed851-a5b0-4740-9679-558a4e8028fb	6a5febfb-8	Microflows$NoCase			/014ae60d	/Microflow:	FALSE
af68b827-(	5	014ae	6a5febfb-8e24-4e2c-a315-7cec1e4b0baa	014ae60d-	Microflows$SequenceFlow	destination	e793cad4-00d1-4ece-a8c1-8bf2c	/014ae60d	/Microflow:	TRUE
af68b827-(	5	014ae	6a5febfb-8e24-4e2c-a315-7cec1e4b0baa	014ae60d-	Microflows$SequenceFlow	isErrorHandler	FALSE	/014ae60d	/Microflow:	FALSE
af68b827-(	5	014ae	6a5febfb-8e24-4e2c-a315-7cec1e4b0baa	014ae60d-	Microflows$SequenceFlow	originBezierVector	{"width":30,"height":0}	/014ae60d	/Microflow:	FALSE
af68b827-(	5	014ae	6a5febfb-8e24-4e2c-a315-7cec1e4b0baa	014ae60d-	Microflows$SequenceFlow	originConnectionInde	1	/014ae60d	/Microflow:	FALSE
af68b827-(	5	014ae	6a5febfb-8e24-4e2c-a315-7cec1e4b0baa	014ae60d-	Microflows$SequenceFlow	destinationBezierVect	{"width":-30,"height":0}	/014ae60d	/Microflow:	FALSE

Figure 2.6: Snippet of a Mendix Model transformed to a tabular format.

In principle, it is possible to restore the original Mendix project from this tabular format, however, some properties are stored under a different name, and nested properties of an object are sometimes transformed into separate objects with a parent-child relationship, it is thus not a trivial task to recreate the original model.

Considering this schema and the data limitations discussed in the previous section, the next Chapter will delve into our proposed approach.

Chapter 3

Approach

In this chapter, we outline our approach and provide a detailed exploration of each of its components. To contextualize and explain the rationale for our chosen methodology, we begin by briefly reviewing several key related works that form the foundation of our approach.

3.1 Existing Approaches

As we discuss in Chapter 2.2, the nature of the available dataset poses significant challenges in terms of the quality and frequency of data collection. In turn, this limits our ability to apply some of the state-of-the-art techniques that are used for pattern detection in similar datasets. Thus, we focus on approaches that do not require additional artifacts and can deal with potentially large edit transactions.

When considering possible solutions to the problem of pattern discovery in edit transactions, one might initially think of leveraging an approach based on frequent sub-set mining, where we mine and analyse frequently occurring sub-sets of edits present in all edit transactions. Negera et al. [29] implemented a similar approach, however, rather than mining frequent sub-sets, they focus on sub-sequences from chronologically ordered edit transactions, utilizing the time dimension to capture part of the semantics of the code. In an earlier study, Negara et al. [28] argue that data provided by Version Control Systems (VCS) is often incomplete and too imprecise for effective pattern detection. Therefore, their method relies on edit data collected directly from an Integrated Development Environment (IDE), where every keystroke during 1,520 hours of development by 24 developers was recorded [29], resulting in a precise and complete dataset. We do not have access to data with such granularity, nor do we have chronologically ordered data that reflects the sequence of Low-Code edits, preventing us from adopting this approach.

In a later study, Nguyen et al. state that applying frequent sub-set mining on Github commits indeed comes with two major issues [30]. First and foremost, in a VCS you lose information about the order of changes. Moreover, if we use an algorithm that does not consider the order of changes, like frequent sub-set mining, we lose all information on the semantic dependencies between changed code. They argue that applying such an algorithm on VCS commits yields trivial patterns of changes that are common, yet unrelated. Removing common edits does not help to remedy this problem, as common changes are part of true patterns. We show in Chapter 2.2 that the data we have available is even less granular than that available in common VCS, thus further amplifying these issues. Figure 2.5 shows the number of edits within each of the revisions of the projects that are part of our analysis, the upper outliers are likely due to the merging of branches, see Chapter 2.2 for a more detail explanation of why this is the case. The high number of edits in transactions makes it difficult, if not impossible, to use an approach based on frequent sub-set mining.

To address these issues different authors propose a graph-based approach applied to a VCS's commit data [19, 30], which they show outperforms the sub-sequence mining approach suggested by Negara et al. [30, 29]. In these approaches, they generate edit graphs on which a frequent sub-graph mining algorithm is applied. Using graphs rather than a list of edits allows for the capture of semantics of the underlying code, without relying on chronologically ordered edit data. We propose a similar graph-based setup, specifically adapted for Low-Code development and incorporating a multi-layered time component.

3.2 Graph Based Approach

In this section, we propose our novel edit pattern mining approach based on a multi-layered graph setup. In contrast to the current state-of-the-art, our approach is capable of mining patterns that span over multiple revisions. Consequently, it can provide insights into the long-term architectural changes to applications. Additionally, our approach focuses specifically on supporting Low-Code applications.

To limit the scope of this thesis, we apply our approach only on Microflows, the Low-Code part of Mendix applications, any edit data with regards to other components such as pages, has been filtered out as a first step.

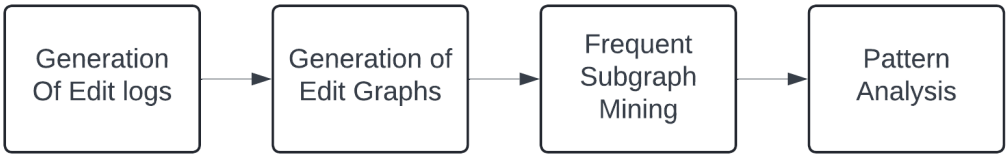


Figure 3.1: A schematic overview of our implemented approach.

In Figure 3.1 we show our approach from a global perspective. The first step consists of converting the data as supplied by Mendix to a set of edit logs for each project that contains the changes from one revision to the next. We use these logs to create graphs that encompass the changes made to a specific Microflow. Lastly, we mine the common sub-graphs from this set of edit graphs and analyse these frequent edit patterns to learn about developer edit behaviour. The next sections will go over each of these individual steps in more detail.

3.2.1 Creating Edit Logs

The initial step of our approach involves generating edit logs that capture the differences between each pair of consecutive revisions. These logs are necessary for creating edit graphs in the next step. To generate the edit logs, we first chronologically order the revisions of each project by their timestamps. As illustrated on the left side of Figure 3.2, revisions 0 through 4 are presented in order from earliest to latest. The next step involves creating revision pairs by pairing each revision with the next one in the sequence, such that revision 0 is paired with revision 1, and revision 1 with revision 2, etc. as depicted in Figure 3.2.

To determine the differences between two revisions within a revision pair, we exploit the structure in which Mendix models are stored. Each revision is represented as a large table that enumerates all components and their properties within a specific Mendix project in a row-based format, see Section 2.2.3 for more details. When comparing the tables of two consecutive revisions, we mark rows that have been added, removed, or modified and add them to the edit log specific to these two revisions. Given how Mendix stores its data, a single component may be represented by multiple rows, with each row containing a single

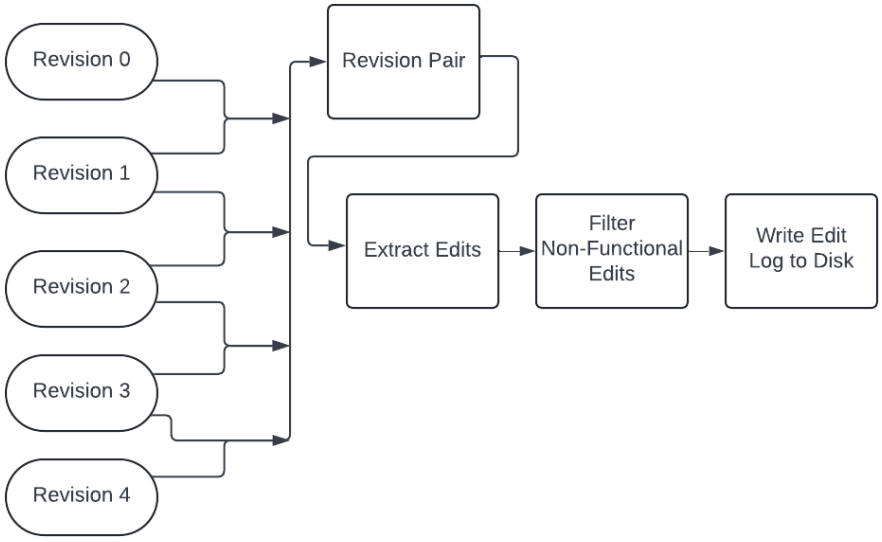


Figure 3.2: Overview of Edit log generation.

property linked to that component. For instance, a button might have separate rows for its color, and for the x and y coordinates that define its position. The resulting edit log consists of rows corresponding to every change made between two revisions, with each row having a label that signals it being either a deletion, addition, or modification.

Mendix models contain a significant amount of data that is irrelevant to our analysis. For instance, each node in a Microflow has a position property associated with it, which is necessary for creating a visual representation when a developer wants to edit a Microflow, however, these properties do not convey any semantic meaning about the Microflow. Therefore, as a final step, we filter out all non-functional edits from the edit log. This ensures we are left with an edit log that only contains those edits that impact the functionality of the Microflow. This final step completes the process of generating the individual edit logs for each pair of revisions.

3.2.2 Generating Edit Graphs

Now that we have edit logs for each pair of sequential revisions, the next step is to construct an edit graph for each edit log. An edit graph consists of nodes, representing the components within a Microflow, and undirected edges, representing the semantic connections between the components in a Microflow. Hence, for each unique component in an edit log, we create a node. As we note in the previous section, due to the way Mendix components are stored internally, the number of nodes does not necessarily have to be equal to the number of rows of the edit log.

Nodes have two labels associated with them, one of these is the edit type of the node, a node can have one of three edit types, either its associated component was modified, removed, or inserted in the edit log we are translating. Each node gets assigned a color, blue, green, or red, depending on if it was modified, inserted, or deleted respectively. Secondly, the node represents the component as how it functioned in the particular Microflow, and is given the component type as a label.

Within a Microflow numerous components perform similar functions, and this poses a challenge to mining edit patterns. For example, various components can convey information to the user, by loading a new page or displaying a message box. This functional similarity

among different components complicates the process of isolating patterns within Microflow edits because even if two developers have the same intent, one might use a page to display new information to the user, while the other uses a message box. Through experimentation, we found that this would cause certain groups of components to never occur in patterns as each component would not appear with high enough frequency in our dataset. To limit the number of unique components we, under the guidance of a Mendix expert, group certain components together based on their similarity in functionality. This comes at the cost of having more generalized patterns, instead of showing precisely which specific grouped component was used. But provides the benefit of more easily identifiable patterns.

After creating a node for each component, we add edges to establish their connections. We categorize these edges into four distinct types based on the relationships they represent.

We create edges of type 0, coloured red, to represent the sequential flow within a Microflow. When two nodes are executed consecutively within a Microflow, they are connected by a type 0 edge in the edit graph. Additionally, some components have sub-components that we also connect through this edge type. For example, a component that retrieves an object has a different sub-component depending on whether it retrieves the object from memory or the application's underlying database. Figure 3.3 contains an example where we see three nodes, each representing a component of the underlying Microflow, connected through a type 0 edge, meaning the components are directly connected in the Microflow as well. In this particular case, the nodes are red, which tells us the three components were deleted from the underlying Microflow.

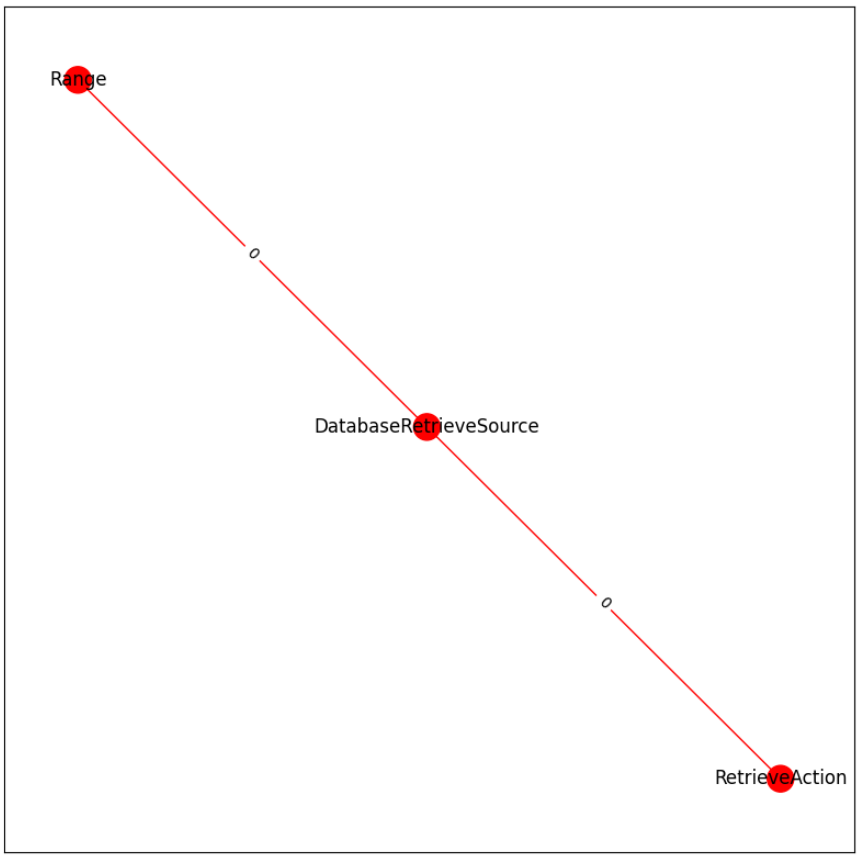


Figure 3.3: Example snippet of a generated edit graph containing edge type 1.

Edges of type 1, coloured green, are based on the shared use of variables between nodes. For instance, if two nodes transform the same variable, they will be connected by a type 1 edge. This also applies in scenarios where one node retrieves data and stores it in a variable,

which is later utilized by another node.

Edges of type 2, represented in blue, provide additional details about properties associated with a node. For instance, the property named ReturnVariable is associated with each end node and captures whether a Microflow returns a variable when the end node is reached. For properties, that can only be set to a limited number of values, such as booleans, the value to which the property is set is included. The labels of other property types are limited to the property name. This edge type enables the identification of patterns where specific properties are edited or modified.

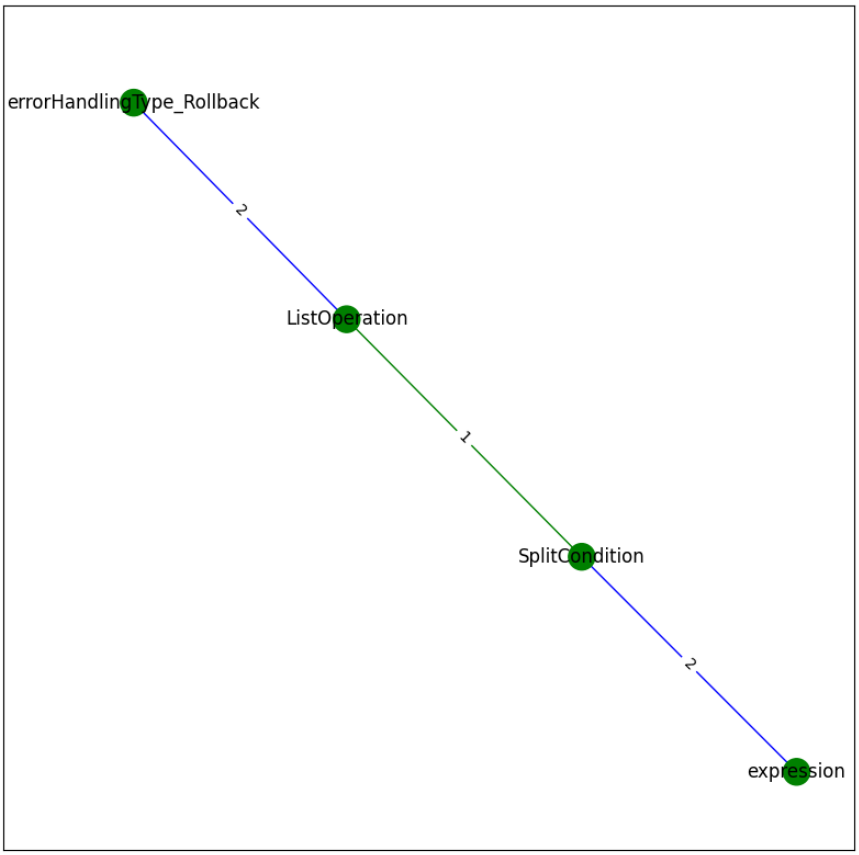


Figure 3.4: Example snippet of a generated edit graph with edge types 1 and 2.

An example snippet of a graph using Edge types 1 and 2 is shown in Figure 3.4. We see two components of a Microflow being represented as nodes, namely the SplitCondition node and the ListOperation node. Additionally, both of these components have one property assigned to them, identifiable by the edge of type 2, the ErrorHandlingType property, and the Expression property. The ErrorHandlingType contains information on how to handle the list operation failing, while the expression property contains the expression of the SplitCondition. Since the ErrorHandlingType can only be assigned a few specific values, its assigned value, Rollback, is included in the label of the node. The edge of type 1 between the SplitCondition and Listoperation nodes signals that both the SplitCondition and the Listoperation make use of the same variable in their operation.

Additionally, there is a fourth type of edge, which is unique to multilayered edit graphs. We will discuss this edge type in the next section.

3.2.3 Multi-layered Edit Graph

To gain a better understanding of how Microflows evolve, which is not feasible with state-of-the-art approaches, we propose a multi-layered graph approach that enables the identification of patterns spanning multiple revisions.

In essence, we construct a multi-layered edit graph where each layer corresponds to an individual edit graph. To create this multi-layered graph, we first take the set of edit graphs belonging to a specific Microflow generated in the previous section and arrange them in chronological order. Starting with the first edit graph in the set, we consider each node and, if the node appears in a subsequent edit graph, we link it to its first future occurrence using an edge of type 3, coloured yellow. This process is repeated for each subsequent edit graph in the ordered set, resulting in a multi-layered graph where each node is connected to its future occurrences.

For example, consider the scenario where a specific conditional node is added and subsequently edited twice in a row. By incorporating these multi-layered edges, we can detect and track such patterns, which would be more challenging or impossible to identify without a multi-layered approach. Figure 3.5 illustrates an example of a multi-layered graph, where a MicroflowCallAction component is introduced in one revision and later removed, alongside its two associated variables. We can identify the MicroflowCallAction being removed in a later revision through the type 3 edge.

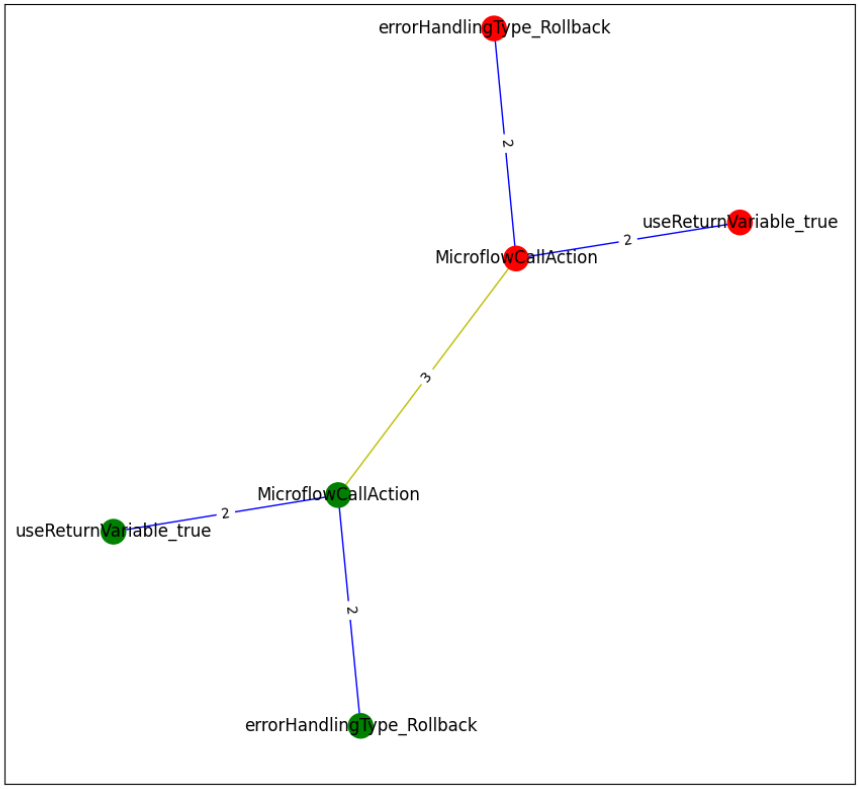


Figure 3.5: Example snippet of a generated multi-layered edit graph.

This approach enables the identification of patterns that involve nodes added but later edited or removed in subsequent revisions, thereby providing additional context to draw conclusions on developer edit behaviour.

3.2.4 Pattern Mining

After we generate the multi-layered edit graphs, the next step is to mine frequently occurring sub-graphs, note that each multi-layered edit graph represents one Microflow. To discover frequent sub-graphs we utilize cgSpan [1], a library that is built upon the well-known gSpan library often used for frequent sub-graph mining [41, 44]. However, cgSpan differs from gSpan in that it only mines closed sub-graphs as opposed to any frequently occurring sub-graphs. A sub-graph g is closed if and only if there exists no super-graph of g that appears with equal occurrence. This is a desirable property since if we have a pattern of which a larger super-graph g exists with at least equal occurrence, we refer to only include the super-graph g in our set of mined patterns.

3.2.5 Pattern Analysis

cgSpan outputs all sub-graphs that satisfy configurable support and size parameters. However, after mining, these patterns still need to be manually interpreted to reveal the intent of the developers using the pattern. To aid in this process and make manual analysis of many patterns more feasible, we can combine patterns that are very similar by ignoring small differences between two discovered patterns, e.g., a difference of 1-2 edges. This can happen quite easily, for example, if one edit graph out of all edit graphs misses one edge in a pattern that is contained in every edit graph, we will get two patterns: One with support value n, not including the missing edge, and one with support value n-1 which does include the missing edge. We use the Jaccard coefficient of two patterns to decide on similarity.

Additionally, we rank pattern according to the formula 3.1, where |N| is the number of nodes, S is the support value of a pattern p, and r is the resulting rating of pattern p. This is the same rating as used by Nguyen et al. [30], and prefers larger patterns over smaller ones if there are patterns with equal support.

$$r = |N| * S \tag{3.1}$$

This ranking helps to prioritize patterns that not only have widespread support but also are sufficiently complex to provide valuable insights. In the case of a large number of patterns, with which Nguyen et al. struggled [30], this ranking intends to find promising patterns more easily.

[1] https://github.com/NaazS03/cgSpan

Chapter 4

Results

This chapter presents the results we obtain by applying our approach set out in Chapter 3 to the dataset supplied to us by Mendix. We begin this chapter with a description of the data selection process and the pre-processing steps we apply. Next, we provide an analysis of the dataset's characteristics. Finally, we apply our approach and present an overview of the edit patterns we mine with cgSpan. A comprehensive discussion on the patterns we identify and the edit behaviour of Citizen Developers can be found in the next Chapter.

4.1 Data Selection

The dataset available at Mendix contains the revisions of over 10,000 projects, extracting all revisions for all projects proved too costly in terms of time required and budgetary constraints. Moreover, even if the extraction of all revisions would be feasible, there would be no way to process such vast amounts of data in the first place.

To ensure the computational feasibility of processing the data provided by Mendix and to narrow the scope of the data extraction, we reduce the size of the dataset. Since our focus is on analyzing the edit behavior of Citizen Developers in modifying their applications over time, we opt to collect all available revisions from a limited number of projects.

There are several factors based on which we decide to filter out part of the projects. First, Mendix has a team of experienced Mendix developers who develop Low-Code applications for internal and external use. To exclude any biases and focus on our effort to support Citizen Developers we exclude all internal Mendix projects from our dataset, to remove any possible bias towards more experienced developers that could be present in these projects.

Secondly, since solely the main branch is tracked there may be a significant time between revisions due to infrequent merges to the main branch. To partly remedy this limitation and to point our focus on projects that were in active development at some point in their lifetime, we only consider projects where the median time between revisions is less than 3 days. Figure 4.1 shows a box plot of the time between revisions, where we observe a significant number of outliers that have a relatively high median time between two revisions, we thus exclude those projects. The median sits at about 10^9 milliseconds between revisions, which converts to roughly 11-12 days, with the minimum being about 10^8 milliseconds, which is equivalent to 1 day, the minimum time between any two revisions due to the edit data being gathered only once per day, as we explain in Section 2.2.3.

Furthermore, we exclude the five percent outliers based on app size to keep our focus on averagely-sized Mendix projects. Figure 4.2 shows a box plot of the project sizes which again show high variance and a significant number of outliers on the upper end.

Additionally, we excluded apps that do not contain any Microflows, or apps that have fewer than fifty revisions overall. To exclude applications that merely contain static pages without any Low-Code or those that have not been actively developed.

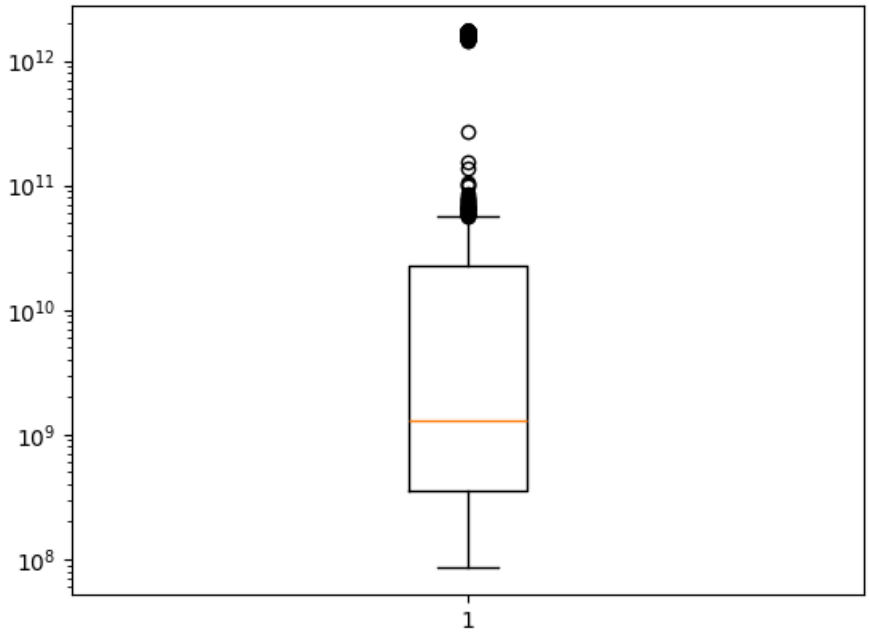


Figure 4.1: Number of milliseconds between two sequential revisions on a Logarithmic Scale.

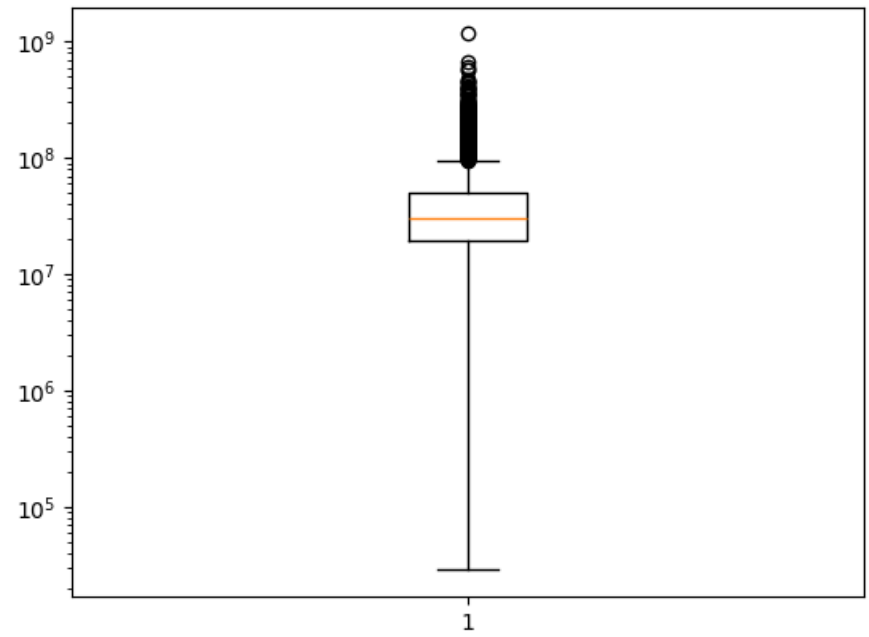


Figure 4.2: Box plot showing byte size of Mendix projects on a Logarithmic Scale.

After applying these restrictions we are left with a sub-set that is still over 5000 projects in size. From this set of projects, we select 13 at random. For those 13 projects we collect

all revisions that we have available, this provides us with a total of 3965 revisions over 13 projects to perform our analysis on. While a sub-set of more than 13 projects would have been beneficial for the generalizability of our research, selecting more projects was not feasible due to the resource constraints described earlier.

The 13 projects contain 28950 unique Microflows in the 3695 revisions, with the combined Microflow count of the latest revisions of each project being 16976. Thus we have a significant number of Microflows on which to base our analysis.

4.2 Edit Characteristics

To the best of our knowledge, no scientific research that investigates how developers use and edit Low-Code in practice exists. This section aims to provide insight into the characteristics of the 13 projects we analyse to partly address this knowledge gap.

To gain an insight into the evolution of Microflows during the development of a Mendix project, we first examine the number of times a Microflow is modified throughout its lifetime. We consider a Microflow to be modified in a revision if any edit, be it an insertion, deletion, or modification, in at least one of the components of the Microflow takes place. This could range from simply changing one property associated with a component in a Microflow to the introduction or the deletion of functionality. In other words, if the Microflow is contained in the edit log of two revisions, as generated via the steps explained in 3.2.1, we consider it to have been modified.

Mendix tracks Microflows and their components across project revisions using unique identifiers. These identifiers enable us to observe which Microflows were modified by the developer between revisions. Furthermore, they allow us to distinguish when a developer edits the properties of a component within a Microflow, as opposed to inserting or deleting components. This distinction enables us to separate insertions and deletions from modifications, without requiring additional interpretation.

Figure 4.3 shows the distribution of the number of times each Microflow in our 13 analysed projects has been modified. At first glance, we notice that the distribution is right-skewed towards a low number of revisions per Microflow. With the median number of revisions of a Microflow throughout a project being 2, and the average sitting below 3. Note that the introduction of a Microflow is considered to be a modification, hence a median of 2 tells us that the median Microflow is modified just once after its creation.

It is noteworthy that Microflows are edited such few times over their lifespan. We propose three explanations for this phenomenon. Firstly, Microflows are significantly smaller than classes in traditional object-oriented programming languages. As discussed in Section 2.1.3, one could consider them to be closer to functions in terms of both functionality and size. Their functionality is typically focused on a single, specific task, which means that edits are generally only necessary if the developer needs to modify the specific function that the Microflow performs.

Secondly, it is important to consider the data limitations outlined in Section 2.2 when interpreting these characteristics. Particularly the fact that only the main branch is tracked and that changes are recorded only once per day. This means that if a developer edits a Microflow numerous times on a separate branch before merging the changes to the main branch, we will interpret this as the Microflow having been modified once. Likewise, multiple edits a day, even if committed to the main branch, go unnoticed. Despite these limitations, the limited number of edits we observe in Microflows on the main branches of these projects implies that, in general, Microflows do not undergo multiple significant changes after their introduction to the main branch.

The third reason relates to the edit behaviour of developers, across all 3695 revisions of the 13 projects a total of 28950 Microflows were introduced, and 11974 Microflows were

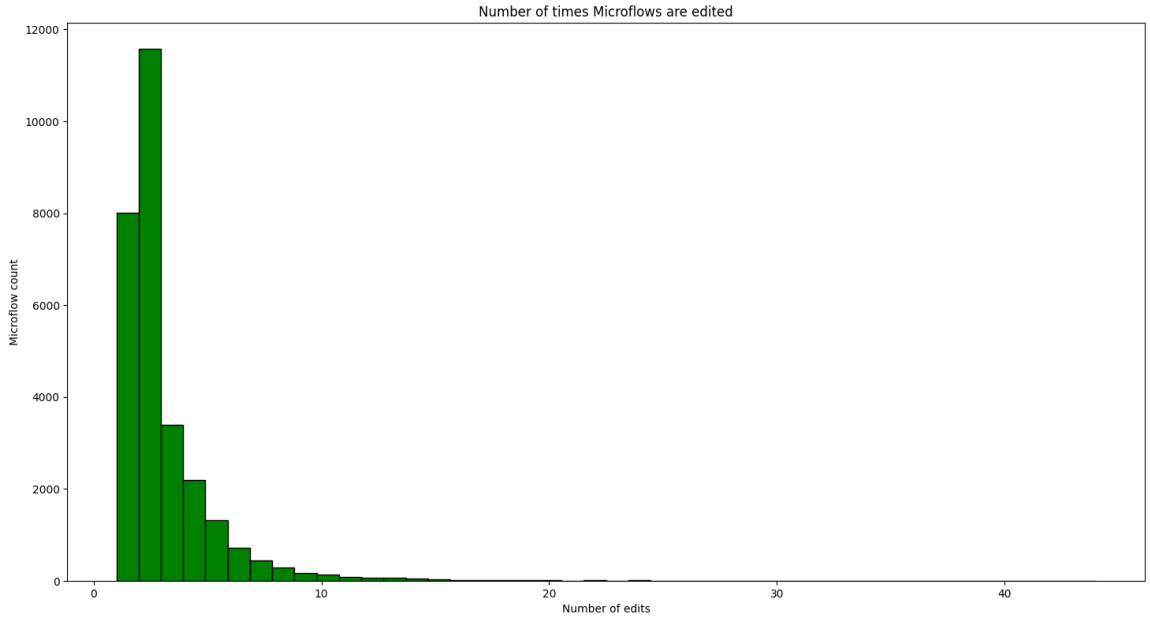


Figure 4.3: A Histogram presenting an overview of Microflow Edit Counts.

removed. Since Mendix tracks Microflows across revisions with a unique identifier, we can directly interpret this result as developer behaviour; Microflows are often deleted while new ones are introduced. This could explain why existing Microflows do not undergo frequent changes. If developers delete existing Microflows and recreate their functionality in new ones when significant changes are required, this will result in a lower number of edits per Microflow. This aligns with anecdotal evidence of how developers interact with the platform. We refer to these types of edits as hidden edits.

Notice, however, that even if we assume all 11974 removed Microflows are part of 11974 hidden edits, we would still end up with a relatively low number of edits per Microflow.

Before we provide an in-depth discussion on how the lack of edits impacts our edit patterns and how this relates to Microflow re-usability in the next Chapter, we will first outlay the results of our analysis in the next section.

4.3 Analysis

We apply our graph-based approach outlined in Chapter 3 to the revisions of the 13 projects we analyse. This approach involves generating edit logs, that capture the differences between revisions, creating edit graphs, and identifying sub-graphs which frequently occur across different projects.

For each of the 3965 revisions in the 13 projects we select for further analysis, we generate edit logs. Where each edit log represents the difference between one revision and the next. Using these edit logs, we generate edit graphs, where each graph represents the evolution of a Microflow, as detailed in Chapter 3. To exclude irrelevant Microflows we exclude those that do not have any edits within their lifetime, moreover, we only consider edit logs that contain less than 100,000 edits due to resource constraints. This results in 8229 edit graphs gathered from the 3695 revisions, each representing one Microflow.

Following this, we employ cgSpan to extract common patterns from the set of 8229 edit graphs. We set a minimum support value of 0.05 and a minimum pattern size of 3. The minimum pattern size of 3 was selected through experimentation, which indicated that patterns

smaller than this threshold lacked sufficient information to be useful. This resulted in 1168 patterns, with the highest support value being 0.28 and the lowest 0.05.

In the final step, we rank these patterns using Equation 3.1. The complete set of extracted patterns is available[1] [31]. In the next section, we will explore the mined patterns in greater detail.

4.3.1 Common Edit Patterns

In this section, we analyse the mined edit patterns. As the resulting number of patterns is too large to discuss each of them in depth individually we first group similar patterns with the Jaccard Similarity set so that we allow a difference of about one node in the larger patterns, about 0.90, see Section 3.2.5 for more details, and take the group's highest ranking pattern as its representative. This reduces the number of patterns from 1168 to 780. We then manually analyse each representative pattern and create groups of similar patterns. For each group, we provide a short description to introduce their purpose and show one or two example patterns to inform the reader.

To introduce the reader we will first provide a short overview of common nodes, or components, in Microflows. Afterward, we will examine one of the most common types of edit patterns, small frequently occurring patterns in our dataset, with an example.

As we describe in detail in Section 2.1.3, Microflows consist of one start point, one or multiple endpoints, and nodes in between that either perform an action or direct the flow of the Microflow. The flow of the Microflow, i.e. the execution order of nodes, is translated to edges of Type 0 in our edit graphs. Note that the edges in our graph are undirected due to constraints concerning the sub-graph mining algorithm cgSpan. Additionally, some components have sub-components that are directly related to each other, such as a Split Node with its split condition, these are also encoded by an edge of Type 0.

Below we provide an overview of the most commonly appearing nodes and their function within a Microflow for the reader to refer back to.

- **Flow Controllers** There are three types of nodes within Microflows that can control the execution flow of the nodes within it. The most important one is the Exclusive Split node which can be found throughout many patterns. The Exclusive Split node has one input flow and multiple output flows, based on a condition given by the developer one of the output flows is chosen. Flows have to either end in an endpoint or be merged together through a merge node, which is another flow controller. The last one is similar to the Exclusive Split node but decides based on the type of a given object.

- **Retrieve Actions** In Mendix Retrieve Action nodes fetch one or more objects from the application's underlying database. There are two types of retrieves, an Association Retrieve retrieves objects from memory, thus also fetching not yet committed objects or changes, while a DatabaseRetrieve only retrieves objects explicitly committed to the database.

- **Call Actions** Call action nodes are a group of nodes used to invoke external entities, such as web services, other Microflows, or custom Java code. The potential output of these calls can be stored and later be used in the Microflow.

- **Parameter Mappings** Parameter Mappings work with Call Actions nodes and facilitate the translation of parameters required as input by these external services and the output they return. For example, they allow the conversion from Json returned by a web service to a Mendix object that can be used in the Microflow. Call Action nodes and Parameter Mappings thus often appear together in patterns.

[1] DOI: 10.6084/m9.figshare.27223998

An example of an actual small edit pattern we mined is depicted in Figure 4.4, which shows three nodes: a Retrieve Action node and two Association Retrieve Source nodes. As explained in Chapter 3, a green node indicates a new component introduced by the developer. In this example, the Retrieve Action and Association Retrieve Source nodes were initially added, and the same Association Retrieve Source node was later edited, as indicated by the type 3 edge and the blue color of the node. In Mendix, the AssociationRetrieve node fetches one or more objects from the application's database. Unlike a database retrieve, which only retrieves objects explicitly committed to the database, an association retrieve also fetches uncommitted or "dirty" objects. The Retrieve Action node serves as a parent for the Association Retrieve Source and can have other types of retrieves as its children.

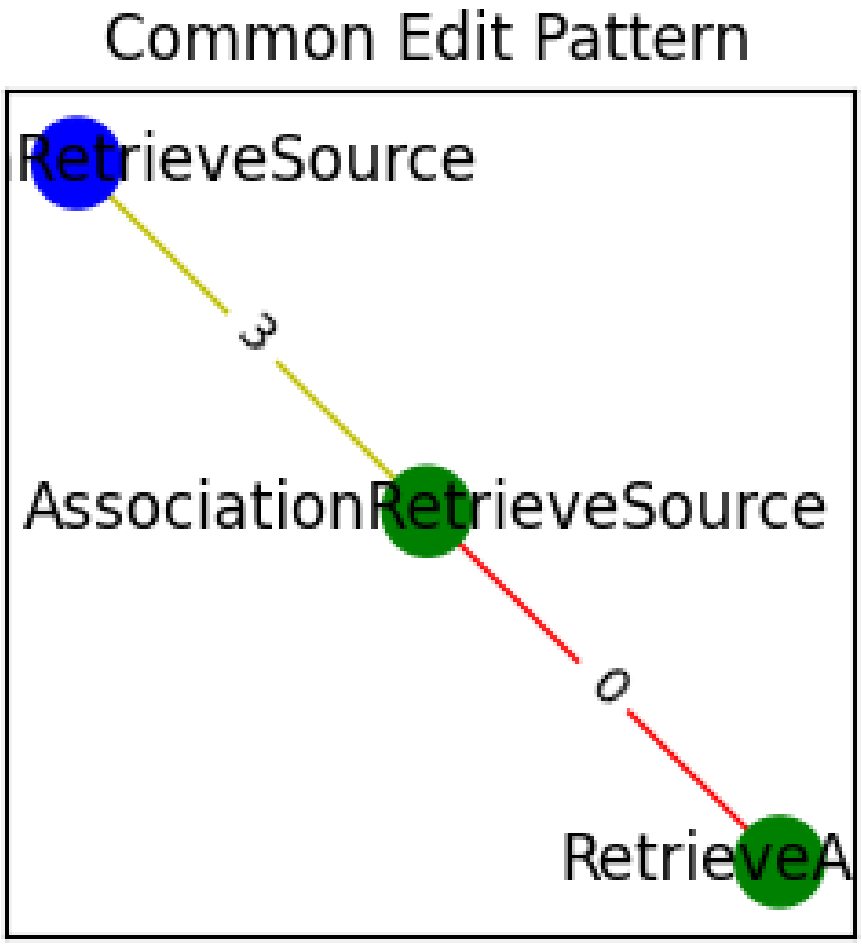


Figure 4.4: An edit graph illustrating the edit of an association retrieve source node.

To illustrate how this might look in practice, consider the Microflow shown in Figure 4.5. In this example, we see a Microflow used in an app for a primary school. An object representing a child is passed to the Microflow. The Microflow retrieves the child's parents by association and checks if this information is available. If not, a message is displayed to the user, in this case, a teacher.

Rather than specifically retrieving the parents of a child, we might have intended to request the current registered caretaker of a Child. To change this we could edit the Microflow

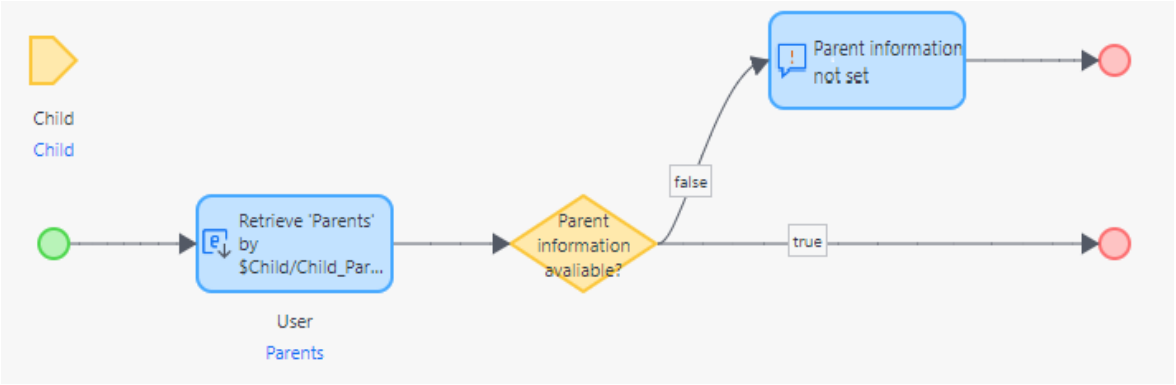


Figure 4.5: An example Microflow showing Parent information being retrieved associated with a provided Child object.

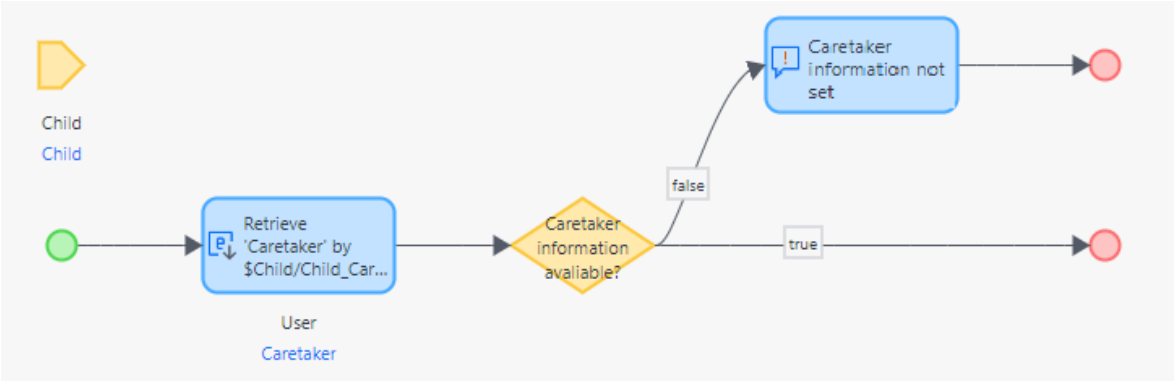


Figure 4.6: An example Microflow showing Caretaker information being retrieved associated with a provided Child object.

to resemble the Microflow depicted in Figure 4.6, where the parents are not retrieved, but rather the caretakers. If the Microflow were modified this way, a resulting edit pattern would match the one shown in Figure 4.4. Specifically, the Association Retrieve Source node would be updated to retrieve the caretakers instead of the parents.

Note that this is not necessarily the only pattern that could be matched to this edit. For example, we also edit the condition of the Exclusive Split node and change the message displayed to the user. Thus a pattern where the Exclusive Split's condition is modified along with the message displayed to the user is also equally applicable to this edit. If it appears often enough throughout our dataset we could even see a pattern encompassing the whole change, i.e. the Association Retrieve, Exclusive Split, and Message nodes all being updated in one pattern.

Another example of a short pattern similar to the one we just saw is shown in Figure 4.7, a sub-graph with a support value of 0.18. In this pattern, we observe the deletion of five nodes: a Microflow Call Action node with its associated properties. In Mendix, Call Action nodes invoke external entities, such as web services, custom Java code, or such as in this case, other Microflows.

One of the most common types of edit patterns is similar to the one shown in Figure 4.8, where a Microflow call action node is used to call another Microflow and is consequently deleted in a later revision. The support value of this specific edit graph is 0.11, e.g. one in ten Microflows contains this structure. A related sub-graph is shown in Figure 4.9, which has a support value of 0.13, where the output of the call action node is later used in a Split Condition, indicated by the edge of Type 1 between the SplitCondition node and the Microflow-

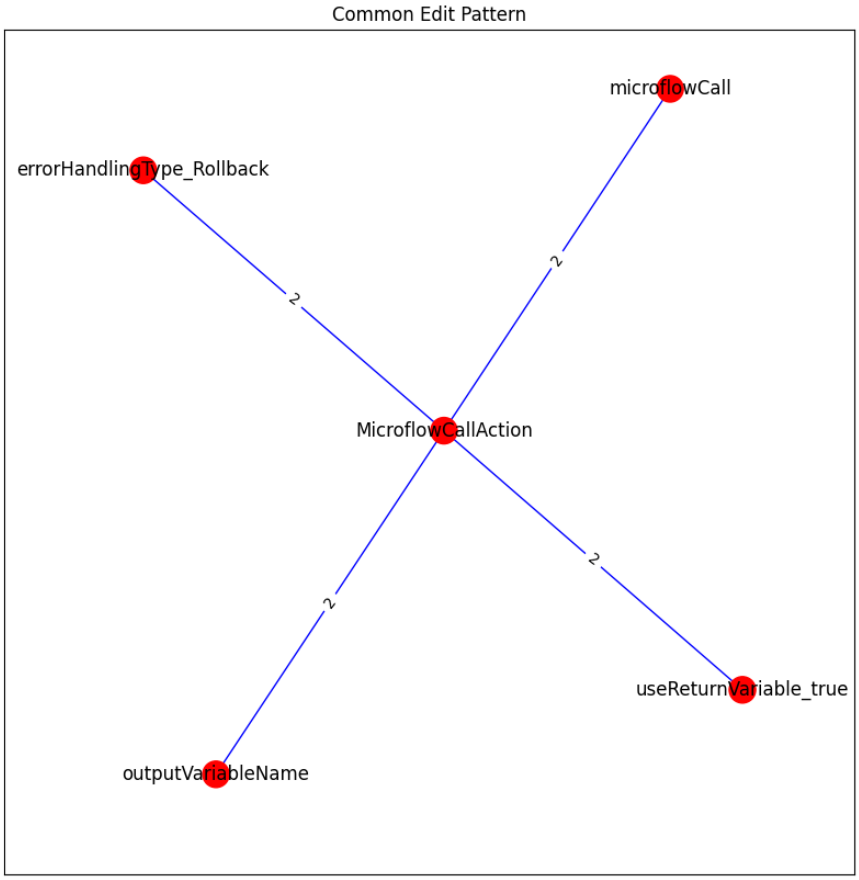


Figure 4.7: An edit graph illustrating the deletion of a Microflow call action node with accompanying nodes.

CallParameterMapping. A MicroflowParameterMapping node is part of a larger group of parameter mappings that facilitate the translation of parameters required as input by external services or Microflows and the output they return, allowing them to interface with Mendix objects within the Microflow.

Another observation is the significant number of patterns which consists of nodes that are added, while being deleted in a later revision, possibly with minor edits in between. Figure 4.10 shows such a sub-graph with a support value of 0.09, where the split condition, alongside accompanying nodes, is deleted after being introduced, without any edits.

A different common edit pattern is shown in Figure 4.11, with a support value of 0.05, where a split condition is used in combination with multiple Member Changes. Member change nodes are nodes that change a variable in some way, depending on the type of the variable different transformations can be applied. Thus, if interpreting the pattern literally, it shows a variable that was changed multiple times, in combination with being used in a split condition node. Similarly, we observe a lot of patterns where a variable is retrieved using a Retrieve or Call node, in Figure 4.12, a sub-graph is shown where a variable is used in a split condition after being retrieved, with a support value of 0.15.

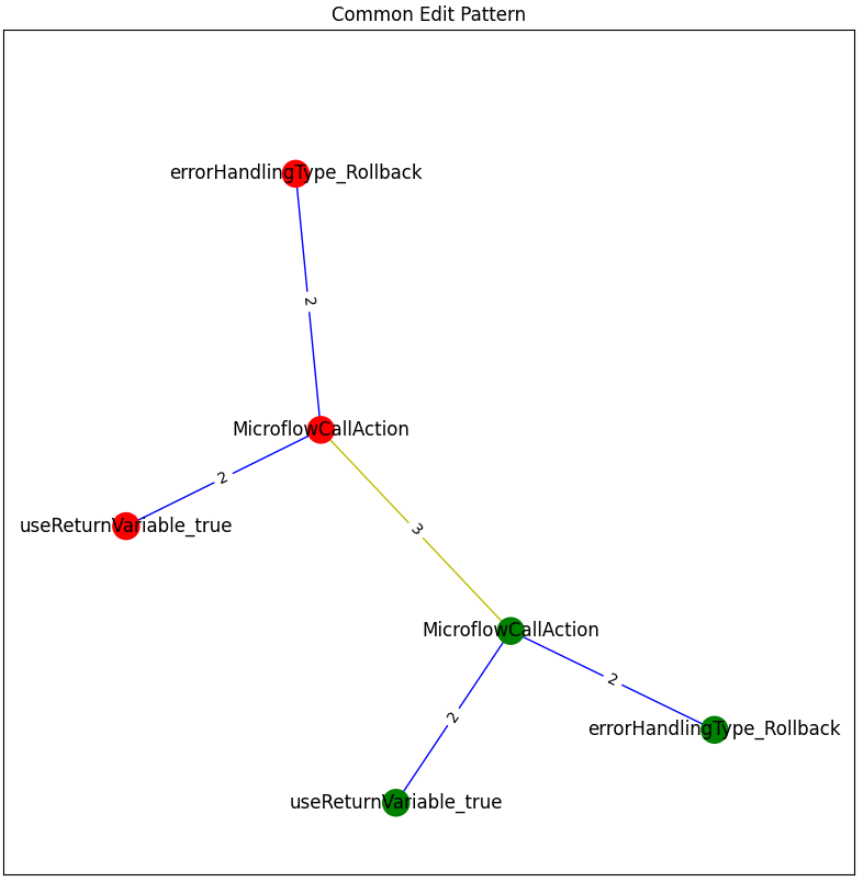


Figure 4.8: An edit graph illustrating a call to retrieve a variable from another Microflow, which is subsequently deleted in a later revision.

Other patterns are often combinations of those previously discussed or differ only slightly. Most commonly, patterns involve parameter mappings, call actions, and split conditions. The key takeaway is that we see few modifications to existing structures or nodes, rather patterns contain insertions of new nodes or the deletion of existing ones. Moreover, most edit graphs revolve around the same few components as indicated above. In the next chapter, we will explore in greater depth what these patterns reveal about edit behavior and potential anti-patterns in Low-Code development.

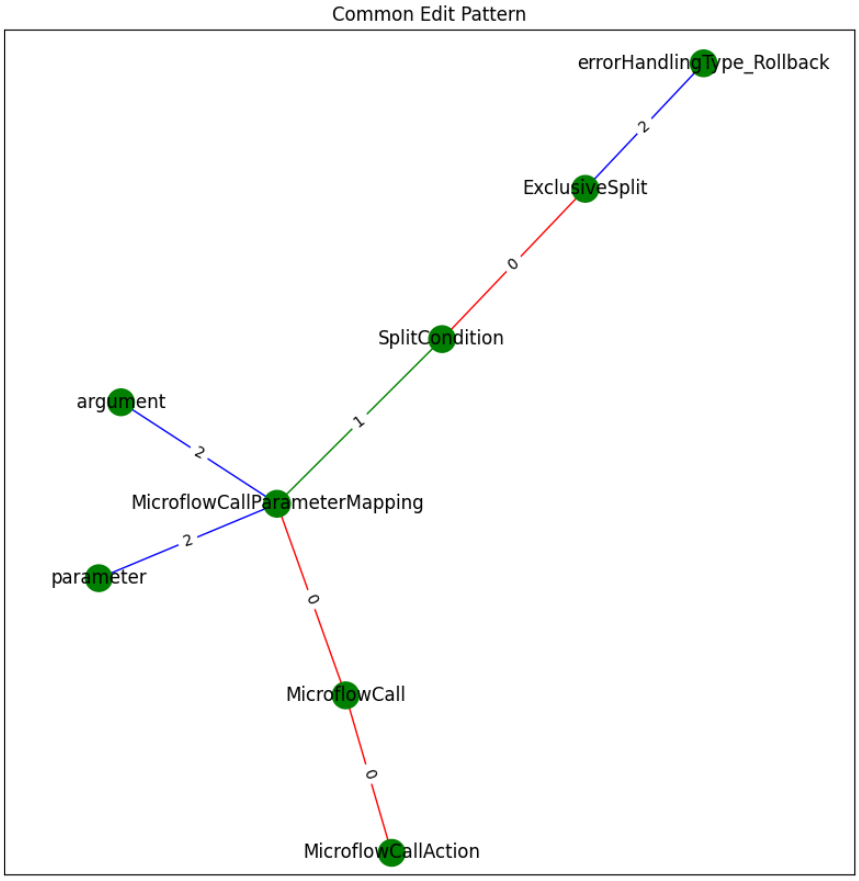


Figure 4.9: An edit graph illustrating a call to retrieve a variable from another Microflow and then used in a split condition.

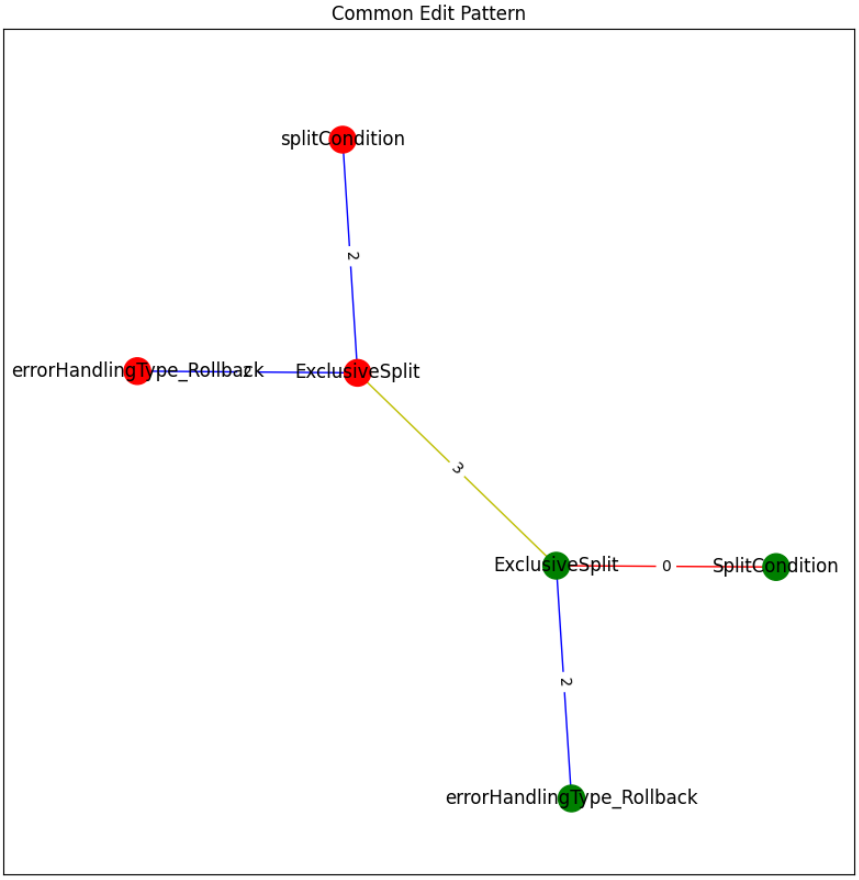


Figure 4.10: An edit graph illustrating the deletion of a split condition.

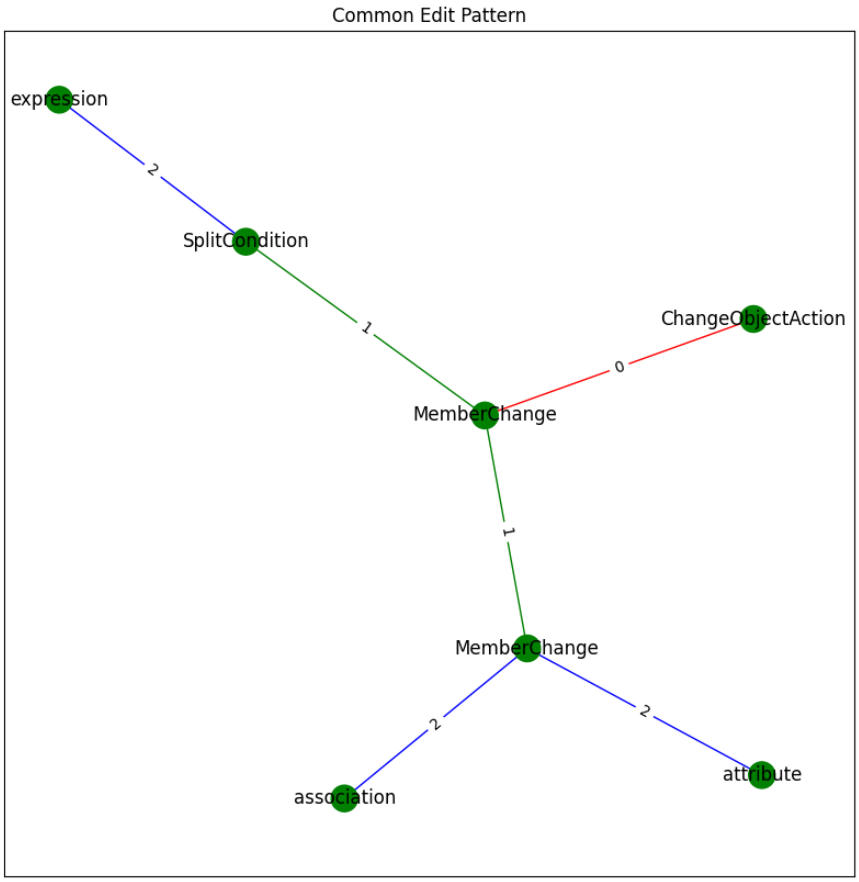


Figure 4.11: An edit graph illustrating the usage of a split condition in combination with multiple Member change nodes.

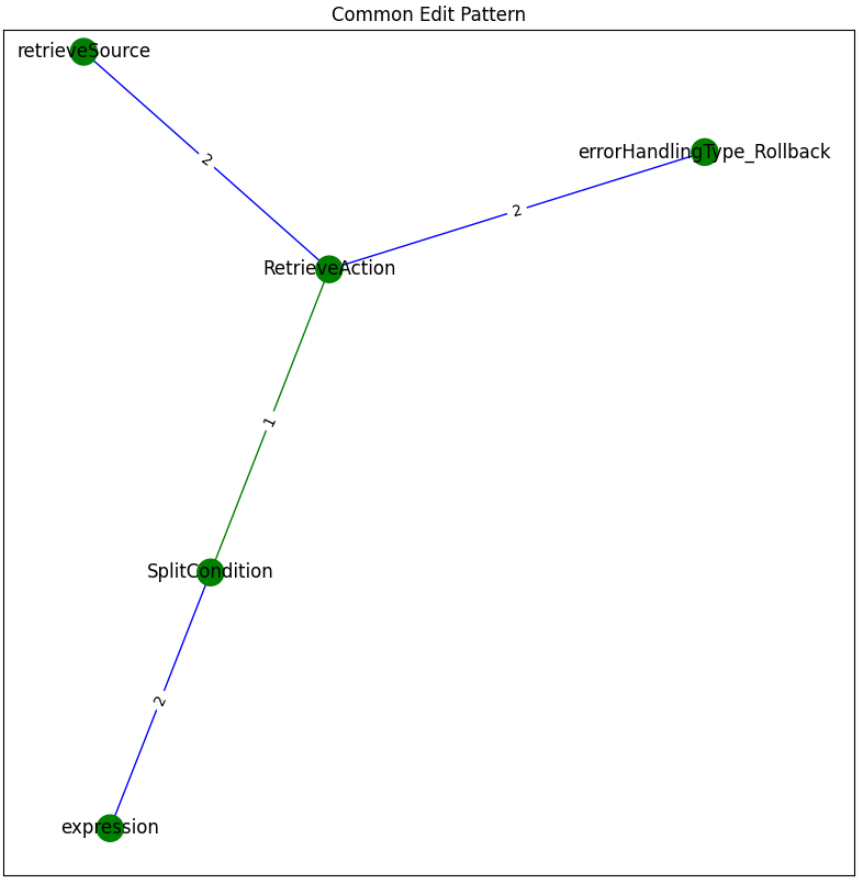


Figure 4.12: An edit graph illustrating the retrieval of a variable which is later used in a split condition.

Chapter 5

Discussion

This chapter provides a comprehensive discussion of the research findings. We analyse the edit patterns identified in Chapter 4, exploring how these reflect the edit behavior of Low-Code developers. Subsequently, we examine existing guidelines, address identified threats to validity, and suggest potential avenues for future research.

5.1 Edit Pattern Analysis

In this section, we will present our insights on what the edit patterns combined with the edit characteristics examined in Chapter 4, reveal about Microflow editing behavior and the presence of potential anti-patterns.

When considering the mined edit patterns described in Chapter 4, we observe several groups of patterns that contain similar structures. Most patterns contain Parameter Mappings, Call Action, or Exclusive Split nodes. Given that these are components that, on average, appear frequently in Microflows this is not a surprising observation. Interestingly, for example, Call and Retrieve Actions appear at a much higher rate than what one would expect to be typical in any general application. The focus of the analysed applications is clearly on retrieving and processing data. We do not find any patterns that involve complex structures, e.g. having nested branches, meaning that at least for our dataset these structures did not occur frequently.

It gets more complicated when we try to extract a deeper meaning from the discovered patterns. In the previous Chapter, we note that the average Microflow is modified less than 3 times, which translates to not being able to discover meaningful edit patterns using our approach. Instead, we see patterns that mostly contain insertions and deletions, which match with the edit characteristics discussed in Section 4.2, and can be interpreted as the frequent creation and deletion of Microflows. We do see some edits, for example, the Retrieve Source node being edited in Figure 4.4, but not to the degree of clear discernible anti-patterns.

Mendix does not have a concept such as classes and, in principle, functionality is implemented via Microflows. Them being edited little directly raises questions about the re-usability of Microflows, which would seem to be limited when considering this data. However, it is difficult to make such a definitive conclusion as this behaviour could potentially be entirely attributed to the way developers interact with the platform, rather than it being a true problem concerning re-usability. There is anecdotal evidence that developers often re-make Microflows when they make non-trivial edits, rather than edit the existing ones. In our analysis, due to the unique id of the Microflow changing, this would show up as a deletion of an existing Microflow and the insertion of a new one. While in reality it could be considered a modification of an existing Microflow. This could be due to the nature of low-code development, where rapid prototyping and experimentation might lead to the frequent creation of new Microflows. Additionally, the low number of edits indicates that Microflows

might be less about incremental refinement and more about replacement, which could be a characteristic of low-code development practices.

Identifying whether this is truly the case is difficult since any arbitrary property of the Microflow could have changed between revisions, combined with the large diffs we are dealing with it is not a trivial task to track a Microflow across revision borders when the developer recreated it and then deleted the original.

An alternative would be to give up on incorporating semantics and consider the whole edit set within a transaction. However, related work, as discussed in Section 6.2, shows that frequent item set mining using VCS commits is not a viable solution due to the creation of sets that contain common, but unrelated changes [28, 29, 30]. Our transactions are even less granular thus amplifying this issue. Setting a limit on the size of transactions, only considering those under a certain threshold, might help to alleviate the problem, albeit not solve it. With the large sizes of our transactions, this is not a workable solution with the current dataset.

Another approach would be to do a more precise analysis of the commits that are known to contain (bug) fixes, or even look at other artifacts related to development. However, as we state in Section 2.2, we do not have access to the data to perform a more fine-grained analysis using commit messages or other artifacts.

In summary, our findings suggest that Microflows undergo minimal modifications post-creation, raising questions about their re-usability in Low-Code environments. However, due to the constraints of our data and the nature of Low-Code development, it remains challenging to draw definitive conclusions. Future research with more granular data may provide deeper insights into these patterns and their implications.

Research Question #2: Identifying Anti-Patterns with the available data from Mendix remains a difficult task. While our approach does find patterns in Microflow edit behaviour, these do not translate to anti-patterns due to the nature of these patterns. Since individual Microflows are not often directly edited in practice. There are some signals that they might be recreated rather than directly edited, future research could investigate this avenue either through improving data granularity, including additional data artifacts, or attempt to track recreated Microflows across revision borders.

Research Question #3: In terms of the frequency of pattern appearance, we can recognize certain types of edits and groups of components that appear frequently in our mined patterns. As we are unable to locate any anti-pattern it is difficult to establish their presence within Microflows.

5.2 Existing Guidelines

In this section, we discuss the existing guidelines for developing applications using the Mendix platform and insights from relevant literature on Low-Code Development Platforms (LCDPs).

5.2.1 Development Guidelines

Like most development platforms, Mendix provides a comprehensive set of guidelines aimed at ensuring the development of high-quality and maintainable applications [25]. These guidelines cover a broad spectrum of development best practices, including naming conventions, project structure organization, security protocols, maintainability strategies, user

interface (UI) design, and performance optimization. A significant emphasis is placed on the readability and manageability of Microflows, particularly through naming conventions, size limitations, and the strategic splitting of larger flows to reduce complexity. Developers are encouraged to enhance clarity by incorporating annotations and following structured design principles.

However, while these guidelines focus on general usability and application performance, they do not delve deeply into the internal logic of Microflows. A notable exception is the emphasis on Input Validation, which underscores the importance of validating and updating input data during development. Additionally, Mendix provides concrete performance-related recommendations, offering developers actionable steps to optimize application performance. Despite these inclusions, the guidelines generally lack explicit measures for the identification and mitigation of problematic code patterns or anti-patterns within Microflows.

Similarly, OutSystems, another prominent Low-Code platform, also offers a set of best practices designed to guide developers towards building efficient, scalable applications. Like Mendix, these practices emphasize structural clarity, security, and performance but do not focus heavily on the detection or correction of anti-patterns, highlighting a broader gap in Low-Code development platforms' current best practices.

Beyond the platforms, broader research on Low-Code development highlights similar challenges and the need for better tools and guidelines to support developers. Studies by Khoram et al. [20] and Pinho et al. [34] emphasize the usability of Low-Code platforms and the importance of providing comprehensive testing frameworks and clear coding standards to aid Citizen Developers, who often lack formal programming education [20]. These studies underscore the necessity for tailored guidelines that not only cover general development practices but also offer concrete methodologies for identifying and mitigating anti-patterns.

Research Question #1: The existing guidelines provided by platforms like Mendix and OutSystems are primarily oriented towards general best practices and operational efficiency. These guidelines promote reusable Microflows and advise against the overuse of constants and event handlers. However, they fall short of explicitly addressing the identification and mitigation of specific code patterns or anti-patterns. While these general recommendations are useful, they do not offer detailed strategies for detecting and correcting problematic patterns in Low-Code environments.
Beyond platform-specific guidelines, the literature on Low-Code Development Platforms (LCDPs) highlights similar gaps. Research by Khoram et al. and Pinho et al. points to the challenges faced by Citizen Developers, who often lack formal programming education, and the need for more comprehensive tools and guidelines [20, 34]. These studies emphasize the importance of providing not only general development practices but also concrete methodologies for identifying and mitigating anti-patterns, suggesting that current guidelines are insufficient in this regard.

5.3 Threats to Validity

Here, we outline several potential threats to the validity of our findings and describe the approaches used to address them.

5.3.1 Generalizability to Other Low-Code Platforms

Our study focused exclusively on the Mendix platform, which raises concerns about the applicability of the results to other Low-Code Development Platforms (LCDPs). While Mendix is a leading platform in the Low-Code space, our findings may not be directly transferable

to other LCDPs with different architectures and functionalities. However, the fundamental principles of anti-pattern detection and the methodological approach we used are relevant for similar platforms like Outsystems, which share the core Low-Code principle. Further research involving multiple platforms would be beneficial to confirm the generalizability of our results.

5.3.2 Sample Size and Data Selection Criteria

The number of projects analysed in this study was limited to 13, which may not be representative of the broader spectrum of Mendix applications. We selected a diverse set of projects with a significant number of revisions. Nevertheless, the relatively small sample size means that our findings may not capture the full range of anti-patterns present in the wider population of Mendix applications. Future studies with a larger and more varied sample size are necessary to validate the robustness of our findings across different contexts and improve the reliability of the conclusions drawn.

5.3.3 Data Collection Constraints

The data collected by Mendix was anonymized and lacked certain details such as commit messages, authorship information, and precise timestamps. This absence of detailed metadata limited the depth of our analysis, as we were unable to consider any contextual information. We focused on structural changes and anti-patterns that could be identified through available edit logs. Despite this limitation, our methodology was designed to extract meaningful insights from the data provided. Future research could benefit from more granular data to provide a deeper understanding of the factors driving the observed patterns.

5.3.4 Platform-Specific Bias

The specific features and functionalities of Mendix may have influenced the types of anti-patterns detected, potentially introducing a platform-specific bias. While our results are inherently tied to Mendix's characteristics, the identified anti-patterns are common in Low-Code environments and should be considered when developing general guidelines for LCDPs. To mitigate this bias, further research involving multiple platforms would help in understanding whether these findings are universally applicable or specific to Mendix. This broader approach would enhance the external validity of the study.

5.3.5 Manual Analysis

The manual analysis involved in pattern detection and categorization might introduce subjective biases. To minimize subjectivity, we established clear criteria for when to consider a pattern "common". Moreover, all patterns we identified can be found in our dataset package[1] [31].

[1] DOI: 10.6084/m9.figshare.27223998

Chapter 6

Related Work

As we briefly discuss in Chapter 1, previous work on Low-Code is scarce. Most related work focuses on highlighting the current challenges and roadblocks within the Low-Code space, rather than directly addressing them [5, 17, 20, 38]. To the best of our knowledge, a study covering anti-patterns, bugs present in Low-Code applications, or even edit behaviour of Low-Code developers does not exist.

This is in contrast with the general field of repository mining, which has seen much interest in the past and remains an important scientific field today. Specifically, plenty of work exists on how to mine anti-patterns, code smells, or common developer edit patterns from code repositories.

For this thesis, we will split these existing approaches into two categories. One category uses additional artifacts besides the commit history, such as commit messages, bug reports, logs, or a combination of these alongside the source code to learn developer edit patterns or discover common anti-patterns [6, 21, 22, 33, 39, 45]. The second category consists of approaches that solely consider the changes in source code between different revisions to identify common edit patterns, ignoring any additional data [43, 29, 30, 19, 18, 11].

The second category aligns with the data we have been provided through Mendix, it does not include any additional information besides the historical changes made to an application over time. For example, it lacks commit messages, bug reports, and author data. As such, it is not feasible to analyse the existence of common anti-patterns within Low-Code applications using approaches falling in the first category with the data we have been provided.

While the second category aligns with the data we have been provided through Mendix, the first category contains an important body of research of which we provide a brief overview in the next section. Afterwards, we will discuss the second category of methods in more detail.

6.1 Category 1: Methods using Additional Artifacts

A significant body of research within the field of repository mining focuses on identifying anti-patterns and code smells through the usage of additional artifacts alongside the code changes in a commit.

One of the most common artifacts being used is the commit message provided by the developer to describe the commit. One can use the commit message to identify commits that are, for example, likely to include bug fixes or code smells [6, 33].

Other data types can also be considered, such as bug reports that are often linked to the specific commit in which they were fixed [21]. Similarly, one could use system logs to search for certain events, errors, and user interactions [46].

Some studies attempt to link edit patterns to specific developers or projects, enabling personalized feedback or insights [30].

By combining these artifacts with source code analysis, researchers have developed techniques to identify commits that introduced or resolved bugs, thereby uncovering patterns associated with software defects.

Several of these advancements have resulted in practical tools, such as Dynamine, a tool introduced by Livshits and Zimmerman, which uses dynamic analysis and bug reports to detect recurring bug patterns [22]. Similarly, Yang et al. presented a method for mining fix patterns from bug-fixing commits, aiming to automate the program repair process in their work Mining Fix Patterns for Automated Program Repair [45].

Another significant development is the application of machine learning algorithms to learn bug-fixing patterns by analyzing commit messages alongside the corresponding source code changes [6]. For instance, Rolim et al. proposed a machine learning approach in Learning from Bug-Fixing Changes, which learns bug-fixing patterns from commit messages and source code modifications [39].

While these approaches offer valuable insights into bug patterns, their reliance on additional artifacts poses a challenge for analysing Low-Code applications. The limited data available from Mendix, as discussed in Chapter 2.2, restricts the applicability of these techniques in the context of this thesis.

6.2 Category 2: Methods Solely Based on Change History

Another group of papers focuses on extracting patterns without using any additional artifacts besides the difference between different revisions [11, 14, 18, 19, 29, 43, 30]. One paper proposes locating a specific kind of bug by seeing if a returned variable is tested, e.g. verified if the variable is null or some other check, before being used in some other. If it is not tested initially, but such a check is introduced by a developer in a later revision of the code it is assumed a bug was present that the developer attempted to fix [43]. While interesting this approach requires upfront knowledge of the type of patterns we want to identify, and is thus not useful to identify new ones. Often, other approaches suffer from similar problems in that they are limited in terms of patterns that can be detected, only focus on a specific component, or have a high false positive rate [43].

Negara et al. [29] implemented an approach focusing on mining sub-sequences from chronologically ordered edits, using the time dimension to capture code semantics. In an earlier study, Negara et al. [28] point out that version control systems (VCS) often provide incomplete or imprecise data for effective pattern detection, so they collected edit data directly from an Integrated Development Environment (IDE), tracking every keystroke across over 1,500 hours of development, yielding a precise edit dataset. However due to the dataset required such an approach is not suitable for our problem, as we describe in Section .

When considering graph-based approaches, such as the one we propose, that rely on converting the edited snippet to an AST or graph. Multiple existing approaches exist, all these identify the differences between the AST or graph before and after revision, and cluster based on those differences [30, 11, 14, 19, 18, 10]. However, none of these use a multi-layered approach, and none are applied to Low-Code specifically.

Nguyen et al. propose a graph-based approach applied to a VCS's commit data, which they show outperforms the sub-sequence mining approach suggested by Negara et al. [30, 29]. In this approach, they generate edit graphs on which a frequent sub-graph mining algorithm is applied. They specifically focus on individual projects, and thus edit patterns within the context of a single project.

Other approaches try to find patterns that are to be considered general, or not project-specific [19]. Due to the nature of our problem, our approach also falls into this category.

Another interesting paper proposes the tool FS3-Change, another graph-based approach, as a method for change pattern mining, which focuses on being scale-able and can handle a

large number of data points, compared to existing graph-based approaches [18].

To conclude, while still a relatively recently developed approach, graph-based methods like FS3-Change demonstrate the potential for handling large-scale datasets [18]. They have the benefit of not requiring additional artifacts and can handle large edit transactions. Our method extends these graph-based approaches by applying them to Low-Code and incorporating a multi-layered time component. This allows us to track the evolution of patterns across multiple revisions, offering a more comprehensive view of development trends and anti-patterns.

Chapter 7

Conclusion

This thesis set out to investigate the identification of anti-patterns in Low-Code, with a specific focus on the Mendix platform. Through the application of a multi-layered graph-based pattern mining approach, we sought to address three core research questions concerning Mendix's existing guidelines, the nature of anti-patterns in Mendix applications, their prevalence, and their potential impact on application quality.

Regarding Research Question 1, we explored Mendix's guidelines and observed that while the platform offers foundational best practices, these guidelines remain focused on usability and operational efficiency. Critically, they do not delve deeply into specific code quality concerns. This leaves a considerable gap in available strategies for addressing code quality issues unique to Low-Code platforms. This gap underscores the need for more targeted and granular guidance for developers.

For Research Question 2, while our graph-based approach successfully highlighted recurring patterns, the limitations of the dataset and in particular the lack of detailed revision histories—hindered our ability to discover more complex edit patterns. Which in turn made the extraction of anti-patterns difficult. The frequent appearance of patterns that relate to data retrieval and conditional flows in the results suggests that is an important focus point of Microflows. Additionally, there is anecdotal evidence that developers often remake Microflows when they make non-trivial edits, rather than edit the existing ones. The fact that we observe few edits to the average Microflow, combined with edit patterns mainly concerning creations and deletions, seems to support this hypothesis. This raises questions about the re-usability of Microflows.

Regarding Research Question 3, we established a baseline understanding of the prevalence of various patterns in Mendix applications. The findings reveal that most patterns observed are simple structures, such as basic conditional flows and retrieval actions, and that more complex patterns are rare. The frequent recreation of Microflows instead of modification raises concerns about reusability and development practices within the Mendix ecosystem.

In conclusion, this research highlights important aspects of anti-patterns in Low-Code environments. Although our approach provided some initial insights into the existence and prevalence of these patterns, the limitations of the available dataset call for more refined datasets to support comprehensive anti-pattern detection. Further research, particularly leveraging more granular data, is needed to deepen our understanding of how these patterns affect application quality.

7.1 Future Work

In terms of future work, there remain numerous angles yet to be investigated regarding Low-Code Development Platforms (LCDPs). If one considers patterns and anti-patterns specif-

ically, the single largest leap that could be taken pertains to the quality and granularity of the available data, which would allow for a much more refined analysis of a broader range of applications. Unfortunately, the lack of an open-source Low-Code community hinders progress in this area. However, despite these challenges, several research opportunities are present that could be pursued to deepen the understanding of Low-Code environments.

In addition to the technical aspects of Low-Code platforms, there is a need to explore the sociotechnical dimensions of these environments, particularly how different user groups, including Citizen Developers and professional developers, interact with these platforms. Future studies could focus on the strategies employed by these users while working with Low-Code platforms, investigating how these processes differ between Citizen Developers and traditional developers. Such studies could provide insights into the unique needs and challenges faced by each group, informing the design of more user-centric Low-Code platforms that cater to a broader range of users.

Furthermore, given the increasing adoption of Low-Code platforms in various industries, there is a pressing need to investigate the impact of Low-Code development on software engineering practices and organizational processes. Studies could examine how Low-Code development influences the roles and responsibilities of software engineers, the dynamics of development teams, and the overall software development life cycle.

Bibliography

[1] Mehenaz Afrin et al. "A Hybrid Approach to Investigate Anti-pattern from Source Code". In: *2022 25th International Conference on Computer and Information Technology (ICCIT)*. IEEE. 2022, pp. 888–892.

[2] Carolin Brandt and Andy Zaidman. "Developer-centric test amplification: The interplay between automatic generation human exploration". In: *Empirical Software Engineering* 27.4 (2022), p. 96.

[3] William H Brown et al. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[4] Rina Diane Caballar. "Programming without code: The rise of no-code software development". In: *IEEE Spectr. Tech Talks*. 2020.

[5] Jordi Cabot. "Positioning of the low-code movement within the field of model-driven engineering". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 2020, pp. 1–3.

[6] Eduardo C Campos and Marcelo de A Maia. "Discovering common bug-fix patterns: A large-scale observational study". In: *Journal of Software: Evolution and Process* 31.7 (2019), e2173.

[7] Kamal A El-Dahshan, Eman K Elsayed, and Naglaa E Ghannam. "Comparative Study for Detecting Mobile Application's Anti-Patterns". In: *Proceedings of the 8th International Conference on Software and Information Engineering*. 2019, pp. 1–8.

[8] Aurelien Delaitre et al. "Evaluating Bug Finders – Test and Measurement of Static Code Analyzers". In: *2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS)*. 2015, pp. 14–20. DOI: 10.1109/COUFLESS.2015.10.

[9] Davide Di Ruscio et al. "Low-code development and model-driven engineering: Two sides of the same coin?" In: *Software and Systems Modeling* 21.2 (2022), pp. 437–446.

[10] Malinda Dilhara et al. "Discovering repetitive code changes in python ml systems". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 736–748.

[11] Sedick David Baker Effendi et al. "A language-agnostic framework for mining static analysis rules from code changes". In: *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2023, pp. 327–339.

[12] Gartner. *Magic Quadrant for Enterprise Low-Code Application Platforms*. G0078582. 2023.

[13] Pedro M Gomes and Miguel A Brito. "Low-code development platforms: a descriptive study". In: *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*. IEEE. 2022, pp. 1–4.

[14] Yoshiki Higo, Junnosuke Matsumoto, and Shinji Kusumoto. "Tree-based mining of fine-grained code changes to detect unknown change patterns". In: *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE. 2021, pp. 61–71.

[15] Nirali Honest. "Role of testing in software development life cycle". In: *International Journal of Computer Sciences and Engineering* 7.5 (2019), pp. 886–889.

[16] Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. "A low-code development environment to orchestrate model management services". In: *IFIP International Conference on Advances in Production Management Systems*. Springer. 2021, pp. 342–350.

[17] Alexandre Jacinto, Miguel Lourenço, and Carla Ferreira. "Test mocks for low-code applications built with OutSystems". In: *Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems: companion proceedings*. 2020, pp. 1–5.

[18] Mario Janke and Patrick Mäder. "FS 3 change: A Scalable Method for Change Pattern Mining". In: *IEEE Transactions on Software Engineering* (2023).

[19] Mario Janke and Patrick Mäder. "Graph based mining of code change patterns from version control commits". In: *IEEE Transactions on Software Engineering* 48.3 (2020), pp. 848–863.

[20] Faezeh Khorram, Jean-Marie Mottu, and Gerson Sunyé. "Challenges & opportunities in low-code testing". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 2020, pp. 1–10.

[21] Chen Liu et al. "R2Fix: Automatically generating bug fixes from bug reports". In: *2013 IEEE Sixth international conference on software testing, verification and validation*. IEEE. 2013, pp. 282–291.

[22] Benjamin Livshits and Thomas Zimmermann. "Dynamine: finding common error patterns by mining software revision histories". In: *ACM SIGSOFT Software Engineering Notes* 30.5 (2005), pp. 296–305.

[23] Yajing Luo et al. "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)*. 2021, pp. 1–11.

[24] Mendix. `https://www.mendix.com/company/`. 2024. (Visited on 2024).

[25] Mendix. *Mendix Guidelines*.

[26] Mendix. *Mendix. The State of Low-Code 2021: a Look Back, the Light Ahead*. 201.

[27] Judith Michael and Andreas Wortmann. "Towards development platforms for digital twins: A model-driven low-code approach". In: *Advances in Production Management Systems. Artificial Intelligence for Sustainable and Resilient Production Systems: IFIP WG 5.7 International Conference, APMS 2021, Nantes, France, September 5–9, 2021, Proceedings, Part I*. Springer. 2021, pp. 333–341.

[28] Stas Negara et al. "Is it dangerous to use version control histories to study source code evolution?" In: *ECOOP 2012–Object-Oriented Programming: 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings 26*. Springer. 2012, pp. 79–103.

[29] Stas Negara et al. "Mining fine-grained code changes to detect unknown change patterns". In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 803–813.

[30] Hoan Anh Nguyen et al. "Graph-based mining of in-the-wild, fine-grained, semantic code change patterns". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 819–830.

[31] Wessel Oosterbroek. *Discovering Common Anti-patternsPresent in Low-Code usingMulti-Layered Graph-Based PatternMining Dataset*. 2024. DOI: 10.6084/m9.figshare.27223998. URL: https://dx.doi.org/10.6084/m9.figshare.27223998.

[32] OutSystems. https://www.outsystems.com/1/state-app-development-trends. 2019. (Visited on 2019–2020).

[33] Kai Pan, Sunghun Kim, and E James Whitehead. "Toward an understanding of bug fix patterns". In: *Empirical Software Engineering* 14 (2009), pp. 286–315.

[34] Daniel Pinho, Ademar Aguiar, and Vasco Amaral. "What about the usability in low-code platforms? A systematic literature review". In: *Journal of Computer Languages* 74 (2023), p. 101185.

[35] Niculin Prinz, Christopher Rentrop, and Melanie Huber. "Low-Code Development Platforms-A Literature Review." In: *AMCIS*. 2021.

[36] Clay Richardson et al. "New development platforms emerge for customer-facing applications". In: *Forrester: Cambridge, MA, USA* 15 (2014).

[37] Arthur J Riel. *Object-oriented design heuristics*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[38] Karlis Rokis and Marite Kirikova. "Challenges of low-code/no-code software development: A literature review". In: *International Conference on Business Informatics Research*. Springer. 2022, pp. 3–17.

[39] Reudismam Rolim et al. "Learning quick fixes from code repositories". In: *arXiv preprint arXiv:1803.03806* (2018).

[40] Raquel Sanchis et al. "Low-code as enabler of digital transformation in manufacturing industry". In: *Applied Sciences* 10.1 (2019), p. 12.

[41] Zevin Shaul and Sheikh Naaz. *cgSpan: Closed Graph-Based Substructure Pattern Mining*. 2021. arXiv: 2112.09573 [cs.AI].

[42] Massimo Tisi et al. "Lowcomote: Training the next generation of experts in scalable low-code engineering platforms". In: *STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019)*. 2019.

[43] Chadd C Williams and Jeffrey K Hollingsworth. "Automatic mining of source code repositories to improve bug finding techniques". In: *IEEE Transactions on Software Engineering* 31.6 (2005), pp. 466–480.

[44] Xifeng Yan and Jiawei Han. "gspan: Graph-based substructure pattern mining". In: *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* IEEE. 2002, pp. 721–724.

[45] Yilin Yang et al. "Mining Python fix patterns via analyzing fine-grained source code changes". In: *Empirical Software Engineering* 27.2 (2022), p. 48.

[46] Tianzhu Zhang et al. "System Log Parsing: A Survey". In: *IEEE Transactions on Knowledge and Data Engineering* 35.8 (2023), pp. 8596–8614. DOI: 10.1109/TKDE.2022.3222417.

Acronyms

IDE Integrated Development Environment

LCDP Low-Code Development Platform

MDE Model-Driven Engineering

GUI Graphical User Interface

FSM Frequent Subgraph Mining

AST Abstract Syntax Tree

UI User Interface

VCS Version Control System