UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Dottorato di Ricerca in Ingegneria e Scienze dell'Informazione

Emerging computing models: algorithms, software architectures and intelligent systems

XXXV ciclo

Titolo della tesi

Scalable and Extensible Cloud-Based Low-Code Model Repository

SSD INF/01

Dottorando

Arsene Indamutsa

Coordinatore del corso

Prof. Vittorio Cortellessa

Tutor

Prof. Alfonso Pierantonio

Co-Tutor

Prof. Davide Di Ruscio

a.a. 2021/2022

Abstract

Low-code development platforms (LCDPs) are becoming increasingly common in the software industry. By leveraging visual diagrams, dynamic graphical user interfaces, and declarative languages, these platforms support the development of full-fledged applications in the cloud. However, given the rapid evolution of these platforms, they encounter a fleet of challenges and limitations. To address the challenges in Low-Code Development Platforms, it's essential to study their core concepts and technologies, primarily Model-Driven Engineering (MDE) and cloud computing. Despite MDE's progress, its broader adoption is hindered by challenges faced by practitioners. The first obstacle is efficient support for discovering and reusing existing model artifacts. The development of similar tools and extensions leads to resource wastage, undermining productivity and collaboration in model-based processes. Additionally, local deployment of modeling environments causes scalability, extensibility, collaboration, and performance challenges. Consequently, modelers are required to engage in a process that involves downloading both artifacts and executables to their local machines. This is a prerequisite step before initiating a potentially intricate and lengthy setup of Model-Driven Engineering (MDE) tools prior to their effective utilization.

Throughout this dissertation, we attempted to advance state-of-the-art toward understanding and supporting cloud-based modeling in terms of LCDPs. Therefore, we aimed to enhance the scalability and extensibility of modeling infrastructures by developing a cloud-based low-code model repository. This approach goes beyond the typical implementation of repositories with simple storage and query capabilities. We provide a large-scale repository and services for low-code engineering (LCE). The implemented repository enables access, persistence, discovery, and reuse of modeling artifacts via scalable and extensible approaches and infrastructures. In the LCE context, core services are containerized, orchestrated, and deployed as cloud services. The repository's functionalities can be extended via its remote API or by adding functionality in the form of extensions and services. Finally, an integrated web-based search platform and various domain-specific languages are devised to support various mechanisms for composing, discovering, and reusing persisted artifacts and model management services.

Acknowledgements

First and foremost, I would like to express my gratitude to Jehovah for His unwavering guidance and strength throughout this trying process.

My heartfelt gratitude goes to my advisor Prof. Alfonso Pierantonio, my co-advisor Prof. Davide Di Ruscio, and Dr. Juri di Rocco. Their supervision, constructive criticism, constant support, and critical feedback helped me overcome research challenges during the COVID-19 pandemic. I would also like to thank my colleagues and fellow students in the EU Lowcomote project and the Department of Information Engineering, Computer Science, and Mathematics at the University of L'Aquila. Their stimulating discussions and collegial environment have fostered my academic growth and allowed me to gain valuable insights and new perspectives. In addition, I am grateful for my secondment to UGround Global, S.L., Madrid, where I had the opportunity to expand my professional expertise.

Finally, I could not have survived this tough time without my wife's love and steadfast support. I thank my mother and family for always believing in me - your support meant everything to me and played a crucial role in my academic journey.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

The history of computer science and computational thinking is marked by a continuous pursuit of abstraction and automation to systematically solve complex problems and design intricate systems [76, 53]. These approaches advanced software engineering, not necessarily resorting to recurrent traditional programming paradigms [5]. However, despite gaining wide traction in the 1990s thanks to rapid application development tools and fourth-generation programming languages, they never achieved a dominant position in the software development space until the current digital transformation [206, 53]. This digital transformation has rendered software technologies prevalent staples in our daily life. As a result, traditional programming paradigms can neither keep up the pace of change and complexity of modern applications nor adapt or sustain the increasing demand for rapid development and deployment of software systems [54, 42, 193].

The size and complexity of software artifacts and tools highlight that software development can no longer be considered self-contained. As a matter of fact, as software systems become more intricate, effective communication and collaboration become crucial at various levels of abstraction [48]. This interaction and collaboration occurs between technical staff, customers, and other stakeholders and bridges the gap in understanding throughout the application development lifecycle [42]. Additionally, due to the ubiquity of software technologies in today's society, there is a constant need for updates and functional enhancements of existing software artifacts and tools. However, there is a shortage of skilled software developers in the labor market to meet this demand [43].

Today, a new breed of technologies that tackle this challenge head-on is emerging as one of the industry's significant new trends [191, 25]. Commonly known as low-code development platforms (LCDPs), they sprung out from recent technological maturity seen in software development, model-driven engineering (MDE), and cloud computing [76]. Particularly,

advancements in visual programming, automatic code generation, and cloud infrastructures
have assisted LCDPs' surge in popularity [186]. Furthermore, LCDPs promote greater access
to application development for a more extensive range of digital-savvy users from different
competence spectrums [76]. However, this increased utilization and adoption requires
maximizing efficiency through efficient resource management, discovery, and reuse of model
artifacts among the LCDP user community [137]. Hence, model repositories come into play
by enabling the persistence, organization, discovery, and reuse of model artifacts and tools
within and across LCDP applications. In doing so, model repositories facilitate collaborative
development among citizen developers and reduce upfront investment costs [206].

1.1   Motivation

Although LCDPs have increased usage in enterprises, their rapid evolution is fraught with
challenges hindering their widespread adoption across all software development domains.
Our work is motivated mainly by the following challenges [206, 19]:

- **Scalability**: The scalability of LCDPs is a crucial aspect of their evolution as they
  expand beyond small applications into larger projects. To fully meet the needs of
  large-scale and mission-critical enterprise applications, these platforms should acquire
  technical capabilities to handle large amounts of data and computation while supporting
  numerous collaborating users in a social network-style environment.

- **Discovery and reuse of modeling artifacts and tools**: The limited discovery and
  reuse of model artifacts and tools within these platforms and MDE generally leads
  to the wasteful recreation of similar artifacts and tools. The lack of adequate reuse
  environments and mechanisms in these platforms further hinders the discovery and
  reuse processes. This issue must be addressed to ensure optimal resource utilization
  and effective collaboration to reach their full potential.

- **Migrating local-based modeling environments to the cloud**: LCDPs have the
  potential to revolutionize modeling; however, their evolution is highly dependent
  on their widespread adoption in the MDE community. Currently, modelers often
  work in isolated local environments, which limits the flexibility and scalability of
  resources and collaboration. In addition, complex, error-prone constant installations
  and configurations hinder the reusability of model artifacts and tools. For modelers
  and modeling engineers to fully embrace this paradigm shift, it is critical that cloud-
  based modeling provide simplicity and improved functionality compared to current
  local-based modeling methods. This includes seamless stakeholder collaboration and
  increased flexibility and scalability regarding the resources involved.

- **Heterogeneity support**: LCDPs have been fostered for citizen developers or non-programmers with specialized knowledge in a particular engineering domain to create applications specific to their field [206]. Modeling engineers, therefore, often expect to easily incorporate their domain expertise into the application using standard formalisms and levels of abstraction. However, the current integration of heterogeneous model artifacts and tools from different engineering disciplines requires more support, potentially limiting the effectiveness of these platforms.

- **Vendor lock-in**: Although LCDPs are becoming more popular, LCDP vendors are implementing their unique programming models and paradigms [186]. Unfortunately, this leads to technological fragmentation and difficulties in cross-platform integration. This phenomenon, known as vendor lock-in, can prove problematic for large-scale projects requiring the reuse of multiple sources of functionalities and artifacts. In addition, implementing existing languages in a cross-platform language would require significant resources and rework, making integration economically unviable.

- **Quality of artifacts**:

  Due to the burgeoning phase of LCDPs, advanced aspects such as quality control are often overlooked. Therefore, reusability or sharing artifacts without adherence to modeling standards often leads to quality degradation and the proliferation of undesirable practices. Assessing artifacts' quality prior to reuse can increase modeler productivity, ultimately contributing to delivering high-quality systems through the robustness and resilience of system performance.

- **Intellectual property**: As we promote the discovery and reuse of modeling artifacts within these platforms, we must implement motivational measures to safeguard the intellectual property rights of users and modelers. Active participation in a collaborating community can easily be maintained by nurturing a sense of trust and fairness. As a result, a dynamic ecosystem is formed wherein the collective wisdom enriches the experiences and benefits all stakeholders involved.

1.2 Research Objectives

The high-level overview of this work is shown in fig. 1.1. The figure can be interpreted from top to bottom. The conceptual map has been organized into layers, each representing a specific research activity to address challenges in section 1.1. The objective is to develop a scalable and extensible cloud-based model repository, with the architectural design revolving around scalability and extensibility. Hence, the API of the repository is designed to support remote access and provide a scalable deployment model for integrating new tools and extensions. The

primary role of the repository is to offer a scalable infrastructure for persisting model artifacts and to facilitate the discovery and reuse of artifacts and tools both within the repository and externally. The following research objectives guided our work:
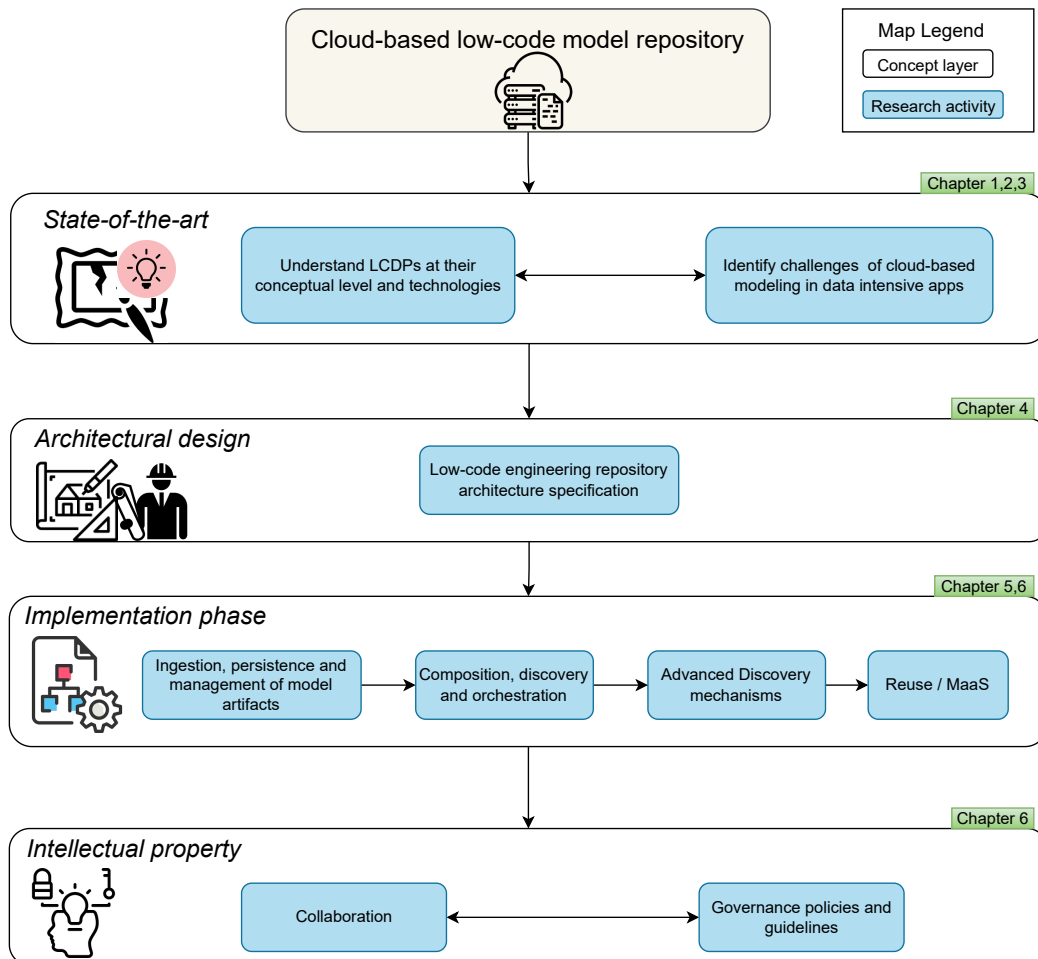
Fig. 1.1 Conceptual map of the dissertation.

- **State-of-the-art of LCDPs**: The LCDP industry is relatively new and has yet to be explored extensively, resulting in a lack of taxonomy and a shared understanding of their state of the art. For LCDPs to be widely adopted in mainstream application development, we must understand their challenges and opportunities in their engineering workflow. While LCDPs have the potential to simplify and speed up application development, they also come with certain limitations and potential security risks. Through thorough investigation and comprehensive research into LCDPs, we can gather valuable insights and data to establish a solid foundation of best practices for their optimal utilization when building mission-critical applications.

- **Open challenges and opportunities of cloud-based modeling**: The increasing adoption of LCDPs brings new challenges and opportunities for cloud-based modeling. As a matter of fact, as modeling engineers are used to local environments, switching to new gears can prove inconvenient. Nevertheless, these challenges and opportunities provide valuable opportunities for further research in the modeling community. As more resources are migrated to the cloud, these challenges provide valuable research targets for the modeling community. Additionally, studying the opportunities presented by cloud-based modeling paves the way for increased adoption and improved efficiency of LCDPs.

- **Scalabale and extensible cloud-based model repositories**: At the heart of the work done in this dissertation is addressing challenges related to the scalability and extensibility of modeling environments, especially model repositories. The intended repository is not just a storage of model artifacts with simplistic retrieval mechanisms but a complex system of services and tools involved in the persistence and management of low-code model artifacts and tools. Users can easily access on-demand artifacts and tools without downloading them or relying on third parties. Below we address objectives directly related to this concept:

  - *Organization and persistence of heterogeneous artifacts*: Ideally, model repositories should enable effortless persistence, discovery, and retrieval of model artifacts from various domains. Hence, proper reverse-engineering mechanisms should be used to extract critical information from these artifacts to facilitate their access and organization. In addition, these mechanisms should allow a dynamic resolution of appropriate artifact extractors at runtime. Furthermore, integrating artifacts from different technologies and domains should not lead to fragmentation in the repository but instead provide a flexible and scalable unified approach for efficient data storage, discovery, and retrieval.

  - *Scalability*: As MDE is applied to larger and more complex systems, the infrastructures and approaches must scale simultaneously, respective to the workload. Additionally, these infrastructures should exhibit resiliency, self-healing, and high availability to support significant model artifacts, computationally expensive operations, and a large user community.

  - *Extensibility*: The ability to easily extend new functionality in a model repository is still limited due to technological dependency, local-based modeling environments, and non-extensible architectures. These local environments often have limited capacity and require time-consuming installations and configurations,

making it difficult to add new features. Despite these challenges, extensibility is crucial for promoting collaboration in the repository without sacrificing its core functionalities.

- **Architecture specifications of a scalable and extensible cloud-based model repository**: The system's architectural design should reflect the current breakthrough approaches and technologies to achieve these features. Hence, it is imperative to consider the expectations of the MDE community during the design phase. Thus identifying the challenges and opportunities of cloud-based modeling is the foundation for successful architectural design to ensure that the needs of the MDE community are met. Furthermore, the process should focus solely on the essential core functions of the system to avoid a quagmire. Such a design paradigm ensures that system key features are maintained and noticed from the onset.

- **Discovery mechanisms**: Discovery and reuse in the repository should be considered from two points of view: model artifact discovery and service discovery. Model artifact discovery involves establishing mechanisms that locate model artifacts based on user-defined criteria. These criteria should not be limited to the internal properties of the artifacts but also to the megamodel or other measures such as their quality or the operations in which they were involved. Therefore, fast retrieval of model artifacts requires scalable indexing mechanisms to support large models and many users. On the other hand, service discovery requires a service registry to enable the discovery of model management tools and extensions. In addition, users must be able to find available services and how they can reuse them.

- **Reusability**: This concept is at the heart of model repository and software development. It enables resource sparing hence reducing upfront efforts either in investments or infrastructures. Reusability in LCDPs is vital to provide mature components tested by different users and platforms, resulting in scalable and stable development. Model artifacts can be manipulated with visual editors or through application programming interfaces (API). Once manipulated, they can be reusable in model management operations. This way, both model artifacts and model management tools and extensions are mutually reused. Reusing them on the same platform where the artifacts were discovered should be possible without downloading them locally or in a different environment before their use.

- **Composition of model management tools and extensions**: They can be composed into workflows by designating model management operations as participants in a higher-level goal. This way, users can upload an artifact, validate it, and transform

it into a component that serves them in later stages of application development in LCDPs. Thus, an orchestrator is required to implement a cluster of services executed on demand, with the user declaratively specifying the desired outcome, with no manual intervention or oversight.

- **Model management tools as-a-service (*Maas*)**: Since model management tools are typically deployed locally as software packages on top of complex environments, it is difficult for citizen developers to keep track of dependencies and frequent updates in different technologies. By offloading modeling resources to the cloud, citizen developers can engage only with application development that reflects their domain. By deploying model management tools and extensions as services (MaaS) in the cloud, these services are packaged into self-contained environments and consumed remotely via APIs. In addition, API specifications that facilitate their use remotely should be created and deployed online for developers to reuse.

- **Artifact quality control**: As repositories become more popular, incorporating quality control measures during the reuse phase can foster a culture of reuse and improve the overall efficiency of the repository. Without this type of quality assurance, the overall integrity and utility of the repository may be compromised. Because the quality requirements for different model artifacts may vary, these requirements must be clearly defined for each type where possible.

- **Access and reuse policy**: Access control and reuse policies are part of the most important aspects of a repository. These mechanisms allow users to manage the distribution, terms of use, dissemination, and utilization of contents within and outside a repository. Hence, by preserving the intellectual property rights of users, it is easier to encourage collaboration and reuse among citizen developers.

1.3   Research Methodology

We tailored our research process to tackle the challenges highlighted in section 1.1. We grounded our process in systematic strategies that advance the current frontiers of Low-Code Development Platforms (LCDPs). Our methodology is anchored on two key tenets: iterative refinement for consistent enhancement and adaptability to incorporate evolving insights and developments with agility. Our research began when the LCDPs had relatively limited research coverage. Hence our initial endeavor was to bridge this gap by contributing to the foundational understanding of these platforms. We developed a comprehensive taxonomy delineating their distinctive features.

We continued our research by meticulously exploring the state-of-the-art of modeling and cloud-based modeling. We conducted thorough literature reviews to glean valuable insights into prevailing practices, identifying potential obstacles, opportunities, and research gaps in the current landscape. The insights gathered during this stage set up the stage for the subsequent steps of our research.

Transitioning from the exploratory stage, we embarked on the design of a cloud-based low-code model repository. The design was iterative and incremental, allowing us to integrate emerging solutions and thus bolster the repository's architecture in terms of efficiency and utility. The extensibility principle underpins the current architecture, allowing for expansion without undermining the repository's core foundations. The architecture also followed peer-reviewed architecture in software engineering [131].

The implementation phase transformed the above architecture into a functional system. Throughout this process, we maintained a strong focus on testing and refining new features, primarily those addressing the challenges in section 1.1. This phase included ensuring quality control of persisted model artifacts through peer-reviewed approaches that assess their quality before their persistence.

The next phase involved testing the major implementation features of the repository with other modeling systems. To this end, we considered Droid, an MDE recommender system from Universidad Autónoma de Madrid (UAM), to help us assess the practicality and feasibility of our implementations [7]. Lastly, the repository implemented features that enforce safeguarding intellectual property rights. By enforcing intellectual property rights, we foster a secure environment for information exchange, empowering a vibrant community of modelers to collaborate and share their artifacts with other repository users without compromising their safety. The following section outlines the structure of this thesis.

1.4 Thesis Structure

Figure 1.2 represents the structure of this thesis. The grey boxes represent the main activities, which are organized into chapters. These are the activities that guided our research throughout this thesis. The rounded boxes represent the main findings, contributions, and approaches related to each activity. In a nutshell, the chapters are organized as follows:

Chapter 2 - *Low-code development platforms*: As the demand for software continues to proliferate, so does the need for efficient development methods. This is where LCDPs come into play, promising to revolutionize traditional software development. In this chapter, we conducted a technical investigation of these platforms to understand their underlying concepts and technologies and the challenges and opportunities they present. We studied
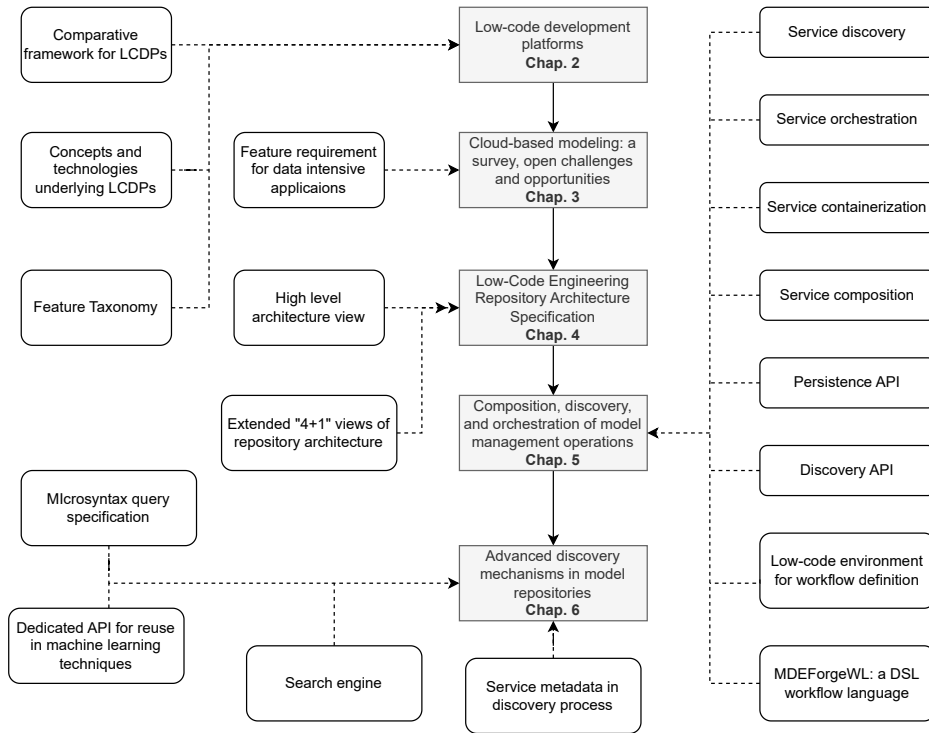
Fig. 1.2 Thesis structure.

eight representative platforms and identified their respective supported features to support the state-of-the-art of these platforms. We created a feature taxonomy to facilitate selection among current LCDPs. Our comparative framework allows us to analyze the functionalities and services of each platform, ultimately leading to a better understanding of LCDPs as a potential new paradigm for software development.

Chapter 3 - *Cloud-based modeling: a survey, open challenges and opportunities*: The use of cloud-based modeling tools and platforms is becoming increasingly popular in the development of data-intensive applications, particularly in the Internet of Things (IoT) field. Through a comprehensive review of 625 articles, we investigated 22 different approaches to cloud-based IoT system development. Our findings highlighted both the benefits and challenges associated with this trend and helped inform the design of our repository for cloud-based modeling infrastructure.

Chapter 4 - *Low-code engineering repository architecture specification*: In our ongoing effort to address the issues of scalability and extensibility in LCDPs and model-driven engineering in general, we have been exploring the potential of a cloud-based model repository. This chapter presents the architectural design of the developed system, drawing inspiration from the challenges highlighted in the previous chapter. The resulting architecture is designed

to be highly decoupled and distributed, allowing for improved scalability and reuse of model artifacts and tools across platforms such as LCDPs. This architecture aims to alleviate all challenges mentioned in section 1.1

Chapter 5 - *Composition, discovery, and orchestration of model management operations*: Developing complex systems often requires coordinating multiple model management services and repositories. Although several service composition proposals have been made in model-driven engineering, no satisfactory solution has yet been found. This chapter proposes a low-code development environment and a domain-specific language for citizen developers to plan, organize, specify, and execute model management workflows. Exposing model management operations as services enables service composition and relieves developers from managing low-level details. This includes managing the required model management services' discovery, orchestration, and integration. This is achieved through a modular and distributed microservice architecture that facilitates the reuse and extensibility of services within our cloud-based cluster.

Chapter 6 - *Advanced discovery mechanisms in model repositories*: This chapter presents a novel approach to discovering heterogeneous model artifacts in MDEForge, a cloud-based model repository. We have developed advanced mechanisms for contextually retrieving and reusing these artifacts across model management services. We also developed a domain-specific query specification in the form of a microsyntax that enables query formulation using keywords, search tags, conditional operators, quality model assessment services, and a transformation chain discoverer. Finally, the usability of our approach was assessed in a recommender system modeling framework and a search facility application. Both applications enjoy more than 5,000 model artifacts currently persisted in our cloud-based model repository.

Chapter 7 - *Conclusion*: This chapter presents the summary contribution of this thesis. First, we present the tools that have been developed during this research and the publications that have been made. Finally, we describe the potential future work for this research.

Chapter 2

Low-code development platforms

The increasing growth of the *internet* use and ever-changing market demands are leading to an unprecedented digitalization of the world [129, 203]. Companies can no longer rely on creating large, monolithic applications tailored to the needs of their particular business or industry. They are under increasing pressure to be responsive, resilient, and competitive to meet today's market demands for software systems [89]. As a result, software systems are becoming steadily more complex, and their development is becoming more costly and time-consuming [55, 90]. Furthermore, the demand for software applications is growing at a rate that outpaces the available supply of skilled software engineers. In fact, market research firm Gartner has already predicted that demand for software systems will not match the delivery capacity of software companies [143, 210]. As a result, companies are looking for faster, cheaper, and safer ways to meet market demand without compromising quality, productivity, and ethics [90].

In light of the current technological landscape, low-code development platforms (LCDPs) have shifted the paradigm of software development [8]. Aimed at addressing the current challenges of digital transformation, LCDPs provide reliable means for enterprises to create fully-operational applications with minimal effort [203]. Recent studies show that companies are switching to LCDPs to streamline their operations [47]. In addition to reducing the strain on IT resources, LCDPs allow fast product delivery to the market at a lower cost. These platforms can also be used in the prototyping and design phases to effectively collaborate with stakeholders and foster better communication and collaboration across the enterprise [177].

With LCDPs, programming expertise is no longer required to develop highly functional software applications. With these platforms, citizen developers can build software applications without needing a team of professional developers [218]. This allows citizen developers to focus on the business aspects of the application and leave behind tedious details such as

setting up infrastructure and maintaining the system. The use of high-level abstractions and models also makes application debugging, scalability, and extensibility on these platforms easy, fast, and maintainable [150]. In addition, when needed, custom code can be specified to further customize the application to the preferences of the individual citizen developer.

This chapter provides a technical survey to distill the relevant functionalities provided by different LCDPs and accurately organize them. In particular, eight major LCDPs have been analyzed to provide potential decision-makers and adopters with objective elements that can be considered to guide the selection and consideration of these platforms. The contribution of the chapter is summarized as follows:

- Identification and organization of relevant features characterizing different low-code development platforms;

- Comparison of relevant low-code development platforms based on the identified features;

- Presentation of a short experience report on adopting low-code development platforms for developing a simple benchmark application.

To the best of our knowledge, this was the first work to analyze different low-code platforms and discuss them according to a set of elicited and organized features.

The remaining sections of this chapter are organized as follows: Section 2.1 presents the background of the work by showing the main architectural aspects of low-code development platforms. Section 2.2 introduces the eight LCDPs considered in this work. Section 2.3 presents the taxonomy, which has been conceived for comparing LCDPs as discussed in Section 2.4. Section 2.5 presents a short experience report on adopting LCDPs to develop a simple benchmark application. Section 2.6 presents the related works. Section 2.7 concludes the chapter and discusses some perspective work.

2.1 Background

In the early days of computing, software development was more concerned with writing functional code [111]. This was simple in concept but quickly became impractical as the size and complexity of programs increased. In response, software engineers in the 1960s began developing ways to abstract away from the underlying hardware and make higher-level languages easier to work with [99]. These efforts paved the way for automating many tasks that had previously been done by hand and significantly increased productivity. In the 1980s, a number of new tools and methods were proposed under the term computer-aided software engineering (CASE) to increase productivity, shorten time to market, and reduce human error

in large software projects [76, 191]. The development of CASE tools laid the foundations for new software paradigms such as visual programming and model-driven engineering (MDE) [191].

With the advent of MDE in software development, the focus in the software engineering research community shifted from the then-trending object technology to models as the central point, bringing abstraction, separation of concerns, and automation of development processes to the forefront [51, 47]. Meanwhile, in today's world, digital transformation has made software almost existential in our daily lives. We rely on complex systems we can hardly operate without using software [8, 47]. However, as the demand for more and more software increased, the complexity of software systems and the lack of enough skilled developers to meet the demand became a challenge [76, 8]. As a result, a new approach, low-code development platforms (LCDPs), began to gain a foothold. They enabled domain experts with varying levels of expertise and technical know-how, known as citizen developers, to develop fully functional, production-ready software applications [47].

LCDPs are software platforms that sit on the cloud [176]. They capitalize on recent developments in cloud computing technologies and modeling, such as Platform-as-a-service (PaaS) and proven software design patterns and architectures, to ensure effective and efficient development, deployment, and maintenance of the wanted application [206]. They operate in model-driven facilities to abstract and automate each step of an application's lifecycle [47]. With the primary goal of dealing with the shortage of highly-skilled professional software developers, LCDPs allow end-users with no particular programming background (called citizen developers in the LCDP jargon) to contribute to software development processes without hindering the productivity of professional developers. They enable citizen developers - from novices to professionals, as well as subject matter experts and seasonal developers, even business stakeholders and decision-makers - to build value-driven enterprise applications in less time, without requiring a lot of resources and effort, as is the case with traditional programming [206]. These applications take advantage of cloud infrastructures, automatic code generation, declarative languages, and high-level and graphical abstractions to develop entire functional software applications [211]. At the heart of LCDPs lies MDE principles that mainly promote automation and abstraction enabled by adopting modeling and metamodeling [206, 47].

Recent research has chronicled the expansion of these technologies with significant market growth in revenue [47]. Forrester and Gartner documented the growth of LCDPs in their reports [176, 184, 211], which forecasted significant market growth for LCDP companies in the coming years. In fact, major PaaS vendors such as Google and Microsoft have already

integrated LCDPs into their mainstream offerings (Google App Maker and Microsoft Power Platform, respectively). Grand View Research forecasts that the market for LCDPs will proliferate in the coming years, from \$3.02 billion in 2016 to \$86 billion in 2027 [174]. In 2019, these platforms accounted for a market value of \$11.45 billion and were expected to grow at a CAGR(Compound Annual Growth Rate) of 22.7% between 2020 and 2027 [173]. Gartner predicts that by 2023, 50% of medium enterprises will use LCDPs, and by 2024, LCDPs will account for 65% of application development, with 75% of large enterprises using these platforms to develop their software systems [209, 47].

A bird-eye view on low-code development platforms

From an architectural point of view, LCDPs consist of four main layers, as shown in fig. 2.1. The top layer (see `Application Layer`) consists of the graphical environment that users directly interact with to specify their applications. The toolboxes and widgets used to build the user interface of the specified application are part of this layer. It also defines authentication and authorization mechanisms applied to the specified artifacts. Through the modeling constructs made available at this layer, users can specify the behavior of the intended application. For instance, users can specify how to retrieve data from external data sources (e.g., spreadsheets, calendars, sensors, and files stored in cloud services), how to manipulate them by using platform facilities or utilizing external services, how to aggregate such data according to defined rules, and how to analyze them. To this end, the `Service Integration Layer` is explored to connect with different services using corresponding APIs and authentication mechanisms.

A dedicated data integration layer allows data manipulation from heterogeneous sources. To this end, the `Data Integration Layer` concerns data integration with different data sources. Depending on the LCDP used, the developed application can be deployed on dedicated cloud infrastructures or on-premise environments (`Deployment Layer`). Note that the containerization and orchestration of applications are handled at this layer together with other continuous integration and deployment facilities that collaborate with the `Service Integration Layer`.

```
┌─────────────────────────────────────────────┐
│              Application Layer                │
│            (e.g., toolbox, widgets)           │
└─────────────────────────────────────────────┘
┌─────────────────────────────────────────────┐
│           Service Integration Layer           │
│          (e.g., Dropbox, Git, IFTTT)          │
└─────────────────────────────────────────────┘
┌─────────────────────────────────────────────┐
│            Data Integration Layer             │
│         (e.g., spreadsheets, databases)       │
└─────────────────────────────────────────────┘
┌─────────────────────────────────────────────┐
│              Deployment Layer                 │
│         (e.g., cloud, local infrastructures)  │
└─────────────────────────────────────────────┘
```
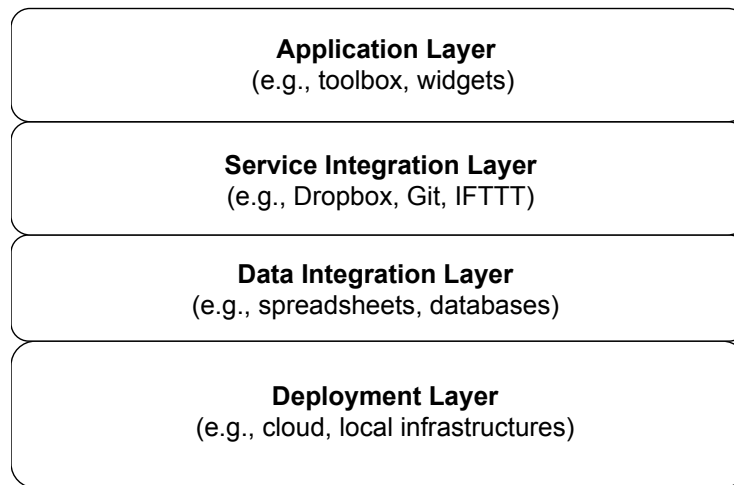
Fig. 2.1 Layered architecture of low-code development platforms.

Main components of low-code development platforms

By expanding the layered architecture shown in fig. 2.1, the peculiar components building any low-code development platform are depicted in fig. 2.2, and they can be grouped into three tiers. The first tier comprises the application modeler; the second tier is concerned with the server side and its various functionalities; and the third tier is concerned with external services integrated with the platform. The arrows in fig. 2.2 represent possible interactions occurring among entities belonging to different tiers. The lines in the middle tier represent the main components building up the platform infrastructure. As previously mentioned, modelers are provided with an *application modeler* enabling the specification of applications through provided modeling constructs and abstractions. Once the application model has been finalized, it can be sent to the platform back-end for further analysis and manipulations, including the generation of the full-fledged application, which is tested and ready to be deployed on the cloud.
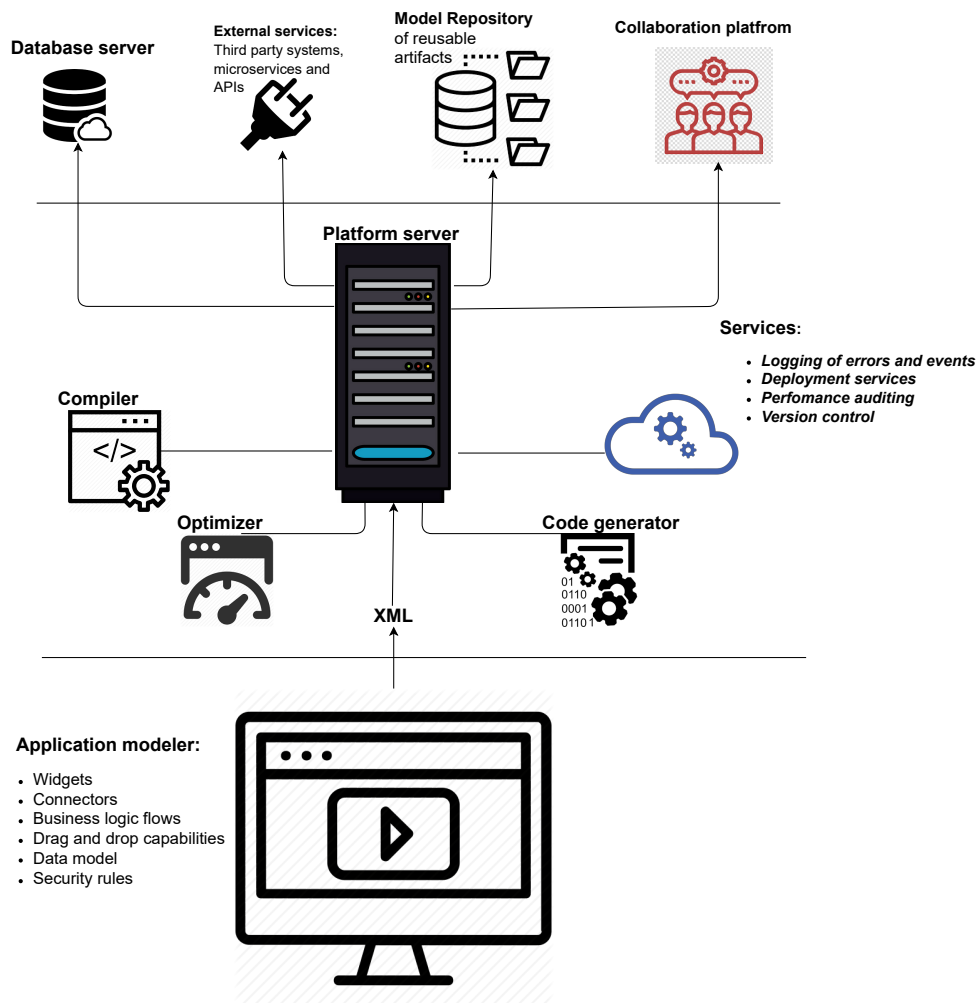
Fig. 2.2 Main components of low-code development platforms.

Figure 2.3 shows the application modeler of Mendix [151] at work. The right-hand side of the environment contains the widgets that modelers can use to define applications, as shown in the central part of the environment. The left-hand side of the figure shows an overview of the modeled system, e.g., the elements in the domain model and the navigation model linking all the specified pages. The application modeler also permits running the system locally before deploying it. To this end, as shown in fig. 2.2, the middle tier takes the application model received from the application modeler and performs model management operations, including code generations and optimizations, by also considering the involved services, including database systems, micro-services, APIs connectors, model repositories of reusable artifacts, and collaboration means [161].

Concerning *database servers*, they can be both SQL and NoSQL. The application users and developers are not concerned about the type of database employed or mechanisms ensuring
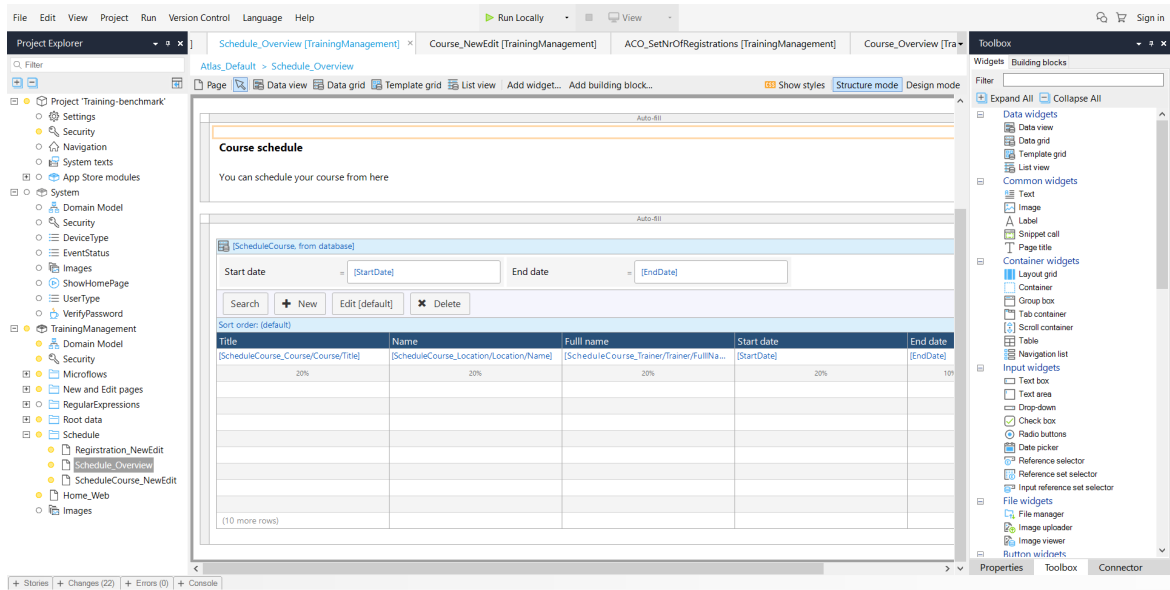
Fig. 2.3 The application modeler of Mendix at work.

data integrity or query optimizations. Generally, the developer is not concerned about the low-level architecture details of the developed application. All the needed *micro-services* are created, orchestrated, and managed in the back-end without user intervention. Although the developer is provided with the environment where she can interact with external *APIs*, specific connectors are responsible for consuming these APIs in the back-end. Thus, developers are relieved from manually managing technical aspects like authentication, load balance, business logic consistency, data integrity, and security.

Low-code development platforms can also provide developers with repositories that can store reusable model artifacts by taking care of version control tasks. To support *collaborative development* activities, LCDPs include facilities supporting development methodologies like agile, kanban, and scrum. Thus, modelers can easily visualize the application development process, define tasks, and sprints deal with changes as soon as customers require them and collaborate with other stakeholders.

Development process in LCDPs

The typical phases of developing applications utilizing LCDPs can be summarized as follows.

1. *Data modeling* - usually, this is the first step taken; users make use of a visual interface to configure the data schema of the application being developed by creating entities, establishing relationships, and defining constraints and dependencies generally through drag-and-drop facilities. A simple data model defined in Mendix is shown in fig. 2.4.
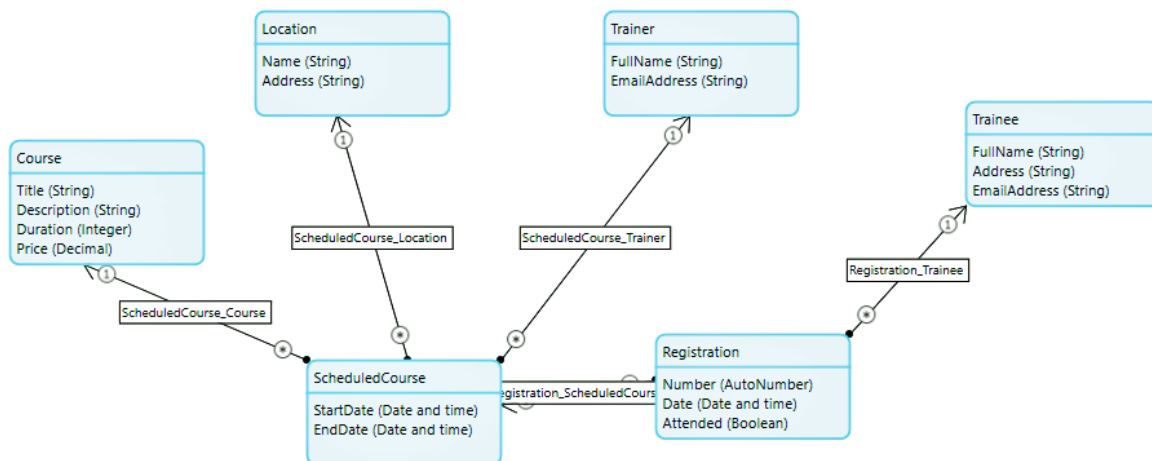
Fig. 2.4 A simple data model defined in Mendix.

2. *User interface definition* - secondly, the user configures forms and pages (c.f. fig. 2.3) used to define the application views and later define and manage user roles and security mechanisms across at least entities, components, forms, and pages. Here, drag-and-drop capabilities play a significant role in speeding up development and rendering the different views quickly.

3. *Specification of business logic rules and workflows* - Third, the user might need to manage workflows amongst various forms or pages requiring different operations on the interface components. Such operations can be implemented in terms of visual-based workflows, and to this end, BPMN-like notations can be employed as shown in fig. 2.5.

4. *Integration of external services via third-party APIs* - Fourth, LCDPs can provide means to consume external services via integrating different APIs. Investigating the documentation is necessary to understand the form and structure of the data that the adopted platform can consume.

5. *Application deployment* - In most platforms, it is possible to quickly preview the developed application and deploy it with a few clicks.

2.2 An overview of representative low-code development platforms

This section provides an overview of eight low-code development platforms (LCDPs) that are considered leaders in their respective markets in Gartner [211] and Forrester [184] reports. We assume that these eight LCDPs are representative of our analysis, which includes various features listed in Tables 2.1 and 2.2.
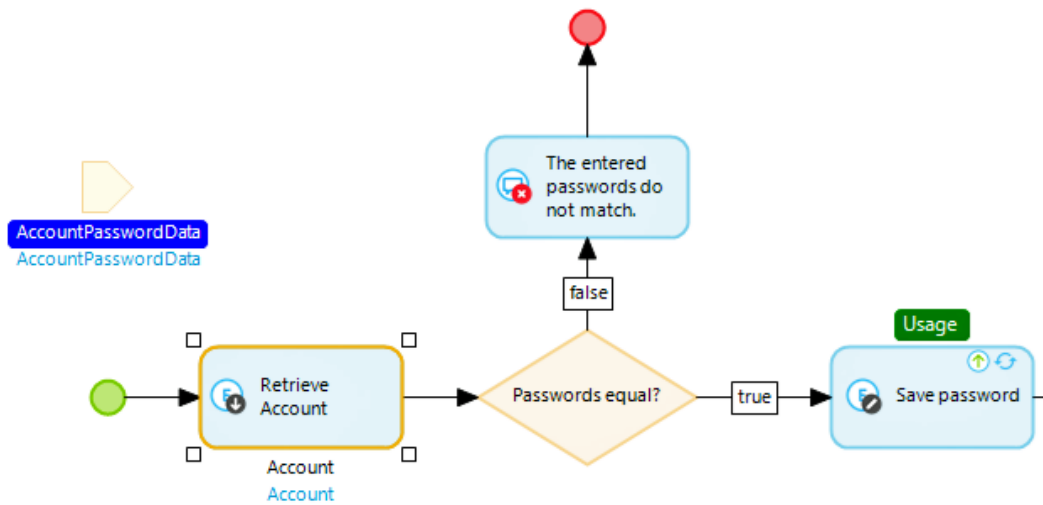
Fig. 2.5 A simple logic defined in Mendix.

OutSystems [161] enables enterprises to develop and deploy applications fast and efficiently while keeping up with the pace of innovation. It ensures developed apps are secure, ready, and scalable. OutSystems is a pioneer and industry leader in low-code application development and is among few vendors regularly ranked as a leader in this domain.

OutSystems supports desktop and mobile applications that can run in the cloud as well as on-premises infrastructures. It offers built-in capabilities that allow an application to be published from a URL with a single click. In addition, OutSystems has an Intermediate Studio for database connectivity via .NET or Java and a Service Studio to specify the behavior of the application being developed. Lastly, it supports a wide range of applications, including billing systems, CRMs, ERPs, extensions to existing ERP solutions, operational dashboards, and business intelligence.

Mendix [151] provides an easy-to-use platform for developing applications with minimum coding. This approach accelerates software development, streamlines deployment, and speeds up application delivery. As a result, enterprises benefit from the platform through features such as cloud deployment, intelligent automation, and data integration capabilities that further streamline business operations.

The majority of Mendix's features are accessible through drag-and-drop as users collaborate in real-time. The visual development feature is very advanced and allows the reuse of various components to speed up the development process, from creating user interfaces to building the data model. With pre-built connectors, machine learning techniques, users can develop

context-aware applications. Mendix's Solution Gallery[1] is an additional resource that permits users to start from already developed solutions.

Zoho Creator [67] is an all-in-one platform that streamlines the digitization of business processes without the overhead of traditional development. With its easy-to-use drag-and-drop and user-friendly, users create forms, pages, and dashboards with a responsive design that adapts to the resolution of any screen (mobile or desktop). With Zoho Creator, organizations can manage their data and processes, gain insights from their database, and easily integrate their current applications with other services such as Salesforce[2] connectors. Additionally, the platform's scalability and flexibility make it easy to customize applications to meet specific needs, regardless of the size of the organization. With the ability to create prototypes and mockups quickly, Zoho creator lowers barriers to entry for web development.

Microsoft PowerApps [165] is a cloud-based tool that enables the creation of custom, cross-platform mobile applications with little to no coding required. It is aimed at non-technical users to allows them to design applications tailored to their unique needs thanks to pre-built packages and components in the Microsoft suite. The platform allows users to create apps using drag-and-drop capabilities and a library of pre-built templates. It enables the reuse of artifacts, and the linking of PowerApps to a range of data sources including Excel, SharePoint, SQL Server, OneDrive, and other products from their Microsoft tool suite. PowerApps supports both model-based and canvas-based methods for application development, depending on the project's requirements. Alongside Power BI and Power Flow, PowerApps is part of the Power Platform.

Google App Maker [73] this is a low-code development platform powered by G Suite [3]. It allows organizations to create and publish custom enterprise applications without having to invest in expensive infrastructure or developers. Google App Maker provides a cloud-based development environment with advanced features such as in-built templates, drag-and-drop user interfaces, database editors, and file management facilities. The platform employs standard languages such as HTML, Javascript, and CSS to create a rich user experience. As a low-code development platform, App Maker enables users to quickly develop and deploy business applications with no or less coding. In fact, users can configure or drag-and-drop components to build their app. As part of google ecosystem, it integrates with G Suite, and users can easily connect their app to google services, and provides access to a wide range of resources including documentation, training materials, and community support forums. Both

---

[1]https://www.mendix.com/solutions/
[2]https://www.salesforce.com/it/
[3]https://gsuite.google.com/

small businesses and large enterprises can use this tool to speed up application development, lower costs, and enhance agility and flexibility.

Kissflow [140]: Kissflow is a cloud-based workflow software designed to help organizations manage and automate their business processes hence increase productivity and streamline operations. It features a visual drag-and-drop interface and a library of templates for common business processes, as well as collaboration, task management, document management, and reporting capabilities. Targeted towards small business applications, the platform provides a simple and visual way to create and manage workflows. It integrates with third-party APIs, such as Zapier[4], Dropbox[5], IFTTT[6], and Office 365[7]. Kissflow is user-friendly and strives to automates routine tasks while providing a clear visual representation of workflows.

Salesforce App Cloud [187] as a low-code development platform, Salesforce App Cloud allows companies to easily create and deploy custom applications without programming skills. Through a visual interface, users such as citizen developers can build apps by drag-and-drop predefined components. Once created, the app is ready and it can be deployed to the Salesforce App Cloud and later accessed via web browser or mobile device to manage sales data, track leads, and generate reports. This platform offers a wide range of features and integrations to meet the specific needs of any business, as well as providing secure and scalable cloud-based applications. Professional developers can leverage the platform's out-of-the-box tools and operations for automation through integration with external services. It also has an extensive AppExchange marketplace[8] where users can find pre-built applications and components, reusable objects, and elements, a drag-and-drop process builder, and built-in Kanban boards.

Appian [11] is one of the oldest low-code development platforms. This platform allows users to build personlized mobile and web apps where users can collaborate and manage their tasks. Its decision engine is useful for complex modeling logic and enables enterprises to build and deploy business applications quickly. Hence enterprises can automate complex business processes and improve organizational agility in their applications. The platform also offers comprehensive analytics and reporting capabilities to track performance and make data-driven decisions, thus saving time and resources while delivering high-quality applications.

---

[4]https://zapier.com
[5]https://www.dropbox.com/
[6]https://ifttt.com/
[7]https://products.office.com/it-it/home
[8]https://appexchange.salesforce.com/
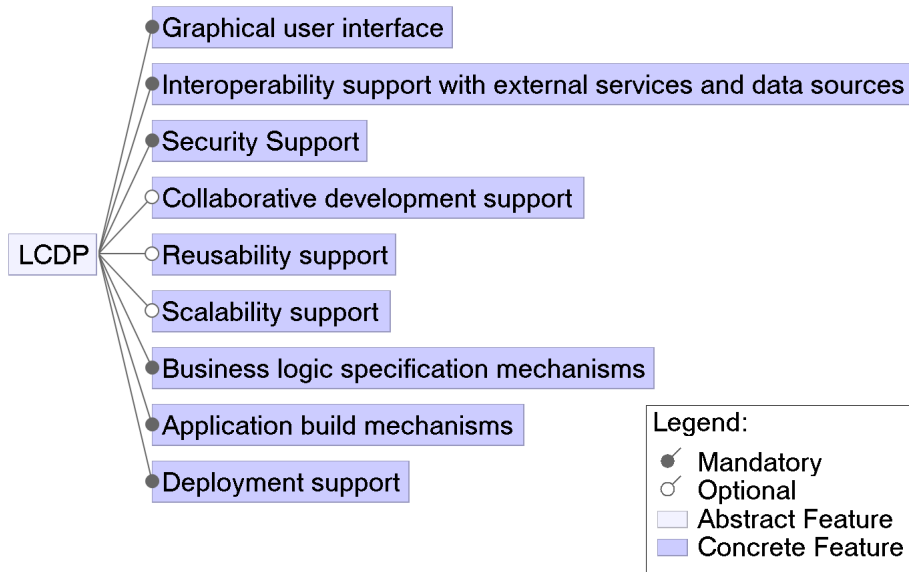
Fig. 2.6 Feature diagram representing the top-level areas of variation for LCDPs.

2.3   Taxonomy

In this section, we introduce preparatory terms that can facilitate selecting and comparing different LCDPs. The features are derived by examining the requirements in building an application along with the capabilities that a low-code platform could provide in building an application. Specifically, by analyzing the low-code development platforms described in the previous section, we identified and modeled their differences and similarities. Our results are documented using feature diagrams [70], which are a common notation in domain analysis [71]. fig. 2.6 shows the top-level feature diagram, where each sub-node represents a main variation point. Tables 2.1, 2.2 provide details of the taxonomy described below.

- *Graphical user interface:* This group of features represents the functionalities available in the front-end of the platform under consideration to support customer interactions. Examples of functionality included in this group are drag-and-drop tools, forms, and advanced reporting capabilities.

- *Interoperability support with external services and data sources*: This group of features refers to the ability to interact with external services such as Dropbox, Zapier, Sharepoint, and Office 365. The ability to connect to various data sources to create forms and reports is also included in this group

- *Security support*: The features in this group relate to the security aspects of the applications developed with the platform being used. Features in this group include

authentication mechanisms, adopted security protocols, and user access control infrastructures.

- *Collaborative development support*: This group refers to the models of collaboration (e.g., online and offline) that are established to support the collaborative specification of applications between developers in different locations.

- *Reusability support*: it refers to the mechanisms used by each platform to enable the reuse of previously developed artifacts. Examples of reusability mechanisms include predefined templates, pre-built dashboards, and built-in forms or reports.

- *Scalability support*: Such a group of features enables developers to scale applications according to various dimensions, such as the number of manageable active users, traffic, and storage capacity that a particular application can handle.

- *Business logic specification mechanisms*: refers to the tools provided for specifying the business logic of the modeled application. These tools include a business rules engine, a graphical workflow editor, and API support for communication with other applications. The business logic can be defined through one or multiple API calls.

- *Application building mechanisms*: refers to the way in which the specified application is created, i.e., by using code generation techniques or by models at runtime. In the first case, the source code of the modeled application is generated from the specified models and then implemented. In the second case, the specified models are interpreted at runtime and used to control and automate deployment.

- *Deployment support*: refers to deployment mechanisms supported. For example, the generated application can be deployed to various app stores and local or cloud infrastructures.

In addition to the information shown in fig. 2.6, top-level features are shown. LCDPs can also be classified in terms of the type of applications they support. In particular, each LCDP can support the development of one or more types of applications, including web portals, business process automation systems, and quality management applications.

	OutSystems [161]	Mendix [151]	Zoho Creator [67]	MS PowerApp [165]	Google App Maker [73]	Kissflow [140]	Salesforce App Cloud [187]	Appian [11]	
Graphical user interface									
Drag-and-drop facilities	✓	✓	✓	–	✓	✓	✓	✓	This feature improves the user experience by allowing drag and drop of elements involved in the creation of an app, including actions, responses, connections, etc.
Point and click approach	–	–	–	✓	–	–	–	–	This is similar to the drag and drop feature, except that you must point to the object and edit some fields such as names
Pre-built forms/reports	✓	✓	✓	✓	✓	✓	✓	✓	These are ready-made and usually reusable editable forms or reports that a user can use when developing an application.
Pre-built dashboards	✓	–	✓	✓	–	✓	✓	–	These are the default dashboards used by the user while developing an application.
Forms	–	–	✓	✓	–	–	–	–	This feature allows you to integrate forms in their applications. They are usually customizable and include custom forms, surveys, checklists, etc.
Progress tracking	✓	✓	✓	✓	✓	✓	✓	✓	This feature helps developers collaboratively track their progress while developing an application.
Advanced reporting	–	–	–	–	–	✓	–	–	This feature provides graphical report capabilities where users can use graphs, tables, charts, etc.
Built-in workflows	–	–	✓	–	–	✓	✓	–	some business logic workflows are identified and packaged in the platform for reuse. Hence users leverage these workflows when building applications.
Configurable workflows	–	–	✓	–	–	✓	✓		In addition to built-in workflows, users can have customizable workflows.
Interoperability support									
Interoperability with external service	✓	✓	✓	✓	–	✓	✓	✓	This feature allows users to integrate parts of the artifacts developed in one application in other LCDPs.
Connection with data sources	✓	✓	✓	✓	✓	✓	✓	✓	This feature connects the application to data sources such as Microsoft Excel, Access, and other relational databases such as Microsoft SQL, Azure, and other non-relational databases such as MongoDB.
Security Support									
Application security	✓	✓	✓	✓	✓	✓	✓	✓	This feature enables the security mechanism of an application, such as access control, authentication, authorization, and so on.
Platform security	✓	✓	✓	✓	✓	✓	✓	✓	these platforms usually provide security capabilities at the platform level where user can access their resources or configure roles for other users.
Collaborative development support									
Off-line collaboration	✓	✓	✓	✓	✓	✓	✓	✓	some platforms enable distributed collaboration for users to synchronize their work using version control mechanisms.
On-line collaboration	✓	✓	–	–	✓	✓	✓	✓	Different developers collaborate concurrently on a application. Conflicts are managed at run-time.
Reusability development support									
Built-in workflows	–	–	✓	–	–	✓	✓	–	This feature allows reusing features such as workflows when creating an application.
Pre-built forms/reports	✓	✓	✓	✓	✓	✓	✓	✓	These are standard forms and the most common reusable, editable forms or reports that a user can integrate into an application.
Pre-built dashboards	✓	–	✓	✓	–	✓	✓	–	These are customizable dashboards that users can directly reuse rather than starting from scratch when developing an application.

Table 2.1 Taxonomy / Comparison for analyzed Low-Code Development Platforms.

	OutSystems [161]	Mendix [151]	Zoho Creator [67]	MS PowerApp [165]	Google App Maker [73]	Kissflow [140]	Salesforce App Cloud [187]	Appian [11]	
Scalability									
Number of users	✓	✓	✓	✓	✓	✓	✓	✓	The platform is configured to autoscale based on the workload imposed by the number currently using the platform.
Data traffic	✓	✓	✓	✓	✓	✓	✓	–	The platform can scale up and down based on the volume of data being processed at a given time.
Data storage	✓	✓	✓	✓	✓	✓	✓	–	The ability for the storage to autoscale based on the data volume that is needed to be persisted or processed.
Business logic specification mechanisms									
Business logic engine	✓	✓	✓	✓	✓	✓	✓	✓	This engine empowers the platform with the ability to execute business logic such as workflows.
Graphical workflow editor	✓	✓	–	–	–	✓	✓	–	This feature allows users to define one or more business logic in a graphical manner.
AI-enabled business logic	✓	–	–	–	–	✓	✓	✓	This is an important feature that uses artificial intelligence to learn the behavior of an attribute or object and replicate that behavior using learning mechanisms.
Application building mechanisms									
Code generation	✓	–	–	–	–	–	–	–	This feature generates the source code of the modeled application, and the user takes responsibility for deploying the application.
Models at run-time	–	✓	✓	✓	✓	✓	✓	✓	The model of the specified application is interpreted at run-time, automating deployment phase.
Deployment support									
Deployment on cloud	✓	✓	✓	✓	✓	✓	✓	✓	With this feature, users can deploy an application to a cloud infrastructure.
Local deployment	✓	✓	–	–	–	–	✓	✓	This feature allows an application to be deployed locally, mainly for testing purposes.
Type of supported applications									
Event monitoring	✓	✓	✓	✓	✓	✓	✓	✓	This type of application is about collecting data, analyzing events that the data may cause, and reporting events that occur in the data to the user.
Process automation	✓	–	✓	✓	✓	✓	–	✓	This type of application focuses on automating complex processes, such as workflows, that can run with minimal human intervention.
Approval process control	–	–	–	–	✓	–	–	–	This type of application consists of processes for creating and managing work authorizations. For example, payment tasks can be managed through the approval of authorized personnel.
Escalation management	–	–	–	–	–	✓	–	–	these types of applications are common in customer service. They focus on automating and streamlining the process of escalating issues and resolving problems efficiently and efficiently.
Inventory management	✓	✓	✓	✓	✓	✓	✓	✓	This type of application is used to monitor the flow of goods, from procurement to delivery, and allow businesses to manage stock levels. In addition, they can help to automate purchase and sales processes.
Quality management	–	✓	✓	✓	✓	✓	✓	✓	This type of application is used to manage the quality of software projects, e.g., by focusing on the planning, assurance, control, and improvements of quality factors.
Workflow management	✓	✓	✓	✓	✓	✓	✓	✓	These types of applications manage sequences of tasks monitored during their execution.

Table 2.2 Taxonomy / Comparison for analyzed Low-Code Development Platforms(2).

2.4   Comparison of relevant LCDPs

In this section, we use the taxonomy presented earlier to compare the eight low-code development platforms overviewed in Section 2.2. Table 2.1 and 2.2 show the result of the comparison performed by listing the corresponding supported features for each platform. The data presented in these tables are mainly taken from the official resources of each platform OutSystems [161], Mendix [151], Zoho Creator [67], Microsoft PowerApps [165], Google App Maker [73], Kissflow [140], Salesforce App Cloud [187], Appian [11], and taking into account the experience we had while developing a benchmark application described in the next section.

2.4.1   Features and capabilities

The main features and capabilities of the analyzed LCDPs can be summarized as follows: *OutSystems* provides developers with a fast mechanism allowing its users to publish their applications with a single click. It provides the ability to connect to various services while developing responsive mobile and web applications. This platform highlights security mechanisms and provides real-time dashboard capabilities. *Mendix* supports collaborative project management and end-to-end development and provides pre-built templates with an app stores. Mendix's Interactive application analytics and rapid development make it suitable for companies looking for efficient collaboration among developers and stakeholders. *Zoho Creator* is known for a user-friendly form builder and a user-friendly, mobile-friendly interface. The platform supports Salesforce and CRM application integration. For organization seeking simple and adaptable application development, its pre-built templates and customized procedures can prove handy in their workflow.

Microsoft PowerApp supports integration with Office 365, pre-built templates, easy conversion of apps for cell phones and tablets. It can connect with third-party apps for basic app development. *Google App Maker* has a drag-and-drop feature similar to most of the LCDPs analyzed, app preview, reusable templates, deployment settings, means to set access roles, built-in tutorials and Google Analytic integration. *Kissflow* supports progress tracking, custom and pre-built reports, collaboration features and the ability to use third-party services such as Google Doc and Dropbox documents. It also supports Zapier to integrate different systems. *Salesforce App Cloud* has an extensive app marketplace for pre-built apps and components, reusable objects and elements, built-in Kanban boards and a drag-and-drop process builder. *Appian* supports native mobile apps, drag-and-drop tools, collaborative task management and a decision engine with AI-powered complex logic.

2.4.2 Additional aspects for comparing LCDPs

The taxonomy discussed in the previous section plays an important role when users need to compare LCDP candidates and select one from possible alternatives. In addition to the characteristics presented previously, we have identified additional aspects that are orthogonal to the taxonomy presented, and that can be taken into account when decision-makers need to decide whether and which low-code development platform to use.

Anticipated solution and outcome: Two main types of applications can be developed using LCDPs, namely B2B (Business to Business) and B2C (Business to Customer solution). B2B solutions provide users with business process management (BPM) capabilities, such as creating, optimizing, and automating business process activities. Examples of B2B solutions include hotel management, inventory management, and human resources management. Multiple applications can be combined in a B2B solution. B2C solutions provide simpler answers for end users. These solutions are used to develop individual applications such as websites and customer relationship management applications. The interactivity aspect is much more important in B2C than in B2B.

Organization size: Another dimension to consider when selecting LCDPs is the size of the company/organization that will adopt the selected LCDP. Organizations fall under three possible categories: *small* (with less than 50 employees), *medium* (when the number of employees is between 50 and 1000), *large* (when the number of employees is higher than 1000). Thus, the decision-making must consider the size of the company in order to find the optimal solution for its needs. Any organization looking to scale its Business at an optimal cost must select an LCDP based on the strength of the Business. LCDPs such as Salesforce App Cloud, Mendix, and OutSystems support large enterprises and are used to develop large and scalable applications. On the other hand, Google App Cloud, Appian, and Zoho Creator are mainly designed to support small and medium-sized businesses and are relatively inexpensive.

Platform knowledge acquisition cost the time spent on the development, testing, and deployment of an application may vary from one low-code platform to another. To be proficient in such processes, users must spend time learning all the related aspects of that platform. Also, decision-makers have to consider potential training costs that have to be faced for learning the concepts and processes of that particular low-code platform.

Price: It is one of the most critical criteria, especially for small or medium-scale companies. The platform's price can be estimated as the price of using the platform for one developer per month. Moreover, the dimensions that contribute to the definition of the price include

i) the number of applications that need to be deployed, and *ii)* where data are going to be stored, i.e., in on-premise databases, in cloud environments, or hybrid configurations.

Increase in productivity: The adoption possibilities of low-code development platforms have to be assessed by considering the potential number of developed applications with respect to the time spent to learn the platform, the price incurred in training, and to buy the licenses to use the considered platform.

2.5 Using LCDPs: an experience report

Building platforms that enable citizen developers to build full-fledged applications faster and more efficiently comes at a cost. Key architectural decisions are made to ensure minimal programming effort, speed, flexibility, reduced upfront investment, and faster deployment of the application. However, the decisions that are typically made when developing these platforms can lead to issues that surface later. To gain insight into LCDPs, we developed a same benchmark application using different platforms, namely Google App Maker, Mendix, Microsoft PowerApps, and OutSystems. The Benchmark application is a course management system that allows faculty and students to manage their courses, schedules, registrations, and attendance. Despite the simplicity of the application, it has common user requirements that are common in the development of typical software applications such as managing, retrieving, and visualizing data. We also had the opportunity to integrate external services through third-party APIs. We were able to explore how reusable developed code and artifacts can be integrated into other low-code platforms to pave the way for discovery and reuse of already proven artifacts across different platforms.

The first activity to develop the benchmark application was identifying the relevant requirements. We identified the relevant use cases and, thus, the functional requirements for the system. Experience has shown that software applications can be developed in LCDPs using two main approaches:

- *UI to Data* - The developer begins the creation of the application by creating a user interface and then linking it to the required data sources. Forms and pages are defined first, followed by the specification of business logic rules and workflows, which then lead to the integration of external services before the application is deployed. LCDPs such as Mendix, Zoho Creator, Microsoft PowerApps, and Kissflow can take this approach.

- *Data to UI* - This is a data-driven approach that starts with data modeling and then builds the application user interface, followed by the specification of business logic rules and workflows. Then, if necessary, external services are integrated before the

application is deployed. LCDPs such as OutSystems, Mendix, Zoho Creator, Microsoft PowerApps, Salesforce App Cloud, and Appian can take this approach.

The specification of rules for business logic, workflows, and integration of external services can be interchanged in both of the above approaches, depending on the developer's style.

Challenges in LCDPs

Traditional programming paradigms have long been plagued with a number of shortcomings vis-a-vis the current digital transformation. Traditional programming requires specialized skillsets from a current small talent pool. This paradigm is time-consuming and costly and is easily flexible to the dynamic needs of customers [8]. For this reason, the resulting applications, unless developed by highly skilled developers, are of inferior quality and difficult to extend and scale [96]. As a matter of fact, apart from the shortage of highly skilled developers compared to the demand available on the market, recent studies show that a typical software project ranges from $434,000 to $2,322,000 from small enterprises to larger companies, respectively. With 52% costing almost 90% of their initial estimates, almost one-third canceled and only less than 17% completed on-time within the allocated budget [90].

On the other hand, LCDPs offer several benefits to an organization that adopts them [143]. It has been noted that LCDPs increase digital innovation and transformation because integrating citizen development in the application development lifecycle positively impacts productivity. Citizen developers participate in organizations' internal processes and widen the horizon of innovation and transformation [191]. They also have the potential to protect businesses from technology churn while enhancing rapid business response to pressing matters. Empowering citizen developers means that IT professionals earlier involved in technical aspects of the business are reduced to the staff that is needed for critical existential of the organizations [96]. Compared to traditional programming, LCDPs enable quick product delivery, cost reduction, complexity reduction, easy maintenance, business profiles' participation in product development, and minimization of varying customer requirements. In addition, apps developed on these platforms are cross-platform and are cost-effective [203].

By developing the considered benchmark application, we identified various challenges that users and developers may encounter during development. These challenges include interoperability problems between different LCDPs, limitations in terms of extensibility, steep learning curves, and scalability issues [160] [194] [211]. Below we discuss these challenges that transcend most of the low-code development platforms. We will not discuss potential challenges that occur concerning code optimization or security and compliance risks

because we did not profoundly assess these features due to the lack or limited visibility of the considered LCDPs. However, we acknowledge that such aspects should be investigated in the future to give a more broad perspective on potential challenges that might affect LCDPs.

Low-code platforms' interoperability: This characteristic ensures interaction and exchange of information and artifacts among different LCDPs, e.g., to share architectural design, implementation, or developed services. This feature is essential to mitigate issues related to vendor lock-ins. Unfortunately, most LCDPs are proprietary and closed platforms. Hence, a lack of standards in this domain hampers the development and collaboration among engineers and developers. Consequently, modelers and developers do not collaborate and learn from one another, impacting the reusability across these platforms.

Extensibility: Refers to its ability to accommodate new functionalities that are not already offered by the platform. Unfortunately, this is often a difficult task because these platforms tend to be proprietary. Due to lack of standards, some of them require extensive coding to add new capabilities, which citizen developers do not necessarily possess. In addition, when extending is required, developers must also adhere to the architectural and design limitations of the platform, making the process inflexible.

Learning curve: The learning curve associated with most low-code development platforms can be steep due to their less intuitive graphical interfaces. Some platforms have limited drag-and-drop capabilities and a lack of teaching materials, such as sample applications and online tutorials. This can impact the platform's adoption and limit it to those who already have knowledge in software development rather than the main target of these platforms: citizen developers.

Pricing [143]: Despite LCDPs recent success, a substantial investment is necessary to fully take advantage of these platforms' offerings. Although the perks may seem worth the cost, this financial requirement limits the number of potential users. As a result, enterprises are the main beneficiaries of these platforms, while individual developers are limited in what they can do with free or low-cost plans.

Limited customization [203, 143]: As organizations move to LCDPs to become more agile and responsive to business needs, it is important that they consider the level of customizability, flexibility, and control of the chosen platform. It has become apparent that a number of platforms offer these features only to a limited extent, even though they are essential for enterprises.

Scalability: LCDPs have gained recognition for their efficiency in building simple, single-function apps. However, they fall short when it comes to large-scale projects. For instance, due to internal design limitations, creating an app with multiple screens and data sources might not be feasible for many LCDPs. To truly excel, LCDPs must be designed to handle huge load of computations and manage big data produced in high volumes, variety, and speed [179].

Vendor lock-in: A potential downside of these platforms is that the source code of the applications is often inaccessible to users due to reserved commercial rights. This can be problematic when users want to migrate their applications elsewhere or make desired changes without the intervention of the platform engineers' support team. In most cases, this leads to a vendor lock-in scenario, where an organization's entire application portfolio is dependent on a single vendor. This can lead to higher costs and less flexibility.

Debugging and maintenance [223, 96]: Because LCDPs are typically based on visual drag-and-drop interfaces, it can be difficult to troubleshoot when something goes wrong. In addition, LCDPs often rely on proprietary software and closed architectures, making it difficult to find trained personnel who can support and maintain the applications. Therefore, organizations should carefully weigh the benefits and risks of selected LCDPs prior to their adoption.

Lack of standardization [182]: The lack of standardization in LCDPs leads to fragmentation as each vendor develops her domain-specific languages and tools without reusing existing resources within the LCDP ecosystem. As a result, valuable resources are wasted and scattered, making it difficult for citizen developers to reuse components developed in other LCDPs or even in different projects on the same platform. This ultimately leads to less efficient use of resources and slower development overall. To solve this problem, vendors must work together to create standards and encourage the reuse of core components. In this way, they can help LCDPs remain an efficient and effective platform for software development.

Overall, LCDPs are suitable for organizations that have limited IT resources and budgets. They provide a fast and efficient way to create fully-featured products, such as CRM applications. However, the end result may be impacted by the functionalities offered by the platform's modules. Users might need to accommodate their initial requirements depending on the options offered by the employed platform. Furthermore, integration with third-party plugins may be limited depending on the selected platform.

2.6   Related works

Guerrero et al. [119] highlight two phases of software development using MDE approaches. The first stage involves the use of tools like Domain Specific Languages, model transformation processes, and code generators. In contrast, the second stage focuses on non-programmers utilizing the resulting artifacts and benefiting from the higher level of abstraction in creating the desired application. The paper also explores the challenges in using a model-driven approach and identifies ways to overcome these obstacles.

The study by Braams [40] explores the best practices for testing Low-Code Model Driven Development (LCMDD) Platforms. It employs the Behaviour Driven Development (BDD) methodology and provides a comprehensive overview of it. Four leading LCDPs were tested using a software quality framework, aiming to provide a holistic approach to software quality based on BDD principles.

Acerbis et al. [3] propose a tool named WebRatio Mobile Platform for building mobile applications using a model-driven development approach. The tool classifies the specification of the domain and that of the interactive model for applications used in mobile by using OMG standard language known as Interaction Flow Modeling Language (IFML). Also, the tool checks the model and generates code that produces ready-to-publish inter-platform mobile applications.

Waszkowski [218] express the utility of the LCDP Aurea BPM for automating business processes in manufacturing. This application shows the critical usage of low-code solutions in different industries.

Chang et al. [57] discuss the App development tool named Smart Maker Authoring Tool, which is used to develop user-friendly apps for non-developers. The tools exhibit greater software development productivity which is very useful for non-developer to customize their needs for their business.

Bock et al. [37, 36] provided a balanced analysis of the current trend toward LCDP platforms. They outlined the defining features that distinguish LCDP platforms and evaluated the technical advances of these platforms. They are built by combining and integrating conventional system design components. They also highlighted potential research opportunities in this area. Although they have not found revolutionary innovations in LCDPs, they underscored potential productivity while using these platforms as long as the project criteria can be met within their boundaries. They have also identified opportunities for areas such as conceptual modeling from the LCDP wave.

Alamin et al. [6] analyzed insights from Stack Overflow topics. Although LCDPs scored high on data modeling, developers see external API integration and dynamic event handling as their biggest challenges. They observed a deficiency of proper tutorial-based documentation, which hinders the customization of applications developed on these platforms.

Luo et al. [143] cover the benefits and limitations of LCDPs from a practitioner's standpoint. According to Gartner and Forrester, the rapid expansion of these platforms points to a promising future. The authors examined Stack Overflow and Reddit to gain insight into LCDPs' features, benefits (e.g., user-friendly graphical interface and out-of-the-box components), limitations, problems, and application domains. The results show that users prefer LCDPs when it comes to dealing with automated processes and workflows. This paper suggests that researchers should clearly define terms when referring to LCDPs and that developers should evaluate whether LCDP is suitable for their projects.

The study presented in [8] investigates the elements that draw developers and programmers to LCDPs. It highlights some of the barriers to its adoption by developers and programmers. The authors administered an online survey to professional developers in IT departments of different companies and students in computing-related departments of Saudi institutions. The results outlined the reasons for the popularity of LCDPs, including reduced app development time, ease of use, automatic code generation, step-by-step instructions, and lower error rate. On the other hand, the limitations were the low scalability of these platforms and the lack of knowledge on how to work with them.

Sanchis et al. [191] delve into the current state of software development automation tools and potential barriers. The aim was to pinpoint areas in LCDP platforms that deserve further examination. The researchers conducted a thorough evaluation of existing information on LCDPs to provide a current understanding of the landscape, including any potential hurdles and promising solutions. The study compared the performance of LCDPs against the vf-OS EU project. The results showed a shortage of open-standards based platforms, hindering the possibility of automatic application building and deployment.

To demonstrate the advantages and drawbacks of LCDPs, in [203] the authors have developed a user access auditing and control automation by making use of the Oracle APEX LCDP. In addition, they have drawn our attention to the fact that LCDPs do not replace conventional development skills but rather complement them, especially when it comes to maintenance or troubleshooting.

Gomes et al. [96] investigate the gap between what developers envision and the actual needs of the domain when it comes to adopting LCDPs. The paper aims to provide a clearer understanding of these platforms by exploring their fundamental principles, current state

of the art, key components, and key concepts. Through a descriptive analysis, the study evaluates the limitations and benefits of LCDPs in-depth. Lastly, it showcases various use cases of LCDPs that cab be utilized in different fields.

Bucaioni et al. [47] address the increasing popularity of LCDPs and their connection to model-driven engineering that has been the subject of lively discussion in recent times. This multi-vocal systematic study aims to explore the similarities and differences between these two approaches. The findings suggest that while there are indeed parallels between model-driven engineering and low-code, they are still distinct entities. LCDP is commonly seen as a subset of MDE due to their shared focus on maximizing efficiency and simplifying processes. This can bring mutual benefits, as MDE can benefit from the insights gleaned from LCDP, and vice versa. Technology-wise, most LCDPs leverage MDE at their core to facilitate rapid prototyping and ease of use – something which users have consistently praised.

Rokis et al. [182] take a deep dive into the potential opportunities and challenges associated with low-code software development (LCDP). The authors highlight advantages of this approach, such as rapid development and improved software quality. Additionally, it outlines that must be removed to optimize this strategy. Some of the challenges include empowering citizen developers, managing minimum requirements for minimal viable products, and establishing standardization frameworks. Strategies are suggested for eliminating these hindrances and helping to propel low-code processes forward. The study concludes with recommendations for mitigating these barriers and advancing low-code software development processes.

2.7 Conclusion and future work

In recent years, interest in LCDPs has grown significantly in both academia and industry. Understanding and comparing a great number of low-code development platforms [91] can be a tedious and difficult task without a proper conceptual framework to support their evaluation. In this chapter, we analyzed eight low-code development platforms that are considered leaders in their respective markets to identify their commonalities and differences. A set of distinguishing characteristics was defined and used to compare the platforms under consideration. We also wrote about our experiences to discuss some key features of each platform, limitations, and challenges we encountered while developing our benchmark application.

The future of this research may refine the proposed taxonomy by considering additional LCDPs to obtain a comprehensive and thoroughly validated set of characteristics. Such

a refinement process could also involve the various LCDP vendors to further validate the created taxonomy and share with them the challenges and lessons learned in developing the discussed benchmark application.

In addition, a stronger focus on the reusability and interoperability that low-code development platforms require can promote their adoption in both academia and industry.

Chapter 3

Cloud-based modeling in data-intensive applications

In recent years, there has been a growing interest in cloud-based modeling, which refers to using modeling tools and techniques designed to run on cloud infrastructures [45]. This trend is mainly due to the many benefits that cloud-based modeling can offer, such as the ability to design, develop, deploy, and manage models and applications with less effort [45, 153]. Additionally, the proliferation of low-code development platforms (LCDPs) has also played a role in the popularity of cloud-based modeling. LCDPs, as software platforms, allow users to create applications with less or no need to write code. LCDPs run on cloud infrastructures, further boosting the popularity of cloud-based modeling [189]. As industries and enterprises move their modeling infrastructures to the cloud, this transition can be complex and expensive, particularly in industrial contexts where infrastructure is often very intricate. Migration costs can include everything from modifying existing models to account for the change in infrastructure to retraining employees on how to use the new system. In addition, business operations can be significant disruptions during the transition period [192].

The future of modeling will inevitably be cloud-based [49]. Several initiatives, including Visual Studio Code[1], Eclipse Che[2], Theia[3], and others, have shown the potential of moving modeling environments from on-premises and monolithic installations to cloud-based platforms, remove accidental complexity and expand the variety of available functionalities [49]. In particular, the cloud offers the ability to quickly provision and scale resources as needed, essential for modeling applications requiring significant computational resources [45]. Addi-

---

[1]https://code.visualstudio.com
[2]https://www.eclipse.org/che/
[3]https://theia-ide.org

tionally, the cloud provides access to various services integrated into modeling applications, such as storage, databases, and machine learning services [153].

As data-intensive applications such as the Internet of Things (IoT) continue to grow in popularity, there is an increasing need for tools that can help in modeling and simulating the behavior and the implementation of such systems [114]. Cloud-based modeling tools offer several advantages, including scalability and flexibility. However, they also come with several challenges, including data privacy and security concerns [115]. These systems require modeling and development infrastructures considering the heterogeneous aspects of the system's data, communication, and implementation layers. This chapter reviews recent progress in developing cloud-based modeling approaches for data-intensive applications in IoT systems. In particular, we conducted a thorough investigation to see where the IoT community stands concerning the current trend of moving traditional modeling infrastructures to the cloud. We examined 625 articles and identified 22 cloud-based IoT system development tools and platforms. We then take a closer look at some options and discuss the research and development opportunities that arise from adopting cloud-based modeling approaches in the IoT domain.

The rest of this chapter is structured as follows: We delve into cloud-based modeling techniques in Section 3.1, discussing the approaches and rationale for the need to model IoT systems through cloud-based environments. Section 3.2 outlines the research methodology we employed for the survey. The results of our analysis to answer three research questions are presented in Sections 3.3, 3.4, and 3.5. We review related work in Section 3.6 and wrap up the chapter with our conclusions in Section 3.7.

3.1 Background

Significant advances in computing power, data storage, and processing are revolutionizing the development and research of complex systems in various fields, including the Internet of Things (IoT) [147]. IoT systems enable the integration of intelligent features into daily human activities by automating services. In particular, such systems enable the automation of low-level services that were previously error-prone when performed by humans. They also increase the efficiency of current technological solutions and connect various devices that make our environment intelligent. According to recent reports, more than 100 billion devices will be connected by 2025, reaching a global market capitalization of $11 trillion [83]. However, to realize the full potential of these systems, citizen developers must also participate in developing customized IoT applications [189].

Meanwhile, the development and consumption of IoT systems are becoming increasingly complex, and end-user engagement is more challenging due to the heterogeneity of hardware and expertise required [189]. There are several reasons for this complexity. IoT applications are complex systems that use heterogeneous devices and data sources. In addition, IoT systems require a tremendous amount of effort and investment in both implementation and maintenance. Moreover, systems are implemented using code-centric approaches, making it difficult to encourage participation from IoT domain experts and other stakeholders with less IoT programming knowledge [189].

Due to the ever-changing requirements and shortage of technical experts who can develop these systems robustly and securely [42], it is necessary to pave the way for domain experts and other stakeholders to integrate IoT capabilities into their daily tasks [189]. Several approaches have been discussed, although practical solutions have yet to find a way to make the use and development of IoT applications more accessible. One practical solution is the systematic use of models as the primary units of abstraction and automation in developing these complex systems using model-driven engineering (MDE) [115]. Models in the context of MDE are not sketches or drawings used only for design, but they prevail as machine-readable and processable abstractions throughout the development cycle of such systems [26]. MDE favors collaboration between engineers and stakeholders, as both work together towards the completion of the conceived products while promoting the integration of the different engineering processes [42].

However, MDE faces challenges that have shifted the focus of developing such complex and heterogeneous systems from on-premises environments to the cloud [75]. Modeling-as-a-Service is gaining momentum as the MDE research community moves modeling tools and services to the cloud. This migration is fueled by several benefits of cloud computing, such as the ease of discovery and reuse of services and artifacts [115]. This new paradigm enables efficient self-healing mechanisms to detect, diagnose, combat threats, and foster collaboration among stakeholders and engineers [58]. In addition, migrating model artifacts and services to the cloud can provide easy access to end users and support sustainable management and disaster recovery of model artifacts and tools [104].

3.2   Study design

This section analyzes how the IoT domain is coping with the migration of existing modeling and development infrastructures to the cloud. For this purpose, we followed the process shown in fig. 3.1 according to the review methodology presented in [31].

Specifically, the search and selection process was conducted in four main phases. In the first phase, we formally and explicitly presented the problem to get a head start on the search. Second, we defined a search term and selected known academic search databases. Third, we conducted a search to collect papers that answered the clearly defined research questions. Fourth, we narrowed down the potential papers and ranked them according to their similarity and variability. Finally, we analyzed the collected papers and provided recommendations for the identified difficulties.

Phase 1: Problem formalization - At this stage, the main goal was to formalize the problem we were trying to solve by looking at the current trend toward model-driven engineering. One of the sources of inspiration for this study was the work in [49], which deals with the topic *"What is the future of modeling?"*. This is how we came up with the formulation of the following research questions:

- *RQ1: What are the current cloud-based modeling approaches for IoT?*

- *RQ2: What challenges do researchers face when developing cloud-based IoT modeling and development infrastructures?*

- *RQ3: What are the main potential opportunities laying ahead for future researchers and developers in the IoT domain?*
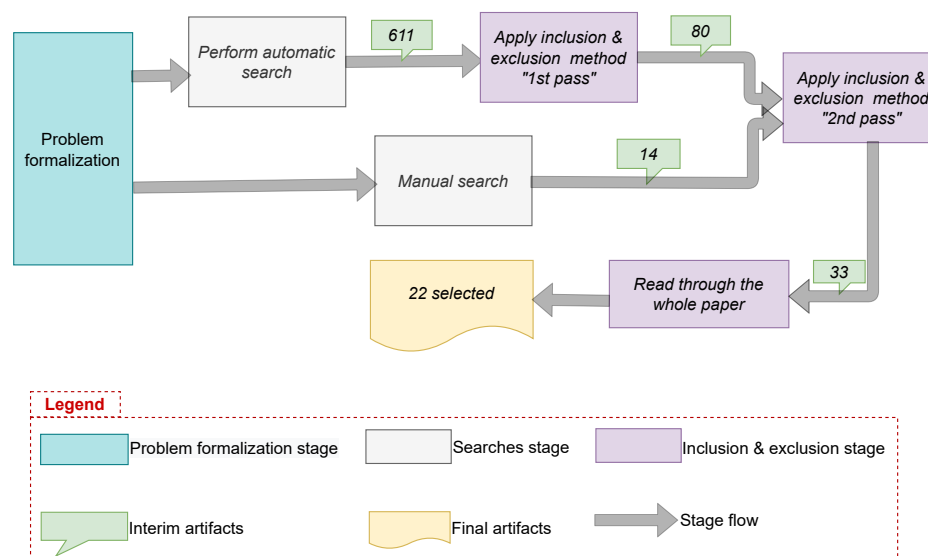

Fig. 3.1 Publications' search and selection process.

Phase 2: Automatic & manual search: In this phase, we applied a search string to several academic databases, i.e., Scopus (Elsevier) [4], IEEE Xplore [5], and ACM library [6], limiting the search to the last 10 years. In addition, we also performed a manual search, mainly using Google Scholar. The query string we used for the automated search was: (`"MDE" OR "Model Driven Engineering") AND ("IoT" OR "Internet of Things") AND ("Cloud" OR "Web"`).

Table 3.1 shows the number of papers we managed to collect in this phase.

Table 3.1 Database results' table

Database	Results
Scopus (Elsevier)	233
IEEE Xplore	263
ACM library	115
Manual search	14
Total	**625**

Phase 3: Inclusion & exclusion, 1st pass: Table 3.1 shows that 611 publications were initially discovered from various sources, in addition to the 14 papers that were manually found and considered relevant to the study. At this point, we reviewed the title, keyword, and abstract of the paper to exclude papers that did not meet the following criteria:

- Studies published in a peer-reviewed journal, conference, or workshop.

- Studies written in the English language.

- Studies that explicitly focus on the topic of the Internet of Things (IoT).

- Studies that propose a cloud-based modeling approach, either explicitly or implicitly.

At the end of this step, we had 80 articles to add to the additional 14 documents we had manually retrieved from Google Scholar.

Phase 4: Inclusion & exclusion, 2nd pass: At this stage, we read the introduction and conclusion of the previously collected contributions. We also removed some duplicates. Several documents were rejected in this phase for different reasons, e.g., because the presented approach does not explicitly offer an IoT-based cloud development environment. At the end of this phase, we had 33 documents.

Phase 5: Reading of the whole paper text: We reviewed the entire articles at this stage, focusing on the proposed approaches and their evaluation sections. Several documents were

---

[4]https://www.elsevier.com/
[5]https://ieeexplore.ieee.org
[6]https://dl.acm.org/

discarded for various reasons. For example, articles that presented hybrid solutions (e.g., local modeling with the ability to store models in remote repositories) were discarded. We also discarded approaches that claimed to create Web-based IoT data wrangling platforms by reusing existing IoT data storage platforms. Finally, we selected 22 papers that used a cloud-based modeling environment to design, develop, or deploy IoT applications.
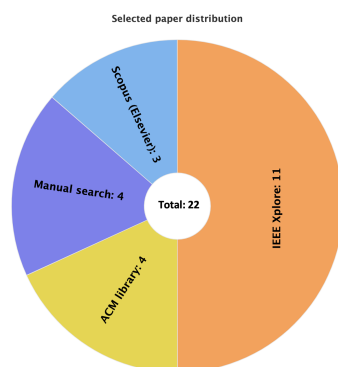
Fig. 3.2 Selected paper distribution.

Figure 3.2 shows the distribution of the selected approaches in relation to their corresponding sources. As shown in fig. 3.2, part of the selected approaches (4 out of 22) were found manually. This is due to our prior knowledge on this topic in terms of frameworks and tools

In the following, we answer the research questions presented in section 3.2 one by one by analyzing the research papers collected as described before.

3.3   Cloud-based modeling approaches (RQ1)

This section focuses on different cloud-based modeling approaches targeting the IoT domain. We have classified the studied approaches into three categories according to the focus of their interest, namely modeling structural IoT aspects, service-oriented approaches, and deployment orchestration. The goal is to answer the research question ***RQ1:*** *What are the current cloud-based modeling approaches for IoT?*

Modeling IoT structural aspects: DSL-4-IoT [188] is a cloud-based modeling tool for the IoT domain that includes a JavaScript-based front-end graphical programming language and an "OpenHAB" runtime execution engine. DSL-4-IoT provides a multi-level model-based approach to IoT application design that supports all phases of the lifecycle of these systems. Automatic model transformations are provided to refine abstract model elements into concrete elements. These transformation results, formatted as JSON arrays, are passed to the OpenHAB runtime engine for execution.

BIoTA [38] provides a cloud-based modeling approach for IoT architectures. Users can perform syntax and semantic analysis using a graphical DSL and supporting tools. BIoTA enables the computational formalization of a user-proposed software architecture using formal automata techniques. Components and connectors are created according to specific rules to meet IoT-specific scenarios, while the resulting software architecture is exported to a Docker-based deployment infrastructure.

Node-RED [158] is a popular and extensible model-driven framework to homogeneously connect IoT devices, APIs, and web services. It provides users with a web-based graphical editor with drag-and-drop capabilities. In Node-RED, users can also create and deploy dashboards in real-time.

AutoIoT [156] is a web-based platform with a graphical user interface (GUI) that allows programmers to deploy and configure IoT systems quickly. The final artifact produced by the system is a Flask project[7] (a Python micro-framework) that can be run as is or extended to meet the needs of developers who may require more complex functionality. The final system can also connect to an MQTT broker, store and query data in a database, present data to users, and exchange data with other systems. AutoIoT was later extended in [157] to allow users to model their IoT systems in the form of JSON files.

In [125], a cloud-based text language and tool for event-based configuration of intelligent environments (ECSE) was proposed. The tool allows end users, whether experts or not, to configure a smart environment using an ontology-based model. In their approach, the authors used the Resource Description Frameworks (RDF) to define the event action rules.

AtmosphericIoT [13] is a cloud-based domain-specific language and tool for building, connecting, and managing IoT systems. AtmosphericIoT Studio is a free online portal IDE that lets users create all kinds of device firmware, mobile apps, and cloud dashboards. It connects devices via Wi-Fi, Bluetooth, BLE, Sigfox, LoRa, ZigBee, NFC, satellite, and cellular networks.

In [29], the authors propose a model-based approach for building responsive and configurable user interfaces in the Web of Things. Models@Runtime are used to build runtime interfaces based on a formal model called Thing Description (TD). This modeling language aims to make attributes, actions, and events of the Web of Things (WT) accessible to the external ecosystem. The modeling language was developed in JavaScript using the VueJS framework and is publicly available[8].

---

[7]https://flask.palletsprojects.com
[8]https://github.com/smar2t/td_interface_builder

In [2], the authors presented a model-driven approach for developing IoT system interfaces. They proposed a design pattern and the necessary components for designing such interfaces in their work. The authors implemented a platform for IoT mobile and web application development based on WebRatio,[9] a generic cloud-based framework for model-driven development and code generation.

FloWare Core [65] is an open-source, model-driven toolchain for building and managing IoT systems. FloWare supports the Software Product Line and Flow-Based Programming paradigms to manage the complexity in the numerous phases of the IoT application development process. The system configures the IoT application according to the IoT system model the developer provided. A Node-RED engine [158] is integrated with FloWare.

Vitruvius [69] is an MDD platform that enables users with no programming skills to create and deploy complex IoT web applications based on real-time data from connected vehicles and sensors. Users can design their ViW applications directly from the web using a custom Vitruvius XML domain-specific language. In addition, Vitruvius provides a variety of recommendation and auto-completion features that help build applications by reducing the amount of XML code that needs to be written.

Service-oriented approaches: This category includes approaches that provide users with cloud-based modeling environments that target service-oriented architectures. In this way, different services are interconnected to build the final IoT systems.

MIDGAR [97] is an IoT platform specifically designed for service generation of applications that connect heterogeneous objects. This is achieved by using a graphical DSL where the user can connect the different things and specify the execution flow. Once the desired model is ready, it is processed by the service generation layer, which generates a tree-based representation model. The model is then used to generate a Java application that can be compiled and executed on the server.

IADev[168] is a model-driven development framework that orchestrates IoT services and generates software implementation artifacts for heterogeneous IoT systems that support multi-level modeling and transformation. This is achieved by transforming requirements into a solution architecture using attribute-driven design. In addition, the components of the generated application communicate via RESTful APIs.

LogicIoT [195] provides a text-based, web-based DSL that facilitates data access and semantic processing in IoT and smart cities environments. LogicIoT is implemented as a

---

[9]https://www.webratio.com/

set of custom Jakarta Server Pages (JSP)[10] in which various custom JSP tags have been implemented to define modeling semantics. The language consists of seven constructs: Relationships, Triggers, Endpoints, Timers, Facts, Rules, and Modules. Using the custom tags, the user can define the operations of the application required for communication between process instances and sensors without having to deal with low-level programming details.

glue.things [126] provides a cloud-based mashup platform for wiring data from web-enabled IoT devices and web services. glue.things handle both the provisioning and maintenance of device data streams, apps, and their integration. In doing so, glue.things rely on established real-time communication networks to facilitate device integration and data stream management. The glue.things modeling tool combines device and real-time communications, enabling users to describe element triggers and actions and deploy them in a distributed manner.

In [201], the authors propose a framework for scalable and real-time modeling of cloud-based IoT services in large-scale applications, such as smart cities. IoT services are modeled and organized in a hierarchical manner.

In [208], a portable web-based graphical end-user programming environment for personal applications is proposed. This tool allows users to discover smart things in their environment and create personalized applications that meet their own needs. Each of the defined smart objects can provide different functions that can be published through a well-defined API. The graphical representation of the system is then generated from the constructed JavaScript objects, in which the user can interact with the system on the fly.

E- SODA [225] is a cloud-based DSL under the Cloud-Edge-Beneath (CEB) architecture ecosystem. In E- SODA, a cloud sensor comprises a set of event/condition/action (ECA) rules that define the lifecycle of the sensor service. This allows the user to abstract and simulate the behavior of the sensor in an event-based manner. This is achieved by having the ECA rules wait for a predefined "event" to occur and responding by performing the "action" when the rule's "condition" is met. Finally, the generated cloud sensor application can be used in any cloud-based application that requires sensor data.

In [149], the authors presented an integrated graphical programming tool based on a goal-oriented approach, where end-users only need to specify their purpose in a machine-understandable way rather than designing a service architecture that satisfies their goal. In this way, the ultimate purpose of an intelligent environment can be represented graphically while hiding the complexity of the underlying semantics. An inferential component uses the

---

[10]https://en.wikipedia.org/wiki/Jakarta_Server_Pages

provided goal and analyzes whether the goal can be achieved given the available services. From this, it infers the user actions (i.e., requests involving REST resources) required to achieve the goal.

InteroEvery [82] promotes a microservice-based architecture to solve interoperability problems in the IoT domain. An IoT system is configured via a web-based graphical interface that displays the functions of each microservice. Then a universal broker connects a dedicated interoperability microservice to different adaptation microservices depending on the choreography patterns used.

Model-based deployment orchestration: Since IoT systems are deployed at different levels of abstraction, this section presents the identified approaches that aim to orchestrate the deployment mechanisms of IoT systems using cloud-based modeling environments.

DoS- IL [155, 154] is a textual domain scripting language for resource-constrained IoT devices. It enables post-deployment modification of system behavior through a lightweight script written using the DoS- IL language and stored in a ne-life gateway. The gateway hosts an interpreter for executing DoS- IL scripts.

TOSCA (Topology and Orchestration Specification for Cloud Applications) [138] aims to improve service management processes' reusability and automate IoT applications' deployment in heterogeneous environments. In TOSCA, common IoT components such as gateways and drivers can be modeled. In addition, gateway-specific artifacts required for application deployment can be specified to facilitate deployment tasks.

GENESIS (Generation and Deployment of Smart IoT Systems) [84] is a textual, cloud-based, domain-specific modeling language that supports the continuous orchestration and deployment of smart IoT systems on edge and cloud infrastructures. GENESIS uses component-based approaches to facilitate the separation of concerns and reusability; therefore, deployment models can be viewed as a collection of components. The GENESIS execution engines support three types of deployable artifacts, namely ThingML model [102], Node-RED containers [158], and arbitrary deployable black box artifacts (e.g., an executable jar). The created deployment model is then passed to the GENESIS deployment execution engine, which is responsible for deploying the software components, ensuring communication between them, provisioning the required cloud resources, and monitoring the deployment status.

Discussion: As presented earlier, there are several approaches to support cloud-based modeling in the IoT domain. Table 3.2 shows an overview of the analyzed approaches; half of them address structural issues, while only a few address deployment issues. The current state of

Table 3.2 Analyzed approaches.

Tool name	Category	Language syntax	Open-source	Tool availability	Underlying infrastructure	Generated artifact
DSL-4-IoT	Structure	Graphical	no	no	js, OpenHAB	JSON config
BIoTA	Structure	Graphical	no	no	Apache Tech. GraphQL	YAML file
IADev	Service	Textual	no	no	ASR,REST,ATL	REST app
Node-RED	Structure	Graphical	yes	yes	Node.js	Node-RED app
AutoIoT	Structure	Graphical & textual	no	no	Python, js	Flask app
[125]	Structure	Textual	no	no	Smart-M3	–
AtmosphereIoT	Structure	Graphical	no	yes	Multi-platform	Multi-platform apps
[29]	Structure	Textual	yes	yes	js,VueJS	UI code
[2]	Structure	Graphical	no	no	WebRatio, IFML	UI code
FloWare Core	Structure	Graphical	yes	yes	JavaScript	Node-RED Config file
Vitruvius	Structure	Textual	yes	no	XML,HTML,js	HTML5 with JavaScrit app
MIDGAR	Service	Graphical	no	no	Ruby,js,HTML, Java	Java app
LogicIoT	Service	Textual	no	no	JSP	-
glue.things	Service	Graphical	yes	no	AngularJS,Meshblu PubNub	NodeRED service
[201]	Service	Textual	no	no	Firebase&Node.js	–
TOSCA	Deployment	Textual	yes	yes	Multi-platform	Config files
[208]	Service	Graphical	no	no	-	-
E-SODA	Service	Textual	yes	no	OSGI cloud	OSGI java bundles
[149]	Service	Textual	yes	yes	ClickScript,AJAX	REST services
InteroEvery	Service	Graphical	no	no	Spring Boot,Rest RabbitMQ,Angular	–
DoS-IL	Deployment	Textual	no	no	js,HTML,DOM	Config files
GENESIS	Deployment	Textual	no	no	multi-platform	Genesis dep. agents

the art indicates that there is no predominant common language, although a graphical syntax is preferred.

Most of the approaches analyzed are supported by tools that are not open source. This goes hand in hand with the public availability of the methodologies. We can state that all tools that are not open source are also not publicly available. In the industrial environment, this is especially true for the internal, proprietary tools.

In analyzing each approach, we also examined the supporting infrastructures and their ability to produce deployable artifacts. In this context, we found that JavaScript-based environments such as Node.js and Angural.js are widely used for tool development. This could be due to the fact that they are among the modern languages for implementing front-end technologies. On the other hand, it seems that most of the techniques produce artifacts, even if few of them are standalone deployable components. It is also worth noting that in most cases the

deployable artifacts produced can only be deployed within the same original environment. To ensure interoperability, scalability, and reusability of the tools, the generated artifacts should generally be deployable anywhere.

3.4   Open challenges (RQ2)

The proliferation of connected, smart, and sensor-driven devices, as well as the increasing use of cloud-based models, has created numerous problems [58]. Since a typical IoT system consists of several complex subsystems, a fully cloud-based environment may become even more complicated. On the other hand, overcoming these obstacles is worth the effort because it opens up more possibilities. This section addresses the current challenges faced by IoT systems in developing and integrating such tools in a cloud-based environment. Essentially, we answer the research question *RQ2: What challenges do researchers face in developing cloud-based IoT modeling and development infrastructures?*

Extensibility mechanisms: Extensible platforms allow new capabilities to be added without having to restructure the entire ecosystem. Since IoT systems are distributed, it is usually recommended to use a microservice architecture throughout the development process [147]. Apart from that, IoT systems may require additional interactions with third-party technologies. As a result of the previous scenario, the development of tools to design and develop such distributed applications in the cloud will require efficient tools that traditional domain specialists may not have. Access mechanisms are presented by tools such as Node-RED [158] and FloWare Core [65], but there is still much to be done. Currently, domain experts need to provide cloud-based automation mechanisms and tools that enable citizen developers to add new functionality without requiring sophisticated skills or changing existing architectures.

Heterogeneity: It is an important challenge in the IoT domain, where different players are developing different applications running on different layers, namely Edge, Fog, and Cloud [147]. In addition, the deployment options and data usage methods are very different, which increases the complexity of traditional code-centric approaches [159]. Cloud-based modeling in IoT brings even more sophistication to the environment in which the system should be designed and developed. The typical cloud-based modeling platform should promote the integration of heterogeneous technological implementations, support reusability, and develop solutions close to the problem domain. Approaches such as [97, 168, 138] have presented different strategies to address such problems, but there is much more to investigate

Scalability: IoT systems are expected to serve a large number of users, perform sophisticated computations, and exchange enormous amounts of data between nodes. Therefore, supporting cloud-based modeling approaches needs to be implemented in a way that mitigates scalability

issues. One of the approaches to address such challenges is the use of container-based orchestration tools such as Kubernetes. The use of such tools can provide out-of-box features such as self-healing, fault tolerance, and elasticity of containerized resources [83]. This will also help automate cognitive processes that can detect scalability needs and autonomously adjust without human intervention.

Interoperability: Interoperability of different tools, services, and resources is critical in the IoT domain. Interoperability of cloud-based modeling platforms, especially in the IoT domain, is currently limited because different tools run in different environments and are of different natures. A tool like [82] promotes micro-service architecture by allowing all parts of the system to communicate with each other. Various regulations, such as standardization, must be implemented to achieve interoperability between different cloud-based modeling environments. To address interoperability issues, technologies such as [158, 188, 156, 65] promote a common JSON-based format for encoding models. It is worth noting that adopting Model-as-a-Service (MaaS) architectures could also promote interoperability of services and artifacts.

Learning curve: Finding professionals who can master and combine the various sophisticated technologies needed to develop and manage IoT systems is not easy. IoT professionals may lack advanced programming skills, while experienced software programmers may lack modeling expertise. For example, designing a cloud-based code generator requires an understanding of various model transformation techniques and specific programming skills. Implementing a visual mashup tool requires knowledge of modern languages such as JavaScript, HTML, and CSS.

Security concerns: Current IoT systems suffer from security concerns as data is collected from a variety of private and public nodes. In addition, data is transmitted through remote IoT gateways, which could be compromised in the process. This heterogeneity of secured and unsecured data could allow attackers to attack devices and compromise the integrity of data and operations [62]. Therefore, appropriate abstractions and automation techniques are needed to help users who may not have the required knowledge of applicable security practices.

To identify the reported challenges, as mentioned in section 3.2, a systematic literature review was conducted by defining research questions. This process involved searching academic databases, applying inclusion and exclusion criteria, and analyzing the selected papers to uncover gaps related to the challenges in the field.

3.5 Research and development opportunities (RQ3)

In this section, we explore several opportunities that we believe researchers and developers can leverage to improve cloud-based development and management of IoT systems. Therefore, we aim to answer the research question ***RQ3:****What are the main potential opportunities for future researchers and developers in the IoT field?*

3.5.1 Tools and platforms

Many tools and platforms are being developed to address cloud-based modeling issues. Therefore, now is the right time to propose powerful and extensible tools that the IoT community can use to solve their domain-specific problems. In this section, we look at several open-source and highly extensible platforms that are popular in the modeling community and that we recommend for the IoT domain.

Cloud-based development tools based on Eclipse: We believe that a significant portion of the MDE community, at least for research purposes, uses Eclipse-based technologies. This is because most Eclipse projects and technologies are open-source, which makes them more accessible and encourages individuals to participate. As of March 2021, the Eclipse Foundation hosts over 400 open source projects, 1,675 committers, and more than 260 million lines of code have been contributed to the Eclipse project repositories [86]. With Eclipse Cloud Development (ECD)[11], the Eclipse community has demonstrated its willingness to move part of its ecosystem to the Web. Eclipse's ECD Tools working group is committed to defining and building a community of best-in-class, vendor-neutral, cloud-based, open-source development tools and to promoting and accelerating their adoption. Some of the best cloud-based technologies that the IoT community can benefit from are the following:

– *EMF.cloud, GLSP, Theia -* Independent of the Eclipse modeling framework (EMF), the EMF.cloud community has recently expressed a strong desire to migrate the Eclipse-based modeling infrastructure to the cloud. This project aims to develop a web-based environment for creating modeling tools that can support the editing mechanisms of EMF-based models. EMF.cloud allows users to interact with models through the EMF.cloud model server, which coordinates the use of GLSP for graphical modeling and LSP for textual modeling. Infrastructures for code generation based on Eclipse Xtend are also included, while Eclipse Theia provides a web-based infrastructure for code editing and debugging. Several resources are available in the community to extend these tools, and we believe that IoT developers can leverage such technologies to create cloud-based IoT DSLs.

---

[11]https://ecdtools.eclipse.org/

– *Sirius Web*[12] - It is an Eclipse Sirius-based modeling tool that provides a powerful and extensible graphical modeling platform for developing and deploying modeling tools on the web. In Sirius Web, the ability to create modeling workbench in a configuration file is supported. In this case, no code generation is required as everything is interpreted at runtime [24]. Since Sirus Web is open-source, it also provides better accessibility and customizability than the desktop version, making it easier for the IoT community to get started with their cloud-based solutions.

Another alternative, such as Eclipse Che[13] makes Kubernetes development accessible to development teams. Che is an in-browser IDE that allows you to develop, build, test, and deploy applications from any machine. Finally, Epsilon playground[14] was recently launched to provide cloud-based tools for runtime modeling, metamodeling, and automated model management.

Low-code development platforms: If we look at LCDPs, the only powerful cloud-based open-source platform for the IoT that we would recommend is Node- RED [158]. Because of its high extensibility and accessibility, Node-RED provides an excellent mashup environment for IoT systems to be designed, developed, and deployed immediately. The Node-RED platform is open, and IoT system developers can create, compile, test, and deploy their own nodes in the Node-RED ecosystem. Several enhancements have been made, such as. [175], which addresses reusability issues for cloud-based modeled components, [94] to address the challenges of heterogeneity and complexity in fog-based development. Finally, in [78], the authors presented SHEN to enable the self-healing capabilities of applications based on Node-RED. As for interoperability, Node-RED models are represented as JSON objects that all third-party tools can easily use. Some of the tools in this area, such as FloWare [65] and GENESIS [84], already support the Node-RED models, which is a clear sign of their great importance. Table 3.3 outlines the main features of the recommended platforms.

---

[12]https://www.eclipse.org/sirius/sirius-web.html
[13]https://www.eclipse.org/che/
[14]https://www.eclipse.org/epsilon/live/

Table 3.3 Recommended cloud-based platforms.

	EMF.cloud	GLSP	Theia	Che	Node-RED
Open-source	✓	✓	✓	✓	✓
Extensible	✓	✓	✓	✓	✓
Scalable	✓	✓	✓	✓	✓
IoT-specific	—	—	—	—	✓
Application	Web-based EMF modeling tools	Graphical language-server-editor	Web-based code editor	Kubernetes-native IDE for DSL deployment	Flow-based programming

3.5.2   Benefits of cloud-based modeling

Although adopting a cloud-based modeling approach in IoT is fraught with difficulties, several opportunities arise, making the investment worthwhile. This section highlights several opportunities that arise once cloud-based modeling is widely adopted in IoT.

- *User communities:* The adoption of cloud-based modeling in the IoT domain has the potential to attract more citizen developers, and it will unravel many modeling opportunities on various devices such as tablets and mobile devices [46, 208, 10].

- *Collaborative Modeling:* Once IoT modeling infrastructures are moved to the cloud, it may be necessary to introduce collaborative modeling capabilities to facilitate interaction between developers and stakeholders. Unfortunately, none of the approaches discussed provide features for collaborative modeling. However, collaborative modeling can take advantage of information sharing, artifact sharing, and service exchange.

- *Productivity:* Allowing users to develop their applications using cloud-based modeling is an important step toward high productivity and shorter time to market [189]. Users can create applications tailored to their problems, and engineers focus on developing features that enable users to develop smoothly at an appropriate level of abstraction. In addition, participants focus on problem-solving in their respective domains and avoid wasting time and resources solving problems that are outside their domain of expertise.

- *Maintenance:* Traditional code-centric methodologies require a significant investment in the ongoing maintenance of developed systems. In addition, systems require periodic upgrades and installations that can be error-prone and time-consuming. System downtime is sometimes required during upgrades or to fix system errors, which impacts production. In addition, the growing demand for software systems in our daily lives and the ever-changing needs of users require an agile approach to quickly resolve these issues without impacting system availability or user access. In many cases, such

challenges are handled by cloud providers, allowing developers and engineers to focus on building applications that directly impact customer needs [72].

- *Monitoring and debugging:* Cloud-based modeling enables monitoring of activities and their archiving by their cloud providers. This is a head start for troubleshooting distributed applications, as developers can track down the microservices that are the root cause of the problems detected. Without proper cloud infrastructures, it would be challenging to troubleshoot these issues, even with features such as self-healing and repair strategies. Current cloud-based solutions have monitoring tools that help diagnose problems and monitor application usage.

- *Improved efficiency and cost-effectiveness:* Cloud-based modeling provides an efficient and cost-effective way of modeling by enabling large-scale data processing. This is because the cloud offers a wealth of capacity and processing power, as well as flexible scaling options [170]. Moreover, these services can be accessed on demand, which helps companies reduce overall costs. By hosting a cloud-based modeling platform, companies can also save on hardware costs as well as other related expenses such as maintenance and upgrades. In addition, using the cloud eliminates the need for upfront investments in infrastructure or software licenses [198].

- *Easier experimentation:* Cloud-based modeling makes it easier to experiment with different types of data sources quickly and accurately. This is because the cloud provides access to efficient integration interfaces through which a range of tools and services can be easily used for experimentation.

- *Data integrity:* The cloud provides an additional layer for sensitive data by offering secure storage options and minimizing the risk of data breaches or data loss. Organizations can maintain control of their data at all times while benefiting from the computing power of the cloud [226].

- *Flexibility:* With a cloud-based modeling platform, organizations have the flexibility to expand or reduce their resources based on their data processing and analysis needs. This allows them to quickly adjust their processing power as needed while providing secure storage solutions for sensitive data [196]. This is especially beneficial for organizations with limited budgets, as they can quickly adjust their resources to meet demand [72].

- *Faster development cycles:* Because a cloud-based modeling platform eliminates the need for local installations and updates, organizations can develop and optimize models

faster than ever before [18]. This also makes it easier to track changes and evaluate the impact of different models in real-time.

3.6   Related Works

We found several studies on MDE and DSL in IoT during our selection process. However, very few of them explicitly focus on cloud-based MDE approaches ([85, 189, 224, 204], to name a few). In this work, we explore the possible approaches to move and adopt the classical local-based MDE in IoT technologies to the cloud.

Our previous study [114] examined the current state of adoption of low-code engineering (LCE) in IoT. LCE combines LCDPs, MDE, machine learning, and cloud computing to facilitate the application development lifecycle, i.e., the design, development, deployment, and monitoring phases of IoT applications. By examining sixteen platforms, we identified a comparable set of characteristics to represent the functionalities and services that each of the examined platforms could support. We found that only 7 of the 16 platforms can be deployed in the cloud, with most of them being LCDPs, while classical MDE approaches rely on a local-based design paradigm.

In [171], the authors conducted a comprehensive assessment of model-based visual programming languages in general before narrowing their focus to 13 IoT-specific visual programming languages. The investigation was based on their characteristics, such as programming environment, licensing, project repository, and platform support. A comparison of these characteristics revealed that 72% of open-source projects are cloud-based, while only 17% of closed-source platforms are cloud-based, confirming a strong upward trend of cloud-based systems in open-source IoT projects.

In [166], the authors discussed tools and methods for developing Web of Things services, particularly mashup tools and model-driven engineering approaches. The techniques were analyzed in terms of their expressiveness, suitability for the IoT domain, and their ease of use and scalability. Although this study is related to the present work, it focuses exclusively on mashup tools and includes only a few approaches. From the preceding discussion, it is clear that few techniques have attempted to implicitly investigate cloud-based MDE approaches. Accordingly, and to the best of our knowledge, this is the first study to analyze the status of cloud-based modeling in the IoT domains.

3.7   Conclusion and future work

Data-intensive systems such as the Internet of Things (IoT) present unique challenges to developers as they build applications that overcome a variety of issues, including heterogeneity, complexity, extensibility, and scalability. Fortunately, migrating to the cloud offers several

benefits, including improved accessibility, productivity, maintenance, and monitoring. This chapter presents the results of a systematic study of the current state of cloud-based modeling methodologies in the IoT domain. After a comprehensive analysis of 22 papers proposing cloud-based modeling environments in the IoT domain, we evaluated each approach based on several characteristics, such as modeling focus, accessibility, openness, and artifact generation. In addition, we critically discussed the challenges that IoT developers may face when using these tools and technologies. In addition, we explored several general-purpose tools and technologies that may have similar applications in the IoT domain.

Chapter 4

Low-Code Engineering Repository Architecture Specification

In today's digital age, streamlining development, persistence, access, discovery, and reuse of low-code artifacts is pivotal to the entire lifecycle of modern model-driven software development. As mentioned earlier, low-code artifacts are based on MDE principles and we refer to them as model artifacts in this dissertation. To achieve our goal (a scalable and extensible cloud-based low-code model repository), we established an architecture that incorporates state-of-the-art MDE principles and tools. The repository architecture draws on current cutting-edge research in MDE techniques and tools [42]. Despite the tantalizing benefits associated with integrating MDE into mainstream software development, there are several limiting factors that stand in the way of its wider adoption; these barriers include the following:

- Although model-based processes have been shown to lead to productivity gains, they still experience limited support for discovery and reuse of model artifacts and tools. Consequently, significant upfront investments are wasted while reinventing the wheel.

- model management are highly dependent on complex development environments such as Eclipse IDE. As a result, they must be downloaded, installed and configured along with distributed software packages prior to their use.

- Limited integration mechanisms that allow developers to build reliable applications, manage new artifacts and tools, add new features, and customize their environments.

The designed low-code engineering repository offers a foundation for efficient solutions to the challenges related to managing model artifacts and tools in LCDPs and cloud-based modeling in general. We designed the proposed architecture from onset with the fundamental properties of a cloud-based infrastructures: scalability, robustness, and extensibility. Hence,

to address issues posed by conventional modeling techniques, the repository enables its users to remotely persist, access, manipulate, discover, and reuse various types of model artifacts and tools. These operations can be performed through modern application programming interfaces (APIs) to facilitate integration with the repository. By offering model management capabilities as a service, the modeling process is streamlined and made more efficient. This approach eliminates the need for local management and maintenance of inconsistent artifacts and tools stored across various local solutions, leading to significant cost reductions and faster delivery times.

The repository has the potential to serve as an essential resource for both professional and citizen developers. Its design allows users to quickly and widely access valuable curated datasets, boosting productivity and streamlining the discovery of model artifacts, and services. It also ensures secure sharing of resources, protected by policies that prevent mismanagement of intellectual property. These policies ensure safe and responsible usage of resources for both educational and commercial purposes. The repository bridges the gap between industry and academia by fostering collaboration while governing sharing of artifacts and tools. Each artifact is bound by a policy that users must follow for reuse for educational and commercial purposes. It is important to note that this architecture extends MDEForge [18] in order to support a scalable and extensible cloud-based low-code model repository. The architecture is also implemented and some aspects of its implementation are discussed in the next chapters.

4.1   Related Works

In this section, we review previous studies and works related model repositories.

AMOR - Adaptable Model Versioning [44]: The authors tailored their approach to use a by-example recorder that enables generic model versioning. In doing so, they have increased the effectiveness of the collaborative software development process. Since models are graph-based, regular line-oriented version control systems are not suitable for model versioning. AMOR provides intelligent and semi-automatic detection of conflicts, development of appropriate resolution strategies, and manual refactoring to correctly resolve potential issues.

Bizycle [133]: The BIZYCLE initiative is a platform that enables more efficient and automated software components and data integration using model-driven techniques and tools. The platform comes with a repository that stores all the metadata of the artifacts related to the integration process of model artifacts, user and role-based access rights, and generated code. They have designed their repository to follow the strategy of model and metamodel-based abstraction from platform-specific aspects to platform-independent and compute-independent aspects.

CDO[1]: CDO is an open-source solution for secure storage of EMF models and metamodels. With compatibility with relational and NoSQL databases and various deployment options such as replicated clusters, offline clones, and embedded repositories, CDO is a versatile choice for applications that require model persistence. It is developed using Java and ensures secure transactions for clients and applications using EMF protocols. In addition, this repository is designed to protect users' transactions with advanced security features.

EMFStore [128]: It is an open-source system for configuring model versioning and is distributed under the Eclipse Public License. It uses an operations-based approach to track and manage changes, detect conflicts and merge them accordingly. It consists of two components: a server that stores models, including their versions, in a repository and provides access control; and a client integrated into an application that tracks changes to the model and allows users to commit, update, and merge them.

GME - Generic Modeling Environment [135]: It's a configurable set of tools that enables users to build personalized, code generation and domain-specific environments for modeling. GME uses graphical models that can represent the application and the environment in which it operates, including hardware resources and their relationships. To this end, they provide a layer that persists these artifacts. This approach has also been successfully used for integrating tools, structurally adaptive systems, and traditional signal processing problems. They have a proprietary object-oriented binary file format that acts as a repository via a server of MS SQL.

ModelBus [105]: ModelBus is a service-oriented architecture (SOA) based approach to distributed model-driven development processes that enables the integration of custom and commercial off-the-shelf tools (COTS) for sharing data across models and services. It features a central communication infrastructure similar to a bus, a range of core services, and model management tools. These tools include an integrated model repository with version control, partial checkout capabilities, and the ability to merge model versions and fragments. ModelBus was used in the EU Modelplex[2] project to enable a distributed development scenario with hundreds of developers in different locations. It was also used in an industrial use case using Intalio Designer for Business Process Model and Notation (BPMN) modeling, Rational Software Architect (RSA) for UML modeling and transformation from BPMN to UML based on Atlas Transformation Language (ATL) and model verification with Object Constraint Language (OCL). All models, artifacts, transformations and rules are stored in the ModelBus repository.

---

[1]https://www.eclipse.org/cdo/documentation/
[2]https://cordis.europa.eu/project/id/034081

It features a central communication infrastructure similar to a bus, a range of core services, and model management tools. These tools include an integrated model repository with version control, partial checkout capabilities, and the ability to merge model versions and fragments.

Morse - Model-Aware Repository and Service Environment [109]: This paper introduces the concept of model-based services that are automatically generated and deployed using a service environment known as MORSE. This approach uses Universally Unique Identifiers (UUIDs) to support versioning of models so that different services can access the desired version of elements or models at runtime. To facilitate this process, MORSE generates and distributes appropriate service clients that allow components to monitor services associated with models in each version. Its transparent versioning enables better monitoring, governance, and self-adaptation in SOAs while reducing the manual effort required to develop services based on runtime models. It also hides complexity from users while respecting UUIDs to ensure that the correct versions are accessed when needed.

ReMoDD - Repository for Model-Driven Development [88]: It is a platform that provides users with access to an array of resources related to model-driven development (MDD). These resources include documented case studies, models, semantic models and metamodels, reference models, model and specification patterns, generic models, transformation descriptions, and modeling experiences. ReMoDD's user interface allows users to search and browse resources and engage with other members in discussion groups or forums. As a hub for knowledge sharing, collaboration, and research-based resource utilization, it acts as a resource for the MDD community. This platform enables users to share their expertise and increase productivity in academia and industry by improving MDD processes.

MDEForge: An Extensible Web-Based Modeling Platform [18]: This paper presents MDE-Forge, an extensible platform for web-based modeling. It aims to foster a modeling community with a repository and enable software-as-a-service of model management tools via REST API. By leveraging these services, it is possible to create extensions that add additional functionality to the platform. MDEForge thus provides several essential features that are missing in existing modeling platforms, reducing their adoption and relevance in an industrial context. It allows users to search and reuse model artifacts without the need for complicated and error-prone installation and configuration procedures.

According to Table 4.1, Although most existing approaches are limited to model persistence, ReMoDD provides impressive support for other model artifacts, such as transformations and metamodels. Rather than programmatically searching and retrieving existing artifacts,

	Managed Artifact	**Main purpose**	**Typical deployment scenario**
AMOR [44]	Modeling artifacts	Versioning of model artifacts	Desktop app
Bizycle [133]	Modeling artifacts	Software components integration	Desktop app
CDO	Modeling artifacts	Model persistence	Client-Server app
EMFStore [128]	Modeling artifacts	Versioning of model artifacts	Client-Server app
GME [135]	Modeling artifacts	Model persistence	Client-Server app
ModelBus [105]	Modeling artifacts	Versioning of model artifacts	Client-Server app
Morse [109]	Modeling artifacts	Versioning of model artifacts	Software-as-a-service
ReMoDD [88]	MDD artifacts	Documentation	Web-based interaction
MDEForge [18]	Model, Metamodel, DSLs	Model persistence, Added value services	Web-based interaction, Software-as-a-service

Table 4.1 Overview of existing MDE tools providing storage features.

ReMoDD provides artifact documentation for its persisted artifacts to support learning. Unfortunately, ReMoDD does not have search and reuse capabilities for existing model artifacts and tools. It is worth noting that MORSE and ReMoDD are among the few approaches that do not require downloading, configuring, or installing executables before users can access the artifacts.

ReMoDD stands out by offering developers features such as web-based search and browsing capabilities. In addition, Morse leverages the same capabilities by allowing developers to use the service as an API. Interestingly, for those seeking a consolidated experience, MDEForge goes a step further by allowing users to access the storage and management of multiple types of artifacts, such as models, metamodels, and transformations, in one place. In addition, it can be used through a web-based interface and allows programmatic use of the features offered. The value-added services offered, such as automatic classification of metamodels, remote execution of transformations, etc., can be used through the APIs provided. The main drawback of MDEForge are the need for more support for storing and managing relevant artifacts of interest in the context of the EU Lowcomote project, including DevOps workflows, quality assurance artifacts, and advanced mechanisms for providing appropriate recommendations to users.

4.2   System Views

This section presents the architecture of the developed low-code engineering repository. The development followed an iterative and incremental approach, where the activities consisted

of multiple sprints and functional deliveries, as shown in fig. reffig:sprint. The iterations included analysis, design and implementation, testing, and deployment, which were organized into sprints based on well-defined functions. We combined different methodologies such as Scrum, Kanban, and Extreme Programming to manage the development. Scrum helped us organize our workflow into sprints to ensure independent, fully functional features, while Kanban helped us visualize the workflow and better organize the backlogs. We used Extreme Programming to improve responsiveness and quickly update changes and requirements.
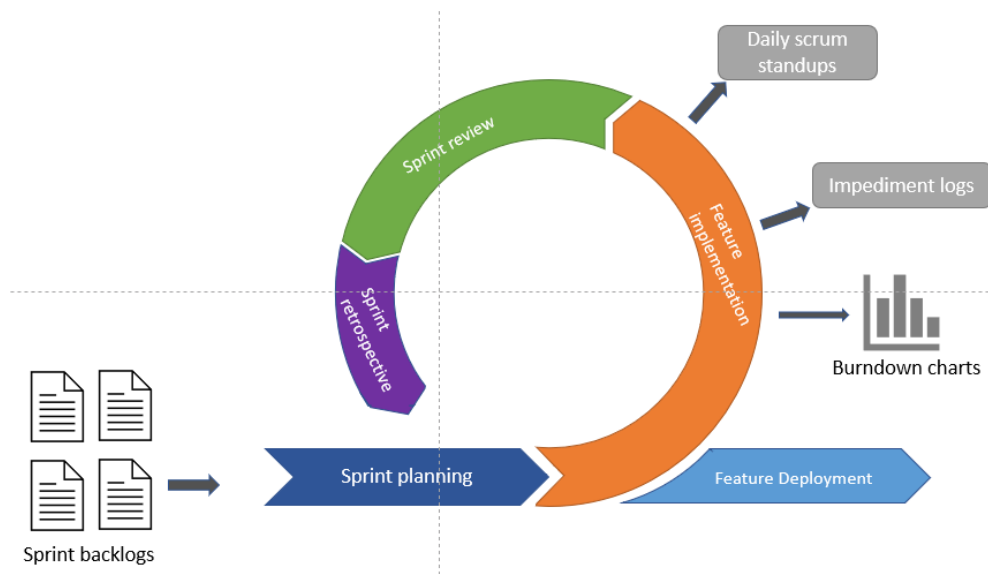

Fig. 4.1 Scrum iterative sprints

To describe the architecture of the proposed system, we used a modified version of the *"4+1"* View model, a model for describing the architecture of software-intensive systems based on the use of multiple concurrent views." [35]. The modification to this model is the addition of a sixth view, namely the data view (see fig. 4.2). Since the repository is intended to persist and facilitate the discovery and reuse of model artifacts, we decided that an additional view was appropriate to show the structure of the data. Thus, the six views used to describe the repository architecture are as follows:

- **Use Case View:** This view is at the center of the "4 + 1" architectural view model because the remaining views orbit around it. It represents the user requirements that capture the system functionality by illustrating user interaction with system stakeholders.

- **Data view:** This view describes the organization and type of data that the system uses to exchange or transfer data during task execution. By identifying the types of data and

their organizational structures, we can easily manage and coordinate tasks and enable efficient task execution, risk mitigation, and improved productivity.

- **Logical View:** The logical architecture view of the system describes the system based on the functional requirements that meet the needs of users and stakeholders. The system is divided into key abstraction sections to better describe the design approach and mechanisms that tackle the identified challenges.

- **Development View:** It is based on the static view and organization of the system. It outlines all components integrated to provide the desired functionality. Hence, the architecture is constructed with extensibility at its core to foster the system's ability to grow and accommodate new ideas without causing chaos through drastic changes. In addition, the static structure can be kept flexible to facilitate maintenance.

- **Process View:** To describe the dynamic aspects and reveal the interdependencies of the repository's functionalities, we mapped different levels of abstraction to understand their unique behavior better. Hence, during integration, developers can have an insightful representation of the processes to allow them to create efficient solutions that remain true to the fundamental architecture of the repository.

- **Physical View:** In this view, we focused on non-functional features (i.e., high availability, performance, reliability, and performance) that emphasize the hardware environment for better component placement to maximize throughput and system efficiency, such as reliability, scalability, performance, and availability. As a result, we strategi-
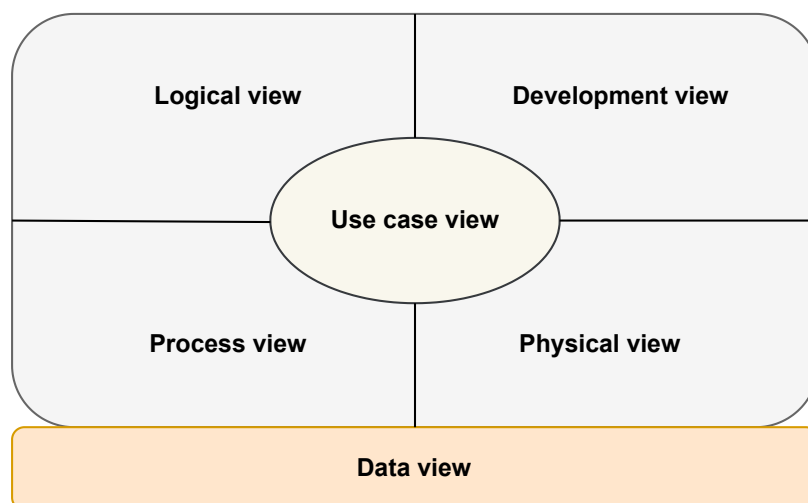

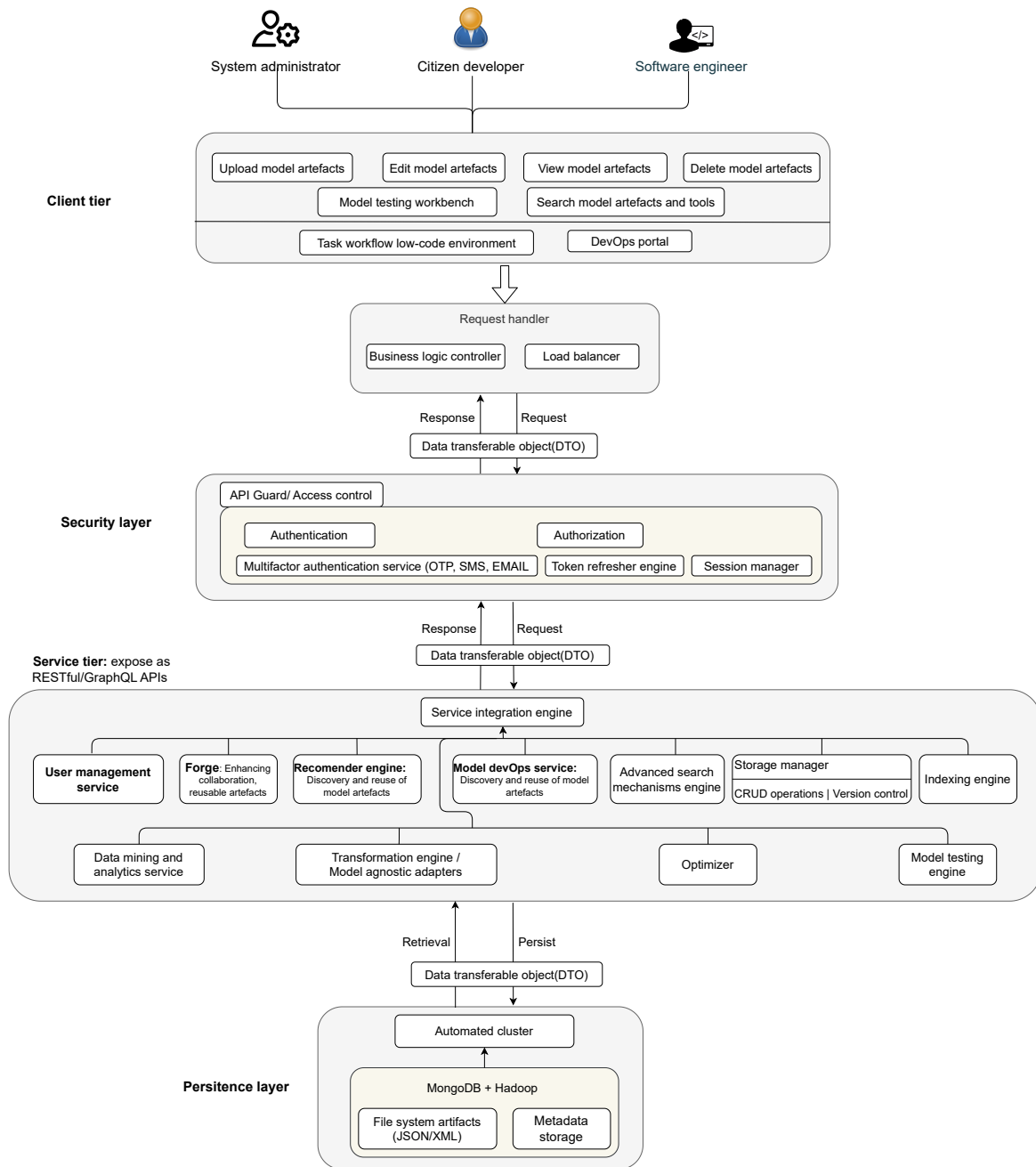Fig. 4.2 The extended *"4+1"* views of the Lowcomote repository architecture

Fig. 4.3 High-level architecture view

cally balanced resource utilization between compute nodes to create an efficient and optimal system architecture.

Understanding the various stakeholders involved in the management and interaction with our repository is critical to its success. The views presented in fig. 4.2 outline the concerns of each stakeholder group, including *system administrators*, *citizen developers*, and *software*

engineers. The high-level diagram depicted in fig. 4.3 summarizes the system and its relationship with key stakeholders. It provides a birds-eye view into the dynamics at play, guides stakeholders' interactions, and ensures optimal information flow governance.

The *client layer* within the system provides a straightforward way to interact via a web-based interface with RESTful/GraphQL API specifications. In this context, the API specifications encapsulate low-level functionalities and expose relevant system functionality externally. This architecture strives to load balance server workloads to prevent overload and adverse effects on performance. Immediately after the load balancer, a *security layer* is added to protect the API from malicious activities. This way, we further improve the application's security features and ensure that resource access is managed correctly through proper authentication and authorization of users and services. This combination of load balancing and robust security measures helps ensure the performance and availability of the system while streamlining low-level processes and operations.

The *service layer* of this system includes core repository services, external integrated services, and third-party extensions. Examples of these third-party services and extensions include Recommendation Engine [112], Model DevOps Operations [61], and Model Testing Engine [123]. The *persistence layer* consists of cluster databases and file systems that are orchestrated and deployed to be highly available, fast retrieval enabled, and scalable. Together, the service and persistence layers provide our main building blocks for effective system scalability and extensibility.

The high-level view of the system is described in detail in the following subsections according to the extended 4+1 model mentioned earlier.

4.2.1 Use case View

From the user's perspective, this view represents the functionalities of the repository. As mentioned earlier, we envision three different stakeholders of the repository, as shown in fig. 4.3, i.e., the *software engineer*, the *system administrator*, and the *citizen developer*. The following paragraphs present the repository functionalities organized according to these identified users.

Software engineer: Software engineers are supposed to be experts in model-driven engineering and the development of low-code development platforms. They can extend the repository by integrating new functionalities. In this respect, Figure 4.4 shows the two prominent use cases involving software engineers as described below:
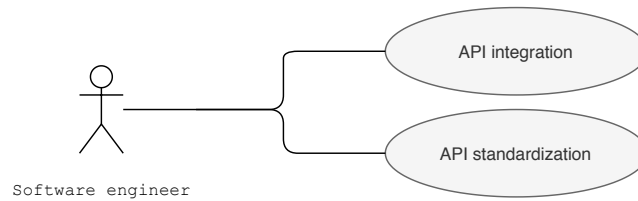
Fig. 4.4 Software engineer use case view

- **API integration:** This integration allows software engineers to programmatically customize and enhance the repository capabilities by either reusing built-in functionalities or adding new capabilities without the overhead. API integration mechanisms on our repository follow dedicated guidelines to ensure the best practices and maintain high-quality standards.

- **API standardization**: Developing and maintaining a modern, extensible software architecture is a challenging undertaking. Therefore, the system mitigates these risks by providing dedicated extension points for software engineers while enforcing the use of API specification services such as OpenAPI[3] and GraphQL[4]. This approach streamlines API documentation, encourages the usage of an API playground to explore resources, and simplifies testing. Besides, it also streamlines development processes and future maintenance.

System administrator: System administrators manage repository infrastructure and resource usage. They also administer access control of resources and user management. As shown in fig. 4.5, system administrators have the following use cases in the repository:
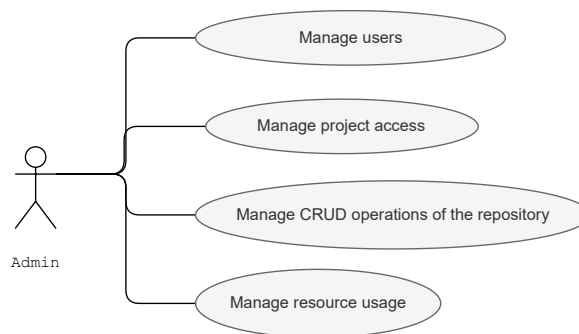

Fig. 4.5 System administrator use case view

- **Manage users:** The system administrator is responsible for managing user accounts and their respective resources. She/he has complete access and administers activities

---

[3]https://swagger.io/specification/
[4]https://graphql.org/

such as adding and deleting users, resetting passwords, temporarily locking user accounts, and controlling user access level of resources and APIs.

- **Manage project access**: The administrator ensures that users have correct permissions for the artifacts and tools presented in a given repository's project. She/he can revoke or grant access to project resources at any time, depending on whether those resources are being used in accordance with repository policies and regulations.

- **Manage CRUD operations of the repository**: The administrator has superior access over the projects and tools managed by the repository. Hence she can perform CRUD operations on projects and tools when needed. For instance, the administrator may deactivate a specific service or other operations that she deems inappropriate.

- **Manage resource usage**: The repository admnistrator access level grants ability to allocate the necessary resources for high-intensive operations. They can downscale or upscale these resources based on demand is essential to ensure that operations are responsive to changing conditions. With advances in automation, adminstrator can maintain a steady level of performance and utilization without compromising closure performance. Hence, she/he has a significant amount of flexibility and control, allowing for minimal latency in service delivery and optimization of resources.

Citizen developer: Citizen developers are the main target of the system, and all the provided functionalities have been defined by considering the potential needs that inexpert developers expect from a low-code model repository. As shown in fig. 4.6, the functionalities provided by the Low-Code repository are grouped under four main categories, i.e., *Model Repository*, *DevOps Model Framework* [61], *Model Recommendations* [112], *Model Mining* and *Model testing framework* [123] which are described in detail below.
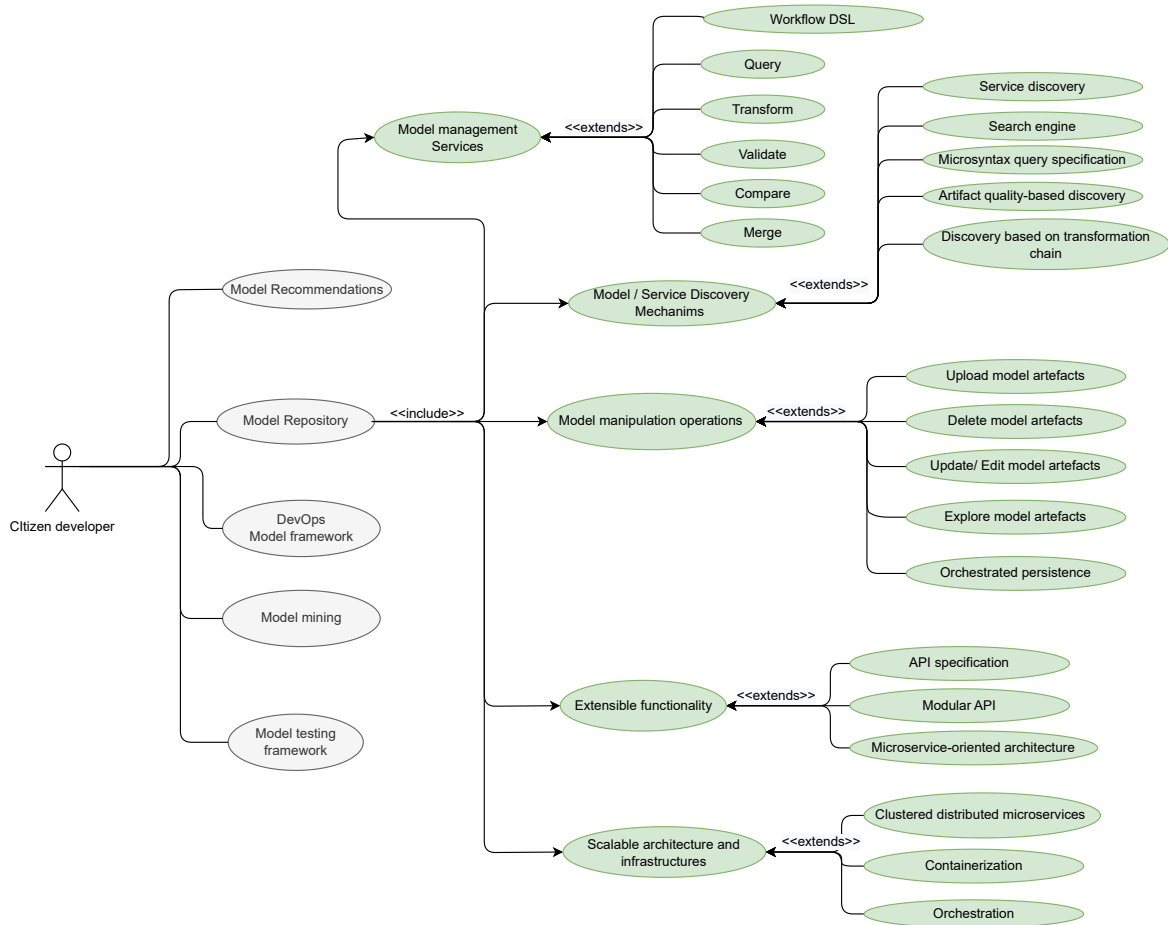
Fig. 4.6 Repository's feature use case view

(i) Model Repository use cases: represent the core functionalities pertaining to managing heterogeneous model artifacts in the repository. Below, we discuss model management use cases and key features of the system:

- **Model Manipulation Operations:** The system implements several functionalities that enable manipulations of model artifacts in the repository. The basic manipulation functionalities are:

  - The citizen developer can upload model artifacts to the repository.

  - The citizen developer can edit/update model artifacts persisted in the repository via provided user interfaces.

  - The citizen developer can discover, view and explore selected model artifacts using various interfaces.

  - A citizen developer can delete model artifacts from the repository.

These operations are modular and represent the smallest logically isolated functionalities. Therefore, higher-level abstraction operations can be implemented by reusing or extending them accordingly.

- **Model / Service Discovery Mechanisms:** This repository feature allows users to find relevant domain models and model management services. It is intended to allow developers to reuse existing domain models and thus avoid reinventing the wheel and starting modeling processes from scratch (c.f. chapter 6).

  The repository's discovery mechanisms target model artifacts mainly, but we have also enabled the discovery of services available in the repository via service registry (cf. chapter 5):

  - *Service Discovery*: This feature allows the user to discover model management services in the repository. A dedicated service registry is responsible for recording available services, their status, and logs.

  - *Search Engine*: This is an integrated, industry-standard engine that indexes a variety of information from uploaded artifacts. This engine enables the retrieval and reuse of artifacts in the repository in the shortest possible time.

  - *Microsyntax query specification*: This is a domain-specific query specification built upon the repository to enable extended and comprehensive queries of artifacts. The query specification can filter artifacts based on multiple criteria in a query string, including their quality metrics and attributes.

  - *Artifact quality-based discovery:* A quality assessment service has been integrated into the discovery mechanisms at the repository to allow the user to retrieve artifacts based on their quality metrics and attributes.

  - *Discovery based on transformation chain:* The user can discover artifacts based on the transformation chain in which the artifact of interest can be consumed.

- **Extensible functionality:** The architecture and design of functionalities at the repository strive to facilitate the extensibility, reuse, and integration of services available at the repository.

  - *API Specification*: The APIs of the repository are provided externally as services. Thus, they are executed and consumed on demand. To enable remote reuse and

integration, API specifications are used to facilitate this endeavor through the use of SWAGGER 2.0 OpenAPI[5] and GraphQL[6] API specifications.

– *Modular API*: Modularity is a technique of breaking down a system into smaller, self-contained components. With a modular API, we have developed an API that consists of self-contained components that are loosely coupled but highly cohesive. This practice promotes the reuse and scalability of current implementations.

– *Microservice oriented architecture*: A microservice is a highly coherent, decentralized single-purpose service. It should have only one purpose, be self-sufficient, and communicate through a well-defined interface. We used this approach to easily track down and fix bugs in isolation without affecting other services and their performance. As a result, services are executed on demand, and adding new services leaves other services in the cluster intact hence their holistic management and maintenance.

• **Scalable architecture and infrastructures:** The repository is built on an architecture that provides automatic scalability and resource management. The infrastructures are cloud-based to facilitate remote service access and reusability.

– *Clustered distributed microservices*: By having microservices in a cluster, the services are managed in a pool, and we benefit from load balancing, strategic monitoring of resources and traffic split across multiple microservices. We can also manage workloads by reserving nodes for some intensive workloads with special requirements. Overall, service clustering is at the core of our architecture to support our main objective: improving repository scalability and extensibility.

– *Containerization*: Containerization of services using Docker technology[7] bolsters decoupling applications from their dependencies from the host system. Hence, the portability of the container is much more straightforward and can run consistently and seamlessly across different environments. In addition, we can easily scale and perform versioning of containers efficiently.

– *Orchestration*: We orchestrated services using Kubernetes. By orchestration, we mean the organization of multiple clusters to function as a single unit. In this way, the current implementation supports easy coordination of containerized services' deployment, scaling, and management. Resources are cons more efficiently, and tasks are automatically distributed to the most appropriate machines, depending

---

[5]https://swagger.io/specification/v2/
[6]https://graphql.org/
[7]https://www.docker.com/

on the available load. High availability and load balancing are also enabled by cluster orchestration.

- **Model management services:** We transformed model management operations that query, validate, transform, compare, and merge model artifacts into services. This allowed remote access to these services and their execution on demand. Based on these services, we developed a *task workflow domain-specific language* that enables the discovery and reuse of model management services. Hence, now services can be executed based on user intended workflow.

The following use cases are part of the design of the lowcomote repository [8] to include functionalites such as model recommendations, continuous integration, and a framework for model testing.

(ii) **Continuous Software Engineering use cases**: DevOps is *"[...] a development methodology aimed at bridging the gap between Development and Operations, emphasizing communication and collaboration, continuous integration, quality assurance and delivery with automated deployment utilizing a set of development practices."* [118]. In this context, Continuous Software Engineering (CSE) is a software engineering approach in which the target system, or at least some of its components, is designed, developed, and deployed in short, regular, iterative, and often (partially-)automated cycles. CSE is a broad term to refer to many continuous-* activities like Continuous Development, Continuous Integration, Continuous Deployment, and Continuous Delivery to form complete DevOps processes [92]. The repository supports model-driven CSE approaches by providing specific CRUD functionalities for DevOps process and platform models [61]. In addition, it aims at supporting the current, and future extension of the DevOpsML conceptual framework presented in [61].

(iii) **Model Recommendations use cases**: During the construction of new models, relevant model artifacts that are stored in the repository can be reused to facilitate the modeling process or to improve the state of an underdeveloped model. The possible reuse of model artifacts are offered to the citizen developer in terms of recommendations [112].

(iv) **Quality Assurance use cases**: In any Low-Code development platform, citizen developers should be involved in all the phases of the application development process, including testing. Therefore, the platform left room for specific functionalities to test the Low-Code system under development. Testing operations are specified in terms of models [123].

---

[8]https://www.lowcomote.eu/

4.2.2 Logical View

The logical view describes our repository by decomposing it into a set of collaborating components. This view's primary focus is to provide technical details of the system functionality and services alongside their interaction. A component diagram, which shows the aforementioned decomposition is illustrated in fig. 4.7 and described in the following.
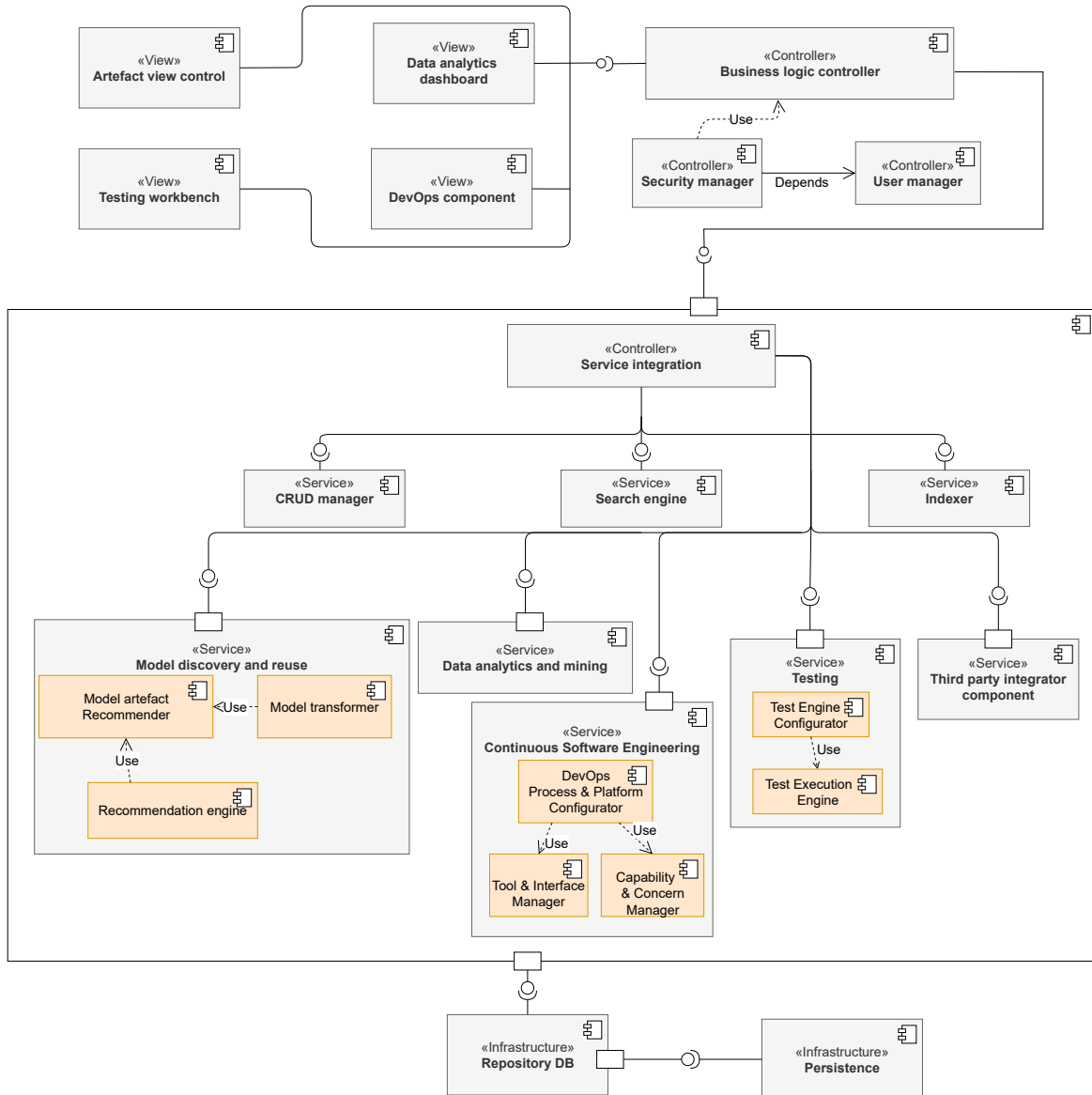

Fig. 4.7 System's logical view

The repository design features core components in its central infrastructure, but also accommodates other additional components that were developed in the parallel under the EU project Lowcomote[9]. The following are the key components of the repository.

- **Artifact View Control:** This component belongs to the client layer and serves as the cornerstone of the repository's basic functionality by presenting its contents such as model artifacts, tools and services. It gives users the ability to manipulate models by uploading, deleting and updating model artifacts in the repository.

- **Business Logic Controller:** This component is critical as the main application logic, providing a wrapper for backend functionality and an access point to the external world. We maintain a consistent structure of publicly available APIs using OpenAPI 3.0 and GraphQL specifications. Therefore, we manage and filter all interactions with the server and external parties from a single access point to ensure optimal load balancing and performance.

- **Security Manager:** This component provides secure access to the repository content. It provides authentication and authorization and protects against unauthorized access and potential security threats. As the gatekeeper of the system, it contains the necessary logic for access control and includes functions for user session management and token-based authorization.

- **User Manager:** It manages the system's security interactions with users and services. The component is responsible for keeping sensitive information, such as credentials, secure while ensuring authorized access throughout the system. In addition, the User Manager component provides user profiling to enhance the overall experience.

- **Service Integration:** This component aims to provide a platform for components to interact via their APIs. It supports the Business Logic Controller for overall coordination of security, load balancing and individual services. We can use this component to monitor and optimize the built-in operations across the system.

- **CRUD Manager:** It is responsible for uploading, deleting, updating, editing and viewing model artifacts. It constantly synchronizes with the persistence layer to ensure data integrity and information updating.

- **Search Engine:** This repository-integrated engine allows users to quickly and conveniently search, discover, and access relevant model artifacts. Advanced discovery mechanisms built into the search engine give users the ability to narrow the search query and navigate through the repository content. The current index can be expanded by

---

[9]https://www.lowcomote.eu/

collecting artifacts from the Internet. However, guidelines for the automatic collection of these artifacts must be documented and considered.

- **Indexer:** It ensures accurate indexing and synchronization of data in the index from our cluster store to the search engine when artifacts are uploaded or updated. Since some services, such as computing artifact metrics, are triggered on these events, this component ensures that the data they generate is indexed along with the artifact document and metadata.

- **Persistence:** This component combines multiple database APIs that store data from a clustered repository and provides a single output of data from the repository. In addition, we have implemented safeguards to ensure that properly authenticated requests are fulfilled and that each data request from the repository database cluster is authenticated and authorized with updated tokens.

- **Repository DB:** It is a central hub for processing, managing, and storing logs collected throughout the system. It is a wrapper for additional data processing tasks that feed metadata into the repository and advanced configurations. In this way, it is used when additional scripts are required that directly manipulate and update persisted data.

Below are the integrated components from the EU Lowcomote project [10]. They use the repository data to build model recommendations, continuous integration, and a framework for model testing.

- **Model Artifact Recommender** [112]: This component provides recommendations to the user during the modeling process. The recommendations can be triggered proactively (by the system) or reactively (by the user). For example, the recommendations can suggest to the user the next modeling step, how to name a new model artifact, and what type of relationship can be specified between newly created model artifacts. The recommendations can also provide useful suggestions for improving the current state of an underdeveloped model that has just been uploaded to the repository.

- **Recommendation Engine** [112]: For a given underdeveloped or under-construction model, recommendations are realized by the query engine, which receives as input the model details and queries the repository to obtain the same or similar models. The models retrieved from the query result are compared with the model under construction.

- **Model Transformer** [112]: After the recommendation engine has retrieved similar models and given the user the appropriate recommendations, the model transformer component is busy applying the selected recommendation to the model being built.

---

[10]https://www.lowcomote.eu/

- **Continuous Software Engineering (DevOps)** [61]: This component acts as a top-level container for CSE-related services and serves as a facade for i) citizen developers interacting directly with the low-code engineering repository and ii) external tools that can be integrated into a dedicated low-code platform configuration (c.f. platform modeling in [61]) that support specific DevOps processes (e.g.., continuous Delivery [61] or test functionalities (c.f. Deliverable D4.3 [124]). In particular, a DevOps process and platform configurator are responsible for two main activities: i) supporting the configuration of DevOps processes and platforms [61] based on available libraries of tools (with their interfaces), according to given requirements (e.g.., required capabilities to address specific problems [61]), ii) collection of tool descriptions and the processes they support in shared libraries (e.g., tool descriptions created by tool vendors [61]). In addition, the DevOps Process and Platform Configurator are supported in the execution of its functions by dedicated manager components (Tool and Interface Manager, Capability and Concern Manager) with specific functionalities for existing libraries (c.f. fig. 4.6) (e.g., collection of predefined queries and recommendations for process and platform models).

- **Testing** [123]: This service provides facilities for quality assurance of low-code systems developed by the Low-Code Development Platforms (LCDP). Several components are considered to support different phases of testing (including test design, test generation, test execution, and test evaluation) for different LCDPs. The *Test Execution Engine* component is a dedicated and configurable engine compliant with the Test Description Language (TDL); it is a standardized language for abstract test case definitions [145]. Citizen developers can design test cases using the TDL language (i.e., create TDL models). TDL is not executable, but the engine makes TDL models executable. It also handles automatic configuration, execution, and evaluation of functional tests at the model level. The engine can be configured for different domain-specific languages (DSLs) via the 'Test Engine Configurator' component so that the test service can be customized and reused by different LCDPs.
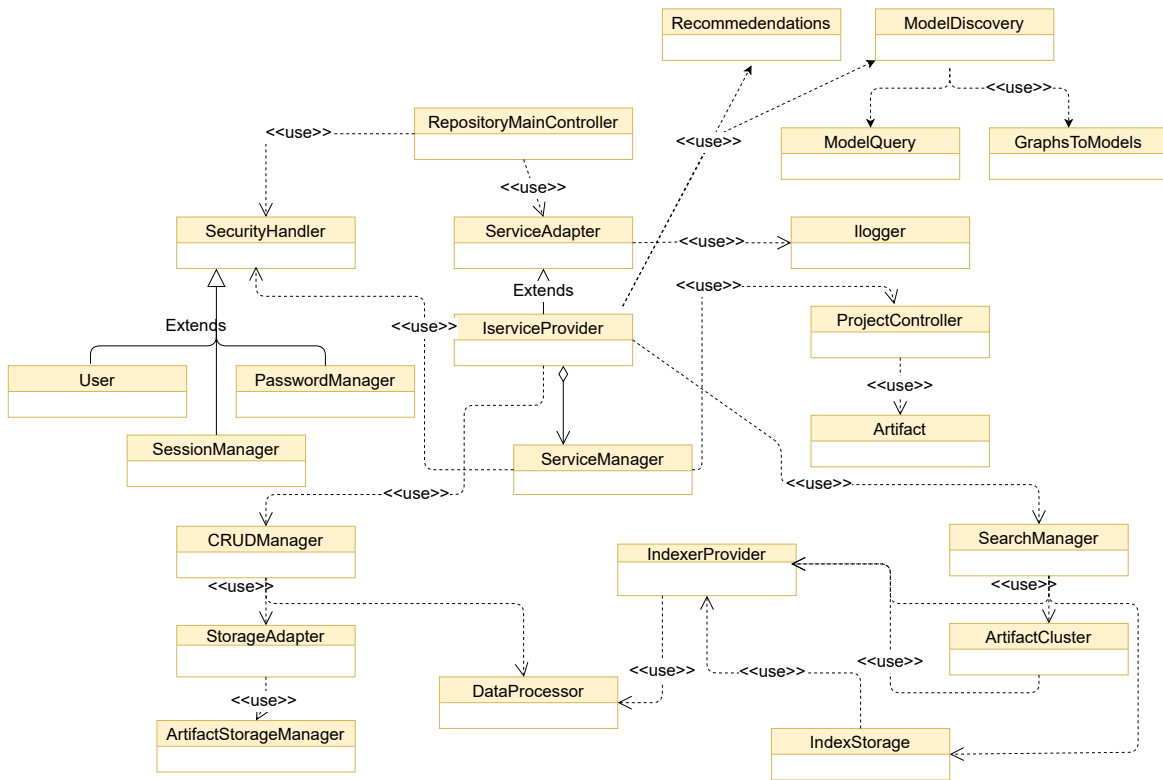
Fig. 4.8 System's development view

4.2.3   Development View

The development view focuses on the actual software design of the repository. The static structure of the system is illustrated in fig. 4.8, and the main elements are described in Table 4.2.

Table 4.2 Description of the main Lowcomote repository static structure elements

Element Type	Name	Description
Class	**RepositoryMainController**	This class acts as system's nucleus, allowing external entities to use internal functionalities. It streamlines and hides the complexities of internal components and functions, offering a simple and modular approach to handle external user requests.
Interface	**ServiceProvider**	This interface allows extending system's functionality by defining new features as contracts, thus ensuring a proper implementation. Hence, we keep the codebase maintainable and extending does not compromise system architecture or major restructuring.

Class	ServiceAdapter	ServiceAdapter enables seamless interaction between different interfaces without the need to alter the original source code. It provides a practical solution for utilizing services that are incompatible, hence enabling integration with legacy systems and external services.
Class	SecurityHandler	The Security Manager class oversees the security of the repository, providing access to security APIs. It acts as a middleware, shielding internal security components and ensuring that API communication goes through it before accessing system functionalities.
Interface	Ilogger	Ilogger interface ensures consistent logging throughout the system for improved performance tracking and debugging. These logs provide valuable insights for troubleshooting, maintenance and data analysis to aid administrators in decision making.
Class	User	It implements user management operations such as deleting, creating or modifying users. It can also manage the access level assigned to each user. It is used by *SecurityHandler* as a parent class to include authentication and validation processes to ensure that all actions are performed securely.
Class	SessionManager	This class manages the creation and deletion of tokens and their validity along a given period of time. All security techniques such as refreshed tokens, OTP or two factor authentication are managed from this class.
Class	PasswordManager	This class manages the passwords of the users, hashing, and other password strengthening techniques are either implemented or reused in this class.
Class	ServiceManager	This class manages internal services. All load balancing, logging, and performance metrics will be calculated in this class. Services are organized and better defined security wise for a better consumption at this point. It is inherited from the *ServiceProvider* abstract class.
Class	ProjectController	This class manages projects that are stored in the repository. A project contains different artifacts and other user can be given access roles on selected artifacts. It uses the *SecurityHandler* class.
Class	Artifact	This is the main entity class that manages the artifacts of the repository. It captures the overall data context of incoming and stored artifacts.
Class	CRUDManager	This class manages the CRUD operations of the different kinds of artifacts stored or to be stored in the repository.
Class	ArtifactStorageManager	Implementation of CRUD operations are implemented in this class.
Class	StorageAdapter	provides access to CRUD operations, including the *ServiceManager* class, serving as a gateway to other external classes. This adapter can resolve incompatible operations before reaching the *ArtifactStorageManager*

Class	DataProcessor	Inherited from the CRUD manager, this class extracts needed metadata. These metadata are organized into a predefined data structure for later consumption by data analytics and machine learning tasks.
Class	SearchManager	This class coordinates the searching of artifacts and tools managed by the repository. This class several utility classes that prepare the query strings to be used for searching processes. The query is then sent to the query engine for information retrieval.
Interface	IndexProvider	This interface exposes indices of artifacts stored in the repository to services such as search and recommendations that use them. Thus, anyone who wants to use indices internally from the codebase should implement this interface.
Class	IndexStorage	This class implements operations that ensures indexing in external custom search engine. It can also be used to implement search capabilities from scratch in the system. Hence, it has predefined data structure that would be used while performing indexing and its management.

4.2.4 Process View

The process view provides an informative overview of the dynamic capabilities of the repository. It clearly and concisely illustrates how each major functionality is performed within the system. Four main processes are presented in this section: Model Management, Continuous Software Engineering, Model Recommendation, and Quality Assurance Services. Through these processes, the repository can be customized to meet the needs of the user by focusing on essential activities along the software development pipeline.

Model Management Services: As explanatory services, we show the sequences of steps performed when searching for model artifacts and uploading new artifacts. We have selected two main use cases for demonstration below:

1. *Search model artifacts*: This use case retrieves model artifacts according to user requests. Model artifacts are indexed along with their metadata to ensure fast searching. For each use case, the process starts with authentication. Then, for each service, authorization is performed against another service to validate sessions and tokens. As can be seen in fig. 4.9, the service integration component acts as an access component since internal components cannot be accessed directly from the implementation components. A single access point helps us ensure and evaluate service performance and data integrity.
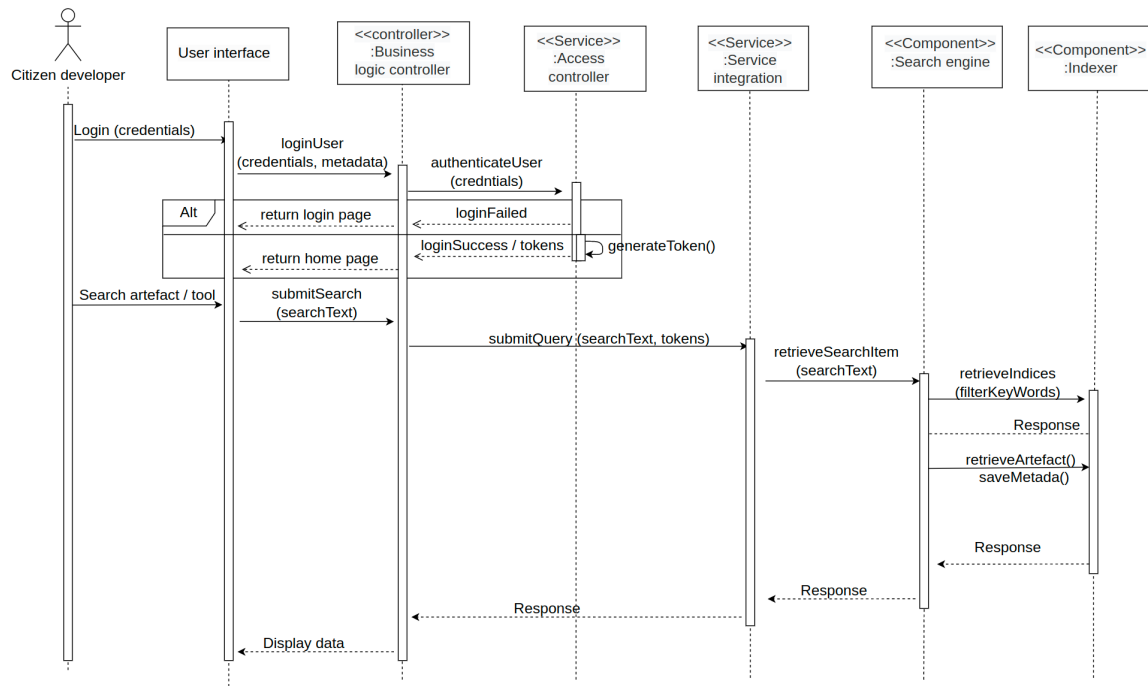
Fig. 4.9 Search model artifact process view

Before executing the query, the data is indexed from the cluster storage or databases into the search engine. The search engine component with the indexer takes care of synchronizing the data in the index and in the cluster storage. We chose this architecture to avoid mixing and instead delegate search activities to an entity specialized in the search domain. The database remains intact to support efficient CRUD operations, and the data is indexed in its respective entity to optimize artifact discovery.

2. *Upload of model artifacts*: Uploading model artifacts is a key feature of the repository. The process starts with user authentication by checking their authorization access level before availing resources. Once authenticated and authorized, the uploaded artifacts are stored and organized in a cluster of databases. The service integration component acts as a facade for the API, delegating the action to the relevant API. The persistence component performs operations on the artifact, which is then crawled to extract information used in discovery mechanisms. The Model Metrics Calculator evaluates the artifact and generates metadata about its quality. The artifact metadata, such as name, size, source, extension, and type, are extracted and stored in a clustered storage. Finally, the indexer component synchronizes the data with the index.

Continuous Software Engineering services: The Low-Code Engineering Repository provides the means for CRUD functionalities for the model-driven artifacts of the DevOpsML
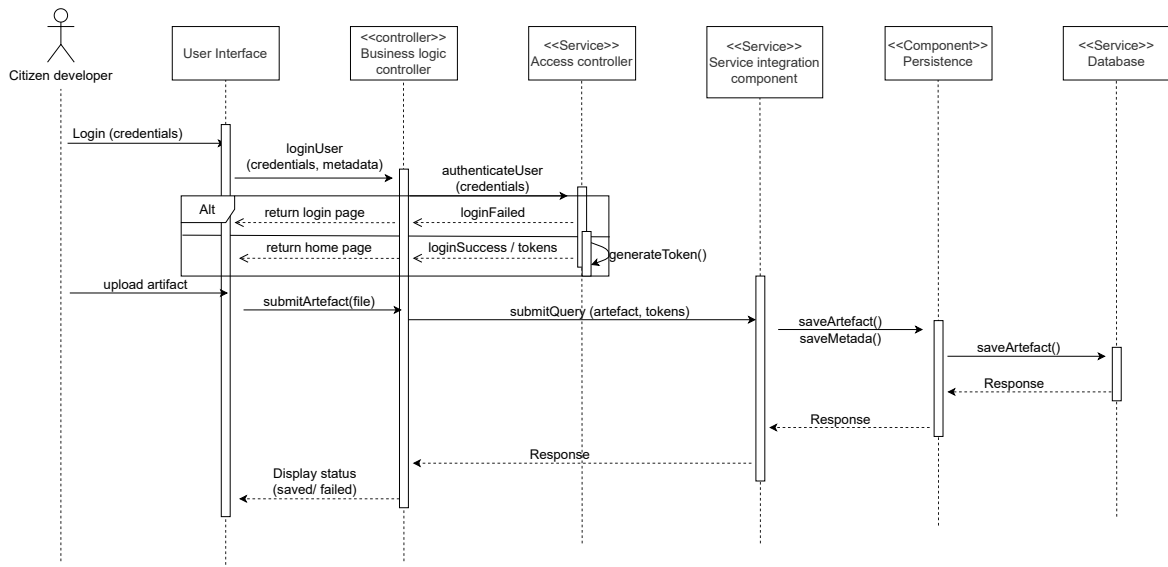
Fig. 4.10 Upload model artifact process view

framework presented in [61], and a prototypical implementation of the DevOpsML framework is available [32]. In [61], an informal activity-like workflow is presented that shows its main four activities, namely, i) *process modeling*, ii) *platform modeling*, iii) *library modeling* (to collect reusable model elements for Platform Modeling), and iv) *process and platform weaving*.

Figures 4.11 and 4.12 provide a high-level view of possible interactions among the Low-Code repository and external tools dealing with process and platform models, respectively.
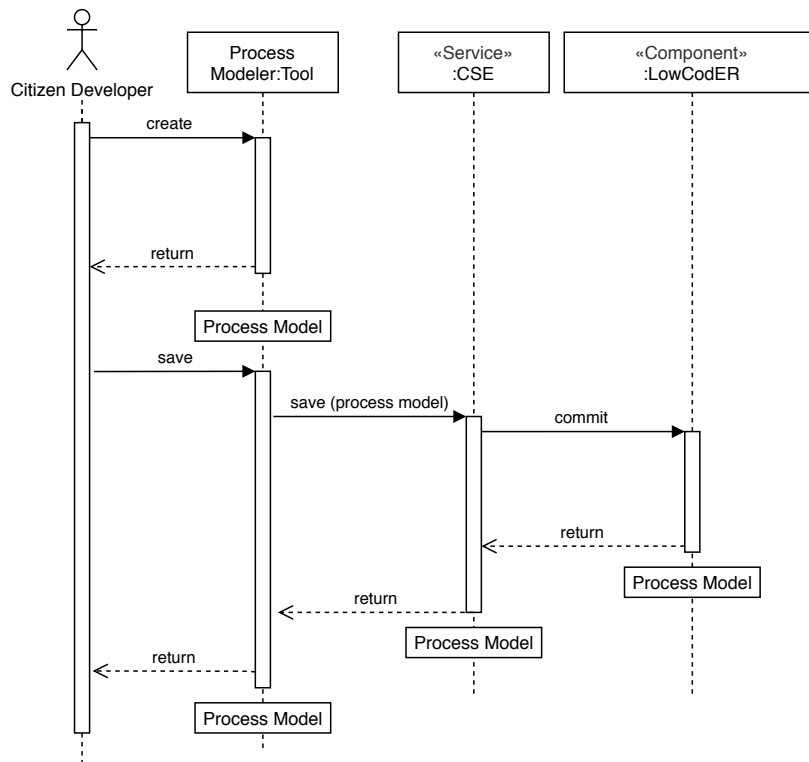
Fig. 4.11 Creating a process model from an external tool and storing it in the low-code engineering repository through the CSE service.
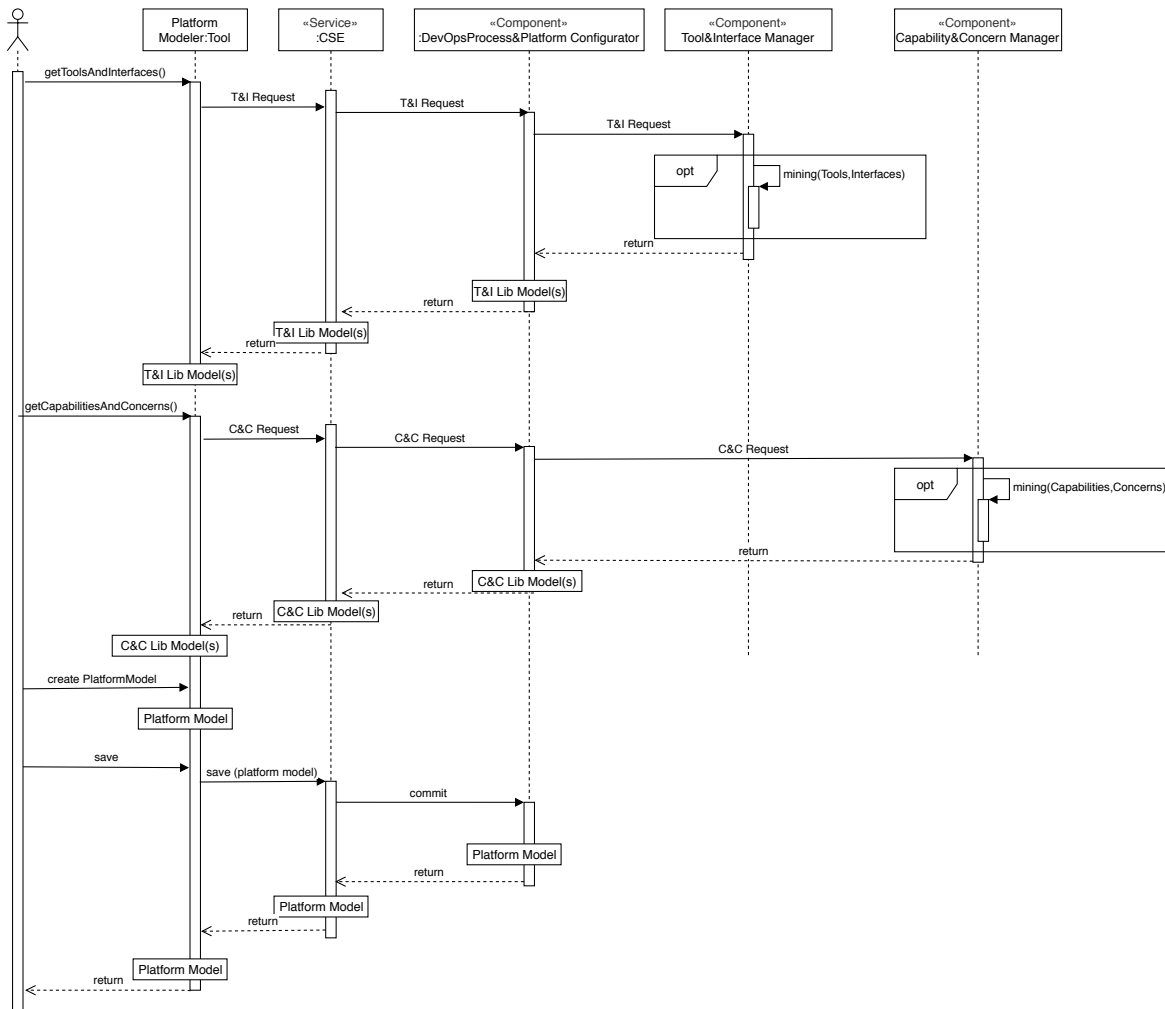
Fig. 4.12 Creating a Platform Model from an external Tool via reusable libraries.

In both interaction scenarios, the citizen developer is performing modeling activities via external tools, which, in turn, communicates with the Low-Code Engineering repository via the dedicated Continuous Software Engineering (CSE) service to support CRUD operations for models.

In DevOpsML [61], the process modeling activity is expected to be mostly supported by existing DSL and functionalities provided by LCPDs [186]. In [61], the OMG SPEM language and compliant modeling tools (e.g., SPEM plugin for MagicDraw UML[11]) have been used.

A Platform Model artifact is obtained at the end of the interaction depicted in fig. 4.11, which is editable by the citizen developer and stored in the repository.

---

[11]https://www.nomagic.com/product-addons/no-cost-add-ons/spem-plugin

Library and platform modeling activities are as well intended to be supported by external modeling tools. In [61], preliminary dedicated metamodels have been presented for specifying platform elements, i.e., *i)* tools with their interfaces, and *ii)* capabilities and concerns that allow their collection in separate libraries.

Figure 4.12 depicts a high-level interaction scenario with a citizen developer that creates a library of platform elements. According to the DevOpsML framework [61], D4.3 [124] shows how the citizen developer can play different roles depending on her background. In particular,

- *Requirement engineers:* she can express requirements by creating libraries of *required* tools, interfaces, capabilities and concerns.

- *Tool provider:* she can store the model of her preferred/available tools as a collection of provided interfaces and capabilities addressing given concerns[12].

The `Tool&Interface` and `Capability&Concern Manager` components (as shown in fig. 4.7) are in charge of providing repository-specific functionalities to support the DevOps Platform Configuration use case (Figure 4.6), like collecting statistics and suggesting recommendations of candidate platform elements for suitable configurations with respect to given requirements. For this purpose, if applicable, data and process mining techniques are invoked on available MDE artifacts (i.e., the data) and CSE processes.

---

[12]Feature models of LCDPs [186] and existing classification of DevOps tools [56] can be used for this task.
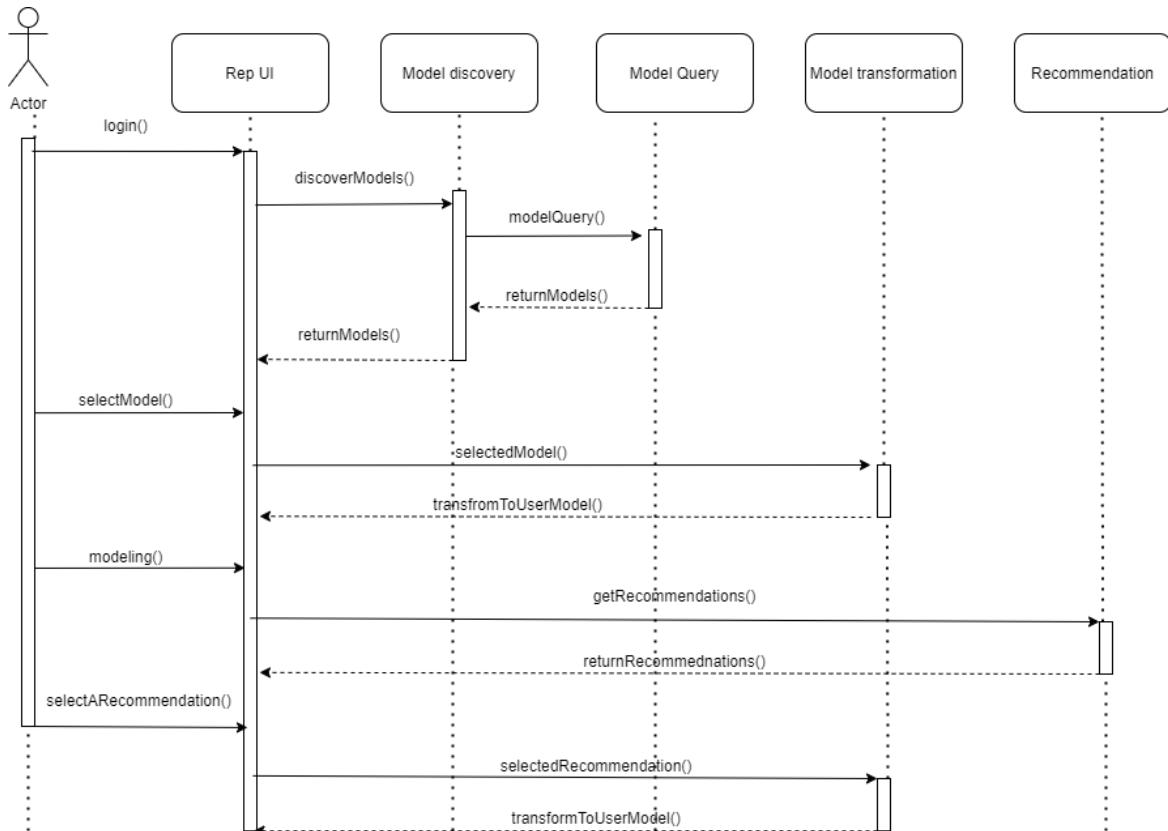
Fig. 4.13 Model discovery and reuse

Model discovery and reuse services: As shown in fig. 4.13, after a citizen developer logs into the repository, she can discover relevant domain models to reuse these models or at least not start modeling from scratch. The discovery process is carried out by using an advanced query facility from the repository. If the developer decides to reuse any given model, then this model are loaded to the UI, and the user can customize it.

If the user starts modeling from scratch or customizing any uploaded model, recommendations are triggered from the repository to the user. For example, suppose the user selects to use any given recommendation. In that case, the selected options are transformed into the model under construction format and merged with the model in the respective context.

Quality Assurance services: Concerning the quality assurance services, we show the processes that are related to the creation of links between test and SUT models (see fig. 4.14) and to get notifications in case of changes occurring in the model of the SUT (see fig. 4.15). In both cases, we focus on the interactions between an *Actor* (that is a citizen developer), a *Test modeler tool* (which acts as a user interface for the actor to model test cases), a *Testing*

service (that is described in Sec. 4.2.2), and the Low-Code repository (hereafter named *LowCodeER* for convenience).

Make links between test and SUT models: When the citizen developer requests to link a specific test model to its related SUT model (i.e., the system model being tested by that test model), through the Test modeler tool, the tool asks the testing service for the path of the SUT Model. The path should be retrieved from LowCodeER since all SUT Models are persisted there. After returning the path to the modeler tool, it then requests the testing service to set the reference to the SUT model in the intended test model, which consequently resulted in an update request from the testing service to LowCodeER to make sure that the reference is saved and can be used later on.
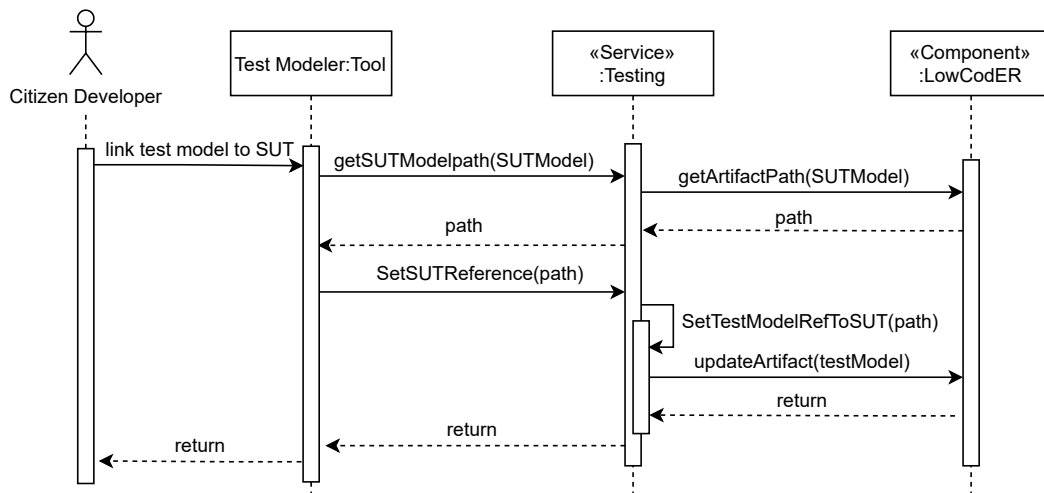

Fig. 4.14 Setting the links between test models and their related system models (SUT models)

Get notification of SUT model changes: The LowCodeER component notifies the testing service of updates in the SUT Model. This notification triggers an operation in the testing service which repeats the related test cases to avoid inconsistency between them and the updated SUT model (this is identical to automatic regression testing). To this end, the testing service requests the repository to retrieve related test models and then executes them. After running tests, two states could happen:

1. all tests passed, meaning no updates in the test models is required.

2. at least one of the test models is failed, meaning that the citizen developer has to update failed test models.

In both cases, appropriate notifications are sent to the citizen developer through the modeler tool.
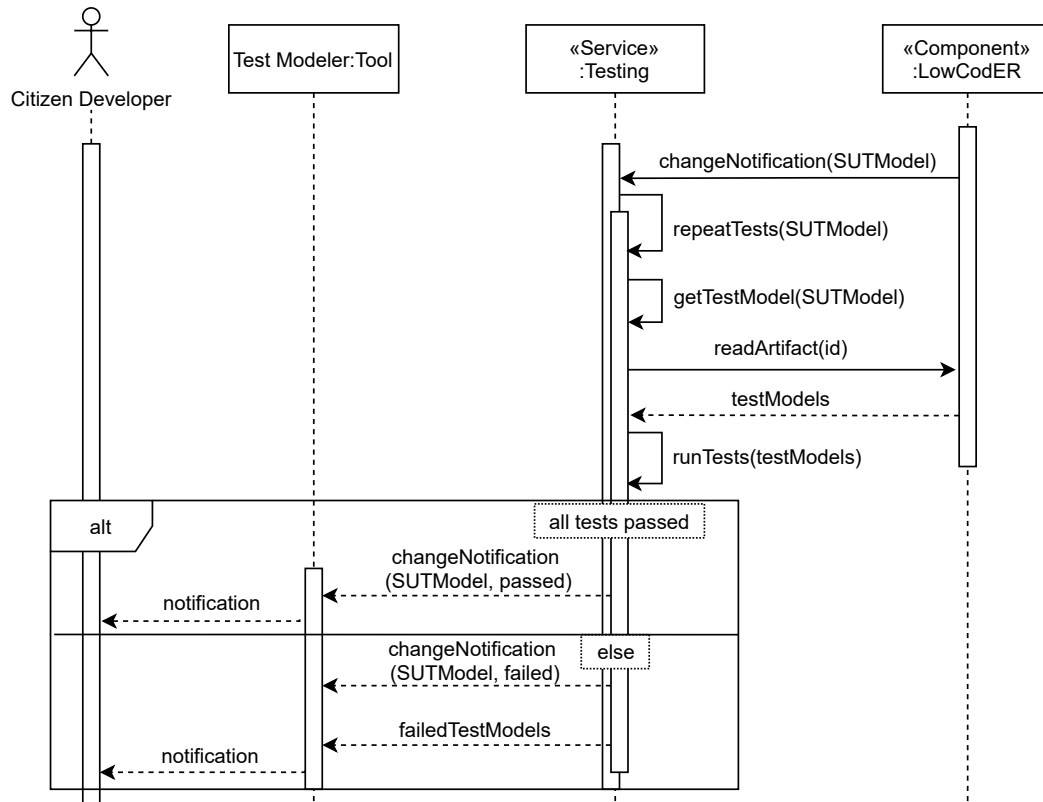
Fig. 4.15 Getting notification of system model changes along with the related failed test models

4.2.5 Data View

The data view captures the essentials of the data transfers that are exchanged between system processes. We implemented services on the repository as remote services, which implies that numerous round-trip calls between the client and the server are performed.

Figure 4.16 shows the data transfer object (DTO) that aggregates the data repeatedly transferred between processes by multiple calls into a single object. This object is stored, retrieved, serialized for transfer, and deserialized for consumption. Manipulation of the DTO occurs in the persistence component, which coordinates the persistence layer of the system. The response consists of two concepts: the message and its content. A message is a JSON object with metadata included, and the content can take any form depending on the service involved.

4.2.6 Physical View

The physical view describes the multi-tier architecture of our cloud-based model repository as shown in fig. 4.17. The system includes different physical components, according to
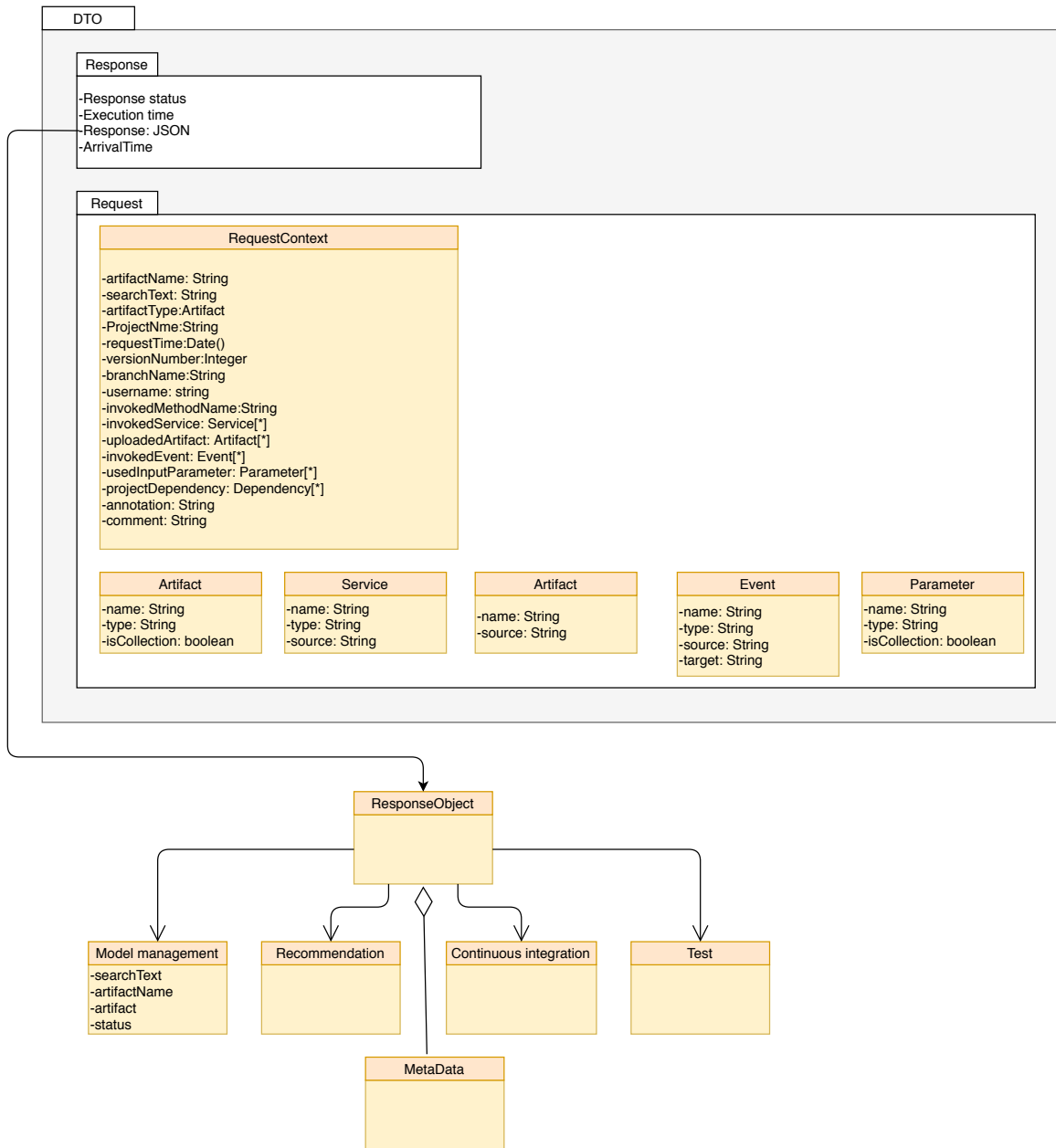
Fig. 4.16 Data view

the high-level architecture shown in fig. 4.3. The client exploits the core repository functionalities but also integrated services from partners [13] that are working on other aspects of low-code engineering. The services are made available in the *Client Layer* and include model management services, recommendations [112], model testing workbench [123], and continuous integration services [61]. Such services communicate with the *Logic Layer* by
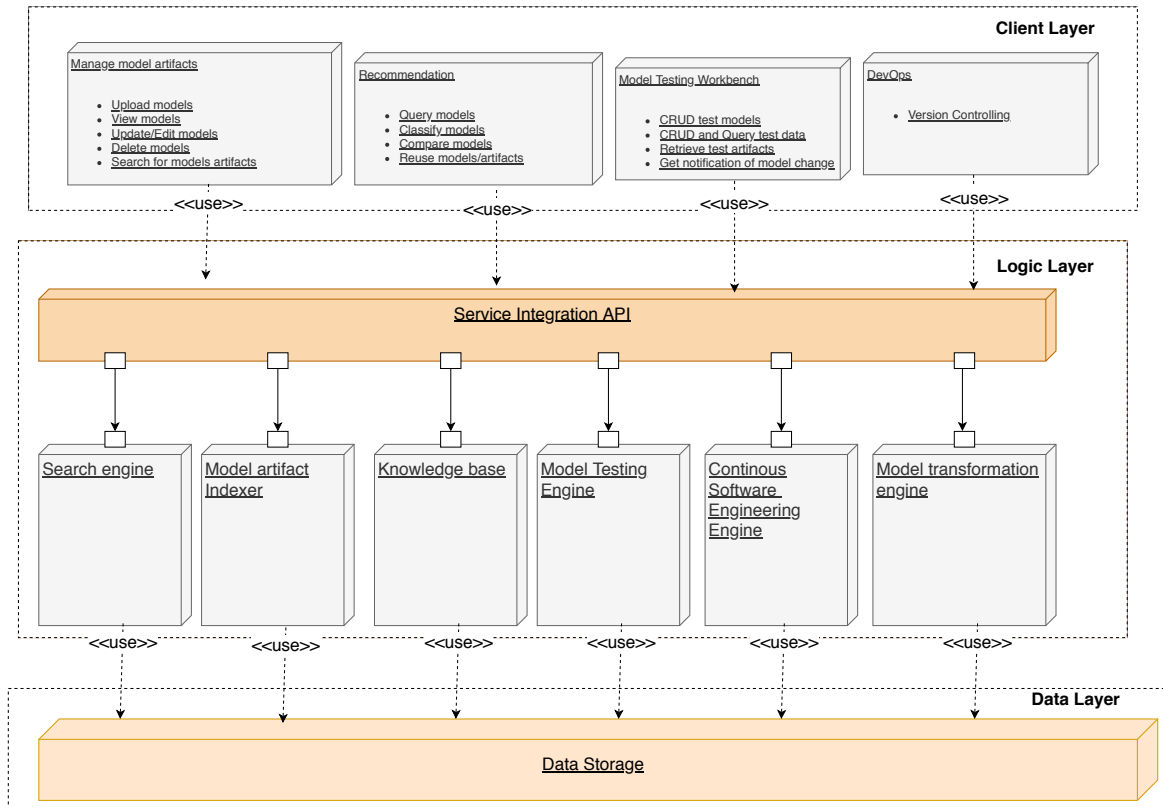

13

Fig. 4.17 System's physical view

using the *Service Integration API* that plays the gateway role for all the other services, which in turn can access the data layer providing storage facilities.

4.2.7 Implementation Overview

Throughout the program, we successfully implemented essential core services directly associated with the model repository, as indicated in the use case view 4.6. As previously mentioned, objective was to create a scalable and extensible cloud-based model repository. We began by implementing a distributed microservices-oriented infrastructure on the cloud. This involved creating an integrated cluster to house all services, implementing monitoring and communication security mechanisms to ensure that our infrastructure is well-maintained and easy to extend or scale. Building upon this orchestrated microservice infrastructure, we proceeded to containerize, shipped and deploy model management services onto the cluster. Subsequently, we developed model manipulation operations that go beyond CRUD (create, read, update, delete) operations, which can be accessed remotely via APIs (REST API or GraphQL). These operations can be remotely accessed through APIs, such as REST API or GraphQL (c.f. chapter 5). Furthermore, we have implemented advanced discovery and reuse

mechanisms within the repository, which are elaborated on in Chapter 6. All services on the repository can be accessed remotely via API specifications mentioned above.

Initially, the architecture of the repository was designed to support various functionalities, including model recommendations, a devops model framework, model mining capabilities, and a model testing framework. Unfortunately, due to time constraints, our colleagues responsible for these services were unable to deliver them on schedule. Nevertheless, our architecture and current implementation have provisions and placeholders in place to seamlessly integrate these services when they become available. We tested the repository's functionalities by integrating it with external systems, as in the case of the Droid recommender framework [7]. As the repository continues to evolve, traditional evaluations can be conducted to further assess its performance.

4.3 Conclusion

This chapter presented the architecture of a cloud-based model repository that was adopted to support extensible and scalable mechanisms to discover, mine, and reuse heterogeneous model artifacts. The repository architecture inherently enforces modeling as a service, starting with management capabilities. Hence, modelers, especially citizen developers, are enabled to access, manipulate, discover, reuse, and persist various model artifacts in an easily accessible and user-friendly repository. In addition, this architecture enables easy maintenance and organization of data regarding the low-code development process. This allows developers to access a plethora of containerized services and enough data to process the artifacts further. With this repository, citizen and professional developers can quickly locate wanted artifacts without tedious searches and reuse them in services that have already been deployed in the repository. In addition, these artifacts can be stored securely using established conventions that allow for reuse in future applications with clear guidelines and governing policies. Overall, this repository architecture aims to increase the productivity of modelers, i.e., citizen and professional developers, by streamlining model management operations to enable the discovery and seamless reuse of model artifacts.

Chapter 5

Composition, discovery, and orchestration of model management operations

Model management services play an essential role while developing complex systems by means of MDE practices. They carry out several model management operations (MMOs), including model transformation, validation, comparison, and merging, which are exposed as remotely consumable services. However, the adoption of MMOs on cloud-based model repositories has raised issues related to their composition, discovery, and orchestration. Notably, it is an arduous and error-prone task to carry out the composition and execution of complex workflows involving different model artifacts consumed by various model management services. For instance, modelers must identify the proper atomic operations among available services, connect to remote model repositories, and figure out their composition to satisfy the final goal. Different composition proposals have been introduced in MDE, even though a satisfactory solution has not yet settled.

The Modeling-as-a-Service (MaaS) [45] initiative promoted the adoption of model management operations as services over the internet. To this end, as presented in previous chapters, several repositories have been proposed in recent years by academia and industry to enable the reuse of model artifacts and their remote execution as services [18]. Thus, the development of complex systems using the MaaS paradigm requires the composition of multiple atomic services that must be properly discovered, containerized, and orchestrated. However, currently available model management activities do not facilitate such operations, as they must provide remote APIs [115] and provide the means to register and discover the services to be composed. Therefore, developers must seek out the required modeling tools and associated model management operations from locally provided infrastructures and work out their composition to develop the desired system from scratch. The main challenges

that make model management operations (MMOs) composition an arduous activity are the following:

- Current composition tools mainly deal with locally available resources.

- Composition mechanisms like ANT tasks are specific for the particular ecosystem at hand (e.g., Epsilon[1]).

- The development of complex engineering processes require technical expertise that citizen developers (i.e., domain experts with limited programming skills) might not necessarily have, though deemed to be aware of involved services.

- Local deployment of modeling infrastructure makes MMO composition error-prone and time-consuming.

- Lack of access to APIs that facilitate on-demand remote execution of MMOs.

- Lack of mechanisms to discover and reuse MMOs over the internet.

In this chapter, we propose the use of a low-code development environment for the development of complex model management processes. The context under consideration is characterized by atomic model management operations provided as services by (potentially) different providers. The envisioned environment supports the discovery and orchestration of the services required to develop the desired composite process. The goal is to develop an event-driven approach based on trigger-action programming as practiced by LCDPs. In these scenario, users can connect different independent services, organize and customize them in a certain flow to achieve their goal [169]. Similarly, the proposed platform supports high-level abstraction and automation to compose model management services provided by different repositories.

We also present MDEForgeWL, a textual DSL with a complete infrastructure to support the execution of MMO workflows available remotely as dedicated services. MDEForgeWL enables efficient composition and discovery of model management services and model artifacts using a dedicated low-code development platform. In addition, a dedicated DSL allows the user to specify complex workflows to orchestrate the underlying model management services. The language is based on a *trigger-action* paradigm, where services can trigger the execution of other services. Users can plan, organize, customize, and execute any model-driven task workflow by incorporating independent model management services into a specific flow to achieve their defined goals [115].

---

[1]https://www.eclipse.org/epsilon/doc/workflow/

The MDEForgeWL engine is implemented in a microservice-oriented architecture that leverages Kubernetes technology. Kubernetes [2] provides out-of-box benefits such as auto-scalability, extensibility, and dynamic selection of services based on workload [115]. In addition, Kubernetes enables model management services to be discovered and used via remote APIs. The code repository of MDEForgeWL is available online.[3] Thus, the main contributions of this chapter are as follows:

- Support service and model artifacts' discovery through the MDEForgeWL platform.

- Empower the user with a DSL to define custom workflows involving model management services.

- Enable orchestration, abstraction and automation of model management services.

- Facilitate extensibility and scalability of model management services in a cloud-based model repository.

The chapter is organized as follows: Section 5.1 discusses the background, and Section 5.2 provides an overview of related work and a comparison of existing approaches to compose model management operations. Section 5.3 presents the architecture of MDEForgeWL and the implementation processes of the platform at its different levels. We also present the MDEForgeWL language in practice with an illustrative example, while section 5.4 concludes the chapter.

5.1   Background

Traditional data-intensive systems are undergoing a digital revolution. This phenomenon has significant implications for the variability of product types and customization possibilities during their life cycle [227, 212]. Significant efforts and investments are required to implement and maintain these complex systems, which limit their acquisition by small and medium-sized enterprises (SMEs)[220]. The complexity of such systems is exacerbated by reliance on code-centric approaches, which have proven to be daunting due to the amount of effort required to program, customize, and integrate complex heterogeneous systems that originate from different engineering domains and processes [60].

This complexity sparked the need for flexible approaches that adapt to system behavior in the face of ever-changing requirements, structural changes, and unexpected conditions [212]. To develop data-intensive systems, MDE promotes the adoption of models as machine-readable and processable abstractions specified using dedicated languages such as System Modeling

---

[2]https://kubernetes.io
[3]https://github.com/Indamutsa/model-management-services.git

Language (SysML)[4]. Dedicated tools are used to support development and analysis tasks, to integrate technical processes and stakeholder perspectives, and to promote information exchange during these processes [26].

To simplify the development of complex systems, trigger-action programming paradigms can be used to facilitate automation and abstraction. For example, such a paradigm is being used in the Internet of Things (IoT) field to develop applications in smart home management, agriculture, e-health, industrial automation, and robotics [199]. Systems such as IFTTT and Zapier are examples of LCDPs that facilitate business process automation by allowing users to specify processes [169]. In particular, they allow the creation of new services, known as recipes, from a user-defined concatenation of services based on conditional statements [215]. For example, a user can like a particular post on Facebook and automatically archive it to a corresponding store in the cloud [162].

The remainder of this section discusses core concepts of this work related to aspects of *service-oriented architectures* and to the development of *domain-specific languages*.

Service-oriented architecture: The current uptrend in service-oriented computing is transforming traditional software systems and infrastructures. This digital transformation involves a shift from a centralized architecture into dynamic and distributed systems that support cloud-based services [163]. This new paradigm uses cloud computing to encapsulate heterogeneous and autonomous services into a service pool that exhibits various functional and non-functional features [120]. Cloud computing is defined by the National Institute of Standards and Technology (NIST) as "*a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*" [120]

Migrating to the cloud facilitates affordable access to reliable and high-performance hardware and software resources and cuts expenses related to system maintenance and security [120]. Moreover, such migration is a pillar in supporting features such as collaboration, remote reuse, high availability, extensibility of model artifacts, and their management services [74]. In addition, cloud computing offers many benefits such as on-demand self-service, broad network access, resource pooling, rapid elasticity, measured service, multi-tenacity and auditability, and certifiability [120]. Besides, it fosters inter-organizational interaction by enabling service discovery, composition, and execution of their business logic [120]. Hence, to achieve a fully operational complex service, atomic services are combined to process data to achieve the user goal, often referred to as composition [136].

---

[4]https://sysml.org/

Generally, a service is defined as an invokable network, an independent high-abstracted and self-contained remote operation that executes low-level functionalities and might return some data [181]. In the MDE context, a model management service is a containerized model management operation (e.g., transformation) along with its engine and auxiliary operations that manipulate input model artifacts and ensure the return of output data. Service discovery is finding and querying from a registry of services that exhibit given functional and non-functional features. In this aspect, composition stands for the operation of discovery, selecting, combining, and executing cloud-based services to achieve the user's goal [148]. To enable these features, service preconditions, effects, inputs, and outputs are encoded in a computer-interpretable form such as a DSL [163].

DSLs used in service composition are designed to support the specification of composite processes, facilitate interoperability between service users and providers, and enable flexible and dynamic invocation of ad-hoc external services [64]. The resulting complex composite services from the complex invocation chain must scale with the number of composing services. Service composition offers two significant benefits to the developer and the user. For the developer, it advances service and artifacts reusability, and from the user's perspective, she has seamless access to a variety of complex services [197].

Domain-specific languages (DSLs): pave the way for domain experts to leverage their knowledge in developing otherwise complex functionalities using intuitive text encoded with instructions for machines to execute [216]. They are preferred for two main reasons: firstly, they unclog a challenging bottleneck in software development: communications among stakeholders and engineers; secondly, they increase productivity among developers [87]. Due to the arduous effort involved in developing a domain-specific language (DSL), MDE techniques are wielded during their design and implementation [27].

MDE techniques express solutions at the same problem domain level in this context. Developing a DSL comprises several phases that result in a compiler capable of reading the text, parsing it, and generating executable code. To realize a DSL, developers take advantage of frameworks such as JetBrains MPS and Xtext [216]. The former offers projectional editing that facilitates parsing the text, thus overcoming the limits of developing DSL editors [64]. As for Xtext[5], it requires a grammar specification and generates the complete customizable infrastructure needed to build a fully-fledged domain-specific language. Xtext provides an out-of-box lexical analyzer, parser, and abstract syntax tree using the EMF model, type checker, compiler, and editing support for the Eclipse modeling framework (EMF). Moreover, it supports the Language Server Protocol (LSP) for client-server communications [27].

---

[5]https://www.eclipse.org/Xtext/

5.2   Composition of model management tools

In this section, we make an overview of existing approaches to compose model management tools (c.f. Sec. 5.2.1). Different criteria are also presented to elaborate a comparative table of the analyzed approaches (c.f. Sec. 5.2.2).

5.2.1   Overview of related work

Berardinelli et al. [26] identified relevant challenges that hinder the adoption of model-driven approaches for cyber-physical production systems engineering and discussed issues related to integrating several modeling tools. An automated engineering toolchain has been proposed to perform early design and validation. Vogel-Heuser et al. [212] presented an approach to support the model-driven engineering of manufacturing systems. The SysML-AT language (SysML for automation) has been proposed to specify both functional and non-functional hardware components' requirements.

Chen et al. [59] presented an approach for automatically translating natural language descriptions into executable If-Then programs. Using neural networks, the system helps users synthesize If-Then programs by proactively predicting triggers and actions related to their descriptions. Dzulqornain et al. [81] also developed a real-time monitoring and controlling smart aquaculture system based on IFTTT and cloud integration. The system facilitates interoperability and integration of sensors, system controllers, client data visualizations, and system monitors.

Quirk et al. [167] presented an approach to map natural language descriptions with If-Then patterns to executable programs. They use semantic parser-learners that utilize already-defined recipe descriptions to train semantic parsers that automatically map these descriptions to executable programs.

Languages such as BPMN, which are general purposes business process languages, tend to be complex due to the vast number of related specifications and notations. Consequently, they sometimes lead to incorrect interpretations of its elements and semantics [66, 214]. Moreover, although graphical specification languages such as BPMN present a solid boundary to achieve defined operations, they tend to score down on flexibility, especially when implementing complex ideas that step out of the fixed boundaries [107]

Build tools such as Gradle require an adequate understanding of their documentation to get started. Moreover, they are not specifically conceived to run model management workflows, requiring extension mechanisms to support MDE artifacts and tools [219]. As a result, they can lead to tedious work that is abstracted by our DSL. If a task fails in Gradle, subsequent tasks that depend on the failed one are not executed [219]. We intend to implement self-

healing mechanisms within our DSL that does not necessarily halt the program's execution but report on the encountered problem to facilitate troubleshooting adequately. Gradle is a very mature build tool, and we intend to use it to implement a part of our workflow engine to facilitate the task execution process.

Alvarez et al. developed MTC Flow [9], a graphical DSL intended to design, develop, and deploy model transformation chains. However, their tools are limited to the Eclipse platform and support only model transformations and validations. In addition, their implementation does not address cloud-based solutions and service discovery features.

Modelflow [190] is a more advanced language towards reactive model management work-flows. Their implementation depends on events that can trigger a given workflow. Their execution engine can react to the modification of resources, and a graph-based execution plan strategy is provided to enable alternative execution paths. However, Modelflow does not involve advanced query mechanisms, service discovery, and other features such as model persistence to remote repositories or cloud deployment. Furthermore, Modelflow is based on the Epsilon language family, and features related to cloud-based solutions were out of scope.

MoScript [127] is based on the Eclipse platform, and the supported model management operations are not cloud-based. It does not support service discovery. Although it can perform model queries within the DSL, they are limited to OCL and directly tied to inner model properties.

Wires [178] is a graphical Eclipse-based tool supporting the orchestration of ATL model transformations. However, it does not support the cloud-based orchestration of model management services and their discovery as for the previously mentioned tools.

MMINT [77] is a tool assisting model management operations employing a graphical editor. It provides an interactive user interface, and the user can choose input models and feed them into a transformation, and the output can be used as input for subsequent transformations.

Moola [219] is a Groovy-based model operation orchestration language. It exhibits several features even though it does not support cloud-based solutions, such as cloud-based orchestration of services and advanced query mechanisms.

5.2.2   Comparison of model management composition approaches

This section compares the most recent tools addressing the problem of composing model management operations. None of the existing approaches implements the model-as-a-service (MaaS) [42] paradigm. Moreover, as shown in Table 5.1, existing approaches can be analyzed concerning the features described in the following.

Table 5.1 Comparison of service composition tools for model management operations.

Feature	MDEForgeWL	Moola	MTC Flow	Modelflow	MoScript	Wires	MMINT
Concrete Syntax							
Graphical			✓			✓	✓
Textual	✓	✓		✓	✓		
Target platform							
Cloud-based (Web integration)	✓						
Local infrastructures (Eclipse,...)	✓	✓	✓	✓	✓	✓	✓
Security Support							
In-built security patterns	✓						
Security pattern	✓						
Collaborative development support							
Artifact sharing capabilities	✓						
Sharing configuration	✓						
Reusability							
Code reuse	✓	✓			✓		
artifacts' reuse	✓	✓	✓	✓	✓	✓	
Scalability support							
Number of users	✓						
Data traffic	✓						
Data storage	✓						
Language features							
Data holder	✓	✓		✓	✓		
Condition	✓	✓			✓	✓	
Iteration	✓	✓				✓	✓
Syntactical & semantic features							
Auto-completion	✓						
Syntax highlighting	✓	✓	✓	✓	✓	✓	✓
Warning & Error markers	✓						
Service heterogeneity							
Model management	✓	✓	✓	✓	✓	✓	✓
Non MMSs	✓						
Service features							
Service-oriented (MaaS, SaaS, ...)	✓						
Service discovery	✓						
Service composition	✓	✓	✓	✓	✓	✓	✓
Cloud-based orchestration	✓						
Third party service integration	✓						
Program execution							
Sequential	✓	✓	✓	✓	✓	✓	✓
Parallel	✓	✓					
Alternative service execution	✓		✓				
Knowledge base							
Documentation	✓						
Query mechanisms	✓				✓		
Advanced features							
Advanced query mechanisms	✓						
Workflow pipelines' specification	✓						
Persistence support							
Cloud-based model repository	✓						
Local file system		✓	✓	✓	✓	✓	✓
Traceability							
Debugging means	✓						
Service call logs	✓						

- *Concrete syntax*: It refers to the language used to specify the composition of the considered model management tools. The language syntax can be textual or graphical.

- *Target platform*: It concerns the platform providing the functionalities of the considered approaches. With cloud-based deployment, the tool can be used on the web or through RESTful APIs. However, most analyzed tools are based on the Eclipse Modeling Framework (EMF). EMF is the leading open-source modeling framework, and it is no surprise that most tools use it. Several initiatives have been migrating that infrastructure to the cloud[6] and enabling features such as collaborative modeling.

- *Security support*: With this feature, we are interested in understanding how the analyzed tool manages the security layer, (e.g., employing security patterns like OAuth2.0).

- *Collaborative development support*: It concerns features that share developed artifacts or enable collaboration between different stakeholders.

- *Reusability*: It concerns available means to reuse already specified artifacts.

- *Scalability*: This feature is related to the architecture used during tool implementation. We evaluate if the system under analysis has some scalability support, e.g., concerning concurrently connected users, data traffic, and data storage.

- *Syntactical & semantic features*: These are features provided by the language server. Although there are several features in this context, we check if their DSLs support auto-completion, syntax/semantic highlighting, and warning and error markers.

- *Language features*: We refer to the availability of language features such as data holders, iterations, and conditional statements. These features are essential in controlling workflow specifications.

- *Service heterogeneity*: We aim to check if the analyzed approach can support services developed and available from different technologies.

- *Service features*: We want to check if the analyzed approach implements the MaaS paradigm. In particular, investigate if related operations are supported, such as service discovery, cloud-based orchestration, and third-party service integration.

- *Program execution*: We check if there is optional execution of the program using sequential or execution means. We also check if the user can specify the service to execute the wanted model management operations. For instance, she might prefer

---

[6]https://www.eclipse.org/emfcloud/

performing model transformations using ATL[7] rather than ETL[8] at run-time based on some service outcome.

- *Information point*: This feature concerns service and information discovery. Although one can query services based on their types to determine which one to use, the user can still discover services using the documentation with illustrative and straightforward demos.

- *Advanced features*: These are features facilitating the development of complex workflows, such as advanced query mechanisms and workflow pipelines specification.

- *Persistence support*: It concerns the technology employed to store developed specifications, which might be locally saved or pushed to a cloud-based repository.

- *Traceability*: Tracing events and problems that occur during the execution of a composite service is an essential feature. We check the availability of debugging means such as console view and the capability to gather the logs of the service calls.

Table 5.1 shows the result of the analysis we performed on the existing tools. In the next section, we present the proposed MDEForgeWL approach to support all the previously presented features.

5.3   The proposed MDEForgeWL platform

Figure 5.1 shows an overview of the proposed MDEForgeWL architecture designed to support the definition and execution of scalable workflows consisting of cloud-based compositions of model management services. The architecture presented in chapter 4 evolves from MDEForge [18]. In this chapter, we discuss the implementatin aspects regarding the composition, discovery and orchestration mechanisms of model managemetn services on the repository. These mechanisms are used to identify and execute model managment services' workflows. The architecture is organized into four tiers: the *front-end*, the *execution engine*, the *cluster of model management services*, and the *persistence layer*. The four tiers of the proposed architecture are described in detail below.

5.3.1   The MDEForgeWL front-end: Low-code development environment

As seen in Figure 5.2, the proposed environment provides a visual and intuitive way for users to create and automate workflows in cloud-based model repositories. With a user-friendly interface that uses drag-and-drop features and custom scripts, user-defined workflows can be created and automated in a domain-specific language (c.f.fig. 5.4). Custom scripting

---

[7]https://www.eclipse.org/atl/
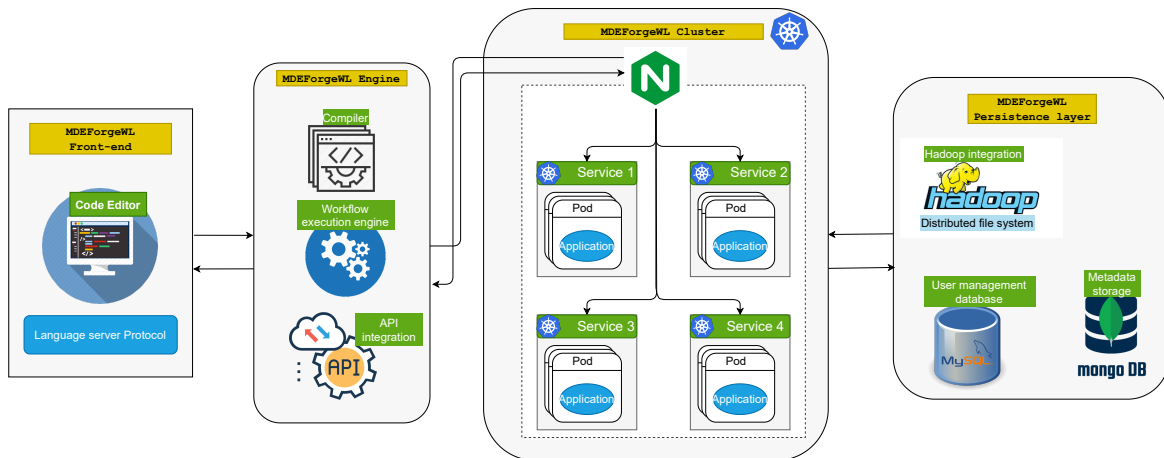[8]https://www.eclipse.org/epsilon/doc/etl/

Fig. 5.1 Overview of the MDEForgeWL architecture.

is enabled by an editor that allows the user to express complex workflow expressions programmatically. Services and extensions on the repositories are organized in decoupled and distributed microservices to emphasize separation of concerns and promote maintainability, scalability, and extensibility of individual services [202].

According to the explanatory workflow shown in fig. 5.2, the citizen developer might want to upload a PIMF (Performance Model Interchange Format) model [139] and generate a corresponding SySML model from it. He can then validate the model, compute special metrics, extract some metadata, and insert the obtained information into another SySML model. The obtained model can be stored in the repository and the user can be notified along with the complete execution logs. The services used in the above scenario are accessed remotely through APIs, and the storage systems are distributed services consisting of multiple network nodes.

Developing and executing model management workflows like the one shown in fig. 5.2 without proper support can be time and resource-consuming, laborious, hard to maintain, and error-prone. The proposed approach aims at enabling citizen developers to create and automate workflows based on selected model management services using a graphical environment with drag-and-drop capabilities. The proposed environment is based on the metamodel shown in fig. 5.3. According to the shown metamodel fragment, workflows consist of nodes, which are an abstract representation of activities referred to as actions and decisions. Several events can trigger activities, and the node can receive different types of inputs, such as model artifacts and variables. Events of interest and their sources are defined as shown in fig. 5.2, and they result from different providers that trigger specific actions as
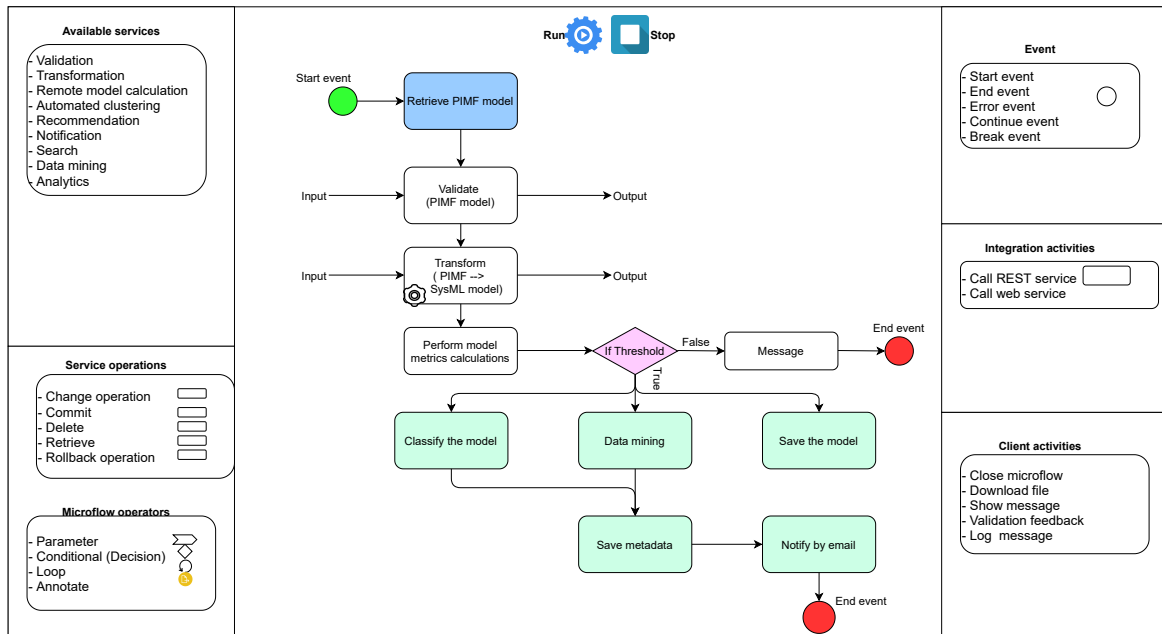
Fig. 5.2 Example mock-up of graphical task workflow environment.

instructed. Nodes represent decoupled and independent micro-services orchestrated when the specified workflow is executed.

Figure 5.4 shows a logical view of the graphical front-end and the corresponding stakeholders, notably two prominent actors involved, i.e., *citizen developers* and *software engineers*. The former can specify task workflows through the provided environment, whereas the latter can extend the repository services by adding new functionalities. The typical user (citizen developer) can access the repository, select services to automate, configure triggers and actions, and authorize task workflows. Interestingly, advanced support is provided to recommend modeling elements while editing workflows, and analyze, test, and deploy models by means of dedicated DevOps support. The workflow definition and analysis component provide such support. The service integration component ensures seamless integration of external and internal services. Once the modeled workflow is ready, the engine transforms and executes the incoming model (task workflow), as presented in the following sections.

5.3.2   The MDEForgeWL front-end: DSL

The MDEForgeWL DSL can perform the necessary low-level functionalities in a program, but also achieve complex functionalities with less user effort. To achieve this goal, the language is declarative: you specify what you want, and we deliver the results according to the given specification. As mentioned earlier, MDEForgeWL was developed in Xtext, and a fragment of the corresponding metamodel is shown in fig. 5.5. Each specification
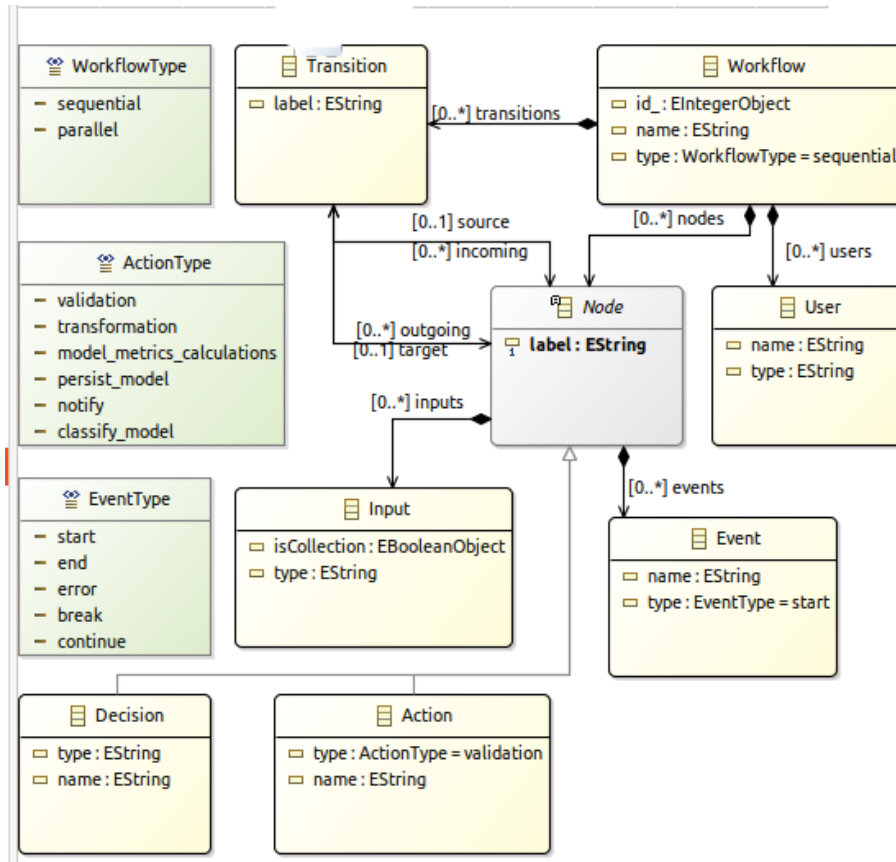
Fig. 5.3 Fragment of the proposed workflow metamodel (graphical environment).

has `WorkflowProgramModel` as its root model element. Each workflow consists of several elements, e.g. statements, workflow blocks, methods or functions. A statement contains features such as variables or other callable statements, including method invocations. Within
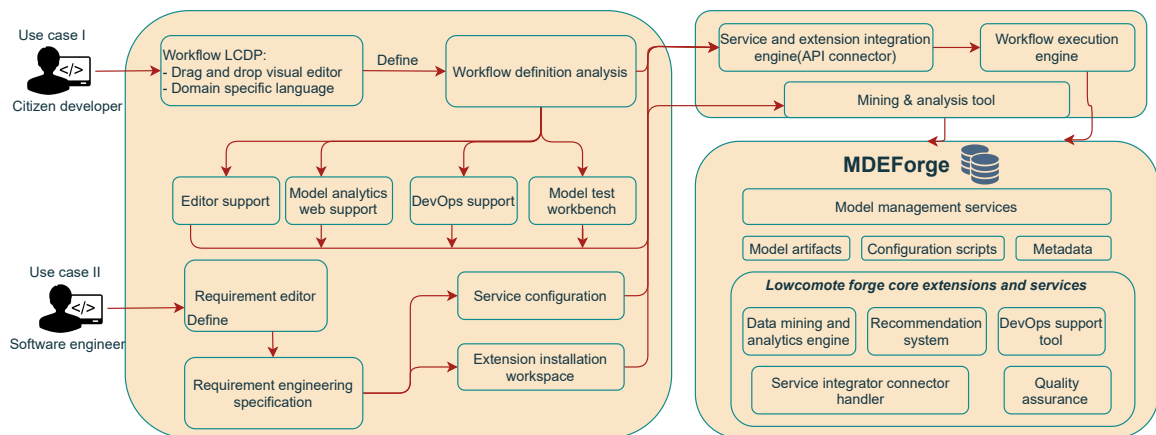

Fig. 5.4 Logical view describing the backend and frontend aspects of the system.

statements, we can use expressions, conditional statements, and loop statements that assist the control flow during the execution of the specification composition. A workflow block consists of steps that contain statements. The statement can also be a service, i.e., one of the model management services managed by the cluster. Furthermore, a statement can be a query to identify artifacts of interest.

In our language, services are an abstraction of model management operations (e.g., model transformations, validations, model queries, and model comparison operations). These are the functionalities that are grouped into containers and deployed individually in the cluster. Performing model management operations such as model transformations using traditional techniques can be complex, time-consuming, error-prone, and typically requires local installation of specific frameworks. Instead, our approach allows specific services to be invoked with appropriate arguments with no installation whatsoever, as shown below.

```
1 // We perform the transformation, the etl script is retrieved by id
2 call service _transfoModel(sourceModel, sourceMetamodel, targetMetamodel, id: 4)
```

Listing 5.1 Service call example

The argument can be a variable or the identifier of the artifact stored in the repository. As shown in Listing 5.2, the user can use advanced query mechanisms to search and find a metamodel based on several criteria, including parameters and properties set at the repository level. This feature is convenient when the users want to specify query predicates to find the artifacts satisfying given properties. In particular, as shown in line 2 of Listing 2 5.2, the language permits the declaration of variables that can be used in their defined scope throughout the script. For instance, the variable `sourceModel` can be queried using its `id`, `type`, and `extension` type. This is a query that returns a single result if successful. The returned results also carry other information, such as the execution status. The user can check if an executed query succeeded before executing subsequent commands.

```
1   //Create variables : This part is improved by advanced query mechanisms. You can query
         ↪ the type of models u want based on your defined criterias
2   var sourceModel = query artifact(id: 1, type: model, ext: xmi)
3   var sourceMetamodel = query artifact(
4           type: metamodel, ext: ecore, hasModel: sourceModel)
5
6   var targetMetamodel = query artifact(
7         user: "john", period: (03,2020 - 2021), hasAttribute: "person",
8         size: <500kb) -> retrieve( startsWith: "catalogue",
9         contains: "class book").first
10
11  var model1 = query artifact( name: "catalogue.xmi", conformsTo: "catalogues.ecore",
```
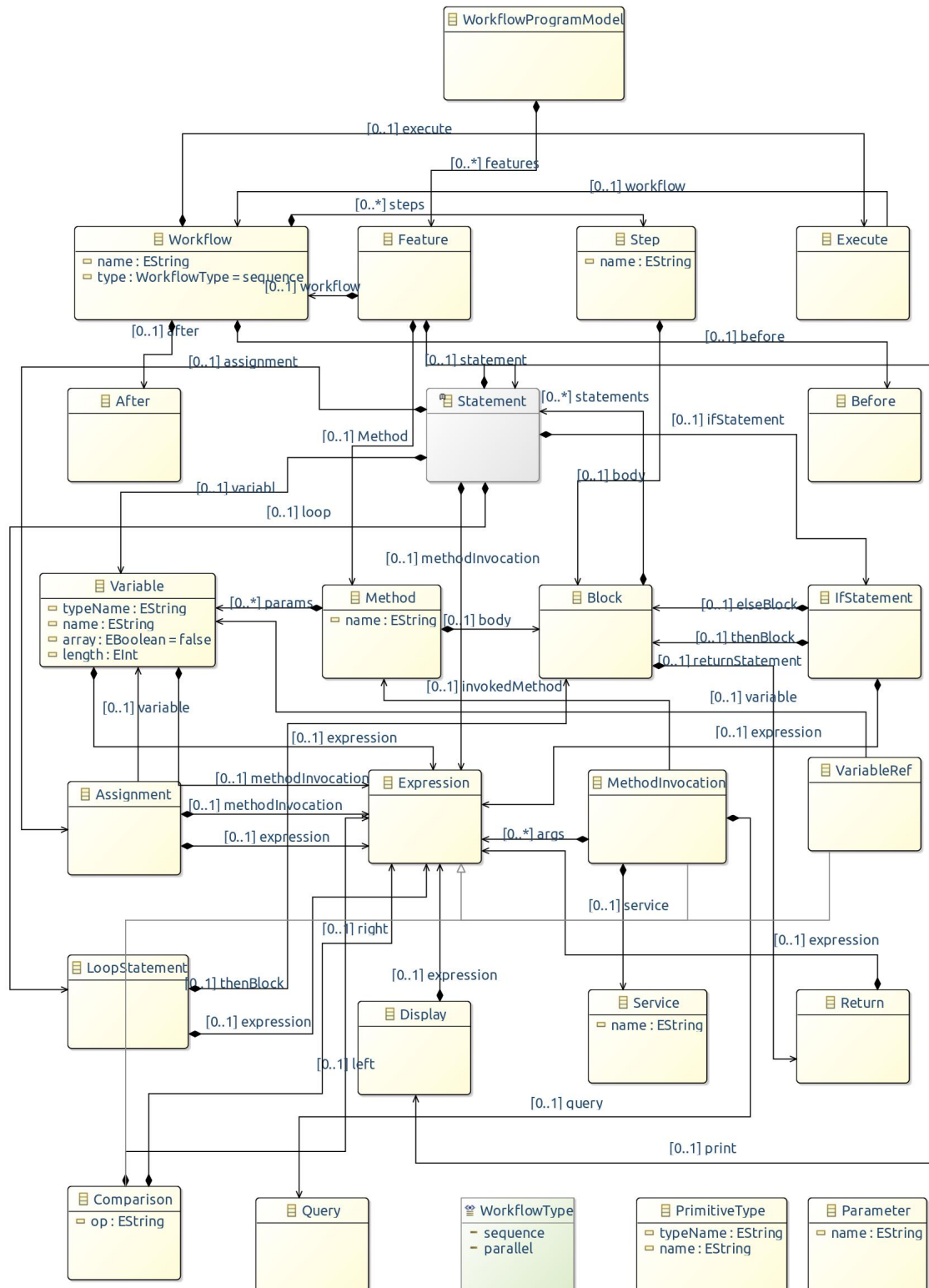
Fig. 5.5 Fragment of the MDEForgeWL metamodel.

```
12            sharedUsername: ["john"], sharedUserNumber: < 3 )
13    var emlscript = query artifact(id: "23")
14    var eclscript = query artifact(id: "44")
15    var eolscript = query artifact(id: "12")
16
17    Workflow workflow type:sequence{
18       step "Validate"{
19          // Let's validate our model with the retrieved ecl script
20          global var eventValid = call service _validateModel(
21                  sourceModel, sourceMetamodel, evlscript)
22          }
23       }
24       step "Compare Transform Merge Persist"{
25          // We will proceed if the validation passed
26          if(eventValid){
27             // We perform the transformation, the etl script is retrieved by id
28             var targetModel = call service _transfoModel(
29            sourceModel, sourceMetamodel, targetMetamodel, id: 4)
30             //If there is a matched trace, we can merge some model aspects
31             var matchedTrace = call service _compareModel(
32            model1, targetModel, eclscript)
33          if(matchedTrace){
34             // We merge the models, and persist the merged model and target model
35                var mergedModel = call service _mergeModels(
36                    model1, model2, eclscript, emlscript)
37                call persistArtifact(targetModel, mergedModel)
38          }
39          }
40       }
41    Post{  // We can notify the user of the outcome of the workflow
42        call notify(email: "johndoe@email.net", message: "message")
43    }
44
45    Execute workflow()
```

Listing 5.2 An illustrative MDEForgeWL specification

The query at line 3 concerns a metamodel with the exact model to which the previous query retrieved the model. The query in lines 6-9 is more complex and is used to search for an artifact with a specific user and persisted on the repository in the specified period. We query the model's content to find if it has a certain attribute and if its size is larger than 500kb. Let us suppose the query returns more than one result that meets the criteria specified. We can pipe the returned results and specify additional criteria, such as the artifact name starting

with a given text or the artifact containing a given text literally. On the returned collection, we can retrieve the first result. It is possible to retrieve models according to their name, the metamodel they conform to, the shared username, and the number of shared users (see line 11). Such properties and parameters are set on the repository level, and our DSL is aware of information regarding the retrieved artifacts. In addition, the user can query the services and choose which one to use based on its functionality, such as a transformation to be executed.

The information about service executions is displayed in a dedicated console. Control flow statements such as conditions and loops are also supported. However, these functionalities are deemed low-level; hence their use is discouraged since the user can still achieve the same results by piping results for further query processing. We currently support basic data types such as booleans, numbers, strings, conditional statements, loop constructs, and functions. We intend to support also data structures such as objects and arrays.

MDEForgeWL specifications can have a single workflow code block. You can specify the execution pattern, sequential or parallel (see the workflow definition on line 17). The user can declare dependencies within steps and among different steps. Step blocks enable pipeline and batch execution of code. For instance, the user might have a step where she wants to validate the model (see lines 18-23) before performing a model transformation or perform some model testing before the subsequent operations (see the step defined in lines 24-39).

In the second step of the explanatory workflow, we use conditional statements (see line 33) to match the traces from the model comparison operation before merging the models (see line 35). The resulting model from the model merging operation is persisted using one line of code (see line 37). The user can specify the models to be persisted by delimiting them with a comma. Underneath, the engine saves models and ensures their relationships with other artifacts, such as the metamodels they conform to. The user can specify `pre` and `post` code blocks for workflows (e.g., see line 41). In this instance, we chose to perform a non-model service regarding notifying the user about the results. Notifying the user using emails or other notification services is not the only way to reflect the progress status of the considered workflow; we can also use the console view to reflect exception logs captured by the platform. In addition, we intend to embed visualization capabilities to reflect logs about the program execution progress and eventual results in real-time to the user.

When the user cancels the workflow execution, the program preemptively forestalls the subsequent executions of the workflow and returns the current status. Although it is out of the scope of the present work, we plan to integrate abilities to pause and resume the execution of the workflows by persisting the current state in the context object and passing it to the

interpreter to resume the paused execution. The execution of the specified workflow has to be triggered by the `Execute` statement as in line 45.

5.3.3   The MDEForgeWL engine

Our engine comprises a compiler, a workflow execution engine, an API integration component, and the language server-side that supports the language editor. Our compiler, a sub-process, consumes the text from the code editor. Next, the lexer lexically analyzes the text, and the extracted tokens conform to the building blocks of our language, such as keywords and statements. Finally, the parser takes the list of incoming tokens and generates the abstract syntax tree (AST). The AST generated by Xtext is an EMF model, and the model is traversed using the EMF API. Once the AST is available from the incoming DSL text, a code generation process is triggered. Technically, code generation in Xtext traverses the AST and translates the tree into executable code that conforms to the language of your choice, in our case, JavaScript. Our language is statically typed, and the data structures we intend to support are arrays and nested objects.

At last, the compiler returns a valid executable code that the execution engine can run. The execution engine runs the provided executable code and uses the API integration component to leverage services provided by the MDEForgeWL cluster. For instance, when a user-defined workflow requires the definition of some available model management services, the engine triggers the orchestration of the involved services at the cluster. They get executed asynchronously (in parallel) or in sequence based on user preferences. Another sub-process, the language server, is also running in the engine behind the scene to provide server-side functionalities to the client code editor. It is important to remark that each service (e.g., a service exposing model transformation functionalities) can have several engines (e.g., ATL and ETL), and the user can choose which one should be used. With our discovery mechanism, the user can find out which engines are available. The system selects the right engine based on different criteria determined by the container orchestrator and API gateway. Moreover, the workflow engine is entangled with logging and monitoring mechanisms that keep track of the execution of workflows. For example, we keep track and visualize API calls using services such as Prometheus[9], Grafana[10] and Zipkin[11]. We have also implemented distributed logging mechanisms within the workflow engine to monitor the workflow executions' progress and facilitate troubleshooting.

---

[9]https://prometheus.io
[10]https://grafana.com
[11]https://zipkin.io

5.3.4 The MDEForgeWL cluster

This DSL can be used as a plugin in the Eclipse platform, but our endeavors aim to migrate model-driven development infrastructures from the environment to the cloud. In this aspect, we can ensure our modeling infrastructures are more scalable and extensible than in traditional modeling. The cluster is built using Kubernetes, an open-source container-orchestration platform [12]. We use it to automate deployment, scale, and manage our containerized model management services into logical units that facilitate their discovery. The Kubernetes cluster offers several features: service discovery and load balancing, self-healing, horizontal scaling, automatic bin packing, storage orchestration, secret and configuration management, and batch execution. In addition, the Kubernetes cluster is designed to be extensible and loose-coupled to facilitate feature updates without hardcore changes to mainstream code-base and architecture [12]. We rely on the Kubernetes ingress controller to accept and load balance the traffic to the microservices. It also manages egress traffic, representing communications from internal to external services out of the cluster. In addition, the ingress controller monitors running pods within the cluster and automatically updates load-balancing rules regarding removed or added services.

The adoption of containerization technology to build cloud-native microservices accelerates the development process. Containers are inherently portable and are built to ensure adequate isolation and efficiency of resources [12]. Furthermore, self-healing mechanisms enable containers to be advertised when they are ready to serve and can be killed, restarted, replaced, or rescheduled to conform to the health check defined by the user. Containers are scaled based on CPU usage to balance the application workload. This is enabled by assigning a single DNS name for a set of pods referred to as a service; thus, all communications are made through the service, and the service load-balance the workload among the bootstrapped pods [12]. Since the MDEForgeWL cluster is deployed using Google Kubernetes Engine[12], the system administrator is allowed to set any resource limits, e.g., on storage, CPU, and memory usage. Kubernetes auto-scales resources based on available maximum and minimum ones or replicas set by the administrator. It has built-in vertical, horizontal, and cluster auto scalers. Auto scalers dynamically adjust the number of running pods or replicas and manage the CPU/memory utilization of machines in the cluster. They can also increase or decrease the number of nodes to meet current usage, desired target, and user demands [132].

Our model management services are self-contained and robust, thanks to their organization in a distributed microservice architecture. This means they offer strong security, flexibility, and scalability while being fault-tolerant, extensible, and loosely coupled. These microservices

---

[12]https://cloud.google.com/kubernetes-engine

are referred to as a *resource server*. Moreover, other services such as automated clustering of model artifacts, search engine integration, and model metrics calculator are integrated at this level. We use the Nginx ingress controller to access our cluster to interact with underneath microservices.

Our orchestration and discovery approach uses current trending containerization and orchestration technologies that automate the manual work related to service discovery activities. Existing discovery approaches mainly rely on WSDL documents [217]. In particular, typically, clients are expected to read and process WSDL files to determine the services exposed by the server of interest. Then, to call the services listed in the analyzed WSDL file, the user employs SOAP over transfer protocols like HTTP [146].

The proposed microservice architecture also includes a service registry, a service API gateway, authorization & authentication server (it implements the OAuth2.0 protocol[13]) and a resource server as shown in fig. 5.6. When a user accesses the web browser via a service endpoint published by the Nginx ingress controller to request the resource server, it goes through the service API gateway. The service API gateway cross-checks the credentials to validate the user authentication. If the user is not authenticated, the service API gateway redirects her to the authorization & authentication server. The server asks the user to authenticate and issues an access token which enables her to access the resource server. The resource server ensures the access token is valid from the authorization server, and then it is set to execute the request. All resource servers implement a client discovery feature to publish their service. The service registry server keeps an open connection to discover and register all self-published services from the resource servers. The service API gateway fetches all available services from the service registry and acts as a proxy server to the resource server. Briefly, registering new services with MDEForgeWL is done by implementing a client discovery that publishes the implemented service. Our engine, by using the service registry server, will discover and register it in our registry. Once the service is registered, it can be used by our API gateway as MDEForgeWL services. Extending services in this manner does not require modifications of the grammar of the DSL.

5.3.5   The MDEForgeWL persistence layer

The persistence layer of the proposed system is divided into three categories. The first category stores structured data using SQL databases such as user management services or other sensitive data. The second part stores in NoSQL databases unstructured data (such as logs or data mined by data mining services from the MDEForgeWL cluster). The last part persists artifacts such as models, metamodels, and transformations. MDEForge, our
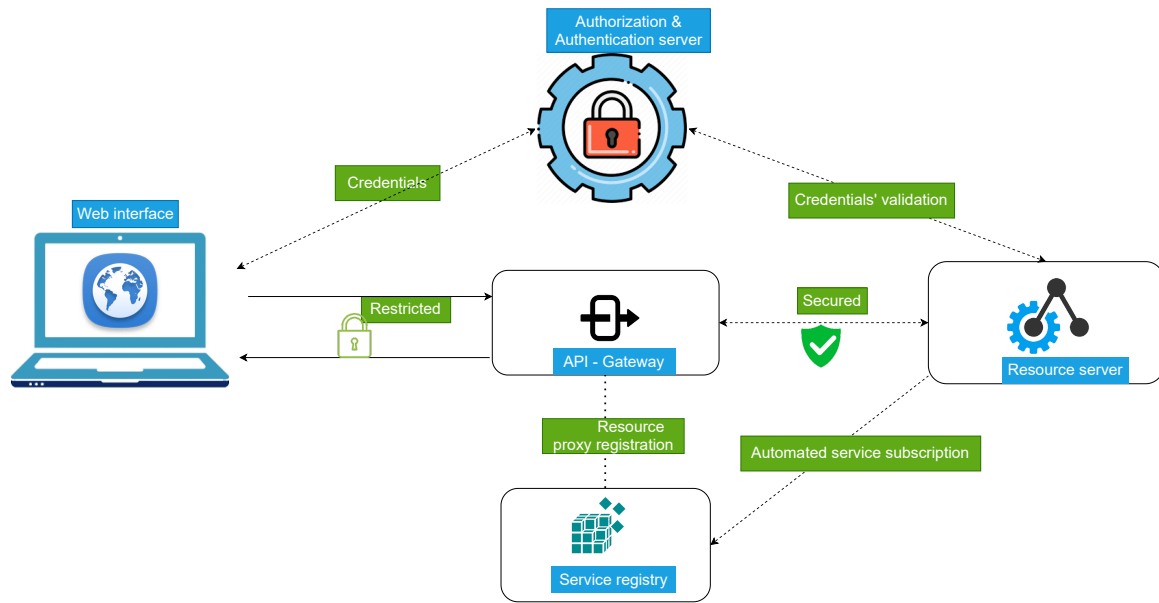
---

[13]https://oauth.net/2/

Fig. 5.6 Detailed view of the MDEForgeWL cluster.

cloud-based model repository, consists of model management services that allow persistence and management of typical model artifacts and tools. Services are accessed and used through RESTful Web APIs [18]. Our repository is built to handle big data with features such as high velocity, volume, and variety and perform analytics and predictions on stored data. A Hadoop cluster is the most pragmatic way to manage big data, break down big problems into smaller elements and enable practical analysis and predictions on the stored data. As in the case of Kubernetes, the Hadoop cluster is self-healing and supports dynamic addition and removal of servers from the cluster [122].

5.4 Conclusion

In this chapter, we introduced MDEForgeWL, a novel approach to support the development of complex model management operations. This platform aims to support the discovery and composition of model management operations that can be consumed as remote services. The goal of the proposed approach is to promote the use of model management service orchestrations and provide modelers with access to a variety of composite services that satisfy specific requirements. A prototype implementation of a workflow DSL MDEForgeWL has been presented along with its architecture. In particular, a low-code development platform similar to the functionalities of currently available LCDPs such as IFTTT and Zapier is presented. Such platforms enable the development of complex processes by integrating and executing different services. The proposed approach envisions a microservice-based

architecture for the integration and execution of model management services orchestrated in the cloud according to user specifications using a BPMN-like modeling language.

The proposed approach aims to overcome the current challenges faced by traditional modeling environments that rely heavily on locally downloaded resources. Such environments are limited in their scalability and extensibility, and their services exhibit strong coupling with the local environment. In the next chapter, we present advanced discovery mechanisms that leverage the architecture and implementation described in this chapter.

Chapter 6

Advanced discovery mechanisms in model repositories

In recent decades, MDE has become increasingly popular due to its advocacy for abstraction, automation, and reuse of artifacts, which significantly impact productivity and quality [42, 180, 16, 14]. Several initiatives in MDE have proposed a variety of technologies and facilities to simplify and automate MDE processes [74, 152]. However, empirical studies still point out barriers that hinder the broader adoption of MDE practices and processes. However, despite significant efforts and progress in this field, achieving efficient advanced discovery and reuse mechanisms to support model discovery, accessibility, retrieval, and reusability is still a challenge [16, 18]. As a result, model artifacts and tools are developed from scratch reinventing the wheel, inflicting unnecessary upfront investments and compromising the productivity benefits of MDE-based processes [205, 186, 205, 221].

In the realm of model-driven engineering (MDE), modelers require techniques and tools to effectively capitalize on existing modeling artifacts that can contribute to the solution at hand. The practices of *discovery* and *reuse* play integral roles in this process. In MDE, *discovery* involves identifying pertinent models, model elements, or transformations already in existence that hold relevance to a specific problem or task. By leveraging existing resources, the objective of discovery is to optimize time, effort, and overall efficiency in model-driven development processes. On the other hand, *reuse* entails utilizing discovered models, model elements, or transformations in the creation of new systems or the evolution of existing ones.

Reusability can take place either locally, where users set up the modeling environment and handle all necessary software dependencies, or through cloud-based model operation executions. In the latter, users interact with a cloud-based modeling platform, providing it with the modeling artifacts to be manipulated remotely, without the need to install the model-management platform locally. This practice enables modelers to effectively draw

upon and adapt discovered artifacts, thereby enhancing efficiency, consistency, and quality throughout the development and maintenance of systems within MDE.

Discovery and reuse require efficient mechanisms at different granular dimensions of modeling artifacts. In addition, the need for efficient persistence and retrieval of artifacts is primal in this quest [18]. In this context, model repositories have been conceived to tackle the issues related to the discovery and reuse of modeling artifacts and tools [20]. Because MDE can be used at various stages of software development life-cycle, developed artifacts can be reused and maintained during development employing a model repository. Hence, model repositories preserve the artifacts and enable their discovery, retrieval, and reuse. Furthermore, MDE relies on model repositories to help collaborative modeling activities [74, 14].

Ideally, discovery techniques should start from basic queries and gradually get complex to filter out the exact artifacts persisted in a distributed environment [200]. In addition, filters should be allowed to model artifact characteristics, such as structural features or other internal aspects that make up the artifacts. It should also consider mega-models, their contextual relationships and the persistence environment of the model [103, 93]. Finally, discovery facilities should facilitate reuse by availing model management operations (MMOs). In this manner, retrieved artifacts can immediately participate in model management activities without setting up new environments.

This chapter presents MDEForge-Search, a novel approach to discovering and reusing model artifacts stored in cloud-based model repositories. Users can exploit pre-defined operators to specify different requirements of the wanted artifacts. Query specifications range from simply retrieving data based on metadata to more detailed and complex information, including quality attributes and relationships within the ecosystem under consideration [116]. For instance, the user can ask the system to search models *i)* that conform to a specific metamodel, *ii)* which can be transformed towards a given target metamodel through direct or transformation chains available in the repository, and *iii)* characterized by a complexity lower than a given threshold. The user will be provided with efficient discovery mechanisms that are *(i)* generic, *(ii)* sensitive to model structure and relationship with other artifacts, *(iii)* based on a dedicated query formulation process, *(iv)* independent from the underlying technologies and data models, and *(v)* able to rank and return the collections of most relevant artifacts that meet specified predicates. In addition, retrieved artifacts can be reused on the same platform along with the deployed MMOs.

The main contributions of this chapter are the following:

- Provide an overview of existing mechanisms to discover model artifacts in model repositories;

- Identify significant challenges to achieving the practical discovery of model artifacts;

- Design key features that enhance discovery in cloud-based model repositories;

- Present a novel approach to perform advanced search queries using a micro-syntax for query specifications over cloud infrastructures.

This chapter is structured as follows: Section 6.1 describes the background of modeling and the concept of model persistence in cloud-based infrastructures. Additionally, we introduce the concepts of discovery and reuse regarding MDE artifacts. Finally, we conclude this section with a motivating example. Section 6.2 introduces the state of the art of discovery tools in MDE and identifies the main limitations and challenges of currently available model discovery tools. Section 6.3 presents MDEForge-Search, the proposed approach providing advanced model discovery features as presented in Section 6.4. Section 6.5 presents a concrete integration of MDEForge-Search with an existing platform aiming at facilitating the development of recommender systems. In Section 6.6, potential threats to validity are discussed. Section 6.7 concludes the chapter and presents future works of MDEForge-Search.

6.1 Background

Modern modeling techniques in data-intensive applications require the discovery and reuse of relevant model artifacts [200]. In this context, there are two phases in the reuse process of model artifacts (c.f. fig. 6.1). The first phase operates on the high-level of the model repository and involves the discovery of relevant artifacts based on a megamodel-aware predicate-oriented approach [50]. MDE offers the concept of a megamodel as a building block for large-scale modeling [207, 50, 52]. A megamodel is used to create and leverage global relationships and metadata on basic macroscopic entities as models and metamodels [207, 50, 52, 80]. A megamodel conceals fine-grained details that impede comprehension of the system in its global perspective [207, 80]. The global picture includes consideration of system architecture, interactions between model artifacts, relationships between artifacts, results of model transformations, etc [80].

Retrieved artifacts are then manipulated through different operations including, merging, comparison, queries and transformations using Model Management Operations (MMOs).

The following aspects characterize modern model repositories:

- Huge number of large inter-related heterogeneous model artifacts of models, transformations, data files, source code, file descriptors;
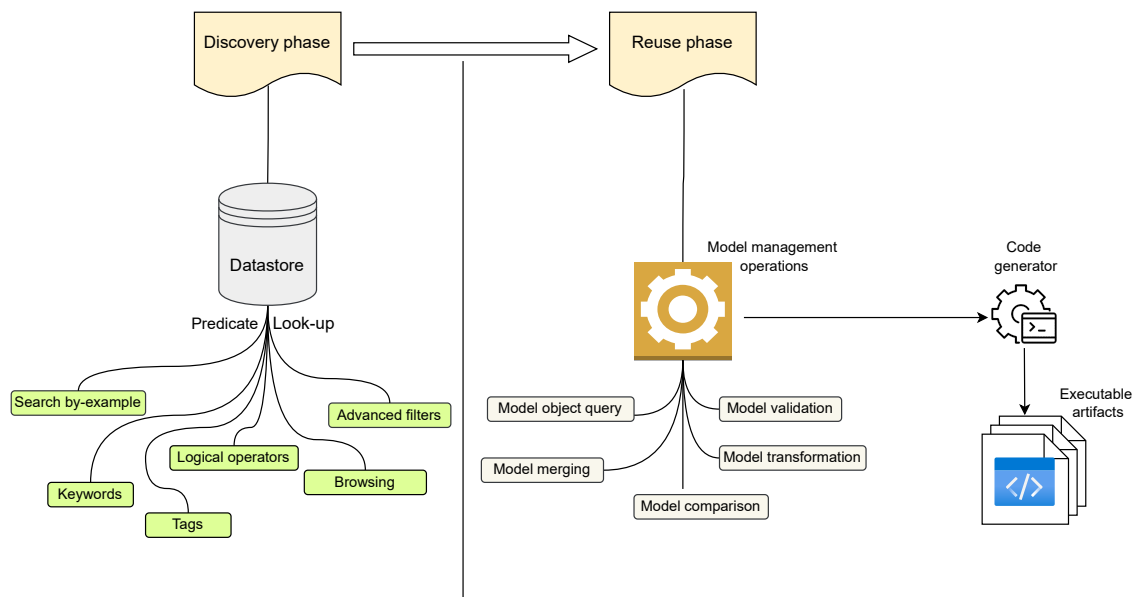
Fig. 6.1 Discovery and reuse process of model artifacts.

- Stakeholders, developers, and business analysts who contribute to the development of the system and hence to the evolution of produced artifacts on the repository;

- Heterogeneous model management tools that carry out MMOs include model transformation, model object query, model validation, model merging, and model comparison.

When the user is at repositories' premises, the discovery processes should facilitate her, for instance, to explore *internal structure* of the artifacts, such as metamodels containing elements of specific types. Also, the user can compute a certain number of elements with a specific size, e.g., bigger than a given threshold. In addition, the user should be able to specify queries specifying the artifacts' *relations with other elements in the ecosystem.* Retrieved artifacts can be reused immediately on the same platform. For instance, the user can specify queries like (in natural language):

I would like to get all the models that:

1. *conform to* metamodel MM_i

2. *contains* elements named $name_1, \ldots, name_n$

3. *can be transformed to* target models conforming to metamodel MM_j by means of single or chained transformations

4. its *quality attribute qa* is valued greater than t
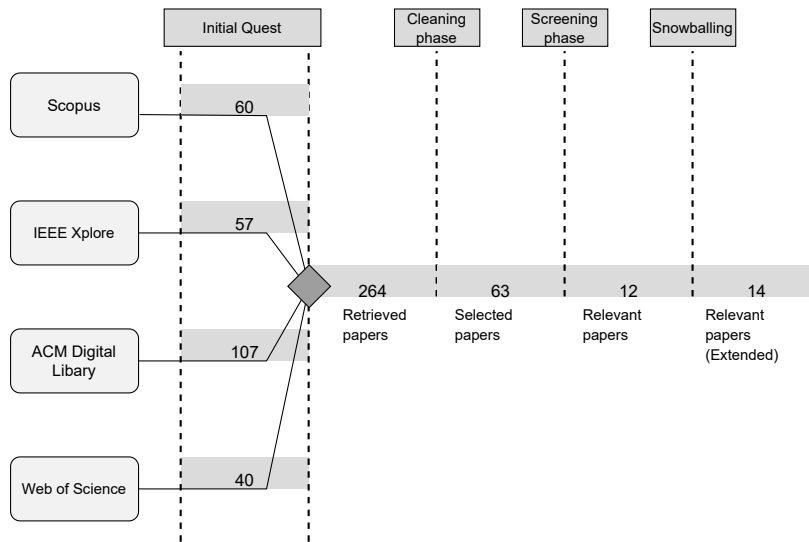
Fig. 6.2 Publication selection process.

Existing technologies permit the specification of queries that predicate the content of the wanted artifacts (2). However, they do not support the specification in a homogeneous manner of *direct relationships* (1), *indirect relationships* (3), or *quality characteristics* that should be satisfied by the wanted elements (4). Moreover, queries like this should be technologically agnostic. Unfortunately, most of the current solutions in this area are still platform-dependent [48, 113]. Furthermore, query mechanisms should be easy-to-use and intuitive but with powerful features such as supporting search tags, keywords and conditional statements. Unfortunately, most existing technologies provide tools that allow query specification at low-level granularity, requiring a high learning curve and technological expertise [113, 48].

6.2   Overview of existing approaches

In this section, we discuss the investigation that has been performed to identify currently available discovery and reuse tools in the field of MDE practices. We have highlighted salient features that advanced discovery tools can provide to support the efficient discovery of model artifacts (c.f. Section 6.2.1). We will also discuss existing tools and approaches in light of the selected features (c.f. Section 6.2.1).

6.2.1   Methodology and scope

This study aims to understand the current state of discovery tools in the MDE environment. We identify the rationale behind their mechanisms and characteristics. The following research questions guided this investigation:

- **RQ1:** *Given the current digital revolution, what features should be supported by existing tools compared to those provided by traditional discovery and reuse mechanisms?*

- **RQ2:** *What are the gaps/challenges in discovery tools that hinder efficient discovery and reuse of model artifacts?*

We performed a systematic investigation to identify the main challenges of artifact discovery on model repositories. Following accepted guidelines [164, 222], we conducted a search, selection, and mapping process. We screened and identified relevant published literature as shown in fig. 6.2 and Table 6.1.

Table 6.1 Database results' table

Database	Results
Scopus (Elsevier)	60
IEEE Xplore	57
ACM library	107
Web of Science	40
Snowballing	2
Total	**266**

Phase 1. *Initial quest*: We formulated a query string executed on Scopus[1], IEEE Xplore[2], the ACM Digital Library[3] and Web of Science[4] (c.f. Table 6.1). Although each database has its query string specifications and search fields, we tried to find publications that contained keywords from each column of Table 6.2. These keywords should be found directly in the title, abstract, or keywords.

We managed to retrieve 264 documents: ACM Digital Library[3] responded to the query with the highest number of 40.2% of the total documents retrieved, as shown in fig. 6.3. Followed by Scopus (Elsevier)[1] providing 22.6% of the total documents, IEEE Xplore[2] with 21.4% and Web of Science[4] with a 15% of the total retrieved documents.

Phase 2. *Cleaning phase*: In the cleanup phase, we removed documents that were not in English or were not directly related to model-driven engineering. In this phase, we also merged and removed duplicates from all the documents we found. After the cleaning phase, we got 63 papers.

Phase 3. *Screening phase*: In this phase, we looked for papers that directly incorporated discovery, query, retrieval, and search techniques in their abstracts. We also excluded articles

---

[1]https://www.scopus.com/home.uri
[2]https://ieeexplore.ieee.org/Xplore/home.jsp
[3]https://dl.acm.org/
[4]https://clarivate.com/webofsciencegroup/solutions/web-of-science/

that were either beyond the scope or irrelevant to this study. Finally, we carefully read the papers in their entirety and only considered documents that could show an implementation of their discovery tool. In the end, 12 articles were selected. The tools are further analyzed and compared in Table 2.1 and Table 2.2.

Phase 4. *Snowballing*: We performed a manual search using ad-hoc methods to find articles that were not found by our query. As a result, we found two articles that were not covered by the formulated queries.

We ended up with the following 14 papers: [141], [144], [200], [117], [104], [22], [142], [127], [15], [95], [33], [108], [34], [130]. In the next section we reviewed the approaches proposed in the selected papers.

Table 6.2 Terms used in the formal search query. Selected articles must contain at least one term from each column in the title, abstract, or keywords.

Modelling / MDE	Search / Query / Discovery
model-driven	search mechanism(s)
model-driven engineering	search technique(s)
model-based software engineering	query technique(s)
model-driven development	query mechanism(s)
model-driven architecture	advanced search
collaborative modeling	advanced query
cloud-based model repository	model finding
model repository	model discovery
	model search
	query by example
	artifact discovery
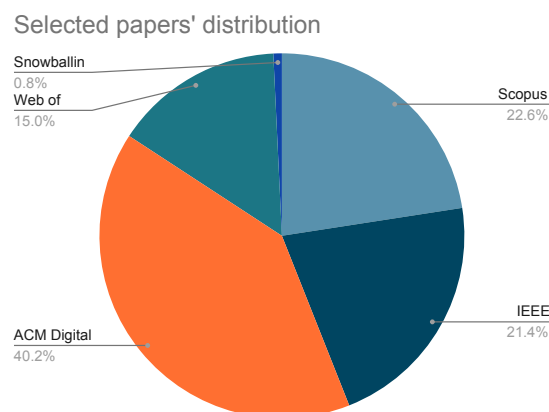
Fig. 6.3 Distribution of queried papers.

Results

MAR [141] provides a generic search engine for heterogeneous model artifacts. Their approach considers the model's structure to enable query-by-examples. They also support keyword-based search (c.f. Section 6.2.1). It uses HBase[5] to create an index that is later used to enable fast query response. Recent updates have integrated the Lucene search engine[12] to support indexing and keyword-based search [144]. In MAR, natural language processing filters out irrelevant paths between model elements. The paths encode the model structure and are stored in the inverted index. Indexed artifacts include ecore metamodels, UML and BPMN diagrams that conform to the EMF metamodel. The results are ranked using a custom scorer algorithm. MAR can be accessed through REST API via a web interface and as an Eclipse plugin.

IncQuery [200, 117] is a proprietary distributed query framework that enables scalable model queries. It is deployed on cloud infrastructures to ensure its scalability and uses incremental graph search techniques. In addition to handling large models and model queries, it has a dedicated indexer and query processor responsible for retrieving model artifacts. The reason for its development was the lack of SQL-like queries, which are not supported by NoSQL databases. Queries are based on a domain-specific language (DSL) called the Viatra query DSL [104]. However, since IncQuery is proprietary, it is not easy to compare its features with current solutions. Moreover, understanding the DSL used when performing queries requires a significant learning curve. It depends on the Eclipse Modeling Framework (EMF) ecosystem, and the reuse capabilities within the framework are not mentioned. Currently, the framework supports scalable queries over collaborative model repositories using the VIATRA query engine [104].

Basciani et al. [22] presents a cloud model search with search tags that enables automatic exploration of model repositories. The tags allow modellers to find relevant artifacts without needing low-level detail expertise. This facilitates the management and reuse of related model artifacts and eliminates error-prone and time-consuming previously necessary processes. Their platform-agnostic approach remains uncluttered even as searches become more complex. It integrates a search engine based on Lucene. However, it could not formulate advanced search mechanisms to enable complex searches. As a result, reusability and exploration are limited, but it was enough to trigger our current research. The tool was used with a dataset of 2,422 metamodels, 350 models, and 115 transformations.

---

[5]https://hbase.apache.org/

Moogle [142] is a model search engine that uses metamodeling information to build richer search indexes and enable complex queries over model artifacts. Moogle improves the general-purpose search engine that performs artifact queries based on a text-based search, ignoring the internal structure of the model. It also presents results in a human-readable format. It is based on Apache Solr[6]. It has EMF parsers that read and parse EMF model artifacts and index them later. Moogle can index different model artifacts as long as a metamodel is provided. Within Moogle, the user can use logical operators to filter results. The indexing processes are much slower, but the evaluation looks promising if a good set of model artifacts is available. Moogle does not support the reusability of retrieved artifacts on the same platform. Moreover, the model artifacts are automatically indexed and kept on local storage, which could be extended to cloud storage.

Moscript [127] is a query DSL designed to handle platform-dependent ad-hoc approaches. It supports complex queries based on internal structures and relationships to other artifacts. It also allows the manipulation of retrieved results. Services can be invoked for model artifacts and written back to the repository. Moscript aims to provide a homogeneous model-based interface to heterogeneous models, considering the concept of mega-modeling. Moscript queries are based on OCL, introducing a significant learning curve for citizen developers, and environments are deployed locally. Since it is a workflow automation tool, it supports the reuse of model artifacts.

Hawk [15] is a modular and scalable model indexing framework designed to enable efficient queries over extensive collections of model artifacts. Model artifacts are fragmented to facilitate their transmission over the network. Hawk is primarily designed to perform global queries on artifacts stored in file-based version control systems. It has component parsers that take input model artifacts (e.g. Ecore) and generate EMF resources. The tool takes the EMF resource and file version number as input and persists them into the index. They have a query API that connects the Hawk index and model management tools that query the index. Although Hawk handles large model artifacts better, it is not designed to facilitate the discovery and reuse of model artifacts. Therefore, it does not directly integrate model management operations that reuse retrieved artifacts. Moreover, the query DSL requires expertise to explore the tool efficiently [16].

In [95], the authors use a keyword-based ontology called WordNet[7] to search and retrieve relevant artifacts using cognitive approaches. The proposed environment enables the retrieval of UML models by combining WordNet and Case-Based Reasoning. The search relies on

---

[6]https://solr.apache.org/
[7]https://wordnet.princeton.edu/

similarity metrics to ensure that relationships between UML elements are considered. The use of ontology in their approach allows for simple matching using synonyms. However, the tool may not scale well when faced with large model artifacts.

Bislimovska et al. proposed MultiModGraph [33], an approach for indexing and searching model repositories. MultiModGraph uses graphs to obtain metamodel information and model structure. The efficient model search is enabled by the approximate mapping of model graph vertices to points in space. The points are used to build and search the index, restricting the search to similar vertices that correspond to the queried vertices. In addition, their approach allows for pruning, which makes the search efficient. However, their methods lack reuse capabilities and are not scalable for large model artifacts.

MORSE [108] is an environment that manages model-driven development (MDD) projects and artifacts in a repository and a model-aware services that enable queries on persisted model artifacts. It is designed to facilitate reflective artifact lookup in a service-oriented application. It dynamically uses information generated from models at runtime to reflect models after they have been created and deployed. These models are then persisted in the repository using Universal Unique Identifier (UUID) to facilitate model identification. Their query is based on a Java implementation for which they are developing an API. However, their implementation does not support efficient search because the artifacts are not indexed and using Java for search is error-prone and requires programming knowledge. The approach's scalability and extensibility are other drawbacks of their implementation.

Bislimovska et al. [34] proposed a content-based query approach (query-by-example) that retrieves model artifacts or model fragments from UML repositories. Their approach involves analyzing and indexing the textual content of the artifacts. They used segmentation granularities and an indexing strategy to allow different configurations of the search engine. The former divides the model into parts that are searched and returned in response to a user query. The latter indexes the content in a search engine index. Their implementation relies on the SMILA framework for content analysis and Apache Solr as the search engine. Their approach lacks reusability and is deployed locally, which poses problems regarding its scalability and extensibility.

Kotopoulos at al. [130] developed QML, a metamodel query language that uses OCL to query UML model repositories. QML is used to formulate fuzzy queries using a Boolean model approach. QML aims to provide efficient mechanisms for accessing metamodels available in the four layers of OMG's Meta Object Facility architecture. The query mechanisms provide

generic access to all types of knowledge (e.g. models, metamodels). This tool is designed to consider semantic information that relational databases cannot handle.

Designed features for productive discovery mechanisms

To answer research question ***RQ1***, we conducted a comprehensive investigation to identify the crucial features that enhance model artifact and tool discovery. Our investigation included a review of 15 existing model discovery tools, and we present a comparative analysis of these tools in Tables 2.1 and 2.2. Our aim is to foster the comparison of the selected tools based on the identified features. In subsequent sections, we explore each identified feature in-depth, elucidating their roles, significance, and necessity in addressing the current limitations within the context of a discovery facility. Features that were deemed important and identified as part of the future work for one or more of the analyzed systems have been incorporated, as they were not supported by any of the analyzed tools.

Through our systematic examination of existing discovery mechanisms and tools, it became evident that limited support was available for crucial features in the analyzed systems. These features include megamodel relations, model-as-a-service, quality assessment support, public standardized API specifications, and scalability features such as service containerization, orchestration, and cloud-based deployment of a service cluster. Furthermore, we observed a lack of adequate attention given to the reusability of artifacts, APIs, and infrastructures within the current systems.

Management of megamodel relations: Megamodeling [30] provides a way to define different types of relationships between model artifacts. The model elements of a megamodel are artifacts such as models, metamodels, and transformations. A megamodel also contains (typed) relationships between artifacts, for example, conformance and transformation. Thus, megamodeling offers the possibility to specify relationships between artifacts and to navigate between them. We expect that the features mentioned below under megamodeling come into play in the discovery of relevant artifacts.

▷*Model conformance*: Model conformance is the relationship between the model and the metamodel with which it conforms. This relationship allows the metamodel to be reused in the creation of subsequent models. Since metamodels can be extended while maintaining the original conformance, this allows the metamodel to be extended and thus new models to be created based on the same metamodel. Most of the tools examined exhibit this feature as shown in Table 2.1 and Table 2.2.

▷ *Transformation conformance*: This feature allows logging mechanisms on the repository to record traces of the execution of transformations, allowing discovery based on the logs of MMO operations. The artifacts involved are recorded along with other metadata about execution status, executors, timestamps, etc. Recording such traces is essential for

creating a transformation dependency graph that can later be used to draw inferences about transformation chain discovery. Unfortunately, the literature found do not reflect this feature.

▷ *User relationships*: To facilitate search, the data store can be designed to support a user-centric architecture. This architecture seamlessly engages users and allows them to configure access permissions and collaboration settings. Some tools built on repositories exhibit this feature, such as MDEForge [18, 116]

Model-as-a-service (MaaS): This cloud computing model enables the provision of model management operations in the form of services [183]. These services can be invoked on persisted artifacts and executed on demand. MaaS is very important to make model management services scalable and reusable within the ecosystem and externally. Enabling MaaS on a discovery platform is important to populate the index with heterogeneous artifacts from different sources and improve reusability.

▷ *Model Management Services (MMS)*: This feature is intended for tools that use a data store along with MMSs. This scheme facilitates the reuse of housed artifacts on the same platform as the discovery mechanisms. Some of the MMS can help populate the index by extracting derived knowledge such as quality metrics from artifacts.

▷ *CRUD operations*: This feature allows users to populate the data store and thus the index. Thus, this feature enables persistence mechanisms that allow users to create, modify, remove, or review artifacts.

▷ *Service execution traces*: Services such as MMS leave traces that should be logged along with the parties involved such as model artifacts and tools. These logs can be used to retrieve artifacts based on the activities they were involved in.

Model heterogeneity support: This feature reflects the support of different types of model artifacts and tools by the discovery mechanisms. In addition, the discovery tool should support model artifacts developed using different technologies.

▷ *Technological independence*: This feature is intended for tools that have a generic discovery approach. Such approaches allow users to retrieve artifacts independent of the underlying technology or data models. Model crawlers and extractors may be domain-specific, but generic metadata should be generalized across persisted model artifacts.

▷ *Artifact type*: MDE frameworks such as EMF have different types of model artifacts. Filtering these artifacts by their type, e.g., models, metamodels, or scripts, is essential for

discovery mechanisms. Moreover, their relationships with each other are the basis of the megamodel, which is also important for the discovery mechanisms of model artifacts.

Access interface: Access options that allow interaction between system and users. These interfaces frame the manner and policy of data access.

▷ *Cloud-based facility*: This refers to tools with a user interface for cloud access, such as a web, desktop, or mobile interface that retrieves data in the form of a remote API. While the developer chooses the user of their tool, it is important to opt for facilities (e.g., web facilities) that involve citizen developers in the discovery process of model artifacts.

▷ *Local-based facility*: This feature refers to model discovery tools used in a local environment. Normally, such tools must be set up in a dedicated environment. Discovery in this type of environment is limited to the memory capacity of the local computers. Moreover, the services associated with this type of discovery mechanisms require intensive resources, making the use of such environments impractical.

▷ *API-based facility*: The discovery tool provides an API that allows you to search and discover model artifacts. Typically, these are RESTful or GraphQL APIs that enable remote search of model artifacts.

Quality assessment support: To facilitate quality-aware searching for artifacts. Search tools should be able to retrieve artifacts based on their quality score to filter out low quality artifacts.

▷ *Automatic quality assessment*: This refers to tools that have quality assessment mechanisms. These services derive quality metrics from the persistence artifacts and can be used in search or discovery.

▷ *Quality persistence*: To enable real-time discovery of quality metrics and attributes [21], this feature allows tools to trigger the calculation of metrics based on emitted event. These events are triggered when a specific artifact is created or updated. The derived metrics are kept along with the metadata of the artifact.

Indexing support: This feature allows quick retrieval of discovered artifacts.

▷ *Automatic Indexing*: In a cloud-based architecture where discovery (search) mechanisms are enabled, the artifact metadata, data file, file descriptors, and index are separated and managed in their self-contained environments. Typically, a live pipeline is established between the data store and the index. The live pipeline provides automatic indexing of newly created artifacts or when they are updated or deleted. In this way, we can outsource reading,

lookup, and other analysis to the search engine and keep the main data store as the single source of truth. In this way, files are automatically indexed when they are persisted, without the need to manually manage or perform indexing of specific artifacts.

▷ *Full-text search support*: This features ensures that the artifact can be retrieved based on its entire content and contextual information such as megamodel related data.

▷ *Integrated search engine*: To enable fast look-up and retrieval of relevant artifacts by discovery mechanisms, it is advisable to set up a search engine. Integrated search engines can be a general purpose search engine such as Lucene[8] [22] or a custom developed search engine such as MAR [141].

Tool interoperability: Features in this aspect allows integration with other tools and facilitate reuse in third-party applications.

▷ *Generated REST API*: This feature allows developers to convert queries made from the frontend to REST API. The generated API can be used directly in the developer application.

▷ *Public standardized API specifications* This feature allows API reuse and extensibility using API specifications such as OpenAPI 3.0 or GraphQL specifications. It is important to develop APIs that are modular to facilitate extensibility of the search/retrieval API.

Query mechanism: Discovery mechanisms support several types of query mechanisms. We have investigated and identified tools and related mechanisms supported as shown in fig. 2.1 and 2.2. Below are some of the key mechanisms used in the search/discovery of model artifacts in these tools:

▷ *Keyword-based query*: This is the basic mechanism in the search and discovery tools. Typically, the user enters a series of words separated by a space. These are extracted into tokens and entered into the search engine for look up. If the search is successful and matches the indexed keywords, the search engine returns the documents that contain the keywords of the query. The retrieved collections are usually ranked based on the relevance score.

▷ *Tag-based query*: The tag-based search mechanism is an advanced feature that allows the user to find a specific element in the artifact. Tags can be used to explore megamodel features such as artifact size, name, persistence time frame, etc. Artifacts can be retrieved based on their internal features such as class names or attributes.

---

[8]https://lucene.apache.org/

▷ *Query by-example query*: This feature allows the user to search for and discover artifacts using examples and a simple template. In the template, the user specifies an example model for artifacts that describes the expected results.

▷ *DSL-based*: tools that use general purpose, domain specific query languages or extension of existing languages.

▷ *Conditional expressions*: This feature allows the user to use conditional operators such as `AND, NOT, OR` to filter out the artifacts. The query can also support grouping clauses to condition the statement in the query.

▷ *Advanced search*: Advanced search allow the combinatorial use of several query mechanisms to filter out artifacts.

▷ *Advanced filters*: Framing queries to return results that exactly match the query is an added benefit. Filters can also include phrase matches, wildcards, and fuzzy searches.

▷ *API*: Tools that have an established API that you can use to filter out artifacts programmatically.

▷ *Browsing*: Browsing is another feature where the user can browse model artifacts through a list of artifacts organized by category. For some users, this is very convenient because it visually displays the available artifacts. Usually, this feature is supported by CRUD, where the user can view, edit, update or delete the selected artifact.

Scalability support: The integration of services into the discovery mechanisms of MDE hasn't been the focus yet. However, once this area is explored, the services involved and their environment must be scalable and extensible to support the workload of a large user community.

▷ *Service orchestration*: This feature allows the deployed services involved in advanced discovery mechanisms to be managed according to the system load. Their execution is done in a seamless way, so that unused resources can be put to sleep and activated only when needed.

▷ *Service containerization*: This feature allows services to be fully packaged with all necessary resources and dependencies to perform their task independently. Advanced discovery mechanisms include a large number of services that need to be strongly decoupled to facilitate their reusability.

Table 6.3 Comparison table of various discovery tools (1/2).

Feature	MAR [141]	IncQuery [200]	MDEForge [22]	Bislimovska et al. [34]	Gomes et al. [95]
Management of megamodel relations					
Model conformance		✓	✓	✓	
Transformation conformance					
User relationships					
Model-as-a-service					
Model transformation execution					
Service execution traces					
Model heterogeneity support					
Technology Independence	✓	✓	✓	✓	
Different kind of artifacts	✓	✓	✓		
User interface					
Cloud-based	✓	✓	✓		
Local-based				✓	✓
Quality assessment support					
Automatic quality assessment					
Quality persistence					
Indexing support					
Automatic indexing					
Full-text search support	✓		✓	✓	
Integrated search engine	✓		✓	✓	
Tools interoperability					
Generated REST API					
Public standardized API specs			✓		
REST/Graphical API support	✓	✓			
Query mechanism					
Keyword-based	✓				
Tag-based			✓		
Logical expressions					
Advanced filters					
API	✓	✓	✓		
DSL-based		✓			✓
Browsing					
Query-by example	✓			✓	
Scalability support					
Service orchestration		✓			
Service containerization		✓			
Cloud-based deployment	✓	✓	✓		
Reusability					
Artifact reusability					

▷ *Cloud deployment*: We will evaluate tools that can be deployed in the cloud, enabling execution over the Internet.

Comparing model search approaches

We selected works with supporting tools from the existing approaches identified in Section 6.1 and as shown in Table 6.3 and Table 6.4.

A search tool in a model-driven environment is expected to have features that allow the user to find relevant artifacts based on complex criteria. For example, the search should be megamodel-aware and locate artifacts based on their relationships to other artifacts in the system. For example, the approach presented by Basciani et al. [22] is one of the

Table 6.4 Comparison table of various discovery tools (2/2).

Feature	Moscript [127]	Hawk [15]	MultiModGraph [33]	MORSE [108]	Kotopoulos et al. [130]	Moogle [142]
Management of megamodel relations						
Model conformance	✓	✓	✓	✓	✓	✓
Transformation conformance						
User relationships						
Model-as-a-service						
Model transformation execution						
Service execution traces						
Model heterogeneity support						
Technology Independence						✓
Different kind of artifacts	✓	✓	✓	✓	✓	✓
User interface						
Cloud-based						✓
Local-based	✓	✓	✓	✓	✓	
Quality assessment support						
Automatic quality assessment						
Quality persistence						
Indexing support						
Automatic indexing						
Full-text search support					✓	✓
Integrated search engine		✓			✓	✓
Tools interoperability						
Generated REST API						
Public standardized API specs						
REST/Graphical API support		✓		✓		✓
Query mechanism						
Keyword-based						✓
Tag-based						
Logical expressions						✓
Advanced filters						
API				✓		
DSL-based	✓	✓		✓	✓	
Browsing						✓
Query-by example			✓			
Scalability support						
Service orchestration						
Service containerization						
Cloud-based deployment						
Reusability						
Artifact reusability	✓					

tools that can retrieve data based on model conformance. However, other relationships, such as user relationships or transformation conformance, are not considered. Search tools such as MAR [141], MDEForge [22], Hawk [15], Moogle [142], and Bislimovska at al. [34] have managed to integrate a search engine into their mechanisms. Using a search engine is important to ensure fast retrieval of relevant artifacts. Thanks to this feature, they implemented tag-based, keyword-based, and query-by-example mechanisms.

Upon our investigation, we noticed a lack of key features that are crucial for enhancing the discovery of model artifacts and tools within the model-driven engineering domain. Features such as transformation conformance, service execution traces, automatic quality assessment,

quality persistence, and automatic indexing are imperative to ensure an effective and robust model discovery process. Moreover, the incorporation of user relationships and the reusability of artifacts, APIs, and infrastructure is instrumental in elevating user experience, fostering collaboration, and promoting query customization capabilities. Furthermore, adopting modern cloud-based environments and service-oriented methodologies is crucial for bolstering scalability, extensibility, and interoperability in model-driven engineering processes. This facilitates seamless integration with a multitude of external and internal services, enabling the reuse of artifacts and tools in distributed and decentralized environments.

Table 6.3 and Table 6.4 show that search/discovery tools have not yet attempted to integrate third-party services that can enrich their index, for example, to compute derived data such as quality metrics. In addition, we have not found tools that enable search based on collaborative features, which are very important in current modeling environments where other users share artifacts. Search tools such as Moscript [127], IncQuery [200], Hawk [15], Kotopoulos et al. [130]. Although these DSLs have high granularity, they tend to be somewhat complex and require a high learning curve. We also found that the indexing mechanisms are not dynamic and do not allow for programmed automatic indexing of model artifacts. Some of the tools that use a search engine in their discovery mechanisms do not consider the internal structure of the models. Therefore, they perform only a text search, which is sometimes limited. In most cases, tools that use MDE DSLs for their search/query take into account the model structure. Although DSL-based queries are complex and pose a high learning curve, they allow artifact exploration with low granularity.

According to the features mentioned in the previous section, Table 6.3 and Table 6.4 also show model heterogeneity, extensibility, and interoperability of tools as some of the desirable features of a model search tool. These features help ensure that various external and internal services can use various models (e.g., model transformation and validation in different languages). Moreover, they ensure artifact reuse by using REST or other API interfaces. Search tools such as MDEForge [22], MAR [141], MORSE [108], Moogle [142], and IncQuery [200] have an API to interact with the external world.

Finally, in a modern cloud-based environment, the operations used in a search tool for MDE artifacts should be service-oriented and executed on demand. Therefore, to support the scalability and extensibility of these services, they should be packaged in containers and orchestrated across multiple nodes. This cloud computing layer, where model management operations are treated as services, facilitates the reusability of artifacts and tools in a distributed and decentralized environment. For example, search tools such as MAR [141] and IncQuery [200] are among the most popular tools that can be used as cloud-based services.

These tools have implemented search/query mechanisms that enables users to execute their query using web-based APIs.

MDEForge-Search elevates MDEForge by implementing all features presented in the tables above apart from query-by-example mechanisms. MDEForge-Search is an extension of MDEForge and built on earlier concepts developed by MDEForge project. MDEForge-Search intends to enable scalable and extensible infrastructures and services in the context of cloud-based model repositories. It also enables advanced discovery mechanisms that are detailed in Section 6.3.

Limitations of current discovery mechanisms in MDE domain

In this subsection, we answer ***RQ2*** and provide an overview of the main challenges identified in the tools analyzed in the previous section. Since search/discovery mechanisms are at the heart of model artifact and tool reuse, addressing these challenges could influence the adoption of MDE practices in software development and enhance collaboration.

▷ *The Big Data Wave*: The advent of cloud computing and Big Data has led to a revolutionary shift in how data is collected, processed, and consumed [228, 100]. Due to the amount of data that both systems and users generate regularly, traditional data processing methods have proven inadequate for the tasks expected [185, 172]. In addition to the amount of data currently being received, data is coming in different varieties from different sources at an unprecedented rate. This data needs to be stored, retrieved, and processed for further use cases [39, 101]. Given the adoption of MDE practices in the industry, the MDE community should consider Big Data (big models) in its discovery mechanisms to consolidate its adoption. However, the tools and their architectures studied indicate that discovery-based architectures and techniques for large model artifacts are either rarely considered or not considered at all.

▷ *Local Infrastructures*: The era of Big Data mentioned above requires a scalable and extensible infrastructure to handle the data stream being processed [113]. The volume of data received from systems and users is too large for traditional computing paradigms [134]. Local infrastructures are not designed for Big Data, so cloud computing is chosen [106]. Although many MDE infrastructures are local-based, the MDE community is looking to move their infrastructures to the cloud to meet requirements such as collaborative modeling, scalability, and extensibility of their infrastructure. It is challenging to persist, process, index, and query large models with local resources. Although Mar [141], Moogle [142], MDEFoge [22] and IncQuery [200] are deployed online, the reusability of discovered artifacts is neglected, forcing the user to download retrieved artifacts for their further use.

▷ *Platform-dependent approaches*: Current MDE discovery tools generally lack efficient generic and technology-independent techniques for finding relevant artifacts. Most of the tools shown in Table 2.1 and Table 2.2 are dependent on the Eclipse Modeling Framework (EMF). This is not surprising since EMF is a predominant framework in the modeling community. Nevertheless, the need for agnostic and technology-independent tools and platforms is very important to democratize MDE practices for citizen developers and ensure collaboration among stakeholders.

▷ *Limited Query Mechanisms*: Using query mechanisms such as keywords or object query DSLs is no longer sufficient to find relevant artifacts in today's complex modeling environment. Therefore, discovery tools need to adapt and provide the user with query mechanisms that can be used to filter relevant artifacts. Furthermore, such mechanisms could aim to combine a large portion of these mechanisms in a more scalable and efficient query. The mechanisms include using keywords, tags, logical operators, advanced filters, API, DSL, browsing, or query by example, as shown in Table 2.1 and Table 2.2.

▷ *Expressiveness of query mechanisms*: Current solutions are either too simple or too complex to retrieve relevant artifacts in complex and large data stores. Typically, query tools use DSLs to enable artifact discovery. Unfortunately, these query DSLs require a significant learning curve, modeling, and programming expertise.

▷ *Reusability*: The goal of the discovery phase is to reuse discovered artifacts. Unfortunately, current discovery tools focus only on discovery and ignore the reuse phase. As a result, users download artifacts locally and upload them somewhere to reuse them, as shown in fig. 6.1. Therefore, we lack a platform that enables the discovery of relevant model artifacts and the reuse of the discovered artifacts in model management operations on the same platform.

▷ *Third party integration*: We also found that there is a lack of integration of third-party services that can evaluate models and compute derived metadata that improve the discovery of relevant artifacts. For example, such services could enrich the index with metadata such as quality metrics of artifacts or other criteria that help users find relevant artifacts.

6.3   Proposed approach

This section introduces MDEForge-Search, a novel approach tailored to overcome the limitations previously identified in Section 6.2.1. The proposed approach is specifically devised to facilitate the discovery of model artifacts and tools. Subsequently, we outline the architectural design and architectural layers underpinning the approach.

Architectural design

This section provides an overview of the high-level architecture of MDEForge-Search, a substantial extension of the MDEForge infrastructure [18, 116]. MDEForge serves as a platform for storing modeling artifacts and executing model management operations remotely. However, the indexing capabilities and query mechanisms of MDEForge are rudimentary. To address these limitations, MDEForge-Search advances the MDEForge repository by enhancing the scalability and extensibility of its underlying infrastructures and services while concurrently facilitating advanced discovery and reusability mechanisms. MDEForge-Search employs a distributed microservice architecture, leveraging Kubernetes[9] [116]. Figure 6.4 provides a high-level view of the building blocks of the platform. The following sections provide a detailed explanation of the architectural design, as depicted in Fig. 6.4.

Storage infrastructure

MDEForge-Search facilitates discovery and reuse phases by relying on a storage infrastructure consisting of a domain-agnostic repository and on a set of integrated services as presented below.

Domain-agnostic repository   This component deals directly with data acquisition and indexing. It is designed to be domain agnostic; hence, it can persist and index data from any modeling domain. Artifacts are stored with metadata such as name, size, type, and so on. An asynchronous extraction module extracts these metadata. This module is also responsible for extracting artifact structure to facilitate structural-based search. In this fashion, the user can easily explore the internal elements of the artifact and retrieve artifacts based on internal elements, cardinalities and relationships such as model conformance. The metadata are stored using a cluster of MongoDB databases[10]. For a durable data store that can handle highly intensive computational jobs and transactions, it is advantageous to have MongoDB[10] as the single primary source of truth for writing operations, rapid data ingestion, and ultimately index data in Elasticsearch[11] [41]. We thus offload search and analytics' activities to Elasticsearch[11]. Elasticsearch[11] is a distributed, open-source, and highly scalable search and analytics engine. It is built on Apache Lucene[12] and facilitates simplified data management, reliability, and horizontal scalability. It offers a more powerful full-text search engine and distributed multitenant capabilities than its competitors [79]. We establish a live data pipeline between MongoDB and Elasticsearch clusters. MongoDB is maintained as the source of truth, guaranteeing data integrity, enabling transactions, and facilitating data

---

[9]https://kubernetes.io/
[10]https://www.mongodb.com/
[11]https://www.elastic.co/
[12]https://lucene.apache.org/

MDEForge integrated repository
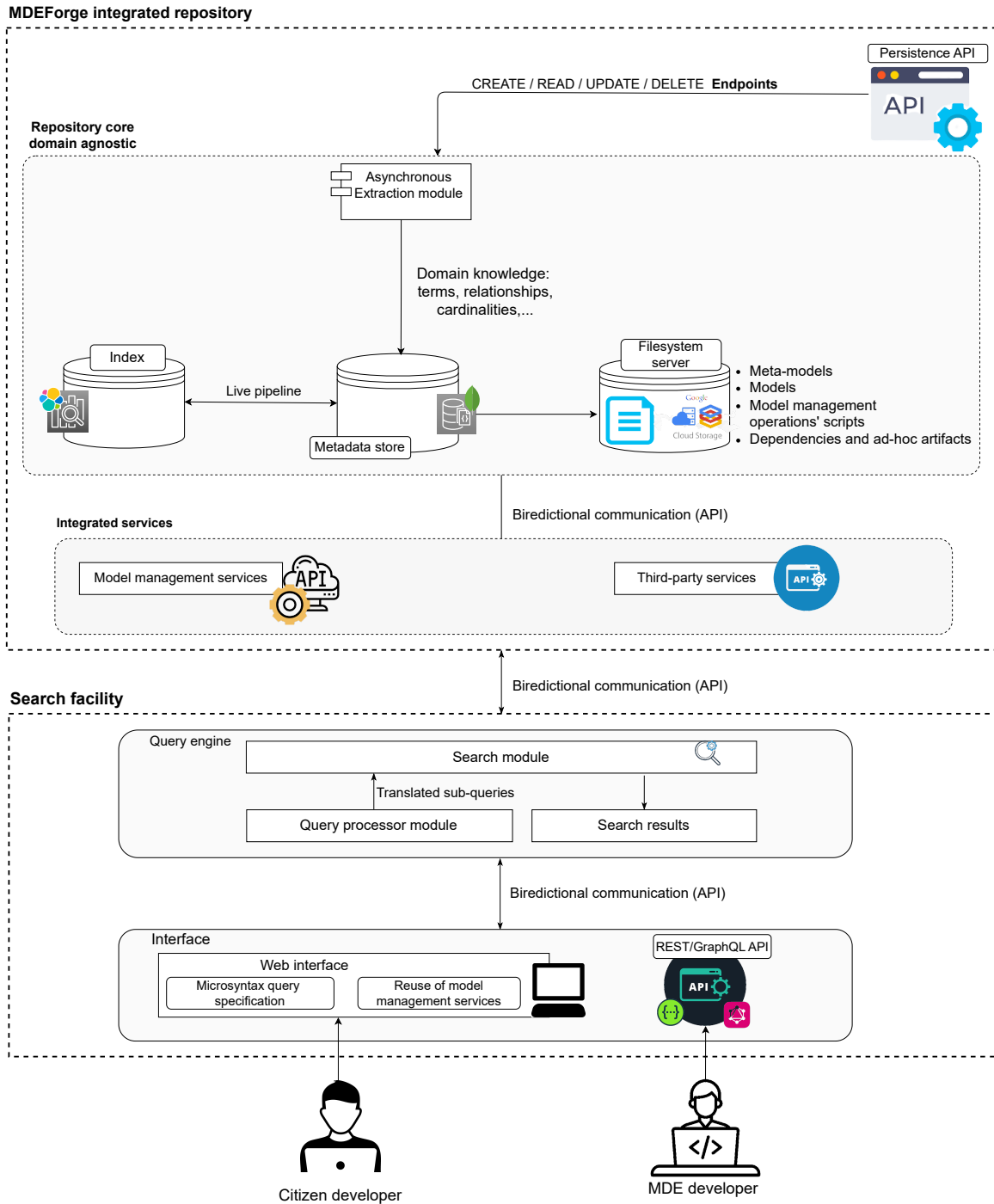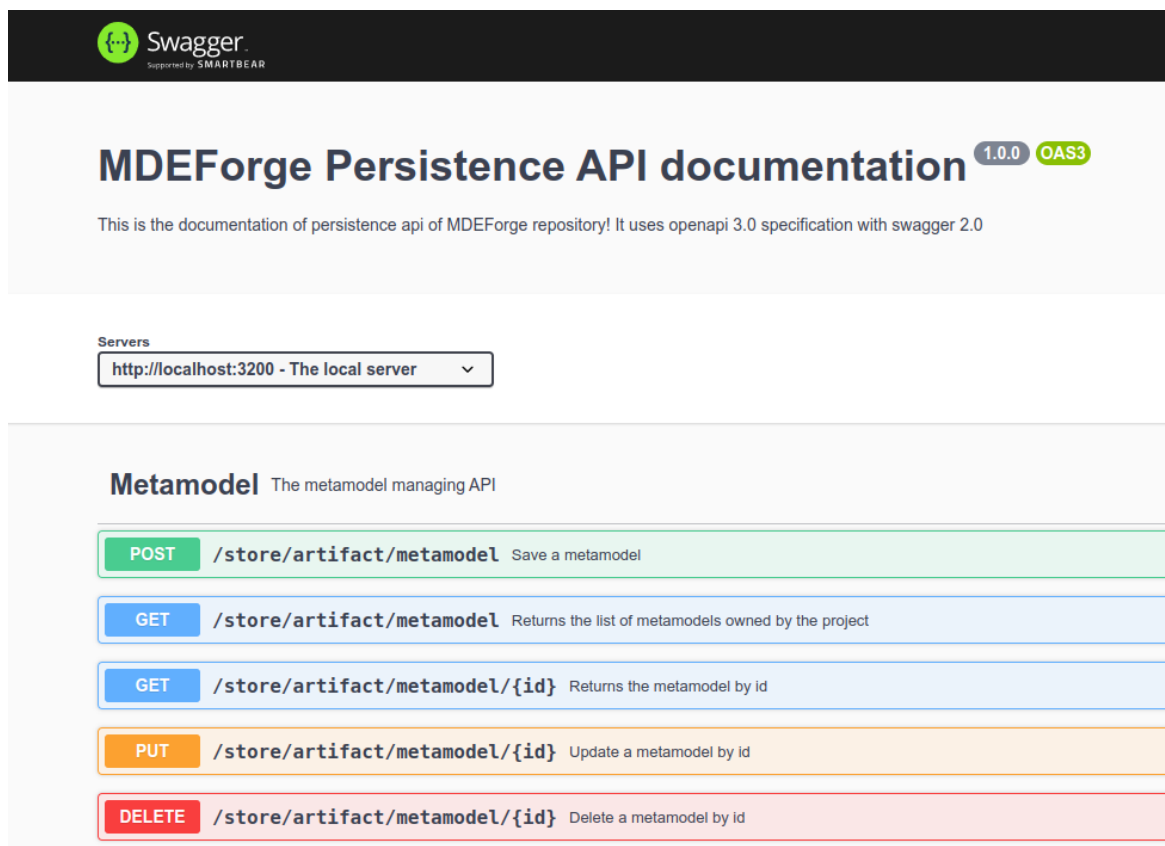
Fig. 6.4 MDEForge-Search high level architecture

backup [213] and Elasticsearch as our integrated search and analytics engine. It is important to note that the document file of the artifacts is persisted in a separate cloud storage server. Only links to actual files are persisted in the metadata data store. The domain-agnostic

Fig. 6.5 MDEForge-Search Persistence API.

repository component manages integrated technologies such as Elasticsearch and MongoDB. These technologies can be replaced or extended to improve performance without requiring changes to the high-level architecture or code base.

Integrated services   This is a wrapper that integrates the repository core domain agnostic block with external services. In particular, it integrates the persistence API (c.f. fig. 6.5), and model management services as shown in fig. 5.4. We have also integrated services that can derive additional metadata from persisted artifacts, such as quality metrics or transformation chains.

MDEForge-Search follows the model-as-a-service (MaaS) paradigm [116]. MaaS enables developers to design and maintain modeling resources and tools that are subsequently made available to end-users as software as a service (SaaS) templates [229]. In this cloud computing tier, web services are the foundation for building distributed applications (c.f. fig. 6.5). Business logic and underlying technology are abstracted into packages, and high-level on-demand capabilities are outsourced via the Internet [229, 19]. Its usage enables on-demand execution of services across the network [19, 183, 23]. Moreover, it facilitates

collaboration among stakeholders, resource optimization and interoperability regardless of underlying technologies [19]. With MaaS, local configuration and infrastructure setup are replaced with cloud-based infrastructure, thus drastically lowering time-to-market [74]. Following this model, our services are containerized and deployed as services. They also are orchestrated in a distributed micro-service environment using Kubernetes on the Google cloud platform [116].

Search facility

The search facility comprises components enabling the discovery and reuse of the repository. It is made of a query engine and its interface as discussed below.

Query engine   This is the backbone of the search facility. It mainly consists of a search module, a query processor module and the returned results' object. However, it has other low-level components that are discussed in the logical layers of the system in Section 6.3.

The *Search module* is comprised of a wrapper API that exposes the engine capabilities to the search facility building block. The module is essentially responsible for receiving and firing translated sub-queries from the query processor module to the search engine. It also collates the search results and can handle errors in case the microsyntax query specification has syntax errors.

The *Query processor module* is responsible of parsing microsyntax query specifications and render its equivalent DSL format to the search engine (c.f. Section 6.4). If the query specification exhibits a syntax error, the parser communicates the error to the search module, and the search module renders the error. The query processor module is implemented to retrieve data derived from third-party services such as quality assessment services (c.f. Section 6.4).

The *Search results* component manages the output of the search module in the JSON format and can be consumed via REST /GraphQL APIs. The returned data is annotated to facilitate its use in, for example, machine learning frameworks, as in the case of the DROID framework [7] (c.f. Section 6.5). In case of an error, the system renders its equivalent object with the appropriate HTTP status to facilitate debugging.

Interface   This component comprises facilities that facilitate searches from citizen and MDE developers. They usually reuse or consume services or data using the application programming interface as RESTful API. We have deployed OpenAPI 3.0 specification and GraphQL specification to enable exploration of the search facility API. The citizen developer (regular user) can search and discover artifact using the microsyntax query specification via a web interface. The retrieved artifacts can be reused directly in model management services
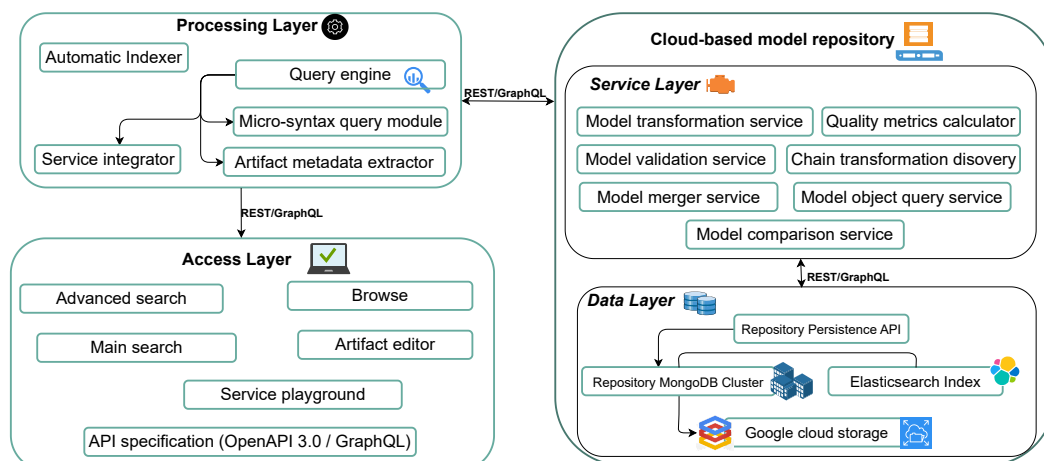
Fig. 6.6 MDEForge-Search logic view.

deployed on the same platform. She can also perform CRUD operations on the retrieved artifacts.

Logical layers

Figure 6.6 presents a detailed view of MDEForge-Search logical layers, i.e., *Data layer*, *Service layer*, *Processing layer*, and *Access layer*. In the remainder of this section, we present in greater details each of them.

Data layer

The journey of data persistence in a cloud-based model repository begins with an API built on the repository warehouse (MongoDB cluster). It is the part of the repository that houses structured data pulled and pre-processed from the data lake. Before further processing, unprocessed data are dumped in the data lake. The data lake is not organized (Google cloud storage), and data is collected in their varied sets of raw data in their native format. MongoDB databases are organized into a cluster. This cluster is our data warehouse where data is processed and organized in the cluster. Elasticsearch is employed for indexing and search operations while MongoDB cluster serves as the primary data repository. Collected heterogeneous modeling artifacts are organized as models, meta-models, or DSL scripts. The actual files, however, are stored at Google cloud storage, as shown in Fig. 6.4. The link to the file is accessible from artifact's metadata and is kept within our database cluster.

All metadata is automatically in sync with data indexed in Elasticsearch in real-time. Additionally, a persistence API was developed on top these tools to manage and provide smooth interactions with the data stored. Heterogeneous artifacts are persisted according to a user-oriented scheme. We gather all the necessary information from the user to ensure she can

control access to her resources within workspaces. The workspace is made up of projects created by the user. The user can share access to her projects. The user can share access, thus granting permissions to the shared user based on her preferences. The projects are made of artifacts.

Service layer

It is a cluster that provides a pool of model management services deployed natively in the cloud-based environment [116]. Each service wraps a corresponding engine responsible for carrying out model operations previously deployed on local infrastructure, as shown in Fig. 6.5. Following the MaaS paradigm, these model operations are exposed externally and can be executed on-demand over the Internet [45]. Their remote execution promotes the reusability of model artifacts in the repository without further configurations necessary before this deployment model.

Currently, this cloud-based model repository integrates model management services that carry out model transformations, model object queries, model validations, model comparisons, and model merging operations. In addition, new services were incorporated into the ecosystem to support the proposed advanced search and query mechanism approach. These services are a quality metrics calculator, a chain transformation discovery service, and a query engine.

Processing layer

It processes the query from the application layer (c.f. fig. 6.5). The layer is made up of a query engine and an automatic indexer. The query engine comprises three components: a Microsyntax query module, an artifact crawler and a service integrator. The service integrator enables external service consumption in MDEForge-Search via APIs. The automatic indexer ensures a live pipeline between MongoDB sharded cluster with Elasticsearch. The query engine integrates together the search engine, a model artifact crawler, a service integrator, and the microsyntax query module.

Access layer

It consists of graphical interfaces and APIs that are used to explore the repository via the query engine (c.f. Fig. 6.4 and Fig. 6.5). The APIs are modular functionality that can be further extended in a given application to deliver the full capability the query engine offers. The graphical interface provides a Web interface to enable the visual exploration of the repository. The user can easily navigate the repository content in the search box with an easy-to-use microsyntax query specification. The main search is used by the microsyntax query specification. The advanced search offers users more in depth screening of the artifacts inside the repository. Users can browse by their types such as model, metamodel or DSL and
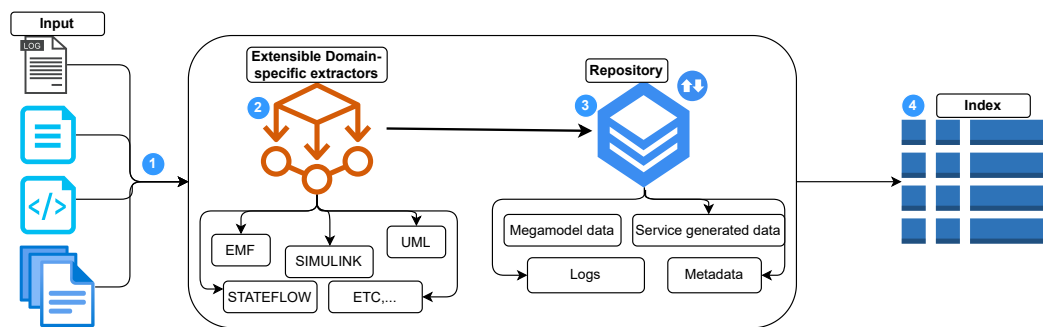
Fig. 6.7 Data flow in MDEForge-Search.

can be screened alphabetically. Retrieved artifacts can be investigated by the means of an editor.

The service playground reuses the artifacts in the service layer, notably model management services. Other services such as the quality metrics calculator service and chain transformation service are already being reused by the microsyntax query specification while performing lookup in the repositories. The search results are displayed based on the relevancy score respectively [79]. By relevancy score, the user searches and queries are analyzed further to return data that reflect the search or query context.

6.4 Enabling advanced reuse-driven discovery

In this section, we explain how we enabled advanced reuse-driven discovery mechanisms on a cloud-based model repository using MDEForge-Search. MDEForge-Search supports the data collection and preprocessing, discovery, and reuse mechanisms as described in the following subsections.

Data ingestion and processing

In a cloud-based model repository with heterogeneous artifacts, ingesting and processing data is critical. As shown in Fig. 6.7, this is the first step toward enabling the discovery and reuse of persisted artifacts. The ingestion process must be reliable and efficient to support a large user community and subsequent operations. As shown in Fig. 6.7, ① data is ingested into the repository using APIs or a web user interface. Our repository is designed to handle a variety of data formats used by different modeling tools. Therefore, incoming data are heterogeneous model artifacts from different sources and technologies. Once data are uploaded to the repository, there are several challenges to overcome to support their discovery and reuse in later phases. First, adopted mechanisms should extract metadata and properly format incoming data input before it is ingested into the repository. Second, modeling processes are diverse and encompass a variety of technologies and data formats. Therefore, extraction

approaches at this stage should be extensible to handle each incoming artifact according to its domain ②.

In our case, we managed to introduce an EMF data extractor. This extractor crawls the file and extracts data of interest, such as class names, attributes, and references. The implementation of this phase is extensible and allows adding more domain-specific extractors from other technologies such as Simulink, UML, and Stateflow model artifacts.This phase is asynchronous to allow tasks and processes to overlap and complete their execution in the background rather than waiting for one task to finish before starting the next. Phase ③ involves organizing the extracted data into structured data that facilitates discovery and reuse. The extracted data are organized into megamodel-related data, including output generated by services such as quality assessment or chain transformation discoverer, logs, or metadata itself, such as artifact size and name. The final phase ④ (c.f Fig. 6.7) involves indexing structured data for rapid discovery and exploration.

As introduced in Section 6.3, Elasticsearch is a powerful distributed search and analytics engine that organizes data within an indexed cluster of nodes that are replicated and sharded for improved fault tolerance, high availability, and scalability [98]. It is recognized as one of the best industry-grade open-source search and analytics engines [110]. At its core, Elasticsearch leverages Apache Lucene, an efficient text search engine library, to index data utilizing an inverted index — an advanced data structure that significantly accelerates the retrieval of matched terms [ela, 98]. This inverted index facilitates full-text search capabilities by mapping terms to their corresponding documents, enabling Elasticsearch to rapidly identify relevant documents based on user queries.

In addition, Elasticsearch adopts the boolean model for locating matching documents. It employs logical operators, such as AND, OR, and NOT, to refine search criteria. Furthermore, Elasticsearch incorporates the *practical scoring function* [98] for determining the relevance of retrieved artifacts. This scoring function considers factors, such as term frequency and inverse document frequency. These factors contribute to a more precise ranking of search results, thus enhancing the user experience and improving the overall efficiency of the search process.

```
score(q,d) =                              1
      .queryNorm(q)                       2
      .coord(q,d)                         3
      . ∑ (                               4
          tf(t in d)                      5
        . idf(t)²                         6
        . t.getBoost()                    7
```

```
. norm(t,d)                     8
) (t in q)                      9
```

The above function `score(q,d)` at line 1 returns the relevance score of a document d for query q. At line 2, `queryNorm(q)` is the query normalization factor [ela] represented mathematically as such:

$$queryNorm = \frac{1}{\sqrt{\sum_{i=1}^{n}[idfT(n-i)]^2}} \tag{6.1}$$

where Ti,...,Tn are query terms and $idfTi^2$,..., $idfTn^2$ are the *squares of inverse-document-frequencies of the terms* or *squared weights*. Following query normalization, the results of one query can be compared to the results of another query.

At line 3, `coord(q,d)` is coordination factor responsible of rewarding the document with a higher percentage of the query terms. It is represented using this equation [ela]:

```
let P = the number of matching terms from query q in document d and
k = total number of terms in query q
```

$$\tag{2}$$

$$coord(q,d) = (\sum_{i=1}^{n}[idfT(n-i)]^2) \times (\tfrac{P}{k})$$

At line 9, the function *sums* the weight for each term t in query q at line 5, for document d. At line 5. `tf(t in d)` assesses term frequency for term t in document d. `idf(t)` at line 6 calculates the inverse document frequency for term t. It responsible of punished repetitive terms such as like, or, and, so, etc. Thus a lower weight is assigned to such terms making less frequency terms relevant to zoom in the relevant document. At line 7, `t.getBoost()` applies the boost to the query to make one query clause more important than the other. At line 8, `norm(t,d)` is the field-length norm. The shorter the field, the higher the weight is assigned because if a term appears in *short field* such as the title, it is likely that the content is about the term. However, if it appears in the *body field*, it might not be very relevant [ela, 98].

Discovery mechanisms

Our cloud-based model repository supports three discovery mechanisms i.e., a *microsyntax query specification*, *advanced searches*, and *browsing facilities* integrated in a web-based search facility, as described in the following.

Microsyntax query specifications   In computing environments, a domain-specific language (DSL) is a computer language that has been tailored to a particular domain for a specific purpose [28]. In contrast, a general-purpose language (GPL) strives to be suitable for writing programs in any domain. The main objective behind DSLs is to simplify complex tasks and problems in a specific domain [63]. In this regard, our domain-specific microsyntax query
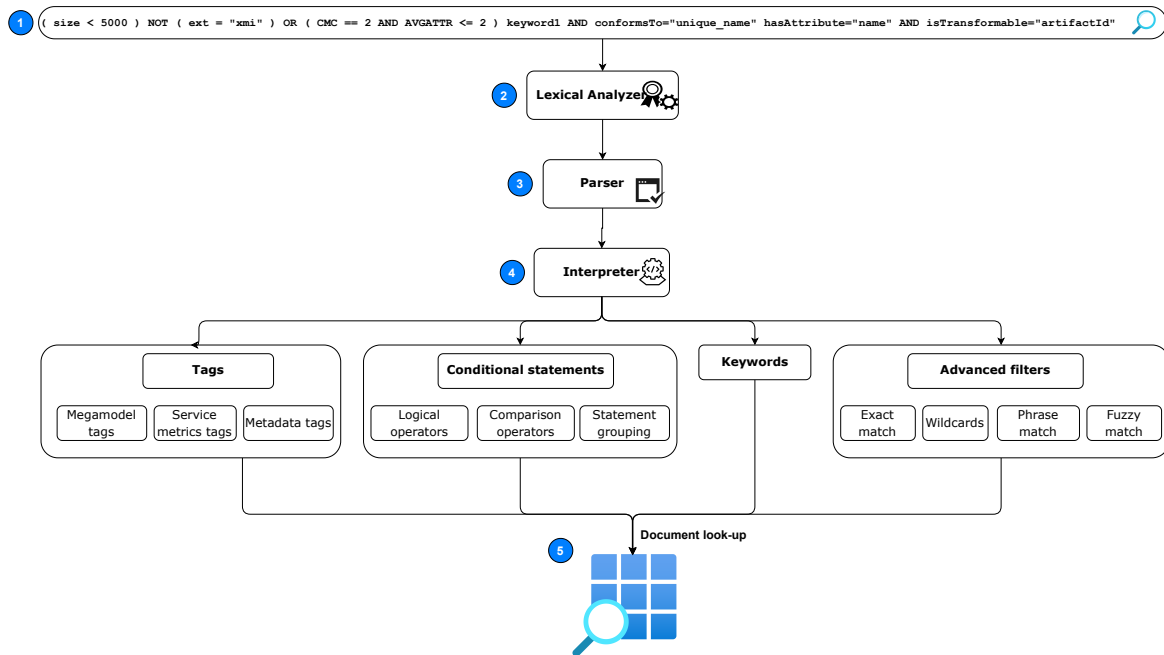
Fig. 6.8 Microsyntax query-specification information flow.

specification is designed to facilitate the discovery and exploration of the repository using an advanced query mechanism. fig. 6.8 shows the flow of information inside the infrastructures that underpin the microsyntax-based query specification mechanism.

As shown in fig. 6.8, in ① the user formulates a query in the form of one query line of text. Next, the text is consumed by a lexical analyzer that breaks down the text into tokens ②. This process involves scanning the input for defined patterns and dividing it into meaningful units used by the parser later on [28]. Once the input has been tokenized, it is used to build an abstract syntax tree (AST) ③. The AST is a tree-like structure that represents the code in a way that is easier for the machine to digest [28]. After the AST is constructed, the AST is traversed and converted into machine code that is executed to produce the desired results. This way, the computer interprets the query text to generate an executable artifact. As shown in fig. 6.8, phase ④ of this process considers essential constructs that compose the microsyntax query specification. These constructs include tags, conditional statements, keywords and advanced filters, discussed in the next paragraph. The final phase is the document look-up in the repository index ⑤, which we explained earlier in the previous section 6.4.

The simplified metamodel given in Fig. 6.9 illustrates the main elements of the query specification DSL. This language has been developed to enhance the efficiency of the discovery process by leveraging two key entities: *Full-text search* and *Contextual search*.
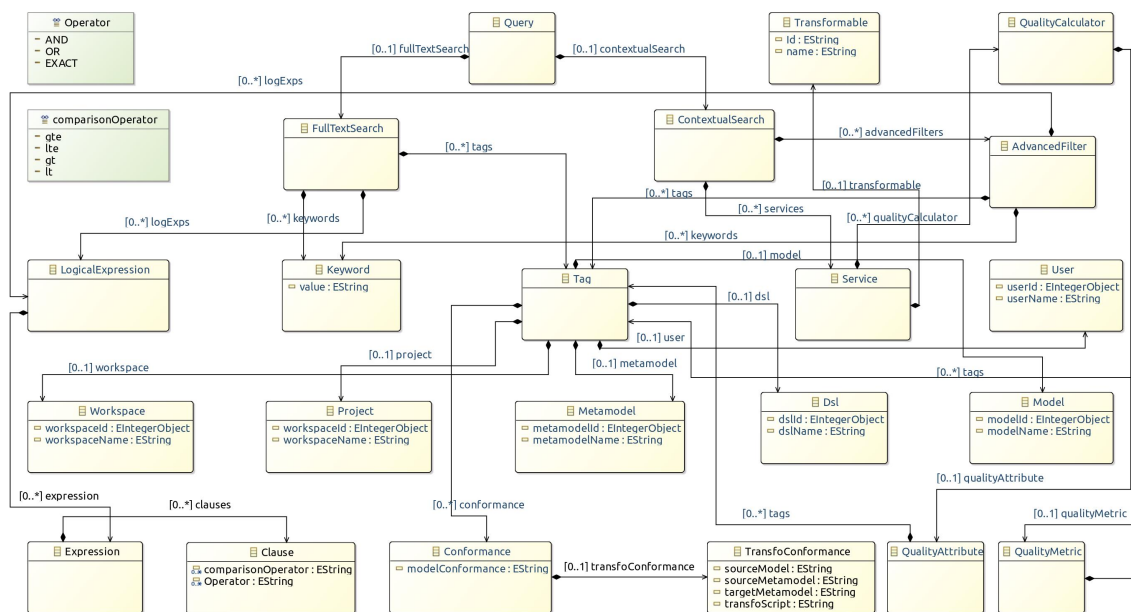
Fig. 6.9 Simplified metamodel of the query specification.

By employing these search categories, users are equipped with powerful tools to retrieve relevant information from cloud-based model repositories.

Full-text search offers a comprehensive mechanism for searching through the content, associated logs, and metadata of stored artifacts. This approach is facilitated by search keywords, tags, and logical expressions. To enable users to construct intricate queries, conditional statements expressed as nested clauses are employed. The incorporation of logical and comparison operators within these clauses (Fig. 6.9) allows for the combination of terms using operators such as AND, OR, and NOT. Consequently, users can refine and filter results more effectively. To optimize the search process, search keywords are tokenized and indexed within an inverted index data structure using Elasticsearch[11].

Tags play a crucial role in the query specification as they serve as navigational markers to direct users towards relevant artifacts. These tags are derived from the organizational structure of the repository and are designed to be user-oriented, hierarchical, and adaptable to different contexts. They encompass users, workspaces, and projects, taking into account access permissions and restrictions. By targeting artifacts based on specific properties or metadata fields, tags allow users to locate models that adhere to a particular metamodel. This enhances the precision of search results, ensuring that the retrieved models align with the desired criteria. Additionally, tags support transformation conformance, enabling users to retrieve artifacts involved in specified transformation operations.

Contextual search is centered around the megamodel relations among artifacts and their defined search space. It examines various aspects such as artifact relationships, involved operations, hierarchical relations, creation timestamps, names, and types (refer to Fig. 6.13). This search method serves as a complement to the full-text search, providing users with an alternative approach to discovering relevant information within a specific context.

By amalgamating the methodologies of full-text and contextual search, a robust and versatile query mechanism is established. As a result, users can efficiently navigate intricate cloud-based model repositories by simply entering a single search text.

Figure 6.10 shows explanatory examples of queries specified by means of the proposed discovery language. Queries are ordered to have a progression in terms of complexity, going from simple tag searches to combining multiple tags, logical expressions, and metric criteria with various boolean and comparison operators.

Q1: The first query selects model artifacts that have their access control set to public and the size of the artifacts is less than 500kb or the extension of the artifacts is of json format.

Q2: This query adds dates and quality metrics in the lookup scope. The query is targeting model artifacts that are publicly accessible and either have a size greater than or equal to 1000. The artifacts should have been created on or before January 2022 or they should have a `attr` metric (number of attributes) greater than 5.

Q3: This query introduces the use of metrics (cflmc - *concrete featureless meta class*) in the search and combines it with criteria on type, unique name, and license. It is looking for model artifacts that are either of 'ecore' type and have a unique name 'uniqueModel', or are licensed under 'Apache' and have a 'cflmc' metric equal to 2.

Q4: This query adds more depth by incorporating attribute, description, date, and metric criteria. The query is for model artifacts that either have a 'color' attribute and are described as 'Complex Model', or have an update timestamp later than 1st January 2024 and have the `amc` metric (number of abstract metaclasses) less than or equal to 3.

Q5: This query combines project, description, and type criteria with a mix of AND and OR operators. It targets model artifacts whose `description` is 'Advanced Model' or model artifacts that belong to 'Project1' and have 'xmi' as their type.

Q6: In this query, there is an AND operator outside the brackets and inside, making it more complex. It combines criteria on conformance, license, and name. It targets

model artifacts conforming to a given 'metamodel_id', and among those, ones that are licensed under 'GPL' and named 'ModelY'.

Q7: This query introduces involved operations criteria and combines it with unique name and metric criteria. It looks for model artifacts with 'operation_id' in their involved operations, and specifically among those, ones with a unique name 'ModelX' and have the metric named `mc` (number of metaclasses) greater than 50.

Q8: This query increases the query complexity by including a NOT operator in the mix. It combines date, extension, metric, keyword, license, project, and transformation capability criteria. Models updated after 1st January 2023 and not in JSON format, or models that have `cmc` metric (number of concrete metaclasses) greater than 5 and `sf` metric (number of structural features) less than or equal to 2, with keyword5, are being searched. They must be licensed under 'MIT', belong to 'Project1', and be transformable to a given model artifact.

```
1   # Example queries of the microsyntax query specification DSL:
2
3   # Q1
4   accessControl = 'public' AND size < 500 OR ext = 'json'
5
6   # Q2
7   (size >= 1000 AND createdAt <= 20220101) OR (project = 'Project2' AND attr > 5)
8
9   # Q3
10  (type = 'ecore' AND unique_name = 'uniqueModel') OR (license = 'Apache' AND cflmc = 2)
11
12  # Q4
13  (hasAttribute = 'color' AND description = 'Complex Model') OR (updatedAt > 20240101 AND amc <= 3)
14
15  # Q5
16  description = 'Advanced Model' OR project = 'Project1' AND type = 'xmi'
17
18  # Q6
19  conformsTo = 'matamodel_id' AND (license = 'GPL' AND name = 'ModelY')
20
21  # Q7
22  involvedOperations = 'operation_id' (unique_name = 'ModelX' AND mc > 50)
23
24  # Q8
25  (updatedAt > 20230101) NOT (ext = 'json') OR (cmc > 5 AND sf <= 2) keyword5 AND license = "MIT" project="Project1" AND isTransformable='ARTIFACT_ID'
26
27  # ----------------------------------------------------------------------------
```

Fig. 6.10 Examples of queries

The Extended Backus-Naur Form (EBNF)[?] shown in Fig. 6.11 defines the grammar of the discover language and consists of the following constructs:

- `searchQuery` is the starting point of the grammar. It consists of a term followed by the End Of File (`EOF`) marker. This ensures that the input is parsed in its entirety and correctly.

- `term` is a sequence of one or more factor expressions connected by boolean operator symbols. This allows the construction of complex queries by combining multiple factors with boolean operators such as `AND, OR, NOT, and SPACE`.

```
1   (* Extended Backus-Naur Form *)
2
3   searchQuery = term, EOF ;
4   term = factor, { boolean_operator, factor } ;
5   factor = text | tag | expression | "(", term, ")" ;
6   boolean_operator = "AND" | "OR" | "NOT" | SPACE ;
7   operator = ">" | "<" | "==" | ">=" | "<=" ;
8   text = KEYWORD | NUMBER | exactText ;
9   exactText = QUOTE, KEYWORD, { SPACE, KEYWORD }, QUOTE ;
10  tag = ( TAG_ID | NUMERICAL_TAG_ID ), "=", tagValue ;
11  tagValue = KEYWORD | NUMBER | exactText ;
12  expression = numericalExpr | metricExpr ;
13  metricExpr = METRIC_ID, operator, NUMBER ;
14  numericalExpr = NUMERICAL_TAG_ID, operator, NUMBER ;
15
16  TAG_ID =
17  'accessControl' | 'content' |  'description' | 'ext' | 'involvedOperations' | 'license' | 'name' | 'project'
18  | 'storageUrl' | 'type' | 'unique_name' |  'conformsTo' | 'hasAttribute' | 'isTransformable' ;
19
20  NUMERICAL_TAG_ID = 'size' | 'createdAt' | 'updatedAt' ;
21
22  METRIC_ID = 'acfmc' | 'aiflmc' | 'amc' | 'attr'    | 'attrh' | 'avgattr' | 'avgref' | 'ccfmc' | 'cflmc' | 'ciflmc' | 'cmc' | 'iflmc'
23  | 'lmc' | 'maxhl' | 'maxhs' | 'mc' | 'mcwsp' | 'mtnb' | 'rec_cont' | 'ref'  | 'refcc' | 'refeop' | 'sf' | 'sfh' ;
24
25  NUMBER = digit, { digit } ;
26  (* digit = "0" .. "9" *)
27
28  KEYWORD = nonQuoteSpace, { nonQuoteSpace } ;
29  (* nonQuoteSpace = any character excluding single or double quotes and spaces  - special characters are espaced with '\'*)
30
31  SPACE = " " ;
32  QUOTE = "'" | "\"" ;
33
34  # -------------------------------------------------------------------------------
35
36  # Constraints:
37  # 1. The search_query should start with a valid token
38  # 2. The search_query should not end with 'AND', 'OR', or 'NOT' or any other operator.
39  # 3. In a metric expression, the metric_operator must be followed by a number because the quality metrics are numerical values.
40  # 4. Tag expressions should have a colon ':' followed by a valid tag_value.
41
42  # -------------------------------------------------------------------------------
```

Fig. 6.11 Simplified EBNF of the proposed discovery language

- `factor` is the fundamental building block of the query language. It can be either text, a tag, an expression, or a term enclosed in parentheses. The usage of parentheses allows the grouping of terms and the alteration of the operation precedence. When a term is within parentheses, it is resolved recursively.

- `boolean_operator` represents logical operators for connecting factors: `AND, OR, NOT,` and `SPACE. SPACE` is interpreted as `OR` when used as a connecting factor.

- `operator` signifies comparison operators used in expressions for manipulating values. It includes greater than ($>$), less than ($<$), equal to ($==$), and their variations with an equal sign ($>=, <=$).

- `text` represents a keyword, a number or an exact text.

- `exactText` is a sequence of one or more keywords enclosed in single or double quotes, indicating that the query is searching for an exact match of the given text.

- `tag` comprises a tag identifier, which could be `TAG_ID` or `NUMERICAL_TAG_ID`, followed by an equals sign and a tagValue. This rule allows querying artifacts based on user-defined metadata fields or tags.

- `tagValue` can be a keyword, number, or exact text, offering flexibility in defining tag values.

- `expression` can be a `numericalExpr` or a `metricExpr`. This is used for forming complex expressions involving metrics or numerical tags.

- `metricExpr` includes a `METRIC_ID` followed by an operator and a number. This facilitates artifact queries based on specific metric values and their relationships.

- `numericalExpr` includes a `NUMERICAL_TAG_ID` followed by an operator and a number, allowing comparison of numerical tag values.

- `TAG_ID` and `NUMERICAL_TAG_ID` are predefined identifiers for tags, while `METRIC_ID` is for metrics. `NUMERICAL_TAG_ID` is made of metadata fields that are quantitative such as the size and dates. The `TAG_ID` are simply the metadata fields of their artifacts such as filenames, description, content, extension types,etc. On the other hand, `METRIC_ID` is designed for metrics, representing identifiers that illustrate various quality metrics and attributes.

- `NUMBER` is a terminal symbol denoting numbers. A number is defined as one or more digits.

- `KEYWORD` is a nonQuoteSpace followed by zero or more nonQuoteSpaces. A nonQuoteSpace is any character excluding single or double quotes and spaces.

- `SPACE` is a terminal symbol representing a space character, which servers a delimiter across different components in the query but also can be interpreted as `OR` operator.

- `QUOTE` represents single or double quote characters used to encapsulate exact text.

- `EOF` is the symbol that denotes the end of input. This ensures that the parser processes the entire input string, and no unprocessed input remains.

It is worth noting that Fig. 6.11 presents a simplified and explanatory version of the grammar language. The tokens for tags and metrics are dynamically generated based on the currently available metrics and indexes. This means that if new metrics are developed or new indexes added to the platform, they automatically become accessible for query specification.

Figure 6.12 provides a use case where a user can enter keywords into the search box to find relevant models. The user can also specify a microsyntax query specification to refine the

Fig. 6.12 Search results.

search results. The results are displayed in a list. The user can click on each result to view the details of the model. The details include information (c.f.fig. 6.7 (iv)) related to megamodel data, data generated by the service, logs and artifact extracted metadata such as artifact name, author, date, description, and a link to download the artifact. As an example, in fig. 6.12, we are looking for artifacts with a size greater than five kilo-bytes that do not have the XMI extension. Or the quality metric number of concrete classes (CMC) is equal to 2, and its average attributes in a class are greater or equal to 2. We can check if it contains *keyword1* and conforms to the metamodel specified by *unique name* or *has an attribute* name and *can be transformed* to the metamodel specified by the ID.

The microsyntax query can get quite complex when nesting conditional statements, search tags or keywords. Conditional clauses can be nested by wrapping the statement into parenthesis. This approach is generic to the technological format of the artifact. Conditional statements can accept logical operators such as great than, greater or equal than, less than and less or equal than (c.f. fig. 6.9). In this manner, the user can query artifacts by specifying thresholds of a given quality metric or attribute. Although the microsyntax query specification can be used to perform an exhaustive search, it can get quite complex when nesting many statements. Hence, at this point, it might be easier to use the advanced search, as discussed in the next section. More tips are provided on the web interface on retrieving exact matches and special characters and how to escape them. We also instruct how to perform fuzzy searches or the usage of wildcards.

Advanced searches Figure 6.13 depicts our advanced search integration in MDEForge-Search to enable the user to perform exhaustive searches of documents based on a variety of factors. This allows users to tailor their search contextually and find what they want without having to wade through irrelevant results. The user can select several artifact

to narrow the search scope. The fields and quality metrics can be added and removed dynamically (c.f. fig. 6.13). In addition, results can be retrieved based on creation or update timestamps. The advanced search integration employs a combination of the microsyntax query specification, the quality assessment service [21], and the optimal transformation chain of a given metamodel to find relevant artifacts [17], as detailed in Sec. 6.4. The microsyntax query specification can be used against artifact fields as well. This allows users to target specific fields that make up an artifact to retrieve documents based on metadata such as model type, size, and complexity.

Fig. 6.13 Advanced search

Browsing facilities   Another discovery mechanism provided by our repository is a listing directory that contains all artifacts found in our repository. These artifacts can be browsed alphabetically and filtered by date or type (c.f.fig. 6.14). The user can click on each artifact to view its dedicated detail page. The selected artifact page contains information such as name, description, access control, and download URL (c.f. fig. 6.15). For each selected artifact, the user can explore and edit the content with a built-in editor or EMF.cloud before their reuse (c.f. fig. 6.16).

Fig. 6.14 Browsing page

Fig. 6.15 View page of artifact and its metadata

The integrated editor allows the user to edit selected artifacts. The editor displays EMF artifacts as document trees as shown inFigure 6.16.
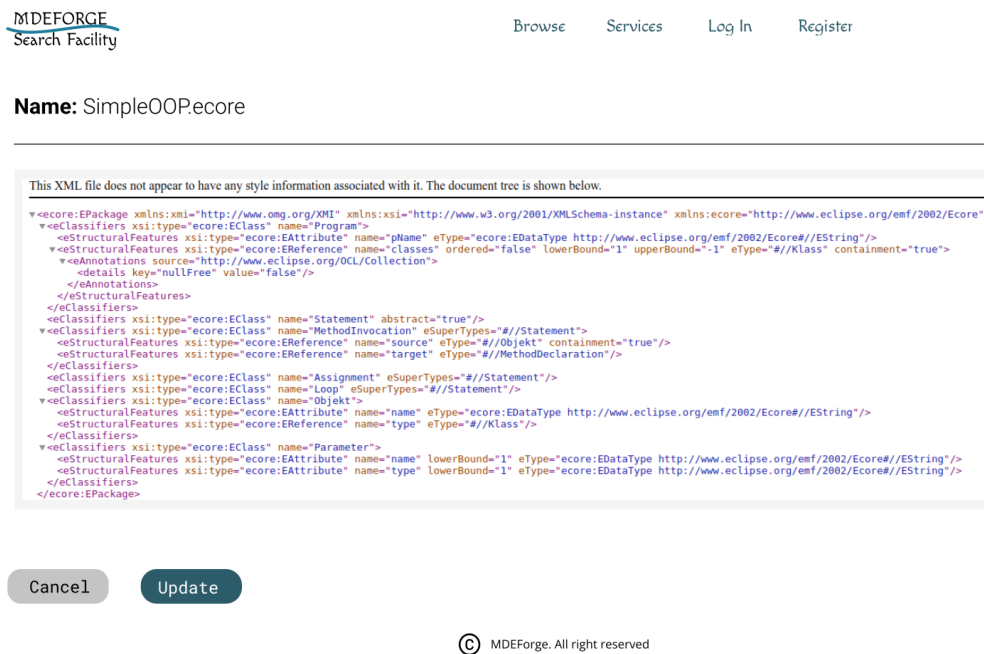

Fig. 6.16 Artifact editor

Platform reuse mechanisms

MDEForge-Search incorporates model management operations (MMO) as services. Hence as shown in fig. 6.17, we have deployed these MMO engines to facilitate the reuse of artifacts persisted in the repository [116]. We developed a playground where the user can edit and manipulate MMOs. The playground is pivotal in reusing artifacts from the repository and the services and extensions deployed there. These operations include but are not limited to model editing, consistency checking, object query, validation, comparison, and transformation. Retrieved artifacts can be further explored and navigated using model object query languages or participating in other MMO at the repository. These services consume artifacts by their specific identifiers from the repository. Besides, the user can upload the file directly from her local machine. We have put at the user's disposal a console to log the execution results of these services (c.f. fig. 6.17).

Integrated services in discovery mechanisms

In this subsection, we present two integrated services we have added atop the core services of the platform and that have been successfully applied to include quality attributes and megamodels relationships in the proposed query mechanism.

Quality assessment service   Software Quality Engineering [121] is a discipline that focuses on improving the approach to software quality. In MDE, quality artifacts are beneficial to identifying quality attributes of interest for specific stakeholders. To enhance the proposed query mechanism with the support of quality measures, we proposed artifact-tailored micro-services to compute the quality measurements defined in [21]. In particular, the authors proposed a generic approach to define and compute quality scores. It includes a DSL to define how quality attributes and metrics may be aggregated. In addition, they implemented an operative environment to apply the defined quality attributes on actual model artifacts enabling automated quality assessment. In our work, a quality assessment service for each supported artifact is provided that takes in a modeling artifact and returns the list of quality scores supported in the underlining quality model specification as output. It is worth noting that once a new quality model has been updated on the quality service, the quality measurements have to be recomputed to be indexed from the query engine.

Transformation chain service   The service for model transformation chain addresses challenges that arise in complex model-driven engineering scenarios where models need to be transformed and manipulated across different representations [17]. The main goal of this service is to identify and optimize the most suitable transformation chain for a given pair of source and target models in a model repository. It is especially helpful when there are multiple transformation options available between input and output models. By taking into account factors such as transformation coverage, information loss, and the number of transformation

Fig. 6.17 Model management services available on the platform.

hops [17], the service assists modelers in determining the optimal chain. Furthermore, it optimizes the execution of the selected chain by efficiently rewriting transformation queries and estimating dependencies between the meta-class and structural features of the involved meta-models in the model transformation process, and then chaining them up [68].

This service proves to be valuable in various contexts of model-driven engineering, particularly in model-based systems, where it facilitates the translation and integration of models across diverse representations. In such systems, the composition of model transformations enables developers to combine and reuse existing models and components, automating the creation of new software systems, improving business processes, and aiding product design across industries like software engineering, business process management, and product design. The automation provided by this service enhances the efficiency and effectiveness of model-driven development efforts. In our approach, this service is consumed as a black box as part of the EU ITN Lowcomote project[13], which is the same context where MDEForge-Search has been devised.

6.5   Applications

Integration of MDEForge with the Droid recommender framework

This section shows the integration of MDEForge-Search into the DROID [7] framework. DROID is a textual DSL that automates the configuration, evaluation and synthesis of recommender systems(RS) for particular modeling languages. RSs are information filtering systems that guides users in selecting items among a potentially large set [4]. DROID allows the configuration of every aspect of an RS, such as the definition of target and items, their corresponding identifiers, pre-processing techniques, recommendation methods, splitting techniques and evaluation protocol. It is important to clarify that DROID is not a component of MDEForge-Search; rather, it utilizes MDEForge-Search as a data source. Additionally, DROID has been developed within the framework of the EU ITN Lowcomote project, similar to MDEForge-Search and the transformation chain service discussed in the previous section.

The motivation behind the creation of DROID is to provide a model-driven solution that facilitates the creation and evaluation of RSs for modeling languages. Existing RSs for modeling are often hardwired to specific modeling languages, which makes it difficult to reuse them for different languages. DROID aims to overcome this limitation by allowing the configuration of every aspect of an RS, such as the definition of target and items, their corresponding identifiers, pre-processing techniques, recommendation methods, splitting techniques, and

---

[13]https://www.lowcomote.eu/

evaluation protocol. This way, RSs can be easily created, evaluated, and reused for different modeling languages without requiring deep knowledge of RSs or programming.

By integrating MDEForge-Search, DROID gains access to a robust search engine that can effectively retrieve models and metamodels. This is particularly relevant since finding relevant metamodels for a particular recommendation system creation case can be difficult. MDEForge-Search allows users to retrieve relevant models and metamodels using a micro syntax to query the search engine. Users can leverage the search capabilities of MDEForge-Search to easily find and retrieve models and metamodels that fit their specific needs for building recommendation systems. This integration enables DROID to improve the quality of recommenders and facilitate the creation of recommendation systems for a broader range of modeling languages.

To create a recommender system for a specific modeling language using DROID, the initial step is to create a DROID project, which necessitates defining a set of information (c.f. Fig. 6.18 part ①). The recommender system developer must specify the name of the recommender and the technology that the recommender system will serve. DROID supports the creation of RSs for both meta-modeling (e.g. Ecore) and modeling (e.g. UML, XMI). The developer can also opt to use the default recommendation settings or create a pre-filled template with a customized set of recommendation settings. The default settings can be particularly useful for those who lack experience in building RSs.

Additionally, in order to train and test the RSs, data must be provided (see Fig. 6.18 part ②). DROID allows the extension of data collection sources via extension points. In this integration, we extended DROID with advanced query mechanisms provided by MDEForge-Search. With this integration, we can execute a query to search for relevant (meta-)models to be used as training and testing data for a new RSs, providing an efficient and effective way to gather the necessary data (see Fig. 6.18 part ③).

This new project will finally require the definition of the target and items to be the subject of the recommendations, in addition to identifiers for each element. The configuration related to the pre-processing techniques, splitting settings, recommendation methods and evaluation protocol can be left with the default configuration or can be modified as desired (c.f. Figure 6.18 part ④). The DSL includes a textual editor with code compilation, a validator and a code generator that synthesizes Java code (c.f. Figure 6.18 part ⑤) from the project specification. By using the recommendation methods, multiple RSs can be trained, evaluated, and compared at the same time. These RSs can then be deployed on DroidREST, which is a generic recommendation service that provides recommendations based on configuration

files generated by DROID (see Figure 6.18, part ⑥). This process ensures that the RSs are optimized and provide the best possible recommendations for the given modeling language.

Once deployed, clients can obtain recommendations by sending a POST request to DroidREST, with the recommender's name and a JSON file that specifies the recommendation target and context. Recommenders can be integrated with Eclipse Modeling Framework's tree editors to provide a pop-up menu for recommendations on objects and can also be added to other external modeling tools.
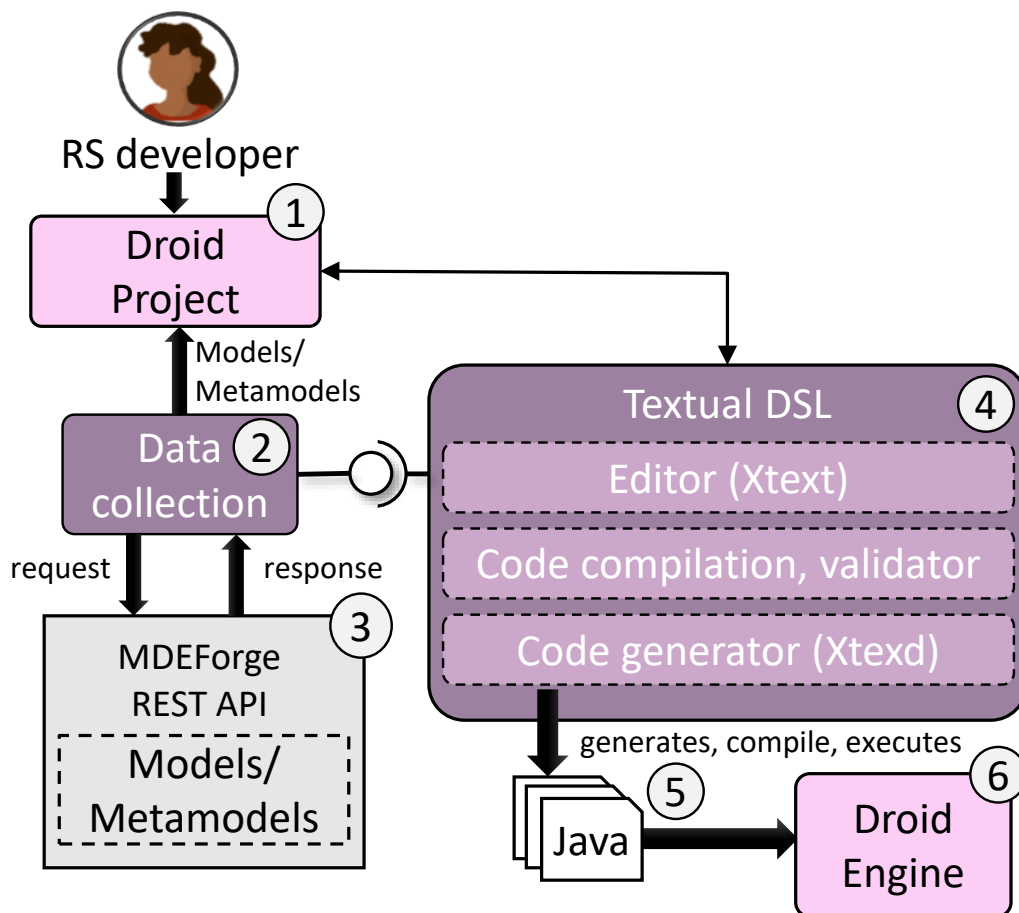
Fig. 6.18 Architecture of DROID with MDEForge-Search

Figure 6.19 shows an example of data collection using the advanced query mechanism engine of MDEForge-Search. In the example, the recommender system developer is creating an recommender system for Ecore meta-modeling. The micro-syntax to query the search engine can be specified using keywords, boolean operators (e.g. *AND*, *OR* and *NOT*), meta-models size and quality metrics to constrict or expand the search. A JSON request is sent to the MDEForge-Search API by pressing the search button.
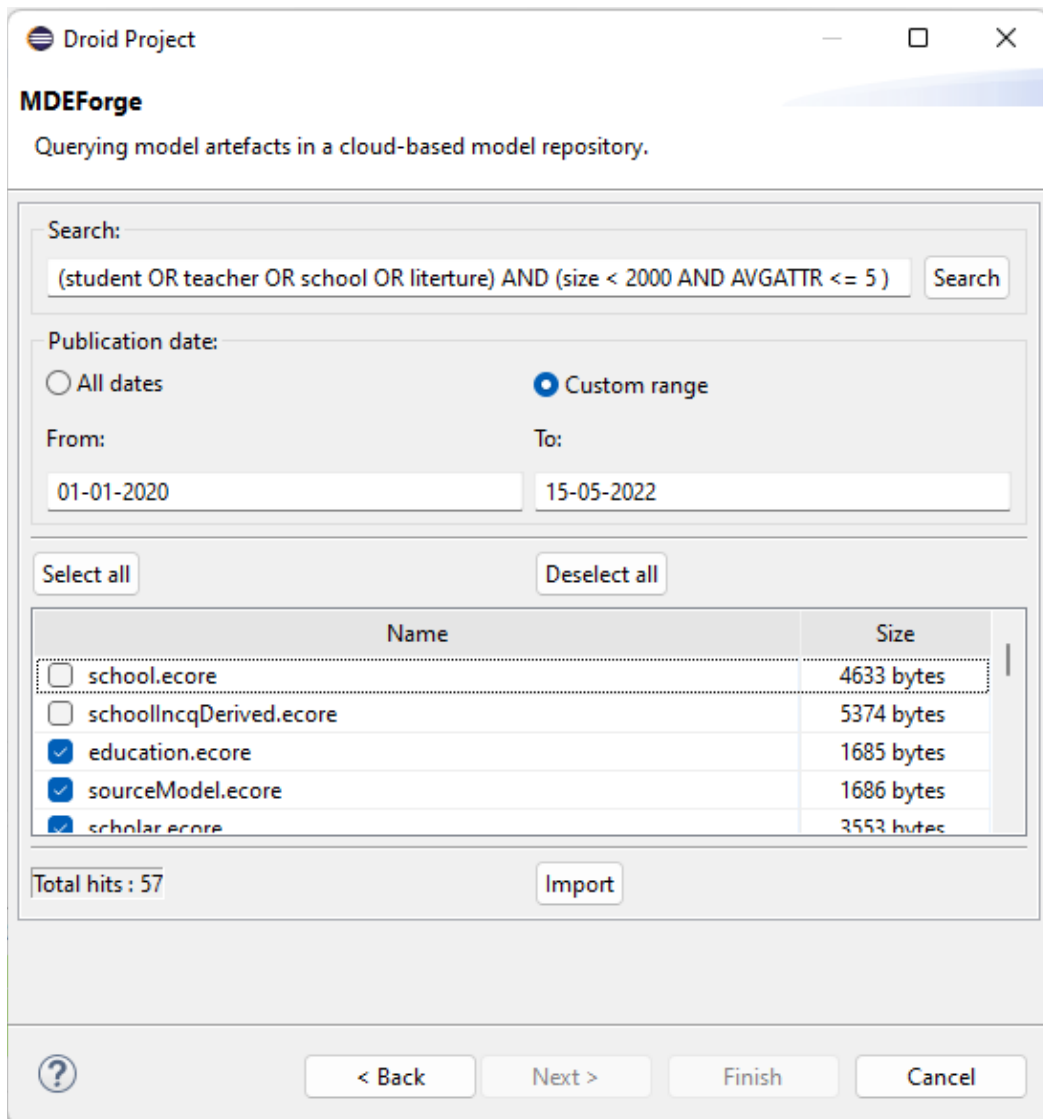
Fig. 6.19 DROID wizard page for the advanced query mechanism.

The returning lists of artifacts are shown in the table viewer. The table presents each artefact's name, extension, size, and total hits. In this example, the query is automatically constrained to Ecore meta-models extensions as the user specified this information on the main page of the wizard. Afterwards, the user can select the desired meta-models and import them into the project by pressing the import button. Finally, the recommender system developer can finish the wizard, creating a DROID project with the collected data.

6.6   Discussion

As highlighted in our research protocol (c.f. Section 5.2.1), we made every effort to ensure the accuracy and validity of our research. In addition, we followed established guidelines

for systematic studies [164, 222], yet it is possible that certain external or internal factors influenced our results. Below, we discuss potential threats to the validity of our research and strategies adopted to mitigate them.

To avoid generalizing causal findings concerning the settings and tools considered, we endeavoured to select all work that represents state of art to date in advanced discovery mechanisms in model repositories in model-driven engineering. We mitigate these external threats by searching five prominent computer science and software engineering databases with comprehensive keywords to ensure broad coverage of related literature (c.f. Section 5.2.1). We have tried to cover all keywords related to the modeling domain. In addition, we further extended our search for relevant articles by manually snowballing. Finally, we took steps to exclude literature not written in English.

Our research study had three main objectives: (1) to identify and evaluate existing tools and approaches for MDE artifact discovery, (2) to identify gaps in current MDE discovery tools and approaches, and (3) to develop an approach for comprehensive, next-generation advanced discovery mechanisms for MDE artifacts that address the identified challenges and gaps. We knew the need to maintain internal validity in designing our research study. Therefore, during the selection process of the tools and approaches, we thoroughly analyzed and considered the literature surrounding each tool and approach. However, one challenge was that most of these tools were unavailable online or for local use. Nonetheless, we worked toward developing a comprehensive discovery mechanism for MDE artifacts that would meet the needs of today's technologically advanced, Big Data-driven world.

6.7 Conclusion

This paper presented a service-oriented megamodel-aware approach to discovering and reusing modeling artifacts stored in a cloud-based model repository. The approach is generic to support the heterogeneous nature of the repository and artifacts. The platform introduces a service-oriented discovery mechanism in the quest for relevant artifacts. This way, the user can retrieve artifacts that meet quality thresholds or participate in a given optimal model transformation chain. The user has at her disposal several query mechanisms to sift through vast information and find exactly what she is looking for using intuitive and easy-to-use methods. With our domain-specific microsyntax query specification, the user can retrieve artifacts by search tags, search keywords, and conditional statements. Other discovery mechanisms, such as advanced search and browsing, are also in place to aid in effective model artifact lookup and filtering.

Our approach supports full-text search of the artifacts that are automatically indexed and persisted across multiple nodes to ensure high data availability, query speed, large data load resilience, and query flexibility. In addition, we have provided a modular API that can be accessed using OpenAPI 3.0 and GraphQL specifications to reuse or extend current functionality in developers' applications. For example, a model-driven recommender system already uses the platform API to train and evaluate machine learning models. As future work, we plan to apply the proposed approach in different integration scenarios other than that performed with Droid.

Chapter 7

Conclusion

Providing reliable access, persistence, discovery, and reusability mechanisms is of crucial importance in light of the rapid evolution and prevalence of low-code development platforms and the challenges posed by the current digital revolution. Furthermore, as LCDPs stem from model-driven engineering principles, current modeling prospects highlight the need for an integrated platform with first-rate capabilities regarding significant data prospects and a collaborative user community. To best respond to the numerous challenges described in Chapter 1 and devise practical solutions, we aimed to develop a scalable and extensible cloud-based low-code model repository. The repository was envisioned not only as a storage fortress but a platform that harbors modeling approaches, provides efficient persistence, discovery, and reuse, and finally enables model management as a service.

This chapter is organized as follows: Section 7.1 discusses the summary contribution of the thesis. Section 7.2 presents the publications produced during the course of the program, whereas Section 7.3 showcases the research supporting tools developed. The tools' documentation and installation processes are described in the appendices [A, B]. Finally, we conclude the chapter in Section 7.4 highlighting the potential prospects of this research.

7.1 Contribution summary

This research aimed to address the challenges described in Chapter 1 by developing a repository that facilitates the management of model artifacts and tools. As a result, the endeavor produced a community-based model repository that supports the persistence, accessibility, discovery, and reuse of heterogeneous model artifacts and tools. The repository now boasts over 5,000 real-world model artifacts collected and organized for convenient access. In this regard, we summarize the main contribution of this work below:

A Comprehensive Overview of Low-Code Development Platforms (LCDPs): In recent years, the use of LCDPs has grown significantly in both academia and industry. Under-

standing and comparing hundreds of LCDPs can be strenuous and challenging without an appropriate conceptual framework underpinning their evaluation. Chapter 2 of this research focused on eight leading LCDPs to identify their commonalities and differences. To compare each platform, we performed a taxonomy where a set of distinguishing features was established. We introduced the intricacies of internal LCDPS information flow. Furthermore, we presented an experience report to discuss the essential features of each platform, limitations, and challenges encountered during the development of a benchmark application. As far as we are concerned, it was the first work that attempted to analyze different LCDPs according to a set of organized features.

Identification of challenges and opportunities toward cloud-based modeling: To develop data-intensive applications such as IoT, developers must overcome various challenges, including heterogeneity, complexity, and scalability. Moving development infrastructure to the cloud opens up opportunities regarding accessibility, productivity, maintenance, fault tolerance, and monitoring. In chapter 3, we present a systematic study we conducted to assess the current state of the art on cloud-based modeling approaches in the IoT domain. After examining 625 articles, we focus on 22 papers proposing cloud-based modeling environments in the IoT domain. The considered approaches have been analyzed to assess their strengths and weaknesses concerning many characteristics, including their modeling focus, accessibility, openness, and artifact generation. This chapter discussed many challenges IoT developers encounter while adopting such tools. We also discussed various generic technologies and tools which can be adopted in the IoT domain. These challenges were taken into account while designing the architecture of the repository.

Architectural design of a scalable, extensible, and community-oriented model repository: We designed the conceived repository using an extended "4 + 1" view model (c.f.Chapter 4). The objective was a well-thought-out architectural design that supports a set of extensible core services and tools on a scalable infrastructure. The result was a community-oriented cloud-based model repository that promotes access, discovery, and reuse of heterogeneous model artifacts and tools. In addition, the repository has mechanisms to ensure intellectual and technical exchange guided by policies and guidelines. By adopting a cloud-based deployment, the architecture of this repository eliminates tedious installations and configurations that need to be installed on complex modeling frameworks before their use. Now, users can leverage model management tools and artifacts via an API or web-based tools that are highly available, scalable, and easy to extend or integrate with other tools.

Scalable and extensible persistence of heterogeneous artifacts Using a clustered database infrastructure, we designed, developed, and published mechanisms to persist heterogeneous

model artifacts in a repository. This infrastructure was designed with scalability, high availability, and disaster recovery built in. The persistence API can be consumed using OpenAPI and GraphQL API specifications. The current data organization could be easily leveraged in prescriptive, descriptive, and predictive analytics, such as in the case of Droid [7]. Our approach ensured that users could quickly retrieve data on demand and minimize downtime by accommodating large workloads without compromising performance.

Model management operations as services Although model management operations are essential when developing complex systems using MDE practices, their reuse is regularly hindered by the numerous local configurations, installations, dependency downloads, and expertise required. To alleviate these difficulties, these operations - namely model transformations, validation, comparison, and merging - have been packaged into containerized services to be remotely consumed over the Internet. Now users no longer have to concern themselves with the minutiae as they can request these operations to be executed upon providing the required inputs. Hence, we increased the reusability of both the artifacts and their management operations by relieving the low-level details preoccupation from the user.

Composition, discovery, and orchestrations of model management services and tools Adopting model management operations on cloud-based repositories poses several challenges regarding their composition, discovery, and orchestration. To address these issues, we introduced a low-code development environment that facilitates the orchestration and integration of various services required for complex workflows (c.f. 5). This approach lets developers plan, organize, specify, compose, and execute model management workflows. In addition, a complete cloud-based infrastructure is provided to support the composition and execution of remotely available MMO services. These services are orchestrated and managed in a self-healing, auto-scaling, highly available cloud-based cluster that is easy to manage and monitor. This implementation provides an MDEForgeWL DSL and supporting engine for defining and executing user-defined workflows of model management services. A service registry registers new services and makes them discoverable for workflow definitions is also provided.

Advanced discovery and reuse mechanisms for a cloud-based model repository: Our service-oriented megamodel-aware approach for discovering and reusing model artifacts in a cloud-based model repository is comprehensive. It provides multiple mechanisms that are both intuitive and easy to use (c.f.chapter 6). For instance, our domain-specific microsyntax query specification allows users to search for relevant artifacts using search tags, keywords, conditional statements, and logical operators. Furthermore, advanced search and browsing features enable effective filtering of persisted artifacts. Additionally, the platform supports

full-text search with automatic indexing across multiple nodes to ensure data availability, query speed, and resilience in the face of large data loads. Moreover, the API provided is modular and can be leveraged via OpenAPI 3.0 and GraphQL specifications for reuse or extension by developers. In this way, the dataset pre-processed and organized for remote consumption can be used in data analytics - as demonstrated by an existing model-driven recommender system already making use of the platform's API capabilities [7] (c.f.Chapter 6.5).

7.2   Publications

The research has been widely disseminated in various academic channels, such as peer-reviewed conferences, workshops, and journals. We have also produced several reports. This program was conducted in the context of the EU Lowcomote project[1], and disseminating the results allowed practitioners to stay abreast of the latest developments. More importantly, the findings have allowed us to evaluate and identify new opportunities for further research.

Below is the list of publications produced during this doctoral program:

- *Supporting the understanding and comparison of low-code development platforms.* Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, Alfonso Pierantonio, 2020. 46th Euromicro Conference on Software Engineering and Advanced Applications (SE2A 2020)

- *A Low-Code Development Environment to Orchestrate Model Management Services.* Arsene Indamutsa, Davide Di Ruscio, Alfonso Pierantonio, Sep. 2021. IFIP International Conference on Advances in Production Management Systems

- *Cloud-based modeling in IoT domain. A survey, open challenges, and opportunities.* Jean Felicien Ihirwe, Arsene Indamutsa, Davide Di Ruscio, Silvia Mazzini, Alfonso Pierantonio, Oct. 2021. 2nd Low-code Workshop at the ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS 2021)

- *MDEForgeWL. Towards cloud-based discovery and composition of model management services.* Arsene Indamutsa, Juri Di Rocco, Davide Di Ruscio, Alfonso Pierantonio, Oct. 2021. MODELS 2021. ACM/IEEE 24th International Conference on Model-Driven Engineering Languages and Systems

- *Advanced discovery mechanisms in model repositories.* Arsene Indamutsa, Juri Di Rocco, Lissette Almonte, Davide Di Ruscio, Alfonso Pierantonio. Software: Practice and Experience (2022) - Under Review, Manuscript ID: SPE-22-0516.

---

[1]https://www.lowcomote.eu/

7.3   Developed tools

We produced the tools mentioned below while addressing issues tackled in this thesis:

- *Persistence API*: this API provides users with a suite of optimized functions for creating, reading, updating, and deleting repository contents. This API is conveniently configured to operate in conjunction with existing data sources and can be deployed as a single unit onto the desired environment. Moreover, because it makes use of the Swagger OpenAPI 3.0[2] and GraphQL[3] API specifications, it can easily be exposed to the public through these open-access standards.

  Code repository: `https://github.com/Indamutsa/model-management-services.git`

- *Search and discovery API*: This API provides users with capabilities that allow them to navigate the repository through advanced discovery mechanisms, such as the combinations of high-level microsyntax query specifications to narrow down their search. The API can be accessed using the open-access standards mentioned above.

  Code repository: `https://github.com/Indamutsa/advanced-query.git`

- *Model management operations remote APIs*: Model management operations can now be executed on demand remotely. These operations were containerized and deployed in a cloud-based cluster to enable the reuse of these operations without exhausting local configurations, dependency downloads, and installations.

  Code repository: `https://github.com/Indamutsa/model-management-services.git`

- *MDEForgeWL*: this Domain Specific Language (DSL) leverages a cluster of model management services(MMSs) APIs to enable users to discover, define and execute task workflows composed of these services. Through this trigger-action paradigm, users can plan, organize, customize, and execute an arbitrary model-driven task in a workflow manner to achieve their desired goals.

  Code repository: `https://github.com/Indamutsa/model-management-services.git`

- *MDEForge Search*: It is a platform that provides advanced discovery and reuse mechanisms in our cloud-based model repository. A search engine is incorporated into the ecosystem to enable indexing and fast retrieval of artifacts. This discovery is enhanced by using a microsyntax query specification that allows us to use search keywords, tags, conditional statements, and logical operators to navigate the repository and narrow our search. The user can also browse by category of the artifacts. The platform also has a

---

[2]https://www.openapis.org/
[3]https://graphql.org/

playground of model management services where the user can reuse retrieved artifacts without needing local configurations or installation.

Code repository: `https://github.com/Indamutsa/advanced-query.git`

Their respective installations and demo highlights can be found in the appendices AB.

7.4   Future work

This thesis investigated the potential of fostering software development efficiency through a cloud-based low-code model repository. This work conceptualized an approach for collecting and managing heterogeneous artifacts and making them available for reuse throughout the entire software development lifecycle. In addition, it provided insights into identifying a set of features required for extracting newly acquired knowledge from models, discovering relevant artifacts, and automating the implementation process, ultimately streamlining the development process.

Below are some of the aspects that can be explored further:

- **Machine learning:** This repository can be leveraged by machine learning models to enhance bug detection, model recommendations, and code completion. Besides, the data and user behavior can also be leveraged to understand trends and guide innovation.

- **Education and research:** This tool can quickly become a potential resource for students and researchers in model-driven engineering and low-code development platforms. However, extending the repository with appropriate interfaces is needed to enable a collaborative and engaging experience with artifacts and tools.

- **Software development:** developers can leverage the repository to enhance productivity and migrate their modeling infrastructure to the cloud. Thus, they can easily collaborate and manage their versioned artifacts without manual local configuration and installations. To achieve this goal, some tools must be developed on top of the current implementation, such as a versioning server and additional interfaces, preferably drag-and-drop features for citizen developers.

- **Simulation and optimization:** Complex systems such as physical systems, manufacturing processes, or supply chains can leverage the repository to store and manage their models while simulating and optimizing their workflow. In this case, the repository is a safe playground for engineers to understand and optimize these systems, leading to improved efficiency and performance.

- **Model transformation and refinement:** The model management operations on the repository can be used to derive low-level and reusable code that will integrate into the software development lifecycle.

- **Integration with external tools:** We have made room for external services and tools that perform recommendations, DevOps, and testing frameworks. However, having these tools directly integrated into the system can prove handy. We can also fully integrate EMF cloud [4] in the repository and leverage the artifacts already persisted.

- **Big data:** Although the repository has automated dataset curating mechanisms, it can be extended to introduce adapters for various extract, transform, and load ETL strategies that cover various domains. Our orchestrated storage can be leveraged to ensure standardization, permission control, and comprehensive data examination.

---

[4]https://www.eclipse.org/emfcloud/

References

[ela] Lucene's practical scoring function: Elasticsearch: The definitive guide [2.x]. https://www.elastic.co/guide/en/elasticsearch/guide/current/practical-scoring-function.html.

[2] (2017). Model-driven development of user interfaces for IoT systems via domain-specific components and patterns. *Journal of Internet Services and Applications*, 8(1):14.

[3] Acerbis, R., Bongio, A., Brambilla, M., and Butti, S. (2015). Model-driven development of cross-platform mobile applications with web ratio and ifml. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, pages 170–171. IEEE.

[4] Adomavicius, G. and Tuzhilin, A. (2005). Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17:734–749.

[5] Akdur, D., Garousi, V., and Demirörs, O. (2018). A survey on modeling and model-driven engineering practices in the embedded software industry. *Journal of Systems Architecture*, 91:62–82.

[6] Al Alamin, M. A., Malakar, S., Uddin, G., Afroz, S., Haider, T. B., and Iqbal, A. (2021). An empirical study of developer discussions on low-code software development challenges. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 46–57. IEEE.

[7] Almonte, L., Pérez-Soler, S., Guerra, E., Cantador, I., and de Lara, J. (2021). *Automating the Synthesis of Recommender Systems for Modelling Languages*, pages 22—-35. Association for Computing Machinery, New York, NY, USA.

[8] ALSAADI, H. A., RADAIN, D. T., ALZAHRANI, M. M., ALSHAMMARI, W. F., ALAHMADI, D., and FAKIEH, B. (2021). Factors that affect the utilization of low-code development platforms: survey study. *Romanian Journal of Information Technology and Automatic Control*, 31(3):123–140.

[9] Alvarez, C. and Casallas, R. (2013). MTC Flow: A tool to design, develop and deploy model transformation chains. *ACadeMics Tooling with Eclipse, ACME 2013 - A Joint ECMFA/ECSA/ECOOP Workshop*.

[10] Antorini, Y. M. and Muñiz, A. M. (2013). The Benefits and Challenges of Collaborating with User Communities. *Research-Technology Management*, 56(3):21–28.

[11] Appian (2020). Appian platform overview. https://www.appian.com/. Accessed on March 23, 2020.

[12] Arundel, J. and Domingus, J. (2019). *Cloud Native DevOps with Kubernetes*.

[13] AtmosphereIoT (2020). Fast time to first data. https://atmosphereiot.com/. Last accessed May 2020.

[14] Barmpis, K. (2016). Towards scalable model indexing. *Doctoral thesis*.

[15] Barmpis, K. and Kolovos, D. (2013). Hawk: Towards a scalable model indexing architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 6:1–6:9, New York, NY, USA. ACM.

[16] Barmpis, K. and Kolovos, D. S. (2014). Towards scalable querying of large-scale models. In *Modelling Foundations and Applications*, pages 35–50, Cham. Springer International Publishing.

[17] Basciani, F., D'Emidio, M., Di Ruscio, D., Frigioni, D., Iovino, L., and Pierantonio, A. (2018a). Automated selection of optimal model transformation chains via shortest-path algorithms. *IEEE Transactions on Software Engineering*, 46(3):251–279.

[18] Basciani, F., Di Rocco, J., Di Ruscio, D., Di Salle, A., Iovino, L., and Pierantonio, A. (2014). MDEForge: An extensible Web-based modeling platform. *CEUR Workshop Proceedings*, 1242(September):66–75.

[19] Basciani, F., Di Rocco, J., Di Ruscio, D., Iovino, L., and Pierantonio, A. (2015a). Model repositories: Will they become reality? In *CloudMDE MoDELS*, pages 37–42.

[20] Basciani, F., Rocco, J. D., Ruscio, D. D., Iovino, L., and Pierantonio, A. (2015b). Model repositories: Will they become reality? 1563:37–42.

[21] Basciani, F., Rocco, J. D., Ruscio, D. D., Iovino, L., and Pierantonio, A. (2019). A tool-supported approach for assessing the quality of model artifacts. *J. Comput. Lang.*, 51:173–192.

[22] Basciani, F., Ruscio, D. D., Rocco, J. D., Iovino, L., and Pierantonio, A. (2018b). Exploring model repositories by means of megamodel-aware search operators. *CEUR Workshop Proceedings*, 2245:793–798.

[23] Basso, F. P., Oliveira, T. C., Werner, C. M., and Becker, L. B. (2017). Building the foundations for 'mde as service'. *IET Software*, 11(4):195–206.

[24] Bats, M. and Begaudeau, S. (2021). Sirius web: 100% open source cloud modeling platform. https://www.eclipsecon.org/2020/sessions/sirius-web-100-open-source-cloud-modeling-platform. 2021-7-1.

[25] Benac, R. and Mohd, T. K. (2021). Recent trends in software development: Low-code solutions. In *Proceedings of the Future Technologies Conference*, pages 525–533. Springer.

[26] Berardinelli, L., Mazak, A., Alt, O., Wimmer, M., and Kappel, G. (2017). *Model-Driven Systems Engineering: Principles and Application in the CPPS Domain*, pages 261–299. Springer International Publishing, Cham.

[27] Bettini, L. (2016a). *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

[28] Bettini, L. (2016b). *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

[29] Bezerra, J. D. H. and de Souza, C. T. (2019). A model-based approach to generate reactive and customizable user interfaces for the web of things. In *Proceedings of the 25th Brazillian Symposium on Multimedia and the Web*, WebMedia '19, page 57–60, New York, NY, USA. Association for Computing Machinery.

[30] Bézivin, J., Jouault, F., Rosenthal, P., and Valduriez, P. (2004). Modeling in the large and modeling in the small. In *Model Driven Architecture*, pages 33–46. Springer.

[31] Biolchini, J., Mian, P. G., Natali, A. C. C., and Travassos, G. H. (2005). Systematic review in software engineering. *System Engineering and Computer Science Department COPPE/UFRJ, Technical Report ES*, 679(05):45.

[32] BISE Institute, JKU, L. (2020). Devopsml. https://github.com/lowcomote/devopsml/tree/1.2.2, last accessed on 28/08/20.

[33] Bislimovska, B., Aluç, G., Özsu, M. T., and Fraternali, P. (2015). Graph search of software models using multidimensional scaling. *CEUR Workshop Proceedings*, 1330:163–170.

[34] Bislimovska, B., Bozzon, A., Brambilla, M., and Fraternali, P. (2012). Search upon uml repositories with text matching techniques. *2012 4th International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation, SUITE 2012 - Proceedings*, pages 9–12.

[35] Board, I.-S. S. (2000). Ieee recommended practice for architectural description for software-intensive systems. *IEEE Std 1471-2000*, pages 1–30.

[36] Bock, A. C. and Frank, U. (2021a). In search of the essence of low-code: an exploratory study of seven development platforms. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 57–66. IEEE.

[37] Bock, A. C. and Frank, U. (2021b). Low-code platform. *Business & Information Systems Engineering*, 63(6):733–740.

[38] Borelli, F. F., Biondi, G. O., and Kamienski, C. A. (2020). Biota: A buildout iot application language. *IEEE Access*, 8:126443–126459.

[39] Boubiche, S., Boubiche, D. E., Bilami, A., and Toral-Cruz, H. (2018). Big data challenges and data aggregation strategies in wireless sensor networks. *IEEE Access*, 6:20558–20571.

[40] Braams, S. (2017). Developing a software quality framework for low-code model driven development platforms based on behaviour driven development methodology.

[41] Bradshaw, S., Brazil, E., and bradshaw2019mongodb, K. (2019). *MongoDB: the definitive guide: powerful and scalable data storage*. O'Reilly Media.

[42] Brambilla, M., Cabot, J., and Wimmer, M. (2017). Model-driven software engineering in practice. *Synthesis lectures on software engineering*, 3(1):1–207.

[43] Breaux, T. and Moritz, J. (2021). The 2021 software developer shortage is coming. *Communications of the ACM*, 64(7):39–41.

[44] Brosch, P., Langer, P., Seidl, M., and Wimmer, M. (2009). Towards end-user adaptable model versioning: The by-example operation recorder. In *Procs.of CVSM '09*, pages 55–60, Washington, DC, USA. IEEE Computer Society.

[45] Bruneliere, H., Cabot, J., and Jouault, F. (2010). Combining model-driven engineering and cloud computing. In *Modeling, Design, and Analysis for the Service Cloud-MDA4ServiceCloud'10: Workshop's 4th edition (co-located with the 6th European Conference on Modelling Foundations and Applications-ECMFA 2010)*.

[46] Brunschwig, L., Guerra, E., and de Lara, J. (2020). Towards access control for collaborative modelling apps. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA. Association for Computing Machinery.

[47] Bucaioni, A., Cicchetti, A., and Ciccozzi, F. (2022). Modelling in low-code development: a multi-vocal systematic review. *Software and Systems Modeling*, 21(5):1959–1981.

[48] Bucchiarone, A., Cabot, J., Paige, R. F., and Pierantonio, A. (2020). Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, 19(1):5–13.

[49] Bucchiarone, A., Ciccozzi, F., Lambers, L., Pierantonio, A., Tichy, M., Tisi, M., Wortmann, A., and Zaytsev, V. (2021). What is the future of modeling? *IEEE Software*, 38(02):119–127.

[50] Béziv, J. (2004). On the need for megamodels. pages 1–9.

[51] Bézivin, J. (2005a). On the unification power of models. *Software and System Modeling*, 4:171–188.

[52] Bézivin, J. (2005b). On the unification power of models. *Software and Systems Modeling*, 4:171–188.

[53] Cabot, J. (2020). 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*, pages 1–3.

[54] Cansu, F. K. and Cansu, S. K. (2019). An overview of computational thinking. *International Journal of Computer Science Education in Schools*, 3(1):17–30.

[55] Carrascal Manzanares, C., Sánchez Cuadrado, J., and Lara, J. d. (2015). Building mde cloud services with distil. In *CEUR Workshop Proceedings*. CEUR-WS.

[56] Ceresani, N. (2016). The periodic table of devops tools v.2 is here. https://blog.xebialabs.com/2016/06/14/periodic-table-devops-tools-v-2/, last accessed on 28/08/20.

[57] Chang, Y.-H. and Ko, C.-B. (2017). A study on the design of low-code and no code platform for mobile application development. *International journal of advanced smart convergence*, 6(4):50–55.

[58] Chen, S., Xu, H., Liu, D., Hu, B., and Wang, H. (2014). A vision of IoT: Applications, challenges, and opportunities with China Perspective. *IEEE Internet of Things Journal*, 1(4):349–359.

[59] Chen, X., Liu, C., Shin, R., Song, D., and Chen, M. (2016). Latent attention for if-then program synthesis. *Advances in Neural Information Processing Systems*.

[60] Chen, X., Nophut, C., and Voigt, T. (2020). Manufacturing execution systems for the food and beverage industry: A model-driven approach. *Electronics*, 9(12).

[61] Colantoni, A., Berardinelli, L., and Wimmer, M. (2020). Devopsml: Towards modeling devops processes and platforms. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA. Association for Computing Machinery.

[62] Conti, M., Dehghantanha, A., Franke, K., and Watson, S. (2018). Internet of things security and forensics: Challenges and opportunities. *Future Generation Computer Systems*, 78:544–546.

[63] Cook, S., Jones, G., Kent, S., and Wills, A. C. (2007). *Domain-specific development with visual studio dsl tools*. Pearson Education.

[64] Cordasco, G., D'Auria, M., Negro, A., Scarano, V., and Spagnuolo, C. (2020). Toward a domain-specific language for scientific workflow-based applications on multicloud system. *Concurrency Computation*, (February).

[65] Corradini, F., Fedeli, A., Fornari, F., Polini, A., and Re, B. (2021). FloWare: An Approach for IoT Support and Application Development. In Augusto, A., Gill, A., Nurcan, S., Reinhartz-Berger, I., Schmidt, R., and Zdravkovic, J., editors, *Enterprise, Business-Process and Information Systems Modeling*, pages 350–365, Cham. Springer International Publishing.

[66] Cortes-Cornax, M., Dupuy-Chessa, S., and Rieu, D. (2012). Choreographies in bpmn 2.0: new challenges and open questions. In *Proceedings of the 4th Central-European Workshop on Services and their Composition, ZEUS*, volume 847, pages 50–57. Citeseer.

[67] Creator, Z. (2020). Zoho creator platform features. https://www.zoho.com/creator/features.html. Accessed on March 23, 2020.

[68] Cuadrado, J. S., Burgueno, L., Wimmer, M., and Vallecillo, A. (2020). Efficient execution of atl model transformations using static analysis and parallelism. *IEEE Transactions on Software Engineering*.

[69] Cueva-Fernandez, G., Espada, J. P., García-Díaz, V., García, C. G., and Garcia-Fernandez, N. (2014). Vitruvius: An expert system for vehicle sensor tracking and managing application generation. *Journal of network and computer applications*, 42:178–188.

[70] Czarnecki, K. (1999). *Generative programming - principles and techniques of software engineering based on automated configuration and fragment-based component models.* PhD thesis, Technische Universität Illmenau, Germany.

[71] Czarnecki, K. (2002). *Domain Engineering*, pages 433–444. American Cancer Society.

[72] David, O., Lloyd, W., Rojas, K., Arabi, M., Geter, F., Ascough, J., Green, T., Leavesley, G., and Carlson, J. (2014). Model-as-a-service (MaaS) using the Cloud Services Innovation Platform (CSIP). *Proceedings - 7th International Congress on Environmental Modelling and Software: Bold Visions for Environmental Modeling, iEMSs 2014*, 1:243–250.

[73] Developers, G. (2020). Google app maker platform guide. https://developers.google.com/appmaker/overview. Accessed on March 23, 2020.

[74] Di Rocco, J., Di Ruscio, D., Iovino, L., and Pierantonio, A. (2015). Collaborative repositories in model-driven engineering. *IEEE Software*, 32(3):28–34.

[75] Di Ruscio, D., Franzago, M., Malavolta, I., and Muccini, H. (2017). Envisioning the future of collaborative model-driven software engineering. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*, pages 219–221.

[76] Di Ruscio, D., Kolovos, D., de Lara, J., Pierantonio, A., Tisi, M., and Wimmer, M. (2022). Low-code development and model-driven engineering: Two sides of the same coin? *Software and Systems Modeling*, 21(2):437–446.

[77] Di Sandro, A., Salay, R., Famelis, M., Kokaly, S., and Chechik, M. (2015). Mmint: A graphical tool for interactive model management. In *P&D@ MoDELS*, pages 16–19.

[78] Dias, J. P., Restivo, A., and Ferreira, H. S. (2021). Empowering visual internet-of-things mashups with self-healing capabilities. *arXiv preprint arXiv:2103.07395*.

[79] Dixit, B., Kuc, R., Rogozinski, M., and Chhajed, S. (2017). *Elasticsearch: A Complete Guide*. Packt Publishing, Birmigham, Mumbai.

[80] Di Rocco, J., Di Ruscio, D., Härtel, J., Iovino, L., Lämmel, R., and Pierantonio, A. (2019). Understanding MDE Projects: Megamodels to the Rescue for Architecture Recovery. *Software and Systems Modeling*, 19(2):401–423.

[81] Dzulqornain, M. I., Harun Al Rasyid, M. U., and Sukaridhoto, S. (2018). Design and Development of Smart Aquaculture System Based on IFTTT Model and Cloud Integration. *MATEC Web of Conferences*, 164.

[82] El Khalyly, B., Banane, M., Erraissi, A., and Belangour, A. (2020). Interoevery: Microservice based interoperable system. In *2020 International Conference on Decision Aid Sciences and Application (DASA)*, pages 320–325.

[83] Farhan, L., Shukur, S. T., Alissa, A. E., Alrweg, M., Raza, U., and Kharel, R. (2017). A survey on the challenges and opportunities of the Internet of Things (IoT). *Proceedings of the International Conference on Sensing Technology, ICST*, 2017-Decem(December):1–5.

[84] Ferry, N., Nguyen, P., Song, H., Novac, P.-E., Lavirotte, S., Tigli, J.-Y., and Solberg, A. (2019). Genesis: Continuous orchestration and deployment of smart iot systems. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 870–875.

[85] Fortino, G., Savaglio, C., Spezzano, G., and Zhou, M. (2021). Internet of things as system of systems: A review of methodologies, frameworks, platforms, and tools. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 51(1):223–236.

[86] Foundation, E. (2021). 2020 annual eclipse foundation community report. hhttps://www.eclipse.org/org/foundation/reports/annual_report.php. Accessed on July 7th, 2021.

[87] Fowler, M. (2010). *Domain-specific languages*. Pearson Education.

[88] France, R., Bieman, J., and Cheng, B. (2007). Repository for model driven development (remodd). In *Models in Software Engineering*, volume 4364 of *LNCS*, pages 311–317. Springer Berlin Heidelberg.

[89] Fritzsch, J., Bogner, J., Wagner, S., and Zimmermann, A. (2019). Microservices migration in industry: intentions, strategies, and challenges. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 481–490. IEEE.

[90] Fryling, M. (2019). Low code app development. *Journal of Computing Sciences in Colleges*, 34(6):119–119.

[91] G2.com (2020). Best low-code development platforms software. https://www.g2.com/categories/low-code-development-platforms. Accessed on May 26th, 2020.

[92] Garcia, J. and Cabot, J. (2019). Stepwise adoption of continuous delivery in model-driven engineering. In Bruel, J.-M., Mazzara, M., and Meyer, B., editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 19–32, Cham. Springer International Publishing.

[93] Gašević, D., Kaviani, N., and Hatala, M. (2007). On Metamodeling in Megamodels. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4735 LNCS(December):91–105.

[94] Giang, N. K., Blackstock, M., Lea, R., and Leung, V. C. (2015). Developing iot applications in the fog: A distributed dataflow approach. In *2015 5th International Conference on the Internet of Things (IOT)*, pages 155–162.

[95] Gomes, P., Pereira, F. C., Paiva, P., Seco, N., Carreiro, P., Ferreira, J. L., and Bento, C. (2004). Using wordnet for case-based retrieval of uml models. *AI Communications*, 17(1):13–23.

[96] Gomes, P. M. and Brito, M. A. (2022). Low-code development platforms: A descriptive study. In *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–4. IEEE.

[97] González García, C., Pelayo G-Bustelo, B. C., Pascual Espada, J., and Cueva-Fernandez, G. (2014). Midgar: Generation of heterogeneous objects interconnecting applications. a domain specific language proposal for internet of things scenarios. *Computer Networks*, 64:143–158.

[98] Gormley, C. and Tong, Z. (2015). *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine.* " O'Reilly Media, Inc.".

[99] Grudin, J. (1990). The computer reaches out: The historical continuity of interface design. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 261–268.

[100] Guerriero, M., Tajfar, S., Tamburri, D. A., and Di Nitto, E. (2016). Towards a model-driven design tool for big data architectures. In *Proceedings of the 2nd International Workshop on BIG Data Software Engineering*, BIGDSE '16, page 37–43, New York, NY, USA. Association for Computing Machinery.

[101] Guo, Q., Lu, J., Zhang, C., Sun, C., and Yuan, S. (2020). Multi-model data query languages and processing paradigms. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, CIKM '20, pages 3505–3506, New York, NY, USA. Association for Computing Machinery.

[102] Harrand, N., Fleurey, F., Morin, B., and Husa, K. E. (2016). Thingml: A language and code generation framework for heterogeneous targets. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, page 125–135, New York, NY, USA. Association for Computing Machinery.

[103] Hebig, R., Seibel, A., and Giese, H. (2012). On the unification of megamodels. *Electronic Communications of the EASST*, 42.

[104] Hegedüs, Á., Bergmann, G., Debreceni, C., Horváth, Á., Lunk, P., Menyhért, Á., Papp, I., Varró, D., Vileiniskis, T., and Ráth, I. (2018). Incquery server for teamwork cloud: Scalable query evaluation over collaborative model repositories. *21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS-Companion 2018*, pages 27–31.

[105] Hein, C., Ritter, T., and Wagner, M. (2009). Model-driven tool integration with ModelBus. *Workshop Future Trends of Model-Driven \ldots*.

[106] Hinkson, I. V., Davidsen, T. M., Klemm, J. D., Kerlavage, A. R., and Kibbe, W. A. (2017). A comprehensive infrastructure for big data in cancer research: Accelerating cancer research and precision medicine. *Frontiers in Cell and Developmental Biology*, 5.

[107] Hjorth, M. (2017). Strengths and weaknesses of a visual programming language in a learning context with children. http://www.diva-portal.org/smash/record.jsf?pid=diva2

[108] Holmes, T., Zdun, U., and Dustdar, S. (2009). Morse: A model-aware service environment. In *2009 IEEE Asia-Pacific Services Computing Conference*, pages 470–477. IEEE.

[109] Holmes, T., Zdun, U., and Dustdar, S. (2012). Automating the Management and Versioning of Service Models at Runtime to Support Service Monitoring. In *EDOC*, pages 211–218.

[110] Hönel, S., Ericsson, M., Löwe, W., and Wingkvist, A. (2020). The journal of systems & software.

[111] Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411.

[112] Ibrahimi, I. and Moudilos, D. (2022). Towards model reuse in low-code development platforms based on knowledge graphs. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 826–836.

[113] Ihirwe, F., Indamutsa, A., Ruscio, D. D., Mazzini, S., and Pierantonio, A. (2021). Cloud-based modeling in iot domain: A survey, open challenges and opportunities. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 73–82.

[114] Ihirwe, F., Ruscio, D. D., Mazzini, S., Pierini, P., and Pierantonio, A. (2020). Low-code engineering for internet of things: a state of research. In Guerra, E. and Iovino, L., editors, *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings*, pages 74:1–74:8. ACM.

[115] Indamutsa, A., Di Ruscio, D., and Pierantonio, A. (2021a). A low-code development environment to orchestrate model management services. In *Advances in Production Management Systems. Artificial Intelligence for Sustainable and Resilient Production Systems*, pages 342–350, Cham. Springer International Publishing.

[116] Indamutsa, A., Rocco, J. D., Ruscio, D. D., and Pierantonio, A. (2021b). Mdeforgewl: Towards cloud-based discovery and composition of model management services. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 118–127.

[117] Izsó, B., Szárnyas, G., Ráth, I., and Varró, D. (2013). Incquery-d: Incremental queries in the cloud. pages 1–4.

[118] Jabbari, R., bin Ali, N., Petersen, K., and Tanveer, B. (2016). What is devops? a systematic mapping study on definitions and practices. In *Proceedings of the Scientific Workshop Proceedings of XP2016*, XP '16 Workshops, New York, NY, USA. Association for Computing Machinery.

[119] Jácome-Guerrero, S. P., Ferreira, M., and Corral, A. (2017). Software development tools in model-driven engineering. In *2017 5th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 140–148. IEEE.

[120] Jula, A., Sundararajan, E., and Othman, Z. (2014). Cloud computing service composition: A systematic literature review. *Expert Systems with Applications*, 41(8):3809–3824.

[121] Kan, S. H. (2003). *Metrics and Models in Software Quality Engineering*. Addison-Wesley Professional.

[122] Karambelkar, H. V. (2015). *Scaling Big Data with Hadoop and Solr Second Edition*.

[123] Khorram, F., Bousse, E., Mottu, J.-M., and Sunyé, G. (2021). Adapting tdl to provide testing support for executable dsls. *The Journal of Object Technology*, 20(3):6–1.

[124] Khorram, F., Mottu, J.-M., and Sunyé, G. (2020). Challenges & opportunities in low-code testing. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–10.

[125] Kiljander, J., Takalo-Mattila, J., Etelapera, M., Soininen, J.-P., and Keinanen, K. (2011). Enabling end-users to configure smart environments. In *2011 IEEE/IPSJ International Symposium on Applications and the Internet*, pages 303–308.

[126] Kleinfeld, R., Steglich, S., Radziwonowicz, L., and Doukas, C. (2014). Glue.things: A mashup platform for wiring the internet of things with the internet of services. In *Proceedings of the 5th International Workshop on Web of Things*, WoT '14, page 16–21, New York, NY, USA. Association for Computing Machinery.

[127] Kling, W., Jouault, F., Wagelaar, D., Brambilla, M., and Cabot, J. (2012). Moscript: A dsl for querying and manipulating model repositories. In Sloane, A. and Aßmann, U., editors, *Software Language Engineering*, pages 180–200, Berlin, Heidelberg. Springer Berlin Heidelberg.

[128] Koegel, M. and Helming, J. (2010). Emfstore: a model repository for emf models. In *Software Engineering, 2010 ACM/IEEE 32nd Int. Conf. on*, volume 2, pages 307–308.

[129] Kotarba, M. (2018). Digital transformation of business models. *Foundations of management*, 10(1):123–142.

[130] Kotopoulos, G., Kazasis, F., and Christodoulakis, S. (2007). Querying mof repositories: The design and implementation of the query metamodel language (qml). *Proceedings of the 2007 Inaugural IEEE-IES Digital EcoSystems and Technologies Conference, DEST 2007*, pages 373–378.

[131] Kruchten, P. B. (1995). The 4+ 1 view model of architecture. *IEEE software*, 12(6):42–50.

[132] kubernetes.io (2021). umentation. https://kubernetes.io/docs/home/. 2021-08-31.

[133] Kutsche, R., Milanovic, N., Bauhoff, G., Baum, T., Cartsburg, M., Kumpe, D., and Widiker, J. (2008). BIZYCLE: Model-based Interoperability Platform for Software and Data Integration. In *Procs.of the MDTPI at ECMDA*.

[134] Laigner, R., Kalinowski, M., Diniz, P., Barros, L., Cassino, C., Lemos, M., Arruda, D., Lifschitz, S., and Zhou, Y. (2020). From a monolithic big data system to a microservices event-driven architecture. *Proceedings - 46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020*, pages 213–220.

[135] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., and Volgyesi, P. (2001). The generic modeling environment. In *Workshop on Intelligent Signal Processing*.

[136] Lemos, A. L., Daniel, F., and Benatallah, B. (2015). Web service composition: A survey of techniques and tools. *ACM Computing Surveys*, 48(3).

[137] Lethbridge, T. C. (2021). Low-code is often high-code, so we must design low-code platforms to enable proper software engineering. In *International Symposium on Leveraging Applications of Formal Methods*, pages 202–212. Springer.

[138] Li, F., Vögler, M., Claeßens, M., and Dustdar, S. (2013). Towards automated iot application deployment by a cloud-based approach. In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, pages 61–68.

[139] Llad, C. M. and Smith, C. U. (2004). Performance model interchange format (pmif 2.0): Xml definition and implementation. In *Quantitative Evaluation of Systems, International Conference on*, Los Alamitos, CA, USA. IEEE Computer Society.

[140] LLC, K. (2020). Kissflow platform overview. https://kissflow.com/process-management/. Accessed on March 23, 2020.

[141] López, J. A. H. and Cuadrado, J. S. (2020). Mar: A structure-based search engine for models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '20, pages 57—-67, New York, NY, USA. Association for Computing Machinery.

[142] Lucrédio, D., Fortes, R. P. d. M., and Whittle, J. (2012). Moogle: a metamodel-based model search engine. *Software & Systems Modeling*, 11(2):183–208.

[143] Luo, Y., Liang, P., Wang, C., Shahin, M., and Zhan, J. (2021). Characteristics and challenges of low-code development: The practitioners' perspective. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11.

[144] López, J. A. H. and Cuadrado, J. S. (2021). An efficient and scalable search engine for models. *Software and Systems Modeling*.

[145] Makedonski, P., Adamis, G., Käärik, M., Kristoffersen, F., Carignani, M., Ulrich, A., and Grabowski, J. (2019). Test descriptions with etsi tdl. *Software Quality Journal*, 27(2):885–917.

[146] Mallick, S., Pandey, R., Neupane, S., Mishra, S., and Kushwaha, D. S. (2011). Simplifying Web service discovery & validating service composition. *Proceedings - 2011 IEEE World Congress on Services, SERVICES 2011*, pages 288–294.

[147] Marjani, M., Nasaruddin, F., Gani, A., Karim, A., Hashem, I. A. T., Siddiqa, A., and Yaqoob, I. (2017). Big IoT Data Analytics: Architecture, Opportunities, and Open Research Challenges. *IEEE Access*, 5:5247–5261.

[148] Martin, D., Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., and Sycara, K. (2005). Bringing semantics to web services: The owls approach. In Cardoso, J. and Sheth, A., editors, *Semantic Web Services and Web Process Composition*, pages 26–42, Berlin, Heidelberg. Springer Berlin Heidelberg.

[149] Mayer, S., Verborgh, R., Kovatsch, M., and Mattern, F. (2016). Smart configuration of smart environments. *IEEE Transactions on Automation Science and Engineering*, 13(3):1247–1255.

[150] Mendix (2020a). An introduction to low-code platform. https://www.mendix.com/low-code-guide/. Accessed on March 23, 2020.

[151] Mendix (2020b). Mendix platform features. https://www.mendix.com/platform/. Accessed on March 23, 2020.

[152] Molina, A. I., Giraldo, W. J., Gallardo, J., Redondo, M. A., Ortega, M., and García, G. (2012). Ciat-gui: A mde-compliant environment for developing graphical user interfaces of information systems. *Advances in Engineering Software*, 52:10–29.

[153] Muzaffar, A. W., Mir, S. R., Anwar, M. W., and Ashraf, A. (2017). Application of model driven engineering in cloud computing: a systematic literature review. In *Proceedings of the Second International Conference on Internet of things, Data and Cloud Computing*, pages 1–6.

[154] Negash, B., Westerlund, T., Liljeberg, P., and Tenhunen, H. (2017a). Rethinking 'Things' - Fog Layer Interplay in IoT: A Mobile Code Approach. In *11th International Conference on Research and Practical Issues of Enterprise Information Systems (CON-FENIS)*, volume LNBIP-310, pages 159–167, Shanghai, China. Springer International Publishing.

[155] Negash, B., Westerlund, T., Rahmani, A. M., Liljeberg, P., and Tenhunen, H. (2017b). DoS-IL: A Domain Specific Internet of Things Language for Resource Constrained Devices. *Procedia Computer Science*, 109:416–423.

[156] Nepomuceno, T., Carneiro, T., Carneiro, T., Korn, C., and Martin, A. (2018). A gui-based platform for quickly prototyping server-side iot applications. In *Smart SysTech 2018; European Conference on Smart Objects, Systems and Technologies*, pages 1–9.

[157] Nepomuceno, T., Carneiro, T., Maia, P. H., Adnan, M., Nepomuceno, T., and Martin, A. (2020). Autoiot: A framework based on user-driven mde for generating iot applications. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 719–728, New York, NY, USA. Association for Computing Machinery.

[158] Node-RED (2020). Node-red: Low-code programming for event-driven applications. https://nodered.org/. Last accessed May 2020.

[159] Noura, M., Atiquzzaman, M., and Gaedke, M. (2019). Interoperability in Internet of Things: Taxonomies and Open Challenges. *Mobile Networks and Applications*, 24(3):796–809.

[160] Opara-Martins, J., Sahandi, R., and Tian, F. (2015). Implications of integration and interoperability for enterprise cloud-based applications. pages 213–223.

[161] OutSystems (2020). Outsystem platform features. https://www.outsystems.com/platform/. Accessed on March 23, 2020.

[162] Ovadia, S. (2014). Automate the Internet With "If This Then That" (IFTTT). *Behavioral and Social Sciences Librarian*, 33(4):208–211.

[163] Pang, S., Gao, Q., Liu, T., He, H., Xu, G., and Liang, K. (2019). A Behavior Based Trustworthy Service Composition Discovery Approach in Cloud Environment. *IEEE Access*, 7:56492–56503.

[164] Petersen, K., Vakkalanka, S., and Kuzniarz, L. (2015). Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64.

[165] PowerApps, M. (2020). Microsoft powerapps platform overview. https://docs.microsoft.com/en-us/powerapps/maker/. Accessed on March 23, 2020.

[166] Prehofer, C. and Gerostathopoulos, I. (2017). Chapter 3 - Modeling RESTful Web of Things Services: Concepts and Tools. In Sheng, Q. Z., Qin, Y., Yao, L., and Benatallah, B., editors, *Managing the Web of Things*, pages 73–104. Morgan Kaufmann, Boston.

[167] Quirk, C., Mooney, R., and Galley, M. (2015). Language to code: Learning semantic parsers for if-This-Then-That recipes. *ACL-IJCNLP 2015 - 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, Proceedings of the Conference*, 1:878–888.

[168] Rafique, W., Zhao, X., Yu, S., Yaqoob, I., Imran, M., and Dou, W. (2020). An application development framework for internet-of-things service orchestration. *IEEE Internet of Things Journal*, 7(5):4543–4556.

[169] Rahmati, A., Fernandes, E., Jung, J., and Prakash, A. (2017). IFTTT vs. Zapier: A Comparative Study of Trigger-Action Programming Frameworks.

[170] Rashid, Z. N., Zebari, S. R., Sharif, K. H., and Jacksi, K. (2018). Distributed cloud computing and distributed parallel computing: A review. In *2018 International Conference on Advanced Science and Engineering (ICOASE)*, pages 167–172. IEEE.

[171] Ray, P. P. (2017). A Survey on Visual Programming Languages in Internet of Things. *Scientific Programming*, 2017:1231430.

[172] Regane, B. (2019). Model-driven engineering for big data.

[173] Research, G. V. (2018). Low-code application development platform market report, 2020-2027. Historical range: 2016-2018.

[174] Research, G. V. (2020). Low-code application development platform market worth $86.92 billion by 2027. https://www.grandviewresearch.com/press-release/global-low-code-application-development-platform-market.

[175] Restivo, A., Ferreira, H. S., Dias, J. P., and Silva, M. (2020). Visually-defined real-time orchestration of iot systems.

[176] Richardson, C. and Rymer, J. R. (2016a). The forrester wave: Low-code development platforms, q2 2016. tech. rep. *Forrester Research*.

[177] Richardson, C. and Rymer, J. R. (2016b). Vendor landscape: The fractured, fertile terrain of low-code application platforms. *FORRESTER, Janeiro*.

[178] Rivera, J. E., Ruiz-Gonzalez, D., Lopez-Romero, F., Bautista, J., and Vallecillo, A. (2009). Orchestrating ATL Model Transformations. *Proc. of MtATL 2009*, (June):34–46.

[179] Rocco, J., Di Ruscio, D., Iovino, L., and Pierantonio, A. (2015). Collaborative repositories in model-driven engineering [software technology]. *IEEE Software*, 32:28–34.

[180] Rodrigues da Silva, A. (2015). Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155.

[181] Rodriguez-Mier, P., Pedrinaci, C., Lama, M., and Mucientes, M. (2016). An integrated semantic web service discovery and composition framework. *IEEE Transactions on Services Computing*, 9(4):537–550.

[182] Rokis, K. and Kirikova, M. (2022). Challenges of low-code/no-code software development: A literature review. In *International Conference on Business Informatics Research*, pages 3–17. Springer.

[183] Roman, D., Schade, S., Berre, A., Bodsberg, N. R., and Langlois, J. (2009). Model as a service (maas). In *AGILE Workshop: Grid Technologies for Geospatial Applications, Hannover, Germany*.

[184] Rymer, J. R. (2019). The forrester wave: Low-code platforms for business developers, q2 2019. *Forrester Research*.

[185] Sachdeva, G. S. (2017). *Practical ELK Stack: Build Actionable Insights and Business Metrics Using the Combined Power of Elasticsearch, Logstash, and Kibana*. Apress, New York, NY, New York, NY.

[186] Sahay, A., Indamutsa, A., Di Ruscio, D., and Pierantonio, A. (2020). Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 171–178. IEEE.

[187] Salesforce (2020). Salesforce app cloud platform overview. https://developer.salesforce.com/platform. Accessed on March 23, 2020.

[188] Salihbegovic, A., Eterovic, T., Kaljic, E., and Ribic, S. (2015). Design of a domain specific language and ide for internet of things applications. In *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 996–1001.

[189] Salman, A. J., Al-Jawad, M., and Tameemi, W. A. (2021). Domain-Specific Languages for IoT: Challenges and Opportunities. *IOP Conference Series: Materials Science and Engineering*, 1067(1):012133.

[190] Sanchez, B., Kolovos, D. S., and Paige, R. (2019). Modelflow: Towards reactive model management workflows. *DSM 2019 - Proceedings of the 17th ACM SIGPLAN International Workshop on Domain-Specific Modeling, co-located with SPLASH 2019*, pages 30–39.

[191] Sanchis, R., García-Perales, Ó., Fraile, F., and Poler, R. (2019). Low-code as enabler of digital transformation in manufacturing industry. *Applied Sciences*, 10(1):12.

[192] Sandobalin, J., Insfran, E., Abrah, S., et al. (2018). A model-driven migration approach among cloud providers. In *XIV Jornadas de Ciencia e Ingeniería de Servicios (JCIS 2018)*. SISTEDES.

[193] Schmidt, D. C. et al. (2006). Model-driven engineering. *Computer-IEEE Computer Society-*, 39(2):25.

[194] Singh, A., Mittal, P., and Jha, N. (2013). Foss: A challenge to proprietary software. *IJCST*, 4.

[195] Sneps-Sneppe, M. and Namiot, D. (2015). On web-based domain-specific language for internet of things. In *2015 7th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, pages 287–292.

[196] Srivastava, P. and Khan, R. (2018). A review paper on cloud computing. *International Journal of Advanced Research in Computer Science and Software Engineering*, 8(6):17–20.

[197] Stürmer, G., Mangler, J., and Schikuta, E. (2009). A domain specific language and workflow execution engine to enable dynamic workflows. *Proceedings - 2009 IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2009*, pages 653–658.

[198] Sultan, N. (2014). Making use of cloud computing for healthcare provision: Opportunities and challenges. *International Journal of Information Management*, 34(2):177–184.

[199] Surbatovich, M., Aljuraidan, J., Bauer, L., Das, A., and Jia, L. (2017). Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of ifttt recipes. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1501–1510.

[200] Szárnyas, G., Izsó, B., Ráth, I., Harmath, D., Bergmann, G., and Varró, D. (2014). Incquery-d: A distributed incremental model query framework in the cloud. In *Model-Driven Engineering Languages and Systems*, pages 653–669, Cham. Springer International Publishing.

[201] Taherkordi, A. and Eliassen, F. (2016). Scalable modeling of cloud-based iot services for smart cities. In *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 1–6.

[202] Taibi, D., Lenarduzzi, V., and Pahl, C. (2017). Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing*, 4(5):22–32.

[203] Talesra, K. and Nagaraja, G. (2021). Low-code platform for application development. *International Journal of Applied Engineering Research*, 16(5):346–351.

[204] Teixeira, S., Agrizzi, B. A., Filho, J. G. P., Rossetto, S., and de Lima Baldam, R. (2017). Modeling and automatic code generation for wireless sensor network applications using model-driven or business process approaches: A systematic mapping study. *Journal of Systems and Software*, 132:50–71.

[205] Tisi, M., Mottu, J., Kolovos, D. S., de Lara, J., Guerra, E., Ruscio, D. D., Pierantonio, A., and Wimmer, M. (2019a). Lowcomote: Training the next generation of experts in scalable low-code engineering platforms. In *STAF (Co-Located Events)*, volume 2405 of *CEUR Workshop Proceedings*, pages 73–78. CEUR-WS.org.

[206] Tisi, M., Mottu, J.-M., Kolovos, D., De Lara, J., Guerra, E., Di Ruscio, D., Pierantonio, A., and Wimmer, M. (2019b). 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)*.

[207] Toure, E. H. B., Fall, I., Bah, A., Camara, M. S., and Ba, M. (2017). Consistency preserving for evolving megamodels through axiomatic semantics. In *2017 Intelligent Systems and Computer Vision (ISCV)*, pages 1–8.

[208] Valsamakis, Y. and Savidis, A. (2018). Personal Applications in the Internet of Things Through Visual End-User Programming. In Linnhoff-Popien, C., Schneider, R., and Zaddach, M., editors, *Digital Marketplaces Unleashed*, pages 809–821, Berlin, Heidelberg. Springer Berlin Heidelberg.

[209] Vincent, P., Iijima, K., Driver, M., Wong, J., and Natis, Y. (2019a). Licensed for Distribution Magic Quadrant for Enterprise Low-Code Application Platforms. pages 1–35.

[210] Vincent, P., Iijima, K., Driver, M., Wong, J., and Natis, Y. (2019b). Licensed for distribution magic quadrant for enterprise low-code application platforms (2019) 1–34. *URL: https://www. gartner. com/doc/reprints*.

[211] Vincent, P., Iijima, K., Driver, M., Wong, J., and Natis, Y. (2019c). Magic quadrant for enterprise low-code application platforms. *Gartner report*.

[212] Vogel-Heuser, B., Schütz, D., Frank, T., and Legat, C. (2014). Model-driven engineering of manufacturing automation software projects – a sysml-based approach. *Mechatronics*, 24(7):883–897. 1. Model-Based Mechatronic System Design 2. Model Based Engineering.

[213] Vokorokos, L., Uchnár, M., and Leščišin, L. (2016). Performance optimization of applications based on non-relational databases. In *2016 International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pages 371–376.

[214] Von Rosing, M., White, S. A., Cummins, F., and De Man, H. (2014). Business process model and notation-BPMN. *The Complete Business Process Handbook: Body of Knowledge from Process Modeling to BPM*, 1(January):429–453.

[215] Vorapojpisut, S. (2015). A Lightweight Framework of Home Automation Systems Based on the IFTTT Model. *Journal of Software*, 10(12):1343–1350.

[216] Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C. L., Visser, E., and Wachsmuth, G. (2013). *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org.

[217] W3C (2007). Web services description language (wsdl) version 2.0 part 1: Core language. https://www.w3.org/TR/wsdl20/. 2021-08-31.

[218] Waszkowski, R. (2019). Low-code platform for automating business processes in manufacturing. *IFAC-Papersmisc*, 52(10):376–381.

[219] Weghofer, S. (2017). Moola-a groovy-based model operation orchestration language.

[220] Weißenberger, B., Flad, S., Chen, X., Rösch, S., Voigt, T., and Vogel-Heuser, B. (2015). Model driven engineering of manufacturing execution systems using a formal specification. In *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–8.

[221] Whittle, J., Hutchinson, J., and Rouncefield, M. (2013). The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85.

[222] Wohlin, C., Runeson, P., Neto, P., Engström, E., Machado, I., and Almeida, E. (2013). On the reliability of mapping studies in software engineering. *Journal of Systems and Software*, 86:2594–2610.

[223] Woo, M. (2020). The rise of no/low code software development—no experience needed? *Engineering (Beijing, China)*, 6(9):960.

[224] Wortmann, A., Combemale, B., and Barais, O. (2017). A systematic mapping study on modeling for industry 4.0. MODELS '17, page 281–291. IEEE Press.

[225] Xu, Y. and Helal, A. (2016). Scalable cloud–sensor architecture for the internet of things. *IEEE Internet of Things Journal*, 3(3):285–298.

[226] Yang, Z., Wang, L., and Song, X. (2016). Secure model based on multi-cloud for big data storage and query. In *2016 International Conference on Advanced Cloud and Big Data (CBD)*, pages 207–214. IEEE.

[227] Zacharewicz, G., Daclin, N., Doumeingts, G., and Haidar, H. (2020). Model driven interoperability for system engineering. *Modelling*, 1(2):94–121.

[228] Zafar, M. N., Azam, F., Rehman, S., and Anwar, M. W. (2017). A systematic review of big data analytics using model riven engineering. *ACM International Conference Proceeding Series*, pages 1–5.

[229] Zou, G., Zhang, B., Zheng, J., Li, Y., and Ma, J. (2012). MaaS: Model as a Service in Cloud Computing and Cyber-I Space. *Proceedings - 2012 IEEE 12th International Conference on Computer and Information Technology, CIT 2012*, pages 1125–1130.

Appendix A

MDEForge cluster

MDEForgeWL

API specifications

Versatile in its structure, the MDEForge Persistence API documentation 1.0.0 offers a variety of options tailored to user needs. All resources are owned by a user, which means a user must first be created before taking full advantage of the system. Henceforth user can create workspaces. projects are contained in a workspace and artifacts are created within a project. Furthermore, users can easily manage their artifacts through standard actions such as creating, modifying, removing and viewing artifacts. The API has been designed to be flexible and intuitive so that users can easily access the necessary tools to manage their resources within MDEForge.

MDEForge Cluster Installation

Welcome to MDEForge microservices!This suite of services are developed using Java-based Spring Boot framework [1] and Node.js [2]. Some utilities might use python or bash scripts.

You can clone the repository and run it locally though it might be computationally expensive. However, you can even deploy it using docker technologies and kubernetes as an orchestrator for these microservices.

Running the cluster

We use helm to run the cluster.
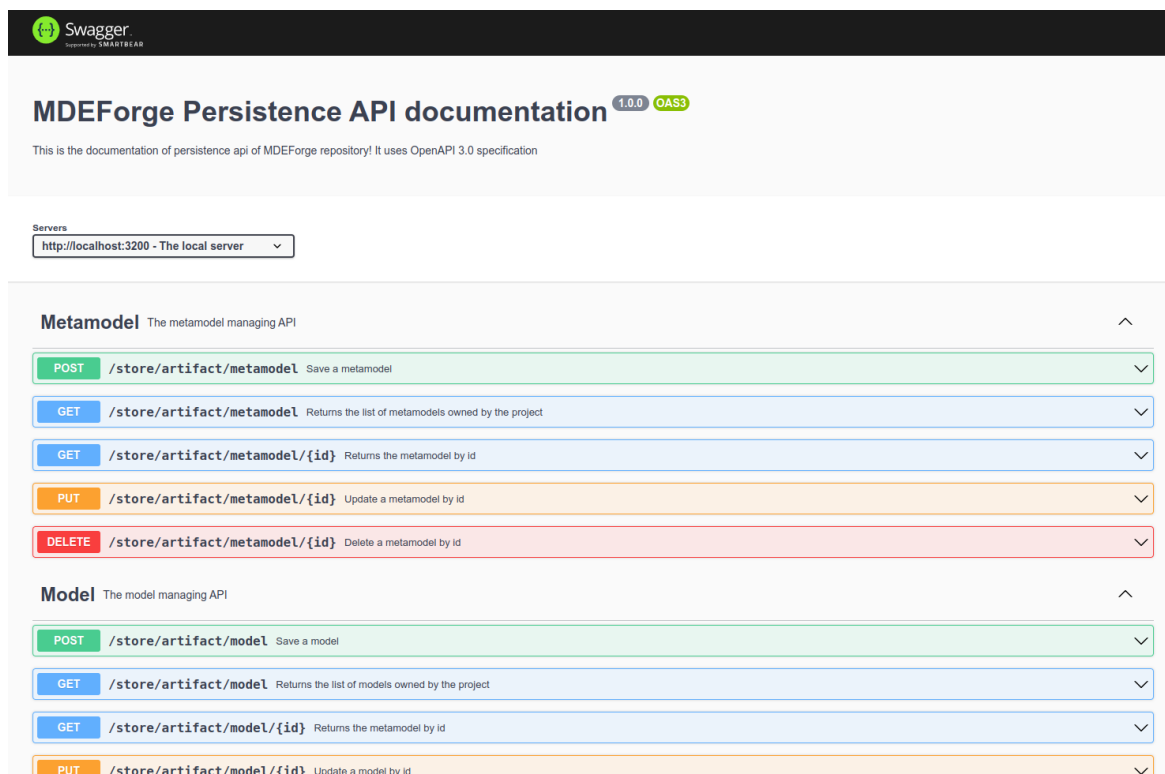
---

[1]https://spring.io/
[2]https://nodejs.org/en/

Fig. A.1 Screenshot of MDEForge OpenAPI 3.0 specification

– Run *helm install lowcomote* to install the chart – If you want to uninstall the charts *helm uninstall lowcomote* – if you want to update the charts *helm upgrade lowcomote* . in the current directory.

Before you can run these commands, make sure you have built and pushed on the cloud the correct containers.

For instance, i already have a script that take the name of the container and version, build it and push it to the registry – build.sh dsl-frontend v2.9

After pushing on the cloud, you have to upgrade the helm charts values, especially the version of the container to retrieve the latest version.

Before you can upgrade the charts, it's good to delete the current deployment you want to update – kube delete deployment.apps/dsl-frontend-deployment service/dsl-frontend-server For instance, here i would like to update the service and deployment above, so i removed them.

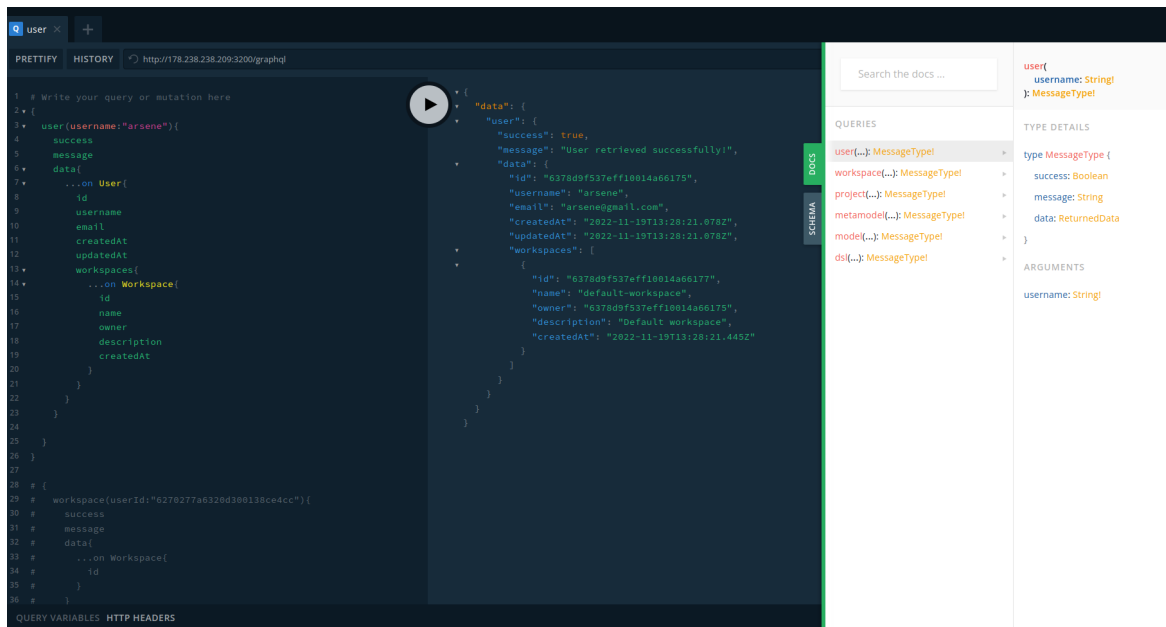Then, now i can update the charts: helm upgrade lowcomote helm-deployment/

Fig. A.2 Screenshot of MDEForge GraphQL API specification

In case you want to forward a given container to localhost, use portforwarding – kube port-forward service/dsl-frontend-server 9999:8080

To enter a container while you are running the cluster: – kubectl exec -it dsl-backend-server – /bin/sh

To get the node where the cluster is deployed: – kube get node -o wide

To connect to GKE cluster – gcloud container clusters get-credentials cluster-name –zone europe-west3-a –project project-name

To allow the cluster – gcloud compute firewall-rules create allowed-node-ports –allow tcp:30100

To log a deployment – kube logs deployment.apps/dsl-backend-deployment

For more information please check the README file on the repository: https://github.com/Indamutsa/model-management-services.git

MDEForgeWL Documentation

This domain specific language orchestrates and enables service composition of model management services.

These are the services supported:

- Persistence for model artifacts(CRUD)

Fig. A.3 Screenshot of MDEForge cluster on Google Cloud

- Model transformation

- Model validation

- Model query

- Model Merging

- Model comparison

- Notifications

The language also provides some general programming language features such as conditional statements, loops, variable assignments and functions.

These features can help you programmatically compose services and process output from previously executed services.

Below is the syntax to get you started:

Variables: Example:

```
var arsene = "Hello world"
Define var indamutsa = "Hello world"
```

_____

Assignment:

```
// You can assign the variable in to another expression or variable
    arsene = indamutsa + "Another hello world"
```

```
If statement:
// You can define a statement.
Example:

var a = 2

if(a == 2){
print("Yes")
}else{
print("No")
}
```

- -

```
Loop statement:
// You can define a loop
Example


var i = 0

loop:if(i <= 0){
print("Inside the loop")
}
```

- -

```
Method:
/** You can define a method, with empty or parameters */

Example:

function method(string: name){
print(name)
}
```

```
Call method:
// You can call a method.
var hello = "The name"

call method(hello)
```

```
Call model management services:
// For instance to call transformation service
call service _transfoModel("Tree.xmi", "Tree.ecore", "Tree.ecore", "Demo.etl")

// For instance to call validation service
call service _validateModel("Tree.xmi", "Tree.ecore", "Demo.evl")

// For instance to call compare service
call service _compareModel("catalogue1.xml", "catalogue2.xml", "Demo.ecl")

// For instance to call model object query service
call service _queryModel("Tree.xmi", "Tree.ecore", "Demo.eol")

// For instance to call model merging service
call service _mergeModels("catalogue1.xml", "catalogue2.xml", "catalogues.ecl",
"catalogues.eml")
```

```
Workflow:
// You can define a workflow

Workflow workflowName type: parallel{
step "Test" {
var name = "inside the workflow"
// You can call define the statements inside here,
// if statements, loops, functions, and services
print(name)
}
}

// You have to run the workflow name
Execute workflowName()
```

```
DEMO PROGRAM: You can copy this program and paste it
in the editor for testing purposes
```

```
// Press ctrl + space to trigger auto-completion



// Create variables : This part is improved by advanced query mechanisms.
// You can query the type of models u want based on your defined criterias
var sourceModel = "Tree.xmi"
var sourceMetamodel = "Tree.ecore"
var targetMetamodel = "Tree.ecore"

var model1 = "catalogue1.xml"
var model2 = "catalogue2.xml"
```

```
var etlscript = "Demo.etl"
var evlscript = "Demo.evl"
var eclscript = "Demo.ecl"
var eolscript = "Demo.eol"


var mergeCompareScript = "catalogues.ecl"
var mergescript = "catalogues.eml"



function dummyFunc(string:name){
print("Executing inside the function")
return name
}



Workflow workflow type:sequence{
step "Test"{

    var name = call dummyFunc("Hello world")

    // Check conditional and loop conditional statements
    if (name == "Hello world") {
        print("Conditional statement testing passed")
        var i = 0

        loop: if(i < 5){
            print("Looping: " + i)
i = i + 1
        }
    }



    // For instance to call model object query service
    call service _queryModel(sourceModel, sourceMetamodel, eolscript)
```

```
    // For instance to call validation service
    call service _validateModel(sourceModel, sourceMetamodel, evlscript)
}


step "Compare and Merge"{
    // For instance to call compare service
    call service _compareModel(model1, model2, eclscript)

    // For instance to call model merging service
    call service _mergeModels(model1, model2, mergeCompareScript, mergescript)
}


step "Transform"{
    // For instance to call transformation service
    call service _transfoModel(sourceModel, sourceMetamodel,
targetMetamodel, etlscript)
}
}
Execute workflow()
```

For a Demo video, please check: https://tinyurl.com/mdeforgewl
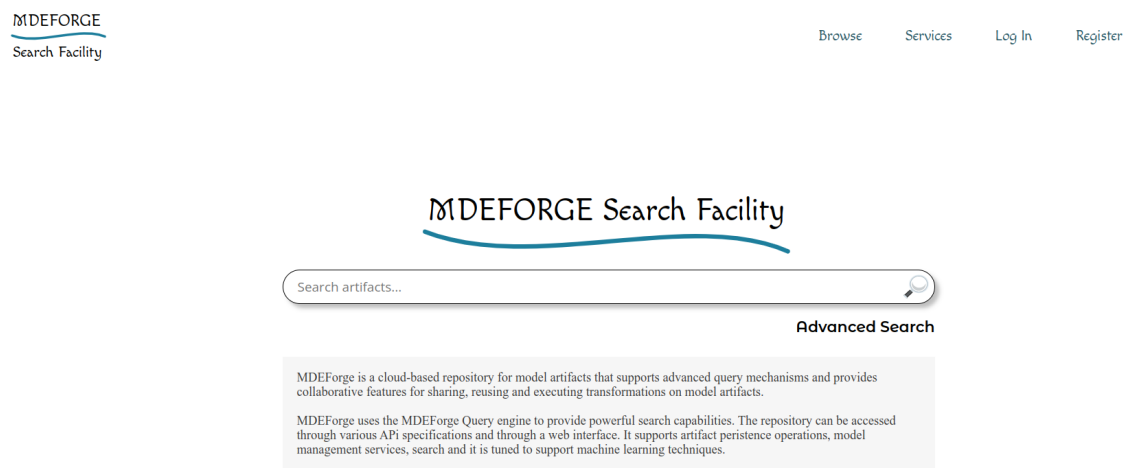
Appendix B

MDEForge search

Fig. B.1 Screenshot of MDEForge Search Interface

Discovery API

Using GraphQL API specification, the user can retrieve data using data entities such as the user, workspace, project and artifact. However, it can also be combined using the microsyntax query specification because it supports search keywords, tags, conditional statement and logical operators. The repository can be found on : https://github.com/Indamutsa/advanced-query.git

MDEForge Search Installation

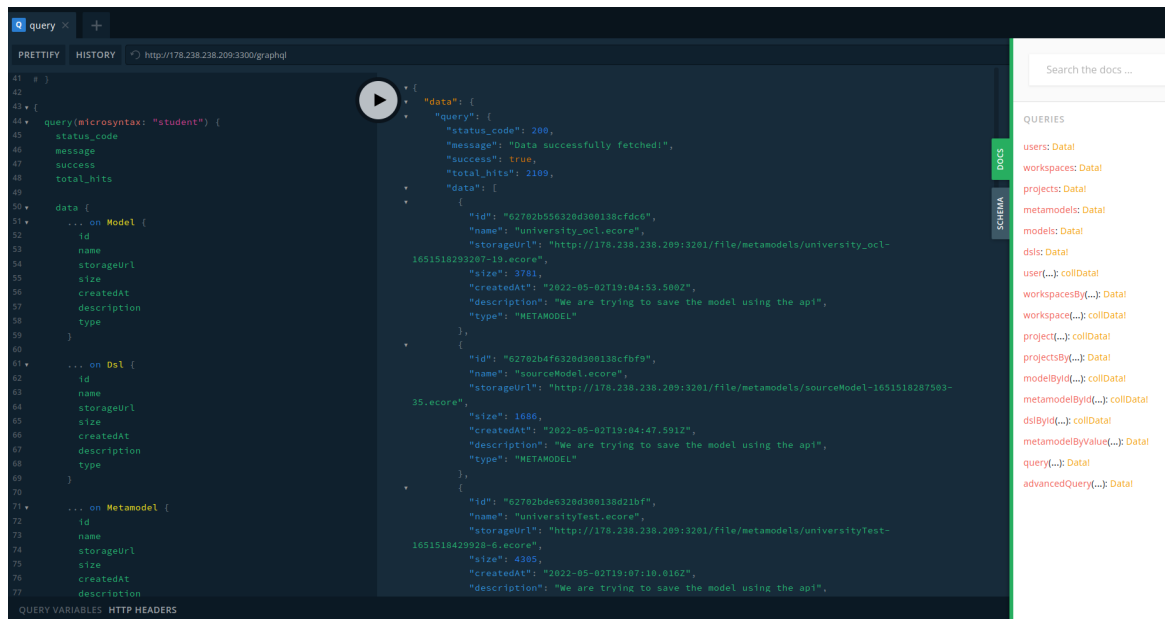The installation of the cluster can be easily done by run the bash script:

Fig. B.2 Screenshot of MDEForge Search GraphQL API specification

```
bash run.sh
```

It will spin up the cluster including the query engine and its frontent. The query engine is running on port 3300 and the frontend is running on port 3500. A cluster of databases is also running on port 27019 with the search engine on port 5601.

Microsyntax Query Specification Documentation

The microsyntax query specification is an integral part of MDEForge Search. It provides means for users to define query criteria to retrieve relevant artifacts by narrowing down the search query by means of operators. The query specification typically consists of multiple components, including a search keywords, conditional statements, search tags and logical operators. The microsyntax query specification provides a powerful yet easy-to-use way to access data persisted in the repository. By understanding the components of query specification, users can take advantage of its power to harness their query specification and effectively retrieve relevant model artifacts from the repository.

Below are the constructs of the query specification:

- *Search Keywords:* These are single words or a number that are used as the basis for the search query. The search engine uses this keyword for lookup in the index and return exact match. The search keywords can be a single keyword or multiple keywords. For example: `student teacher class`. These keywords can be used to compose a complex query with the use of operators described below.

- *Boolean required operator "AND":* The operator used to specify that all search terms or keywords on the left and on the right of the operator must be present in the returned results. Example: `student AND teacher`

- *Boolean exclusion operator "NOT":* This operator is used to exclude the next term, keyword or condtional statement group on its left side. example: student NOT [teacher AND class]

- *Boolean logical operator "OR":* This operator is used to specify that search results may contain any of the specified search keywords on the right and left of the operator respectively. Example: `student OR teacher`

- *SPACE:* this is an operator in our microsyntax operator. It is the same as OR operator. Example: `teacher OR student is the same as teacher student`

- *Left bracket group indicator "[":* This operator is used to group keywords and operators. It needs to be close by the right bracket group indicator. `[student AND teacher] OR class`

- *Right bracket group indicator "]":* Same like above, This operator is used to group keywords and operators. It needs to be close by the right bracket group indicator. `[student AND teacher] OR class`

- *Equality Operator "==":* This operator is used to specify exact match of the value specified. For instance to retrieve an artifact where its size is equal to 2 can be done as `size == 2`. Especially using the metrics tags, we use it like `cmc == 2`. That artifact should have a metric where cmc is equal to 2

  For the operators below, they assess tags that represent numerical values.

- *Less than operator "<":* This operator is used to specify a value on right side that is is less than a specified tag. For instance: `cmc < 2`

- *Greater than operator ">":* This operator is used to specify a value on right side that is is greater than a specified tag. For instance: `cmc > 2`

- *Less than or equal to operator "<=":* TThis operator is used to specify a value on right side that is is less or equal to a specified tag. For instance: `cmc <= 2`

- *Greater than or equal to operator ">=":* This operator is used to specify a value on right side that is is greater or equal to a specified tag. For instance: `cmc >= 2`

- *Exclusion operator "-":* This operator can used to exclude a specific keyword or groups of keywords from the search results.

- *Required keyword operator "+":* This operator is used to indicate that a specific keyword or groups of keywords from the search results are required in the search results.

- *Alternative keyword operator "|":* This operator is used to indicate that the the search results may contain a specific keyword or groups of keywords.

- *Quotation marks "" or " :* using quotation marks around a keyword or phrase indicates an exact match search. The search engine will only return results that exactly match the enclosed keyword or phrase. For `"teacher class student"`, an exact match is sought.

- *Wildcards:* This is used to indicate that A wildcard is a special character that can be used in a search query to match any sequence of characters. Wildcards are often used to search for different variations of a word or to search for multiple words that have the same root.

  - *Fuzzy search operator " ":* This operator is used to specify a search that is not an exact match for a specific search term or keyword. For instance `teachjr` will return `teacher`.

  - *The asterisk *:* this wildcard can match any sequence of characters. For example, `"mach*"` would match `"machine"`, `"machinery"`, `"machinist"`, etc.

  - *The question mark ?:* this wildcard can match any single character. For example, `"mach?ne"` would match `"machine"` and `"machene"`, but not `"machinery"`

  - *The caret ^ :* this wildcard is used to boost the relevance of a search term.

- *Search tags:* ["*accessControl*","*content*","*createdAt*","*description*","*ext*", "*involvedOperations*","*license*","*name*","*project*","*size*","*storageUrl*","*type*", "*unique_name*","*updatedAt*","*conformsTo*","*hasAttribute*","*isTransformable*"]: These are predefined tags that can be used to filter the model artifacts. For `teacher hasAttribute: name`

- *Metrics tags* ["*acfmc*","*aiflmc*","*amc*", "*attr*","*attrh*","*avgattr*", "*avgref*","*ccfmc*","*cflmc*", "*ciflmc*","*cmc*","*iflmc*", "*lmc*","*maxhl*","*maxhs*", "*mc*","*mcwsp*","*mtnb*", "*rec_cont*","*ref*","*refcc*",

"*refeop*","*sf*","*sfh*"]: These are quality metrics of model artifacts especially meta-models. They can be used to filter artifacts that reach some metrics in the search results.

```
name:  simple* AND cmc == 2
```

Remember that if you want to perform a search containing these special operators $+-\&\&||!()[]" *?:/$, you may need to skip them with a forward slash /.