"This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 813884"

Project Number: 813884

Project Acronym: Lowcomote

Project title: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms

D4.4 Low-Code Engineering Repository

Project GA: 813884

Project Acronym: Lowcomote

Project website: https://www.lowcomote.eu/

Project officer: Thomas Vyzikas

Work Package: WP4

Deliverable number: D4.4

Production date: October 18th 2022

Contractual date of delivery: September 30th 2022

Actual date of delivery: September 30th 2022

Dissemination level: Public

Lead beneficiary: University of L'Aquila

Authors: Arsene Indamutsa, Davide Di Ruscio, Alfonso Pierantonio, MohammadHadi Dehghani, Ilirian Ibrahimi, Alessandro Colantoni, Faezeh Khorram

Contributors: The Lowcomote partners

Abstract

Low-code development platforms (LCPDs) are software development platforms on the Cloud, provided through a Platform-as-a-Service model, which allows users to build completely operational applications by interacting through dynamic graphical user interfaces, visual diagrams and declarative languages. They address the need of non-programmers to develop personalised software and focus on their domain expertise instead of implementation requirements.

The availability of model repositories is paramount to reusing already developed low-code artifacts. Consequently, the availability of efficient and accurate ways to retrieve artifacts is highly relevant. Thus, relying on sound and well-formed models for discovering and reusing existing artifacts is key to preserving productivity benefits related to low-code processes.

This deliverable presents a novel repository consisting of core services to store and manage low-code artifacts. Atop such services, it is possible to develop extensions adding new functionalities to the system. Furthermore, all the services can be used via Web access and a REST API that permits adopting the available model management tools as software-as-a-service.

Executive summary

This document presents the low-code engineering repository underpinning the design, analysis and evolution of low-code engineering repository infrastructures. The conceived and developed infrastructures support scalable and extensible cloud-based model repositories. The architecture of the repository is presented using an extended *"4 + 1"* view model. The current repository can manage the persistence and reuse of heterogeneous modelling artefacts, which can be identified and retrieved employing the proposed advanced discovery mechanisms.

Contents

1 Introduction **6**
1.1 Structure of the deliverable . . . 7

2 Low-code development platforms **8**
2.1 Background . . . 8
2.1.1 A bird-eye view of low-code platforms . . . 8
2.1.2 Main components of low-code development platforms . . . 9
2.1.3 Development process in LCDPs . . . 10
2.2 An overview of representative low-code development platforms . . . 11
2.3 Taxonomy . . . 13
2.4 Comparison of relevant LCDPs . . . 14
2.4.1 Features and capabilities . . . 15
2.4.2 Additional aspects for comparing LCDPs . . . 16
2.5 Using LCDPs: a short experience report . . . 16
2.6 Conclusion . . . 18

3 Cloud-based modeling: a survey, open challenges and opportunities **19**
3.1 Background . . . 19
3.2 Study design . . . 20
3.3 Cloud-based modeling approaches (RQ1) . . . 21
3.4 Open challenges (RQ2) . . . 24
3.5 Research and development opportunities (RQ3) . . . 26
3.5.1 Tools and platforms . . . 26
3.5.2 Benefits of cloud-based modeling . . . 27
3.6 Related work . . . 28
3.7 Conclusion . . . 28

4 The Low-Code Engineering Repository Architecture **29**
4.1 Related work . . . 29
4.2 System views . . . 31
4.2.1 Use case view . . . 33
4.2.2 Logical view . . . 36
4.2.3 Development view . . . 38
4.2.4 Process view . . . 39
4.2.5 Data view . . . 44
4.2.6 Physical view . . . 44
4.3 Conclusion . . . 44

5 MDEForgeWL: cloud-based discovery and composition of model management services **46**
5.1 Background . . . 47
5.2 Composition of model management tools . . . 48
5.2.1 Overview of related work . . . 48
5.2.2 Comparison of model management composition approaches . . . 48
5.3 The proposed MDEForgeWL platform . . . 50
5.3.1 The MDEForgeWL front-end: low-code development environment . . . 50
5.3.2 The MDEForgeWL front-end: DSL . . . 52
5.3.3 The MDEForgeWL engine . . . 55
5.3.4 The MDEForgeWL cluster . . . 55
5.3.5 The MDEForgeWL persistence Layer . . . 56
5.4 Conclusion . . . 57

6 Enabling service-oriented discovery mechanisms in model repositories **58**
6.1 Background and Motivation . . . 59
6.2 Overview of existing approaches . . . 60
6.2.1 Methodology and scope . . . 61

6.2.2 Results . . . 62
6.2.3 Designed features for productive discovery mechanisms . . . 64
6.2.4 Comparing model search approaches . . . 67
6.2.5 Limitations of current discovery mechanisms in MDE domain . . . 69
6.3 Proposed approach . . . 70
6.3.1 Architectural design . . . 70
6.3.2 Logical layers . . . 72
6.4 Enabling advanced reuse-driven discovery . . . 73
6.4.1 Data ingestion and processing . . . 74
6.4.2 Discovery mechanisms . . . 75
6.4.3 Platform reuse mechanisms . . . 79
6.4.4 Integrated services in discovery mechanisms . . . 80
6.5 Integration of MDEForge with the Droid recommender framework . . . 80
6.6 Conclusion . . . 81

7 Model slicing on low-code engineering platforms **83**
7.1 Related Work on Model Slicing . . . 83
7.2 Background Information about the zAppDev LCDP . . . 84
7.2.1 Running example . . . 84
7.3 Repositories . . . 85
7.3.1 Input Class . . . 86
7.3.2 Horizontal Slice . . . 86
7.3.3 Vertical Slice . . . 86
7.3.4 The model slicing approach at work . . . 87
7.4 Conclusion . . . 89

8 Conclusion **90**

1 Introduction

Low-code development platforms (LCDPs) are easy-to-use visual environments that are being increasingly introduced and promoted by major IT players to permit *citizen developers* to build their software systems even if they lack a programming background. Forrester and Gartner document the growth of LCDPs in their reports [1, 2, 3] that forecast a significant market increase for LCDP companies over the next few years. Major PaaS players like Google and Microsoft are all integrating LCDPs (Google App Maker and Microsoft Power Platform, respectively) in their general-purpose solutions. According to a Forrester report [2], the low-code market is expected to represent $21B in spending by 2022. In a recent Gartner report [3], 18 LCDPs were analyzed out of 200.

The Lowcomote project aims at training a new generation of early-stage researches (ESRs) in the design, development and operation of new LCDPs, that overcome significant limitations of currently available platforms including *scalability*, *openness*, and *heterogeneity*.

To this end, Lowcomote is focusing on three main research objectives:

- **RO1**: Enabling *Low-code Engineering of Large-Scale Heterogeneous Systems*, by innovative development environments on the Cloud and precise integration of low-code languages with new domains;
- **RO2:** Developing a *Large-scale Repository and Services for Low-Code Engineering*, as a Cloud-based service able to handle a huge number of low-code artefacts;
- **RO3:** Producing advancements in *Scalable Low-Code Artefact Management*, as new algorithms and reusable components.

This document presents final results related to **RO2** developed in the context of WP4 and in particular of *ESR6 - Scalable and Extensible Cloud-based Low-Code Model Repository* and *ESR8 - Capability Discovery and Reuse in Low-code System Models*. According to the Lowcomote Description of Work (DoW), the expected results of ESR6 are related to the development of a "*[. . .] community-based model repository able to manage the persistence and reuse of heterogeneous modelling artefacts (including models, metamodels, and model transformations). The repository will support advanced query mechanisms and will be extensible in order to add new functionality, e.g. remote calculation of model metrics, semantic model differencing, validation and composition of model transformations, and even automated clustering of the stored modeling artefacts. [. . .]*". Furthermore, the expected outcomes of ESR8 are related "*[. . .] to facilitate model-level component discovery and reuse through automated identification of relevant low-code system model fragments from other, related system models. [. . .]*".

In this respect, we provide the evolving the Lowcomote low-code model repository. We describe the architecture, the use cases supported by the repository, and the architectural components that have been

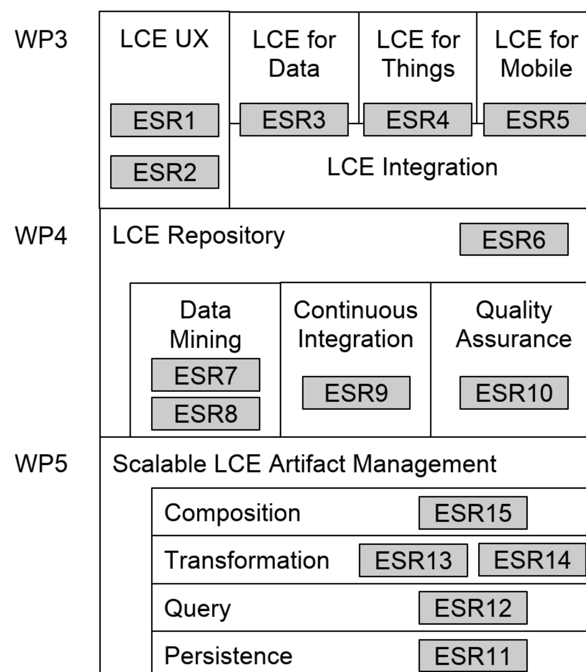

Figure 1: Overview of the Lowcomote project architecture

implemented to best support the use cases. Moreover, we outline the technologies that were used to implement the knowledge base. The system architecture was guided by the IEEE Recommended Practice for Architectural Description of Software-Intensive Systems [4].

1.1 Structure of the deliverable

This deliverable is structured as follows:

- Chapter 1 provides an overview of existing LCDPs with the aim of positioning Lowcomote in such a context and motivating the need of a dedicated repository.
- Chapter 2 provides an overview of cloud-based modeling open challenges and opportunities
- Chapter 3 presents an overview of existing cloud-based modeling platforms and highlights related challenges and opportunities;
- Chapter 4 presents the architecture of the Lowcomote repository. In particular, the different architectural views used in this document are presented, before presenting the architectural view decomposition of the conceived Lowcomote repository;
- Chapter 5 presents an approach to composing model management services;
- Chapter 6 presents an advanced approach to discovering and reusing modeling artifacts stored in model repositories;
- Chapter 7 presents an approach to reuse modeling artifacts employing a proposed model slicing approach;
- Chapter 8 concludes the document.

2 Low-code development platforms

LCDPs are provided on the Cloud through a Platform-as-a-Service (PaaS) model, and enable the development and deployment of fully functional software applications utilizing advanced graphical user interfaces and visual abstractions requiring minimal or no procedural code [5]. Thus, 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 sacrificing the productivity of professional developers.

By using low-code platforms, citizen developers can build their software application without the help of several developers that were earlier involved in along the full-stack development of fully operational applications. Thus, developers can focus on the business logic of the application being specified rather than dealing with unnecessary details related to setting up of the needed infrastructures, managing data integrity across different environments, and enhancing the robustness of the system. Bug fixing and application scalability and extensibility are also made easy, fast and maintainable in these platforms by the use of high-level abstractions and models [6]. Procedural code can be also specified in these platforms to achieve further customization of the application on one's own preferences.

In this chapter, a technical survey is provided 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 when educated selections and considerations have to be performed. The contributions of the chapter are 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 related to the adoption of LCDPs for developing a simple benchmark application.

To the best of our knowledge, this is the first work aiming at analyzing 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 that have been 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 related to the adoption of LCDPs for developing a simple benchmark application. Section 2.6 concludes the chapter and discusses some perspective work.

2.1 Background

Low-code development platforms[1] are software platforms that sit on the cloud and enable developers of different domain knowledge and technical expertise to develop fully-fledged applications ready for production[1]. Such applications are developed through model-driven engineering principles and take advantage of cloud infrastructures, automatic code generation, declarative and high level and graphical abstractions to develop entirely functioning applications [3]. These platforms capitalise on recent developments in cloud computing technologies and models 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.
At the heart of low-code platforms, there are model-driven engineering (MDE) principles [7] that have been adopted in several engineering disciplines by relying on the automation, analysis, and abstraction possibilities enabled by the adoption of modelling and metamodeling [8].

2.1.1 A bird-eye view of low-code platforms

From an architectural point of view, LCDPs consist of four main layers, as shown in Fig. 2. The top layer (see `Application Layer`) consists of the graphical environment that users directly interact with to specify their

[1]Hereafter, the terms *low-code platforms* and *low-code development platforms* are used interchangeably.

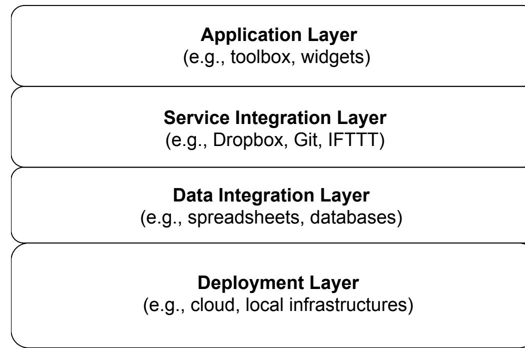

Figure 2: Layered architecture of low-code development platforms

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 authorisation mechanisms to be applied to the specified artefacts. Through the modelling constructs made available at this layer, users can specify the behaviour of the application being developed. 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 utilising external services, how to aggregate such data according to defined rules, and how to analyse them. To this end, the `Service Integration Layer` is exploited to connect with different services by using corresponding APIs and authentication mechanisms.
A dedicated data integration layer permits to operate and homogeneously manipulate data even if heterogeneous sources are involved. To this end, the `Data Integration Layer` is concerned with data integration with different data sources. Depending on the used LCDP, 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`.

2.1.2 Main components of low-code development platforms

By expanding the layered architecture shown in Fig. 2, the peculiar components building any low-code development platform are depicted in Fig. 3 and they can be grouped into three tiers. The first tier is made of 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 that are integrated with the platform. The arrows in Fig. 3 represent possible interactions that might occur among entities belonging to different tiers. The lines shown in the middle tier represents 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.
Figure 4 shows the application modeler of Mendix [9] 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 in terms of, e.g., the elements in the domain model, and the navigation model linking all the different specified pages. The application modeler also permits to run the system locally before deploying it. To this end, as shown in Fig. 3, 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 [10].
Concerning *database servers*, they can be both SQL and NoSQL. In any case, the application users and developers are not concerned about the type of employed database or mechanisms ensuring data integrity or query optimizations. More in general, the developer is not concerned about 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*, there are specific connectors in charge of consuming these APIs in the back-end. Thus, developers are relieved from the responsibility of manually managing technical aspects like authentication, load balance, business logic consistency, data integrity and security.

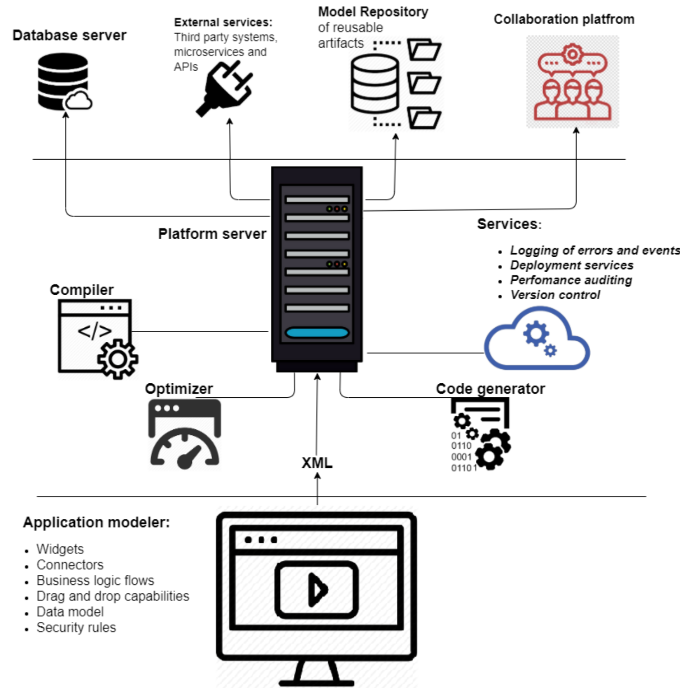

Figure 3: Main components of low-code development platforms

Low-code development platforms can also provide developers with repositories that can store reusable modeling 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, sprints and deal with changes as soon as customers require them and collaborate with other stakeholders.

2.1.3 Development process in LCDPs

The typical phases that are performed when developing applications by means of LCDPs can be summarized as follows.

1. *Data modeling* - usually, this is the first step taken; users make us of a visual interfaces to configure the data schema of the application being developed by creating entities, establishing relationships, defining constraints and dependencies generally through drag-and-drop facilities. A simple data model defined in Mendix is shown in Fig. 5.

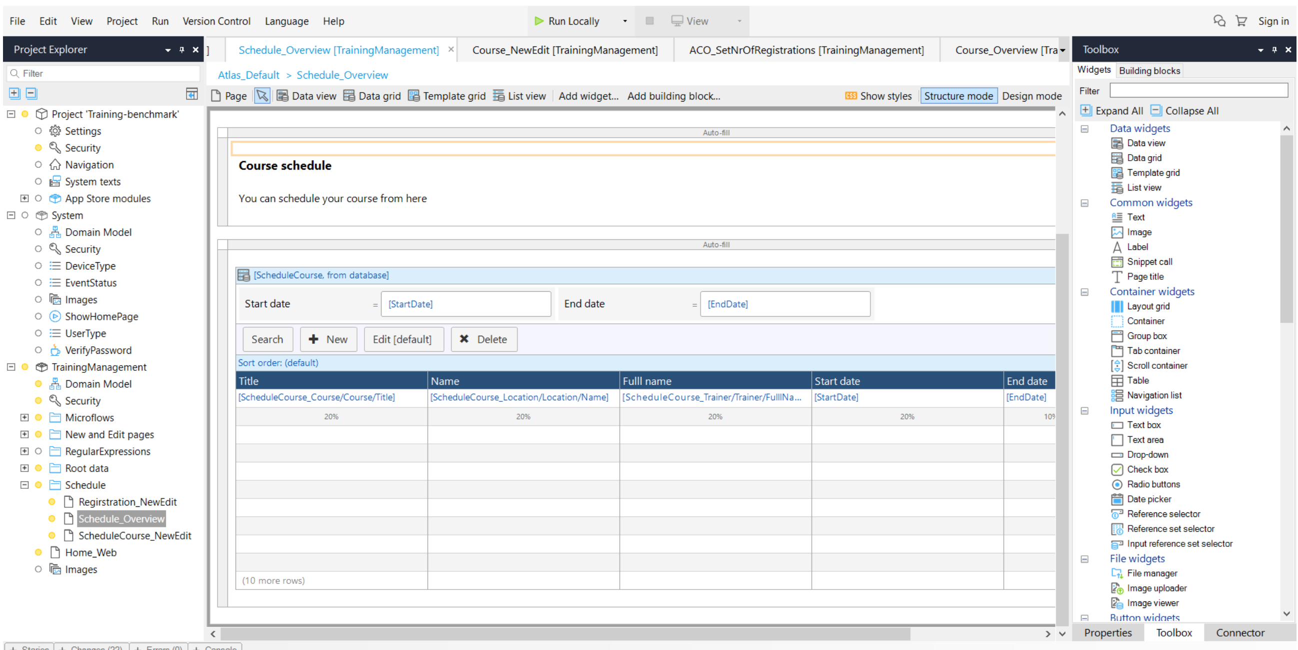

Figure 4: The application modeler of Mendix at work

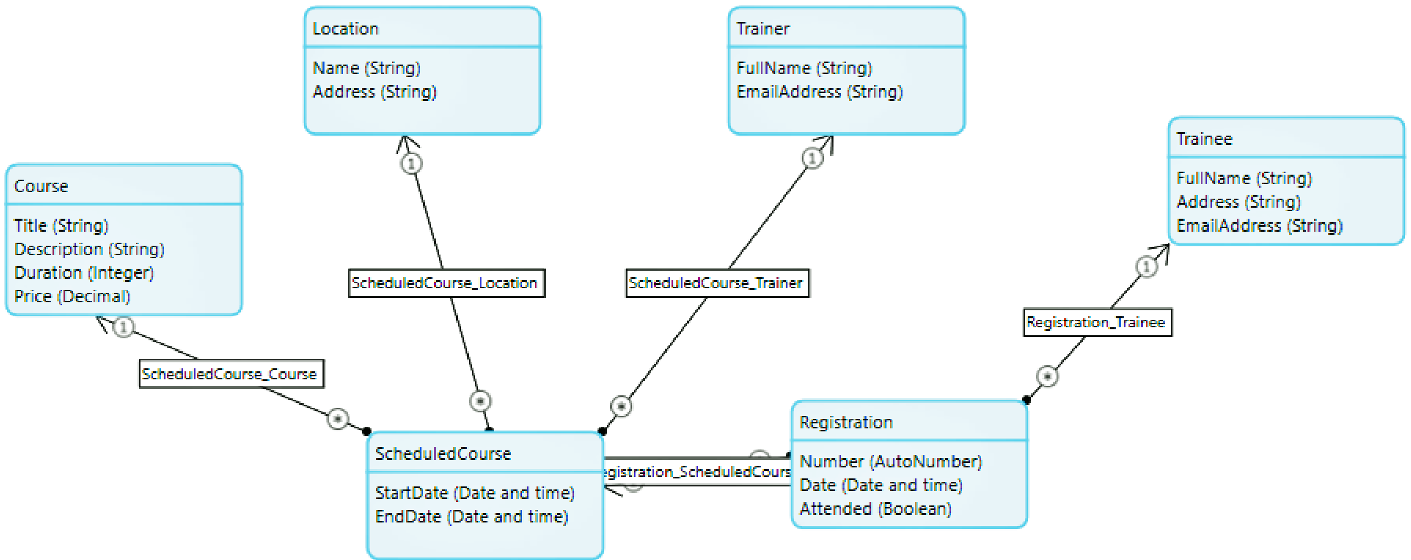

Figure 5: A simple data model defined in Mendix

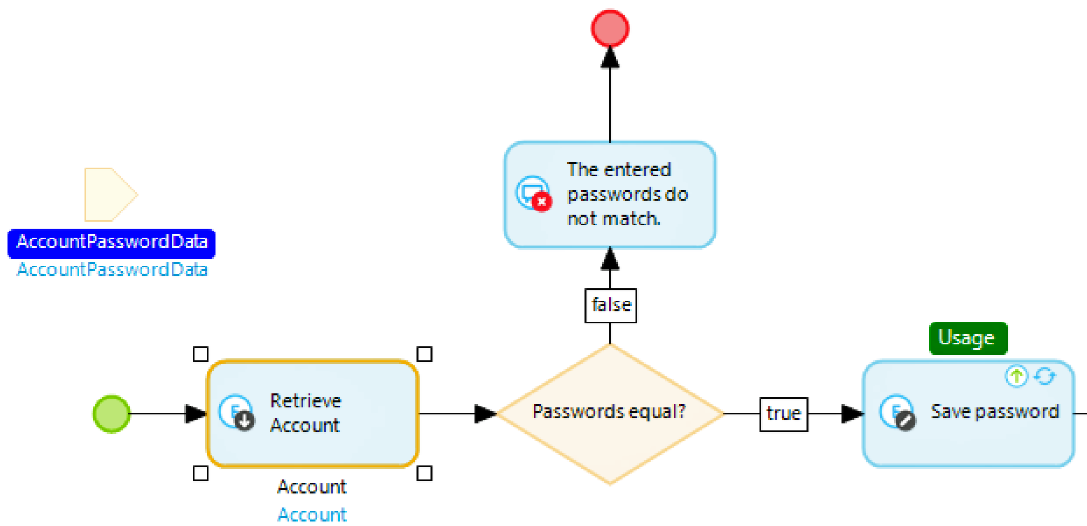

Figure 6: A simple logic defined in Mendix

2. *User interface definition* - secondly, the user configures forms and pages (e.g., see Fig. 4) used to define the application views, and later define and manage user roles and security mechanisms across at least entities, components, forms, and pages. It is here that drag-and-drop capabilities play a significant role to speed up development and render 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, e.g., shown in Fig. 6.
4. *Integration of external services via third-party APIs* - Fourth, LCDPs can provide means to consume external services via integration of different APIs. Investigating the documentation is necessary to understand the form and structure of the data that can be consumed by the adopted platform.
5. *Application Deployment* - In most platforms, it is possible to quickly preview the developed application and deploy it with few clicks.

2.2 An overview of representative low-code development platforms

This section presents an overview of eight low-code development platforms that have been considered as leaders in the related markets from recent Gartner [3] and Forrester [2] reports. These eight low-code platforms are assumed to be representative platforms for the benefit of our analysis that encompasses diverse feature capabilities mentioned in Table 1.

OutSystem [10] is a low-code development platform that allows developing desktop and mobile applica-

tions, which can run in the cloud or in local infrastructures. It provides inbuilt features which enable to publish an application via a URL with a single button click. OutSystems has two significant components. First, it has an intermediate Studio for database connection through .NET or Java and secondly, it has a service studio to specify the behaviour of the application being developed. Some of the supported applications in this platform are billing systems, CRMs, ERPs, extensions of existing ERP solutions, operational dashboards and business intelligence.

Mendix [9] is a low-code development platform that does not require any code writing and all features can be accessed through drag-and-drop capabilities while collaborating in real-time with peers. There is a visual development tool that helps to reuse various components to fasten the development process from the data model setup to the definition of user interfaces. Users can create some context-aware apps with pre-built connectors, including those for the IoT, machine learning, and cognitive services. Mendix is compatible with Docker[2] and Kubernetes[3], and it has several application templates that one can use as starting points. Mendix's Solution Gallery[4] is an additional resource that permits users to start from already developed solutions, and that might be already enough to satisfy the requirements of interest.

Zoho Creator [11] offers drag-and-drop facilities to make the development of forms, pages and dashboards easy. The provided user interface supports web design where the layout of the page reflects the resolution of the screen of the user (e.g., in the case of mobile or desktop applications). It also offers integration with other Zoho apps and other Salesforce[5] connectors. Customized workflows are essential features of Zoho Creator.

Microsoft PowerApps [12] supports drag-and-drop facilities and provides users with a collection of templates which allows reuse of already developed artifacts. A user can follow model-driven or canvas approaches while building applications. PowerApps integrates with many services in the Microsoft ecosystem such as Excel, Azure database[6] or similar connectors to legacy systems.

Google App Maker [13] allows organizations to create and publish custom enterprise applications on the platform powered by G Suite[7]. It utilizes a cloud-based development environment with advanced features such as in-built templates, drag-and-drop user interfaces, database editors, and file management facilities used while building an application. To build an extensive user experience, it uses standard languages such as HTML, Javascript, and CSS.

Kissflow [14] is a workflow automation software platform based on the cloud to help users to create and modify automated enterprise applications. Its main targets are small business applications with complete functional features which are essential for internal use, and human-centred workflows such as sales enquiry, purchase request, purchase catalogue, software directory, and sales pipeline. It supports integrations with third-party APIs, including Zapier[8], Dropbox[9], IFTTT[10], and Office 365[11].

Salesforce App Cloud [15] helps developers to build and publish cloud-based applications which are safe and scalable without considering the underlying technological stacks. It exhibits out-of-the-box tools and operations for automation by integrating them with external services. Some of the peculiar features are the extensive AppExchange marketplace[12] consisting of pre-built applications and components, reusable objects and elements, drag-and-drop process builder, and inbuilt kanban boards.

Appian [16] is one of the oldest low-code platform, which permits to create mobile and Web applications through a personalization tool, built-in team collaboration means, task management, and social intranet. Appian comes with a decision engine which is useful for modeling complex logic.

[2]https://www.docker.com/
[3]https://kubernetes.io/
[4]https://www.mendix.com/solutions/
[5]https://www.salesforce.com/it/
[6]https://azure.microsoft.com
[7]https://gsuite.google.com/
[8]https://zapier.com
[9]https://www.dropbox.com/
[10]https://ifttt.com/
[11]https://products.office.com/it-it/home
[12]https://appexchange.salesforce.com/

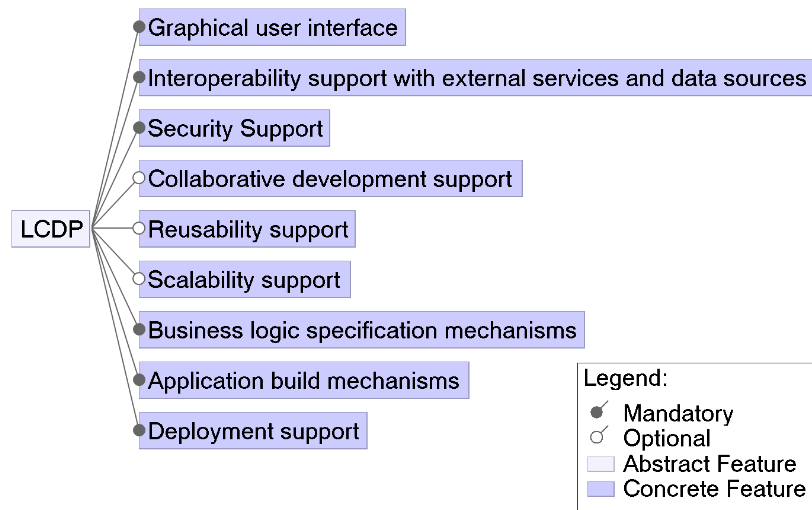

Figure 7: Feature diagram representing the top-level areas of variation for LCDPs

2.3 Taxonomy

In this section we introduce preparatory terms, which can facilitate the selection and comparison of different LCDPs. The features are derived by examining the requirements in building an application along with the capabilities that a low-code platform could offer in achieving the making of an application. In particular, by analysing the low-code development platforms described in the previous section we identified and modeled their variabilities and commonalities. Our results are documented using feature diagrams [17], which are a common notation in domain analysis [18]. Fig.7 shows the top-level feature diagram, where each subnode represents a major point of variation. Table 1 gives details about the taxonomy described in the following.

- *Graphical user interface:* This group of features represents the provided functionalities available in the front-end of the considered platform to support customer interactions. Examples of features included in such a group are drag-and-drop tools, forms, and advanced reporting means.
- *Interoperability support with external services and data sources*: This group of features is related to the possibility of interacting with external services such as Dropbox, Zapier, Sharepoint, and Office 365. Also, connection possibilities with different data sources to build forms and reports are included in such a group.
- *Security support*: The features in this group are related to the security aspects of the applications that are developed by means of the employed platform. The features included in such a group include authentication mechanisms, adopted security protocols, and user access control infrastructures.
- *Collaborative development support*: Such a group is related to the collaboration models (e.g., online and off-line) that are put in place to support the collaborative specification of applications among developers that are located in different locations.
- *Reusability support*: It is related to the mechanisms employed by each platform to enable the reuse of already developed artifacts. Examples of reusability mechanisms are pre-defined templates, pre-built dashboards, and built-in forms or reports.
- *Scalability support*: Such a group of feature permits developers to scale up applications according to different dimensions like the number of manageable active users, data traffic, and storage capability that a given application can handle.
- *Business logic specification mechanisms*: It refers to the provided means to specify the business logic of the application being modeled. The possibilities included in such a group are business rules engine, graphical workflow editor, and API support that allows one application to communicate with other application(s). Business logic can be implemented by using one or more API call(s).
- *Application build mechanisms*: It refers to the ways the specified application is built, i.e., by employing code generation techniques or through models at run-time approaches. In the former, the source code of the modeled application is generated from the specified models and subsequently deployed. In the latter, the specified models are interpreted and used to manage the run-time execution of the application.
- *Deployment support*: The features included in such a group are related to the available mechanisms for

Table 1: Taxonomy for Low-Code Development Platforms

Feature	Description
Graphical user interface	
Drag-and-drop designer	This feature enhances the user experience by permitting to drag all the items involved in making an app including actions, responses, connections, etc.
Point and click approach	This is similar to the drag-and-drop feature except it involves pointing on the item and clicking on the interface rather than dragging and dropping the item.
Pre-built forms/reports	This is off-the-shelf and most common reusable editable forms or reports that a user can use when developing an application.
Pre-built dashboards	This is off-the-shelf and most common dashboards that a user can use when developing an application.
Forms	This feature helps in creating a better user interface and user experience when developing applications. A form includes dashboards, custom forms, surveys, checklists, etc. which could be useful to enhance the usability of the application being developed.
Progress tracking	This features helps collaborators to combine their work and track the development progress of the application.
Advanced Reporting	This features enables the user to obtain a graphical reporting of the application usage. The graphical reporting includes graphs, tables, charts, etc.
Built-in workflows	This feature helps to concentrate the most common reusable workflows when creating applications.
Configurable workflows	Besides built-in workflows, the user should be able to customize workflows according to their needs.
Interoperability support	
Interoperability with external services	This feature is one of the most important features to incorporate different services and platforms including that of Microsoft, Google, etc. It also includes the interoperability possibilities among different low-code platforms.
Connection with data sources	This features connects the application with 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 which involves confidentiality, integrity and availability of an application, if and when required.
Platform security	The security and roles management is a key part in developing an application so that the confidentiality, integrity and authentication (CIA) can be ensured at the platform level.
Collaborative development support	
Off-line collaboration	Different developers can collaborate on the specification of the same application. They work off-line locally and then they commit to a remote server their changes, which need to be properly merged.
On-line collaboration	Different developers collaborate concurrently on the specification of the same application. Conflicts are managed at run-time.
Reusability support	
Built-in workflows	This feature helps to concentrate the most common reusable workflows in creating an application.
Pre-built forms/reports	This is off-the-shelf and most common reusable editable forms or reports that a user might want to employ when developing an application.
Pre-built dashboards	This is off-the-shelf and most common dashboards that a user might want to employ when developing an application.
Scalability	
Scalability on number of users	This features enables the application to scale-up with respect to the number of active users that are using that application at the same time.
Scalability on data traffic	This features enables the application to scale-up with respect to the volume of data traffic that are allowed by that application in a particular time.
Scalability on data storage	This features enables the application to scale-up with respect to the data storage capacity of that application.
Business logic specification mechanisms	
Business rules engine	This feature helps in executing one or more business rules that help in managing data according to user's requirements.
Graphical workflow editor	This feature helps to specify one or more business rules in a graphical manner.
AI enabled business logic	This is an important feature which uses Artificial Intelligence in learning the behaviour of an attributes and replicate those behaviours according to learning mechanisms.
Application build mechanisms	
Code generation	According to this feature, the source code of the modeled application is generated and subsequently deployed before its execution.
Models at run-time	The model of the specified application is interpreted and used at run-time during the execution of the modeled application without performing any code generation phase.
Deployment support	
Deployment on cloud	This features enables an application to be deployed online in a cloud infrastructure when the application is ready to deployed and used.
Deployment on local infrastructures	This features enables an application to be deployed locally on the user organization's infrastructure when the application is ready to be deployed and used.
Kinds of supported applications	
Event monitoring	This kind of applications involves the process of collecting data, analyzing the event that can be caused by the data, and signalling any events occurring on the data to the user.
Process automation	This kind of applications focuses on automating complex processes, such as workflows, which can takes place with minimal human intervention.
Approval process control	This kind of applications consists of processes of creating and managing work approvals depending on the authorization of the user. For example, payment tasks should be managed by the approval of authorized personnel only.
Escalation management	This kind of applications are in the domain of customer service and focuses on the management of user viewpoints that filter out aspects that are not under the user competences.
Inventory management	This kind of applications is for monitoring the inflow and outflow of goods and manages the right amount of goods to be stored.
Quality management	This kind of applications is for managing the quality of software projects, e.g., by focusing on planning, assurance, control and improvements of quality factors.
Workflow management	This kind of applications is defined as sequences of tasks to be performed and monitored during their execution, e.g., to check the performance and correctness of the overall workflow.

deploying the modeled application. For instance, once the system has been specified and built, it can be published in different app stores and deployed in local or cloud infrastructures.

In addition to the top-level features shown in Fig. 7, LCDPs can be classified also with respect to the *Kinds of supported applications*. In particular, each LCDP can specifically support the development of one or more kinds of applications including Web portals, business process automation systems, and quality management applications.

2.4 Comparison of relevant LCDPs

In this section, we make use of the taxonomy previously presented to compare the eight low-code development platforms overviewed in Sec. 2.2. Table 11 shows the outcome of the performed comparison by showing the corresponding supported features for each platform. The data shown in Table 11 are mainly obtained by considering the official resources of each platform as referenced by [10],[9], [11], [12], [13], [14], [15], [16], and by considering the experience we gained during the development of a benchmark application as discussed in the next section.

Table 2: Comparison of analysed low-code development platforms

Feature	OutSystems	Mendix	Zoho Creator	MS PowerApp	Google App Maker	Kissflow	Salesforce App Cloud	Appian
Graphical user interface								
Drag-and-drop designer	✓	✓	✓		✓	✓	✓	✓
Point and click approach				✓				
Pre-built forms/reports	✓	✓	✓	✓	✓	✓	✓	✓
Pre-built dashboards	✓		✓	✓		✓	✓	
Forms			✓	✓				
Progress tracking	✓	✓	✓	✓	✓	✓	✓	✓
Advanced reporting						✓		
Built-in workflows			✓			✓	✓	
Configurable workflows			✓			✓	✓	
Interoperability support								
Interoperability with external service	✓	✓	✓	✓		✓	✓	✓
Connection with data sources	✓	✓	✓	✓	✓	✓	✓	✓
Security Support								
Application security	✓	✓	✓	✓	✓	✓	✓	✓
Platform security	✓	✓	✓	✓	✓	✓	✓	✓
Collaborative development support								
Off-line collaboration	✓	✓	✓	✓	✓	✓	✓	✓
On-line collaboration	✓	✓			✓	✓	✓	✓
Reusability support								
Built-in workflows			✓			✓	✓	
Pre-built forms/reports	✓	✓	✓	✓	✓	✓	✓	✓
Pre-built dashboards	✓		✓	✓		✓	✓	
Scalability								
Scalability on number of users	✓	✓	✓	✓	✓	✓	✓	✓
Scalability on data traffic	✓	✓	✓	✓	✓	✓	✓	
Scalability on data storage	✓	✓	✓	✓	✓	✓	✓	
Business logic specification mechanisms								
Business rules engine	✓	✓	✓	✓	✓	✓	✓	✓
Graphical workflow editor	✓	✓				✓	✓	
AI enabled business logic	✓					✓	✓	✓
Application build mechanisms								
Code generation	✓							
Models at run-time		✓	✓	✓	✓	✓	✓	✓
Deployment support								
Deployment on cloud	✓	✓	✓	✓	✓	✓	✓	✓
Deployment on local infrastructures	✓	✓					✓	✓
Kinds of supported applications								
Event monitoring	✓	✓	✓	✓	✓	✓	✓	✓
Process automation	✓		✓	✓	✓	✓		✓
Approval process control					✓			
Escalation management						✓		
Inventory management	✓	✓	✓	✓	✓	✓	✓	✓
Quality management		✓	✓	✓	✓	✓	✓	✓
Workflow management	✓	✓	✓	✓	✓	✓	✓	✓

2.4.1 Features and capabilities

The essential and distinguishing features and capabilities of the analyzed low-code platforms can be summarized as follows: *OutSystems* provides developers with a quick mechanism to publish developed applications, the capability to connect different services, to develop responsive mobile and web-apps, security mechanisms and real-time dashboards. *Mendix* supports collaborative project management and end-to-end development, pre-built templates with app stores and interactive application analytics. *Zoho Creator* has an easy to use form builder, user-friendly and mobile-friendly user interfaces, and the capability of app-integration among different Zoho CRM apps, Salesforce, etc. It also supports pre-built templates and customized workflows. *Microsoft PowerApp* supports integration with Office 365, pre-built templates, easy mobile and tablet application conversion, and the capability to connect with third-party applications for basic application development. *Google App Maker* has a drag-and-drop function similar to most of the analyzed low-code platforms, app preview, reusable templates, deployment settings, means to specify access roles, built-in tutorials and google analytic integration. *Kissflow* supports progress tracking, custom and pre-built reports, collaborative features, and the possibility 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 market place for pre-built apps and components, reusable objects and elements, in-built 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-enabled complex logic.

2.4.2 Additional aspects for comparing LCDPs

The taxonomy discussed in the previous section plays an important role when users have to compare candidate LCDPs and select one among possible alternatives. Further than the features previously presented, we identified additional aspects that are orthogonal to the presented taxonomy, and that can be taken into account when decision-makers have to decide if a low-code development platform has to be adopted and which one.

Type of solutions to be developed: there are two main types of applications that can be developed employing LCDPs, namely B2B (Business to Business) and B2C (Business to Customer solution). B2B solutions provide users with business process management (BPM) functionalities such as creation, optimization, and automation of business process activities. Examples of B2B solutions include hotel management, inventory management, and human resource management. Multiple applications can be combined in a B2B solution. B2C solutions provide more straightforward answers for end customers. B2C solutions are for developing single applications such as websites and customer relations management applications. The interactivity aspects of B2C is much more crucial than B2B ones.

Size of the user's company/organization: another dimension to be considered when selecting LCDPs, is the size of the company/organization that is going to adopt the selected LCDP. Organizations fall under three possible categories: *small* (with less than 50 employees), *medium* (if the number of employees is in between 50 to 1000), *large* (if the number of employees is higher than 1000). Thus, the decision-maker must keep in mind the organization size to identify the optimal solution according to her needs. Any organization who wishes to scale their enterprise at an optimum cost need to select an LCDP based on the strength of the company. LCDPs such as Salesforce app cloud, Mendix, and OutSystems support large enterprises, and they are used to develop large and scalable applications. Google App Cloud, Appian, Zoho Creator are instead mainly for supporting small to medium scale enterprises and they are relatively cheaper.

Cost and time spent to learn the platform: 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 to learn 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.

The price of the low-code platform: it is one of the most critical criteria, especially for small or medium-scale companies. The price of the platform 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 in 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: a short experience report

The making of such platforms capable of giving citizen developers the ability to build fully-fledged applications faster and efficiently comes on a cost. Critical architectural decisions are made to ensure minimal coding, speed, flexibility, less upfront investment and out-of-box functionalities that deliver the full application faster. However, decisions that are usually taken during the usage of LCDPs can give place to some issues that might emerge later on. In particular, to get insights into LCDPs, we developed the same benchmark application by employing different platforms, and in particular Google App Maker, Mendix, Microsoft PowerApps and OutSystems. The benchmark application is a course management system intended to facilitate trainers and trainees to manage their courses, schedules, registrations and attendance. Despite the simplicity of the application, it exhibits general user requirements that are common during the development of typical functionalities such as management of data, their retrieval and visualization. Moreover, we had the possibility of integrating external services via third-party APIs. We managed to investigate how reusable code and artefacts developed in one platform can be integrated into other low code platforms hence smoothing the path toward discovery and reuse of already proven artefacts across different platforms.

The first performed activity to develop the benchmark application was the elicitation of the related requirements. We came up with the corresponding use cases, and thus with the functional requirements of the system. According to the performed experience, software applications can be built in LCDPs by following two main approaches:

- *UI to Data* - the developer starts building the application by creating a user interface and then linking it with the needed 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 deployment. LCDPs such as Mendix, Zoho Creator, Microsoft PowerApps, and Kissflow can follow this approach.
- *Data to UI* - it is a data-driven approach that starts from data modeling and then builds the user interface of the application followed by the specification of business logic rules and workflows. Afterwards, it leads to the integration of external services, if needed before the deployment of the application. LCDPs such as OutSystems, Mendix, Zoho Creator, Microsoft PowerApps, Salesforce App Cloud and Appian can follow this approach.

Specification of business logic rules, workflows, and the integration of external services can be swapped according to the developer's style in both the approaches mentioned above.

By developing the considered benchmark applications with the considered LCDPs, we managed to identify some challenges that users and developers are likely to face along the course of development in LCDPs such as interoperability issues among different low-code platforms, extensibility limitations, steep learning curves, and scalability issues [19] [20][3]. Below we discuss such challenges that transcend most of the low-code development platforms we surveyed. We will not discuss potential challenges that might also occur concerning code optimization or security and compliance risks, because we were not able to deeply assess these features due to the lack or limited visibility of the considered low-code platforms. However, we acknowledge that such aspects should be investigated in the future to give a more broad perspective about potential challenges that might affect LCDPs.

Low-code platforms' interoperability: this characteristic ensures interaction and exchange of information and artefacts among different low-code platforms, e.g., to share architectural design, implementation or developed services. Such a feature is also essential to mitigate issues related to vendor lock-ins. Unfortunately, most low-code platforms are proprietary and closed sources. There is a lack of standards in this domain by hampering the development and collaboration among different engineers and developers. Thus, they are unable to learn from one another, and the reuse of already defined architectural designs, artefacts and implementations are still hampered.

Extensibility: the ability to add new functionalities not offered by the considered platform is hard in such proprietary platforms or even impossible. Due to lack of standards, some of them require extensive coding to add new capabilities, which have to adhere to architectural and design constraints of the platform being extended.

Learning curve: most of the platforms have less intuitive graphical interfaces. For some of them, drag-and-drop capabilities are limited, and they do not provide enough teaching material, including sample applications and online tutorials to learn the platform. Consequently, the platform adoption can be affected. The adoption of some platforms still requires knowledge in software development, thus limiting their adoption from citizen developers who are supposed to be the main target of these platforms and products.

Scalability: Low code platforms should be preferably based on the cloud and should be able to handle intensive computations and to manage big data, which get produced at high velocity, variety, and volume [21]. However, due to lack of open standards for such platforms, it is very challenging to assess, research and contribute to the scalability of these platforms.

Overall, LCDPs are suitable for organizations that have limited IT resources and budget because they can deliver fully-featured products in a short time, as in the case of CRM applications. However, the development possibilities depend on the functionalities provided by the available modules, and users might need accommodating their initial requirements depending on the options offered by the employed platform. Third-party integration and the management and maintenance of the developed applications can be hampered depending on the extensibility capabilities of the employed LCDPs.

2.6 Conclusion

Over the last years, the interest around LCDPs has significantly increased from both academia and industry. Understanding and comparing hundreds of low-code platforms [22] can be a strenuous and challenging task without the availability of an appropriate conceptual framework underpinning their evaluation. In this section, we analyzed eight low-code platforms that are considered as leaders in the related market, to identify their commonalities and variability points. An organized set of distinguishing features has been defined and used to compare the considered platforms. A short experience report has been also presented to discuss some essentials features of each platform, limitations and challenges we identified during the course of development of our benchmark application.
As shown in the taxonomy drawn in Fig. 7, LCDPs provide users with modeling functionalities and the support for reusing already developed artifacts. In the next section, we discuss cloud-based modeling functionalities provided by existing platforms by focusing on the IoT domain, which is of interest for the Lowcomote project, particularly for *ESR4 - Urban Area Management in Smart Cities*. Chapter 4 instead focuses on the backend of LCDPs, particularly on the proposed Low-Code Engineering Repository Architecture. The section will emphasize on the architectural view decomposition based on *"4 + 1 "* view model architecture.

3 Cloud-based modeling: a survey, open challenges and opportunities

Cloud-based modeling is one of the relevant topics in the model-driven engineering community due to the induced possibilities of designing, developing, analyzing and deploying applications seemingly with reduced efforts. This has also been recently favored by the increasing adoption of low-code development platforms (LCDP). Ideally, domain-specific low-code development platforms have to run on cloud infrastructures, even though in some industrial settings such as IoT, domain-specific modeling environment tends to be local-based [23]. Nowadays, industries and companies are trying to migrate their modeling infrastructures to the cloud. However, especially in industrial contexts, the existing modeling infrastructures are implemented in complex environments in which the migration cost can be far more expensive and very complicated.

The future of modeling will forcefully be cloud-based [24]. Several initiatives, including Visual Studio Code[13], Eclipse Che[14], Theia[15], and others have shown a lot of potential in shifting modeling environments from local-based and monolithic installations to cloud-based platforms in order to eliminate accidental complexity and expand the variety of available functionalities [24].

In the IoT domain, modeling and development infrastructures need to consider several heterogeneous aspects of the system's data, communication, and implementation layers. This chapter looks at what has been done so far in the IoT domain to support IoT systems' development through cloud-based modeling approaches. 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. Following an examination of 625 articles, we identified 22 different cloud-based IoT system development tools and platforms.

We perform an analysis of the various issues that the IoT community is encountering while implementing cloud-based modeling tools. As a result, we take a deeper look at a few options and discuss the research and development opportunities enabled by adopting cloud-based modeling approaches in the IoT domain.

The remainder of this chapter is organized as follows: Section 3.1 provides an overview of cloud-based modeling approaches and highlights motivations and needs for modeling IoT systems by means of dedicated cloud-based environments. Section 3.2 presents the research methodology we have used to conduct the survey. Sections 3.3–3.5 discuss the findings of the performed analysis, which has been performed to answer three dedicated research questions. Section 3.6 discusses the related work, whereas Section 3.7 concludes the section.

3.1 Background

The significant advancements in computing power, data storage and processing are revolutionizing the development and research of complex systems in several domains, including that of the Internet of Things (IoT) [25]. IoT systems enable the integration of intelligent features into daily human activities through the automation of services. In particular, such systems allow automation of low-level services that used to be error-prone if done by humans. Moreover, they increase efficacy in current engineering solutions and connect a range of many devices that render our environment smart. Recent reports predict that more than 100 billion devices will be connected by 2025 and 11 trillions dollars of global market capital will be reached [26]. However, to unleash the full potential of these systems, it is necessary that also citizen developers can take part in the development of custom IoT applications [23].
The development and consumption of IoT systems are becoming way more complex, and involving end-users is more challenging due to the heterogeneity of the hardware and required expertise [23]. This complexity originates from various sources. IoT applications are complex systems that use heterogeneous devices and data sources. Besides, IoT systems require enormous efforts and investments both in their implementation and maintenance. Moreover, the systems are implemented using code-centric approaches that make it challenging to foster the inclusion of IoT domain experts and other stakeholders with less IoT programming skills [23].
Due to the ever-changing requirements and the shortage of engineering experts that develop these systems

[13] https://code.visualstudio.com
[14] https://www.eclipse.org/che/
[15] https://theia-ide.org

robustly and securely [27], the way forward entails the need to pave the way for domain experts and other stakeholders to integrate IoT capabilities in their daily tasks [23]. Several approaches are being discussed, while practical solutions are finding a way to facilitate IoT application usage and development very accessible. Model-driven engineering (MDE) promotes the systematic use of models as the primary abstraction entities all along the development of complex systems by fostering abstraction and automation [28]. Models in the context of MDE are not sketches, drawings that serve purpose only in design, but they prevail until the end of the development cycle of these systems as machine-readable and processable abstractions [29]. MDE favors collaboration of engineers and stakeholders, as both work together toward the completion of the conceived products and foster integration of different engineering processes [27].
However, MDE itself has faced challenges that have shifted the focus of the development of such complex and heterogeneous systems from local environments to the cloud [30]. Modeling-as-a-Service is gaining momentum as the MDE research community is migrating modeling tools and services to the cloud. This migration is encouraged by several out-of-box benefits in cloud computing, such as easy discovery and reuse of services and artifacts [28]. It has enabled efficient self-healing mechanisms to detect, diagnose, and countermeasure threats and foster collaboration among stakeholders and engineers [31]. Furthermore, migrating modeling artifacts and services on the cloud can facilitate end-users easy accessibility, hence supporting sustainable management and disaster recovery of model artifacts and tools [32].

3.2 Study design

This section aims to analyze how the IoT domain is coping with the trend of moving existing modeling and development infrastructures to the cloud. To this end, we followed the process shown in Fig. 8 according to the methodology presented in [33].
In particular, the search and selection process was mainly conducted into four main phases. In the first phase, we formally and explicitly represented the problem to get a head start on the search. Second, we defined a search string and selected well-known academic search databases. Third, we performed a search to gather papers to answer properly defined research questions. Fourth, we narrowed down the potential papers and mapped them based on their similarity and variability. Finally, we analyzed the collected papers and elaborated some recommendations on the identified difficulties.

Phase 1: Problem formalisation - This phase mainly focused on formalizing the problem we wanted to solve by looking at the current model-driven engineering tendency. One of the sources of inspiration for this study was the work in [24] which addresses the topic of *"what is the future of modeling?"*. Thus, we came up with the formulation of the following research questions:

- ***RQ1:*** *How is the IoT community adopting cloud-based modeling approaches?*
- ***RQ2:*** *What challenges do researchers face when developing cloud-based IoT modeling and development infras-*

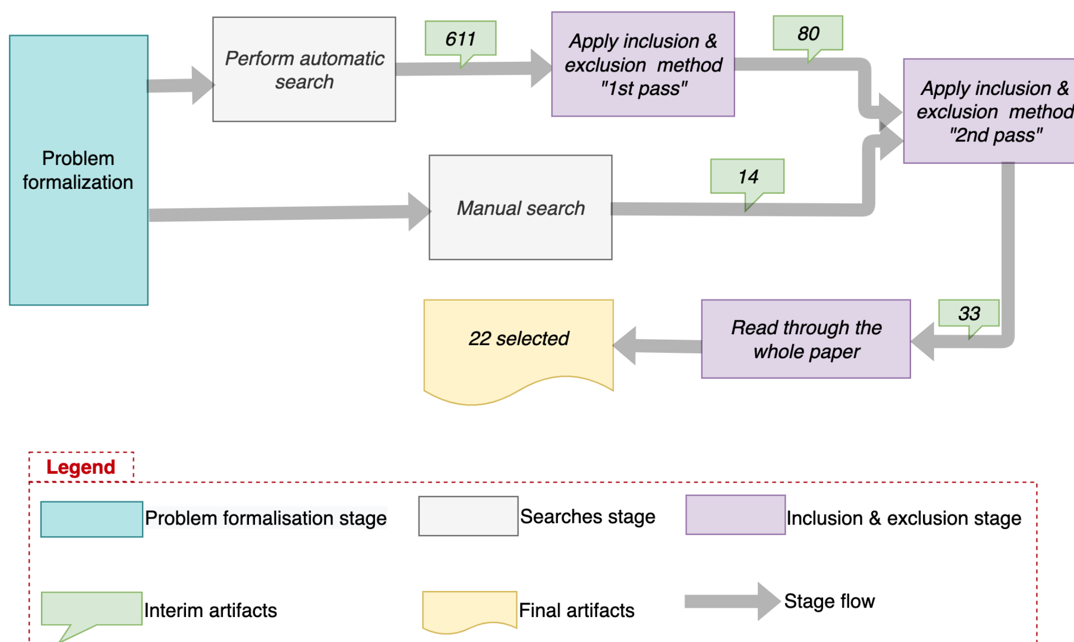

Figure 8: Search and selection process

tructures?

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

Phase 2: Automatic & manual search: In this phase, we applied a search string to different academic databases, i.e., Scopus (Elsevier)[16], IEEE Xplore[17] and ACM library[18] by limiting the search on the last 10 years. Additionally, we also performed a manual search, primarily using Google Scholar. The query string we used for the automatic search was: `("MDE" OR "Model Driven Engineering") AND ("IoT" OR "Internet of Things") AND ("Cloud" OR "Web")`.
Table 3 shows the number of papers we managed to collect in this phase.

Table 3: 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 shows that 611 publications were initially discovered from different sources, in addition to the 14 papers that were manually found and considered relevant for the study. At this point, we have reviewed the paper's title, keyword, and abstract to exclude papers that were not satisfying the following criteria:

- Studies published in a peer-reviewed journal, conference, or workshop.
- Studies written in English.
- Papers that focus explicitly on the Internet of Things (IoT) topic.
- Studies that propose a cloud-based modeling approach, either explicitly or implicitly.

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

Phase 4: Inclusion & exclusion, 2nd pass: In this phase, we read the introduction and the conclusion of the papers previously collected. We also removed some duplicates. Various documents were rejected during this phase for a variety of reasons, for instance, because the presented approach is not explicitly offering an IoT-based cloud-based development environment. At the end of this phase, we ended up with 33 documents.

Phase 5: Reading of the whole paper text: We've gone over the entire articles in this phase, focusing on the proposed approaches and their evaluation sections. Several documents were discarded because of different reasons. For instance, papers that presented hybrid solutions (e.g., enabling local modeling with the possibility of storing models on remote repositories) were discarded. In addition, the approaches that claim to build web-based IoT data-wrangling platforms by reusing existing IoT data storage platforms were also discarded. Finally, we selected 22 documents that leverage a cloud-based modeling environment to design, develop, or deploy IoT applications.
Figure 9 shows the distribution of the selected approaches with respect to their corresponding sources. As you might notice from Fig. 9, a portion of the selected approach (4 out of 22) was found manually. This is because of our previous knowledge on this topic in terms of framework and tools.
In the following, the research questions presented in Sec. 3.2 are answered singularly by analyzing the research papers that have been collected as previously described.

3.3 Cloud-based modeling approaches (RQ1)

This section goes over different cloud-based modeling approaches that target the IoT domain. We organized the analysed approaches into three categories according to their main focus of interest i.e., modeling IoT structural aspects, service-oriented approaches, and deployment orchestrations. The aim is to answer the research question ***RQ1:*** *How is the IoT community adopting cloud-based modeling approaches?*

[16]https://www.elsevier.com/
[17]https://ieeexplore.ieee.org
[18]https://dl.acm.org/

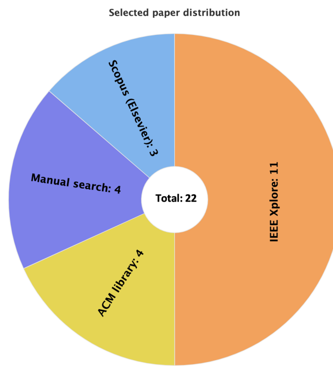

Figure 9: Selected paper distribution

Modeling IoT structural aspects: DSL-4-IoT [34] is a cloud-based modeling tool for the IoT domain, which comprises a JavaScript-based graphical frontend programming language and a runtime "OpenHAB" execution engine. DSL-4-IoT provides a multistage model-driven approach for the design of IoT applications that supports all stages of the life cycle of these systems. Automatic model transformations are provided to refine abstract models elements into concrete ones. Those transformation results formatted as JSON-Arrays are passed to the OpenHAB runtime engine for execution.
BIoTA [35] offers a cloud-based modeling approach for IoT architectures. A graphical DSL and supporting tools allow users to perform syntax and semantic analysis. BIoTA renders it possible to computationally formalize a software architecture suggested by a user according to formal automata techniques. The component & connectors are created following specific rules to meet IoT-specific scenarios while exporting the resulting software architecture towards a Docker-based deployment infrastructure.
Node-RED [36] is a popular and extensible model-driven framework for tying together IoT devices, APIs, and web services in a homogenous manner. It provides users with a Web-based graphical editor with drag-and-drop facilities. In Node-RED, the user can also create and deploy real-time dashboards.
AutoIoT [37] is a Web-based platform with a Graphical User Interface (GUI) that allows programmers to deploy and configure IoT systems quickly. The final system-generated artifact is a Flask project[19] (a python micro-framework) that can be run as is or extended to meet the needs of developers that might require more complex functionalities. The final system can also connect to an MQTT Broker, storing and querying data in a database, presenting data to users, and exchanging them with other systems. AutoIoT had later been extended in [38] to allow users to model their IoT systems in terms of JSON files.
In [39], a cloud-based textual language and tool for Event-based Configuration of Smart Environments (ECSE) had been proposed. The tool enables the end-user, being expert or not, to configure a smart environment by employing an ontology-based model. In their approach, the authors used the Resource Description Frameworks (RDF) to define the event-action rules.
AtmosphericIoT [40] is a cloud-based domain-specific language and tools for building, connecting, and managing IoT systems. AtmosphericIoT Studio is a free online IDE that lets you create all kinds of device firmware, mobile apps, and cloud dashboards. It links devices via Wi-Fi, Bluetooth, BLE, Sigfox, LoRa, ZigBee, NFC, satellite, and cellular networks.
In [41], authors propose a model-based approach for creating responsive and configurable Web of things user interfaces. Models@Runtime are used to produce runtime interfaces based on a formal model named Thing Description (TD). TD's goal is to expose Web Things (WT) attributes, actions, and events to the outside ecosystem. The modeling language has been developed in JavaScript, using the VueJS framework, and it is publicly available[20].
In [42], the authors presented a model-driven approach to the development of IoT system interfaces. In their work, they proposed a design pattern and the required components for designing such interfaces. The authors implemented a platform for the development of IoT mobile and web applications based on WebRatio,[21] a generic cloud-based model-driven development and code generation framework.
FloWare Core [43] is a model-driven open-source 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 stages of the IoT application development process. The system configures the IoT application

[19]https://flask.palletsprojects.com
[20]https://github.com/smar2t/td_interface_builder
[21]https://www.webratio.com/

following the IoT system model supplied by the IoT developer. A Node-RED engine [36] is integrated in FloWare.
Vitruvius [44] is an MDD platform that allows users with no programming experience to create and deploy complex IoT web applications based on real-time data from connected vehicles and sensors. Users can design their ViWapplications straight from the web using a custom Vitruvius XML domain-specific language. Furthermore, Vitruvius provides a variety of recommendation and auto-completion features that aid in creating applications by reducing the amount of XML code to be written.

Service-oriented approaches: This category includes approaches providing users with cloud-based modeling environments targeting service-oriented architectures. Thus, different services are connected to build the final IoT systems.
MIDGAR [45] is an IoT platform specifically developed to address the service generation of applications that interconnect heterogeneous objects. This is achieved by using a graphical DSL in which the user can interconnect and specify the execution flow of different things. Once the desired model is ready, it gets processed through the service generation layer, generating a tree-based representation model. The model is then used to generate a Java application that can be compiled and run on the server.
IADev [46] is a model-driven development framework that orchestrates IoT services and generates software implementation artifacts for heterogeneous IoT systems while supporting multi-level modeling and transformation. This is accomplished by converting requirements into a solution architecture using attribute-driven design. In addition, the components of the produced application communicate using RESTful APIs.
LogicIoT [47] offer a textual web-based DSL to ease data access and processing semantics in IoT and Smart Cities settings. LogicIoT is implemented as a set of custom Jakarta Server Pages (JSP)[22] in which different custom JSP tags have been implemented to define the modeling semantics. The language consists of seven constructs: relations, triggers, endpoints, timers, facts, rules, and modules. Using the custom tags, the user can define the application's operations required to enable the communication between process instances and sensors without being concerned with low-level programming details.
glue.things [48] offers a cloud-based mashup platform for wiring data of Web-enabled IoT devices and Web services. glue.things take care of both the delivery and maintenance of device data streams, apps, and their integration. In this regard, glue.things rely on well-established real-time communication networks to facilitate device integration and data stream management. The glue.things modeling tool combines device and real-time communication, allowing users to describe element's triggers and actions and deploy them in a distributed manner.
In [49], the authors proposed 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 [50], a portable web-based graphical end-user programming environment for personal apps is proposed. This tool allows the users to discover smart things in their environment and create personalized applications that represent their own needs. Each of the defined smart objects can provide various features that can be published via 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 [51] 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 sensor service life-cycle. It allows the user to be abstract and simulate sensor behavior through an events-based fashion. This is achieved by having the ECA rules listen for the occurrence of a predetermined "event" and respond by performing the "action" if the rule's "condition" is met. Finally, the generated cloud sensor application can be used in any cloud-based application which needs sensor data.
In [52], the authors introduced an integrated graphical programming tool based on a goal-driven approach, in which end users are only required to specify their purpose in a machine-understandable manner, rather than designing a service architecture that fulfills their goal. This allows a smart environment's ultimate purpose to be graphically represented, but the complexities of the underlying semantics are hidden. A reasoning component uses the provided goal statement and analyses whether the goal can be achieved given the set of available services and infers whatever user actions (i.e., requests involving REST resources) are required to achieve it.
InteroEvery [53] promotes a microservice-based architecture to deal with interoperability issues of the

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

IoT domain. First, an IoT system is configured through a web-based graphical interface showing each microservice's functionalities. A universal broker connects a dedicated interoperability microservice with various adaption microservices depending on employed choreography patterns.

Model-based deployment orchestration: As IoT system deployment happens at different layers of abstractions, this section presents the identified approaches, which aim to orchestrate the deployment mechanisms of IoT systems using cloud-based modeling environments.
DoS-IL [54, 55] is a textual domain scripting language for resource-constrained IoT devices. It allows changing the system's behavior after deployment through a lightweight script written with the DoS-IL language and stored in a gateway at the fog layer. The gateway hosts an interpreter to execute DoS-IL scripts.
TOSCA (Topology and Orchestration Specification for Cloud Applications) [56] aims at improving the reusability of service management processes and automate IoT application deployment in heterogeneous environments. In TOSCA, common IoT components such as gateways and drivers can be modeled. In addition, the gateway-specific artifacts necessary for application deployment can also be specified to ease the deployment tasks.
GENESIS (Generation and Deployment of Smart IoT Systems) [57] is a textual cloud-based domain-specific modeling language that supports 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 regarded as an assembly of components. The GENESIS execution engines support three types of deployable artifacts, namely ThingML model [58], Node-RED container [36] and any black-box deployable artifact (e.g., an executable jar). The created deployment model is subsequently passed to GENESIS deployment execution engine, which is in charge of deploying the software components, ensuring communication between them, supplying the required cloud resources, and monitoring the deployment's status.

Discussion: As previously presented, several approaches are available to support cloud-based modeling in the IoT domain. Table 4 shows an overview of the analyzed approaches; half of them are concerned with structural issues, whereas only a few deal with deployment concerns. The current state of the art suggests that there is no predominant common language, although graphical syntax is preferred.
Most of the analyzed approaches are supported by tools, which are not open source. This goes hand in hand with the public availability of the methodologies. We can observe that all the tools that are not open-source are also not publicly accessible. When looking at industrial settings, this is especially true when it comes to internal proprietary tools.
While analyzing each approach, we also looked at the supporting infrastructures and their ability to generate deployable artifacts. In this regard, we have discovered that JavaScript-based environments like Node.js and Angural.js are widely used for tool development. This might be due to the fact they are among the modern languages for front-end technology implementation. On the other hand, it appears that the majority of techniques generate artifacts, even though few of them are standalone deployable components. It is also worth noting that the generated deployable artifacts can only be deployed within the same original environment in most of the cases. To ensure interoperability, scalability, and reusability of the tools, the generated artifacts should generally be deployed anywhere.

3.4 Open challenges (RQ2)

Multiple issues have arisen as a result of the expansion of connected smart and sensor devices, as well as the increased usage of cloud-based models [31]. As a typical IoT system consists of multiple complex sub-systems, having an all-in cloud-based environment can become even more complicated. On the other hand, overcoming these barriers is worth the effort because it opens up more opportunities. This section elaborates on the current challenges IoT systems face while developing and integrating such tools in a cloud-based environment. Essentially, we are answering the research question ***RQ2****: What challenges do researchers face when developing cloud-based IoT modeling and development infrastructures?*

Extensibility mechanisms: Extensible platforms allow the addition of new capabilities without having to restructure the entire ecosystem. Because IoT systems are distributed, a typically recommended architecture would be to use the micro-service architecture throughout the development process [25]. Aside from that, IoT systems may require additional interactions with third-party technologies. As a result of the previous scenario, developing tools to design and develop such distributed applications on the cloud need efficient

Table 4: 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
[39]	Structure	Textual	no	no	Smart-M3	–
AtmosphereIoT	Structure	Graphical	no	yes	Multi-platform	Multi-platform apps
[41]	Structure	Textual	yes	yes	js,VueJS	UI code
[42]	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
[49]	Service	Textual	no	no	Firebase&Node.js	–
TOSCA	Deployment	Textual	yes	yes	Multi-platform	Config files
[50]	Service	Graphical	no	no	-	-
E-SODA	Service	Textual	yes	no	OSGI cloud	OSGI java bundles
[52]	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

tools that traditional domain specialists may not have. Accessibility mechanisms are presented through tools like [36, 43], but there is still a lot to be done. Currently, domain experts must provide cloud-based automation mechanisms and tools to allow citizen developers to add new features without requiring sophisticated knowledge or changing existing architectures.

Heterogeneity: It is an important challenge of the IoT domain, which involves different players developing various applications running at different layers, namely the edge, fog, and cloud [25]. In addition, deployments and data consumption methods are very diverse, increasing the complexity of traditional code-centric approaches [59]. Cloud-based modeling in IoT brings even more sophistication regarding the environment in which the system should be designed and developed. The typical cloud-based modeling platform should foster the integration of heterogeneous technological implementations, promoting reusability and developing solutions close to the problem domain. Approaches such as [45, 46, 56] have presented different strategies to tackle such issues, but much more have to be investigated.

Scalability: IoT systems are expected to handle a wide range of users, perform demanding computations, and share enormous amounts of data among nodes. Therefore, supporting cloud-based modeling approaches must be implemented in such a way that scalability concerns are mitigated. One of the approaches to tackle such challenges is to adopt container-based orchestration tools such as Kubernetes. The use of such tools can offer out-of-box features such as self-healing, fault-tolerance, and elasticity of containerized resources [26]. This will also help automate cognitive processes that can detect scalability needs and adjust autonomously without human intervention.

Interoperability: The interoperability of various tools, services, and resources is critical in the IoT domain. The interoperability of cloud-based modeling platforms, particularly in the IoT area, is currently limited since different tools run in different environments and have different natures. A tool like [53] promotes the micro-service architecture by allowing all parts of the system to communicate with each other. Several regulations, such as standardization, will need to be implemented to achieve interoperability among different cloud-based modeling environments. To address interoperability concerns, technologies like [36, 34, 37, 43] promote a common format based on JSON to encode models. It is worth noting that adopting Model-as-a-Service (MaaS) architectures could also promote the interoperability of services and artifacts.

Learning curve: It is not easy to find professionals who can master and combine the different sophisticated technologies involved in developing and managing IoT systems. IoT domain experts may lack modern programming expertise, whereas experienced software programmers may lack modeling domain expertise. For instance, conceiving a cloud-based code generator requires understanding different model transformation techniques and particular programming abilities; Implementing a visual mashup tool will necessitate knowledge of modern languages such as JavaScript, HTML, and CSS.

Security concerns: Current IoT systems suffer from security concerns as data are collected from a wide distribution of private and public nodes. Furthermore, the data is transferred using remote IoT gateways, which might get exposed in the process. This heterogeneity of secured and unsecured data might favor attackers to target devices and compromise the integrity of data and operations [60]. Therefore, proper abstractions and automation techniques are needed to help target users that might not necessarily have the required knowledge of the security practices to be employed.

3.5 Research and development opportunities (RQ3)

In this section, we examine several opportunities that we think researchers and developers can leverage to improve the cloud-based development and management of IoT systems. Therefore, we aim at answering the research question ***RQ3:*** *What are the main potential opportunities laying ahead for future researchers and developers in the IoT domain?*.

3.5.1 Tools and platforms

Numerous tools and platforms are being built to tackle cloud-based modeling concerns. Thus, now is the right moment to suggest powerful and extensible tools that the IoT community may harness to solve their domain-specific issues. In this section, we look at various open-source and highly extensible platforms that are popular among the modeling community and that we would recommend for the IoT domain.

Cloud-based development tools based on Eclipse: We believe that a significant part of the MDE community, or at least for research purposes, uses Eclipse-based technologies. This is because most Eclipse projects and technologies are open-source, making them more accessible and encouraging individuals to participate. As of March 2021, the Eclipse Foundation hosts over 400 open source projects, 1,675 committers, and over 260 million lines of code have been contributed to Eclipse project repositories [61]. Through the Eclipse Cloud Development (ECD)[23] effort, the Eclipse community has demonstrated its willingness to transit a part of its ecosystem to the web. Eclipse's ECD Tools working group strives to define and construct a community of best-in-class, vendor-neutral open-source cloud-based development tools and promote and accelerate their adoption. Some of the best cloud-based technologies that the IoT community can benefit from are the following:

- *EMF.cloud, GLSP, Theia* - Independently from the Eclipse modeling framework (EMF), the EMF.cloud community 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. Code generation infrastructures based on Eclipse Xtend are also included, while Eclipse Theia provides a web-based code editing and debugging infrastructure. Several resources are available in the community for extending those tools, and we believe that IoT developers may use such technologies to construct cloud-based IoT DSLs.
- *Sirius Web*[24] - It is an Eclipse Sirius-based modeling tool that provides a powerful and extensible graphical modeling platform for users to design and deliver modeling tools on the web. In Sirius Web, the ability to create your modeling workbench in a configuration file is supported. In this case, no code generation is required because everything is interpreted at run-time [62]. Furthermore, being open-source, Sirus Web provides greater 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[25] makes Kubernetes development accessible for developer teams.

[23] https://ecdtools.eclipse.org/
[24] https://www.eclipse.org/sirius/sirius-web.html
[25] https://www.eclipse.org/che/

Che is an in-browser IDE that allows you to develop, build, test, and deploy applications from any machine. Finally, Epsilon playground[26] has been recently launched to offer cloud-based tools for run-time modeling, meta-modeling, and automated model management.

Low-code development platforms: Looking at the LCDPs, the only powerful cloud-based open-source platform for IoT we would recommend is Node-RED [36]. Due to its high extensibility and accessibility, Node-RED offers an excellent IoT system mashup environment in which IoT systems can be designed, developed, and deployed on the fly. The Node-RED platform is open, and IoT system developers can build their custom nodes, compile, test, and deploy them in the Node-RED ecosystem. Several extensions have been made, such as [63] tackling the reusability issues in cloud-based modeled components, [64] to tackle the heterogeneity and complexity challenges found in the Fog based development. Finally, in [65], the authors presented SHEN to enable self-healing capabilities of applications based on Node-RED. In terms of interoperability, Node-RED models are represented as JSON objects, which any third-party tools can easily consume. Some of the tools in this domain, such as FloWare [43] and GENESIS [57] already support the Node-RED models, which shows a great sign of its high impact. Table 5 outlines the essential characteristics of the recommended platforms.

Table 5: Recommended technologies

	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 there are difficulties in adopting a cloud-based modeling approach in the IoT domain, various opportunities will emerge, making the investment worth it. This section outlines various opportunities that will emerge once cloud-based modeling is widely adopted in the IoT domain.

- *User communities:* Adopting cloud-based modeling in the IoT domain will have the potential of attracting more citizen developers, and it will unravel a lot of modeling opportunities on different devices such as tablets and mobile devices [66, 50, 67].
- *Collaborative modeling:* Once IoT modeling infrastructures are moved to the cloud, it can be necessary to introduce collaborative modeling features to simplify the interaction of both developers and stakeholders. Unfortunately, none of the discussed approaches provide collaborative modeling functionalities. However, collaborative modeling can harness the power of real-time information, artifacts, and service exchange.
- *Productivity:* Empowering users to develop their applications by embracing cloud-based modeling is a head-start toward high production and reducing time to market [23]. Users can create applications that cater to their problems, and engineers focus on developing features that facilitate the user for a smooth development at an appropriate abstraction level. In addition, the participants will focus on problem-solving in their particular domain and avoid wasting time and resources on solving problems that are outside their competencies.
- *Maintenance:* Traditional code-centric methodologies necessitate a significant investment in the ongoing maintenance of developed systems. In addition, systems require regular upgrades and installations, which can be error-prone and time-consuming. During upgrades or troubleshoots times of the system, sometimes system downtime is necessary, impacting production. Furthermore, the growing need for software systems in our daily lives and constantly changing user requirements required an agile approach for addressing these issues quickly without compromising system availability or user access. In many cases, such challenges are handled by cloud providers, leaving developers and engineers to focus on developing applications that directly impact customer demands [68].

[26] https://www.eclipse.org/epsilon/live/

- *Monitoring and debugging:* Cloud-based modeling enables monitoring of activities and their archive through its cloud providers. This is a head-start when debugging distributed applications because developers can track down the micro-services, which are the root of the detected problems. Without appropriate cloud infrastructures, it would be challenging to solve these issues, even with features such as self-healing and repair strategies. Current cloud-based solutions come bundled with monitoring tools that assist in problem diagnosis and monitor the usage of the applications.

3.6 Related work

We identified several surveys papers on MDE and DSL in the IoT domain throughout our paper selection process. Still, very few of them focuses explicitly on cloud-based MDE approaches ([69, 23, 70, 71] to mention a few). In this work, we are interested in examining the possible approaches helping in migrating the classical local-based MDE in IoT technologies to the cloud and its adoption.
Our previous study [72] looked at the current state of low-code engineering (LCE) adoption in the IoT domain. LCE combines LCDPs, MDE, machine learning, and cloud computing to facilitate the application development life-cycle, namely from design, development, deployment, and monitoring stages for IoT applications. A comparable set of features has been identified by examining sixteen platforms to represent the functionalities and services that each of the investigated platforms could support. We discovered that just 7 of the 16 could be deployed on the cloud, with the majority of them being LCDPs, whereas classical MDE approaches rely on a local-based design paradigm.
In [73], 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 research was carried out based on their characteristics, such as programming environment, licensing, project repository, and platform support. According to a comparison of such features, 72% of open-source projects are cloud-based, whereas only 17% percent of closed-source platforms are cloud-based, which confirms a strong uptrend of cloud-based systems in open-source IoT projects.
In [74], the authors discussed tools and methods for creating Web of Things services, in particular, mashup tools as well as model-driven engineering approaches. The techniques regarding expressiveness, suitability for the IoT domain, and ease of use and scalability have been analysed. Although this study is related to this document, it solely focuses on mashup tools and only includes a few approaches. We can observe from the preceding discussion that only a few techniques attempted to explore cloud-based MDE approaches implicitly. According to this, and to the best of our knowledge, this is the first study analyzing the status of cloud-based modeling in the IoT domain.

3.7 Conclusion

To develop IoT applications, developers must overcome several challenges, including heterogeneity, complexity, and scalability. Moving development infrastructure to the cloud opens up a host of new opportunities in terms of accessibility, productivity, maintenance, and monitoring. In this chapter, we presented the conducted systematic study to assess the current state of the art in cloud-based modeling approaches in the IoT domain. We looked at 22 papers proposing cloud-based modeling environments in the IoT domain. The approaches considered were analyzed to assess their strengths and weaknesses with respect to many characteristics, including their modeling focus, accessibility, openness, and artifact generation. We discussed many challenges that IoT developers face when adopting such tools. We also discussed several generic technologies and tools that can be used in the IoT domain.

4 The Low-Code Engineering Repository Architecture

The development of the Lowcomote platform draws on techniques and tools that have emerged from recent research in MDE (Model-Driven Engineering). MDE is characterized by the systematic use of models as primary units of abstraction throughout the software development lifecycle to promote both abstraction and automation [8]. To this end, domain-specific languages are used to shift the emphasis from third-generation programming languages to the use of models that are close to the problem domain. The models are then analyzed and manipulated to produce the source code of the modeled systems. Despite the benefits associated with the adoption of MDE, there are still some limiting factors that hinder the wider adoption of MDE, including the following:

- Support for discovery and reuse of existing modeling artifacts is very limited. As a result, similar transformations and other model management tools must often be developed from scratch, increasing upfront investment and compromising the productivity benefits of model-based processes.
- Modeling and model management tools are typically distributed as software packages that must be downloaded and installed on client machines, often in addition to complex software development IDEs (e.g., Eclipse).
- Integration mechanisms that allow developers to build applications, manage new artifacts, and add additional functionality.

To overcome these issues, we present the work related to the development of a cloud-based low-code engineering repository. This repository provides its model management capabilities as a service to support remote access, manipulation, and storage of various types of modeling artifacts. The repository supports various third-party extensions and services, especially from research colleagues involved in this project.The architecture of this community-based model repository enables the reuse of heterogeneous modeling artifacts. It also supports advanced discovery mechanisms and is open to adding new features while maintaining robustness and scalability.

4.1 Related work

In this section, we discuss modern approaches to providing repositories for modeling artifacts. We will primarily outline outstanding research challenges related to proper management of model artifacts. We also discuss the discovery, reuse, and deployment of model management tools and artifacts.

AMOR - Adaptable Model Versioning [75]: It is an attempt to leverage version control systems in the field of MDE. AMOR supports model conflict detection and focuses on intelligent conflict resolution by providing techniques for representing conflicting changes and suggesting appropriate resolution strategies.

Bizycle [76]: This is a project to support the automated integration of software components by means of model-driven techniques and tools. Among the various components of the project is a metadata repository that manages and stores all artifacts required for and generated during integration processes, i.e., external and internal documentation, models and metamodels, transformation rules, generated code, users and roles.

CDO[27]: it is a pure Java model repository for EMF models and metamodels. CDO can also serve as a persistence and distribution framework for EMF -based application systems. CDO supports different types of deployments such as embedded repositories, offline clones, and replicated clusters. However, the typical deployment scenario consists of a server that manages the persistence of models by leveraging all types of database backends (such as large relational databases or NoSQL databases) and a EMF client application.

EMFStore [77]: is a software configuration management system tailored to the specific requirements of versioning models. It is based on the Eclipse Modeling Framework and is an implementation of a generic operations-based version control system. EMFStore implements in the modeling domain the typical operations of SVN, CVS and Git for text-based artifacts, i.e. change tracking, conflict detection, merging and versioning. It consists of a server component and a client component. The server runs standalone and provides a repository for models including versioning, persistence and access control. The client component is typically integrated into an application and is responsible for tracking changes to the model as well as committing, updating, and merging.

GME - Generic Modeling Environment [78]: It is a set of tools that support the creation of domain-specific modeling languages and code generation environments. It also provides a repository layer for storing the

[27] http://www.eclipse.org/cdo/

	Managed Artefact	Main purpose	Typical deployment scenario
AMOR [75]	Model	Model versioning	Desktop application
Bizycle [76]	Model	Integration of software components	Desktop application
CDO	Model	Storage	Client-Server application
EMFStore [77]	Model	Model versioning	Client-Server application
GME [78]	Model	Storage	Client-Server application
ModelBus [79]	Model	Model versioning	Client-Server application
Morse [80]	Model	Model versioning	Software-as-a-service
ReMoDD [81]	Any	Documentation	Web-based interaction
MDEForge [7]	Model, Metamodel, Transformation	Storage, Added value services	Web-based interaction, Software-as-a-service

Table 6: Overview of existing MDE tools providing storage features

developed models. Currently, MS repository (an object-oriented layer via MS SQL server or MS access) and a proprietary binary file format are supported.

ModelBus [79]: It consists of a central bus-like communication infrastructure, a set of core services, and a set of additional management tools. Depending on the usage scenario, various development tools can be connected to the bus via tool adapters. Once a tool is successfully connected, its functionality is immediately available to others as a service. Alternatively, it can also make use of the services already available on the ModelBus. Available services include a built-in model repository that can version models, support partial checkout of models, and coordinate the merging of model versions and model fragments;

Morse - Model-Aware Repository and Service Environment [80]: It is a service-based environment for storing and retrieving models and model instances at both design and execution time. Models and model elements are identified by Universally Unique Identifiers (UUID) and are stored and managed in the Morse repository. The Morse repository provides versioning capabilities so that models can be manipulated at runtime and new and old versions of models can be maintained in parallel;

ReMoDD - Repository for Model-Driven Development [81]: It is a repository of artifacts designed to improve the productivity of MDE research and industry and the learning experience of MDE students. By means of a Drupal web application, users can contribute MDE case studies, model examples, metamodels, model transformations, descriptions of modeling practices and experiences, and modeling exercises and problems that can be used to develop class assignments and projects. Searching and browsing capabilities are enabled by a web-based user interface that also provides community-oriented features such as discussion groups and a forum.

MDEForge - MDEForge: an Extensible Web-Based Modeling Platform [7]: It is an extensible modeling framework consisting of a set of core services that allow to store and manage typical modeling artifacts and tools. Based on these services, it is possible to develop extensions that add new functionality to the platform. All services can be used through web access and a REST API that allows to adopt the available model management tools through the software-as-a-service paradigm.

According to Table 6, the majority of existing approaches only provide support for persistence of models. Only ReMoDD supports other kinds of modeling artefacts, like transformations, and metamodels. However, the main goal of ReMoDD is to support learning activities by providing documentation for each stored artefact. Consequently, ReMoDD cannot be used to programmatically retrieve artefacts from the repository or more generally cannot be adopted as software-as-a-service to search and reuse already existing modeling artefacts. Most of the discussed approaches require local installation and configuration. Only ReMoDD and Morse do not require to be installed locally. In particular, the modeling artefacts stored in ReMoDD can be searched and browsed through a Web-based application. Morse provides developers with the possibility to use it as a service. Interestingly, MDEForge provides the means to store different kinds of artifacts including, models, metamodels, and transformations. Moreover, it can be used by means of a Web-based interface, and it permits also to exploit the provided functionalities in a programmatic way. In fact, the provided added value services, like automated classification of metamodels, remote execution of transformations, etc. can be used by means of the provided APIs. The main drawbacks of MDEForge are the lack of support for storing and managing relevant artifacts that are of interest in the Lowcomote context including DevOps workflows, quality assurance artifacts, and advanced mechanisms for providing users with relevant recommendations.

4.2 System views

This section presents the architecture of the Lowcomote repository. Its development follows an iterative and incremental approach, where activities consist of multiple sprints and functional deliveries, as shown in Fig. 10. Iterations included analysis, design and implementation, testing, and deployments organized in sprints and based on well-defined functions. To manage the development, we used various methodologies such as Scrum, Kanban, and Extreme Programming. Scrum helped us organize our workflow into sprints to ensure independent, fully functional features, and Kanban helped us visualize the workflow and better organize the backlogs. To improve responsiveness and rapid updates of changes and requirements, we used extreme programming.

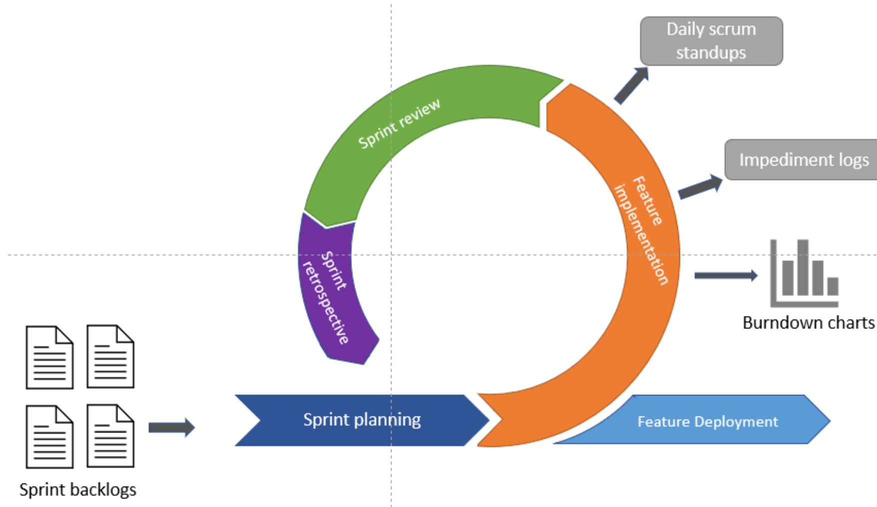

Figure 10: Scrum iterative sprints

To describe the architecture of the proposed system, we used a modified version of the *"4+1"* View model, which is a model for "describing the architecture of software-intensive systems based on the use of multiple, concurrent views" [4]. The modification to this model is the addition of a sixth view, namely the data view (see Fig. 11). Since the repository is intended to store and manipulate modeling artifacts, we decided that adding such an additional view is appropriate to also show the structure of the data to be managed by the repository. Thus, the six views used to describe the repository architecture are as follows:

- **Use case view:** This view is at the center of "4 + 1" architectural view model because the remaining views revolve around it. Use case view presents user requirements that captures system functionality and thus illustrate the envisioned user's interaction with the rest of the system stakeholders.
- **Data view:** This view describes the organization and type that the system will utilize to exchange or transfer data during task executions.
- **Logical view:** The logical architecture view of the system describes the system based on the functional requirements the system shall support to satisfy user requirements and stakeholder needs. The system is divided into key abstraction sections that better describe the design elements and common mechanisms that tackle the problem domain.
- **Development view:** The development view supports the static view and organization of the system. The system is presented in terms of components, which are properly integrated to deliver that wanted

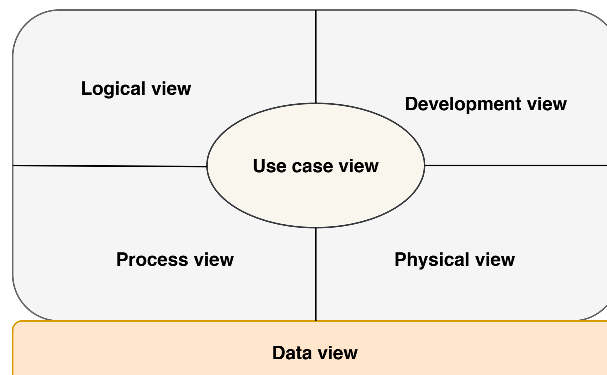

Figure 11: The extended *"4+1"* views of the Lowcomote repository architecture

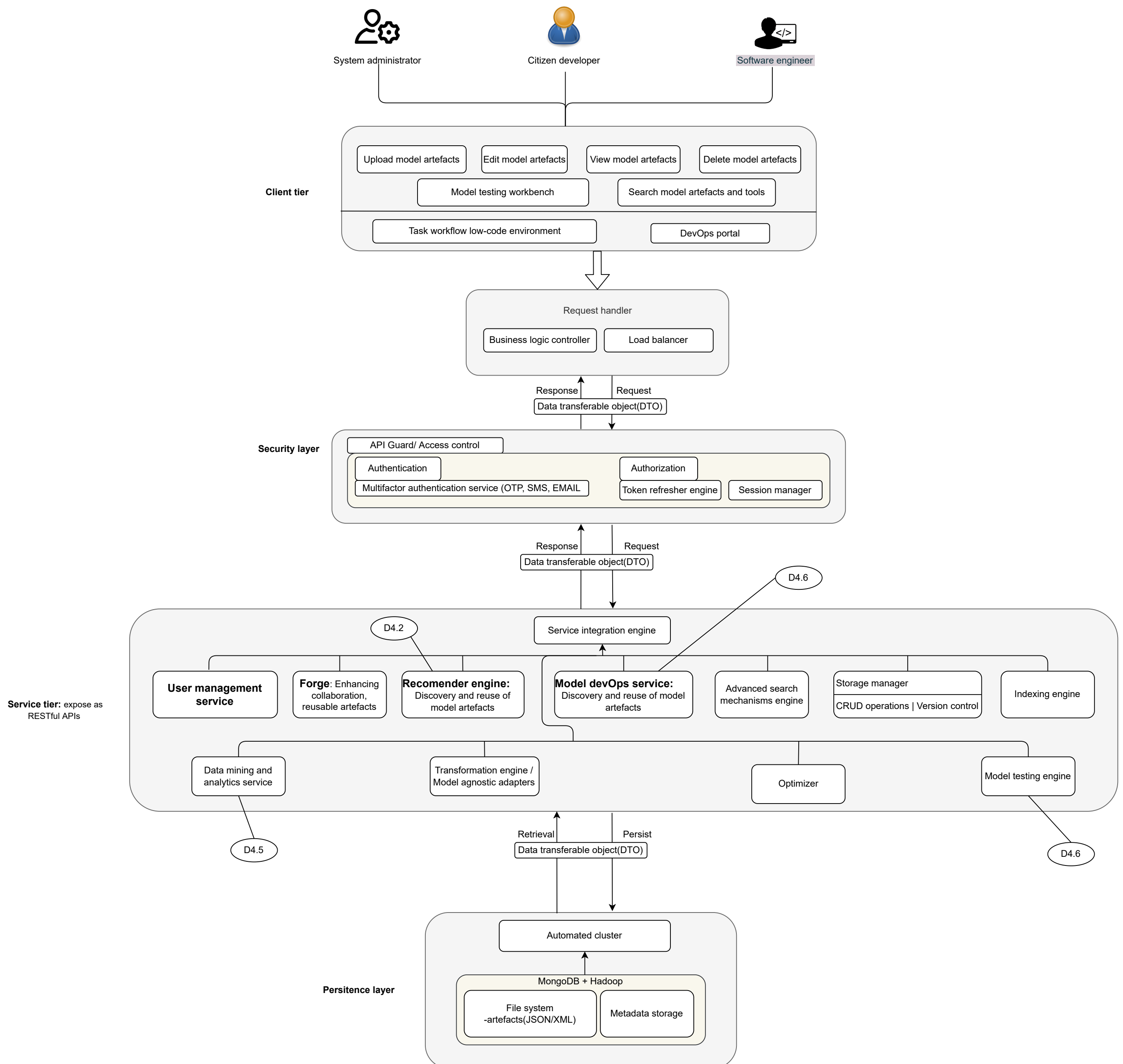

Figure 12: High level architecture view

functionalities. It is essential to bear in mind that the complete development architecture view can be identified only when the system is complete, hence making the current version provisional.

- **Process view:** The process architectural view of the system aims at describing the dynamic aspects in relation with some functionalities of the repository. System processes may be described on different levels of abstraction in order to depict explanatory and peculiar processes.
- **Physical view**: In this view, we primarily elaborate the non-functional requirements such as reliability, scalability, performance, availability. System components are mapped to corresponding processing nodes.

The presented views address concerns that pertain to the different stakeholders (i.e., *system administrator*, *citizen developer*, and *software engineering*) that are involved in the management and adoption of the presented Low-Code repository. Figure 12 shows a high-level view of the system interacting with the different stakeholders. All of them interact with a *client tier* (mainly Web-based), which exposes only the functionalities that are relevant for the corresponding user. A *security layer* takes care of authorization and authentication aspects. All the functionalities provided by the repository are provided by a *service tier*, which properly interacts with the *persistence layer* depending on the requested services. It is important to remark that the services that are not the focus of this deliverable are marked with the identifier of the corresponding deliverables, presenting them with details. For instance, the recommender engine is presented

in the deliverable D4.2. Therefore, the reader can refer to that document to get details about such a service. The high-level view of the system is described in detail in the next subsections according to the extended 4+1 model mentioned earlier.

4.2.1 Use case view

On the user perspective, this view presents the functionalities provided by the Low-Code repository. As previously mentioned, we envision three different stakeholders of the repository as shown in Fig. 12, i.e., the *software engineer*, the *system administrator*, and the *citizen developer*. In the following, the presentation of the repository functionalities is organized with respect to these three kinds of envisioned users.

4.2.1.1 Software engineer Software engineers are supposed to be experts in model-driven engineering and in the development of low-code development platforms. They can extend the system by integrating new functionalities. In this respect, Figure 13 shows the two main use cases involving software engineers as described below:

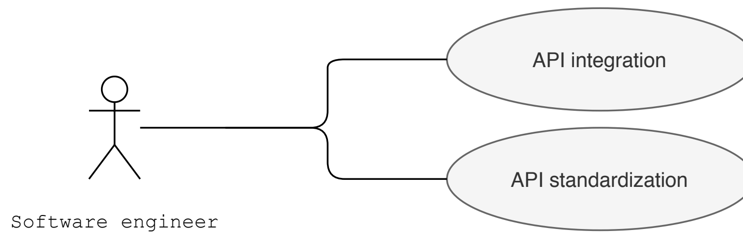

Figure 13: Software engineer use case diagram

- **API integration:** The software engineer can programmatically integrate new APIs or extensions in the repository. Dedicated policies and guidelines will be defined so that additional functionalities can be added by extending an integration service that will be purposely conceived.
- **API standardization**: The system will provide software engineers with dedicated extension points to add the specification of new APIs by using services like OpenAPI [28] for API documentation and further testing.

4.2.1.2 System administrator System administrators will manage the repository infrastructure and the users that are allowed to exploit the repository functionalities. As shown in Fig. 14 the use cases that will be provided by the repository to system administrators are the following:

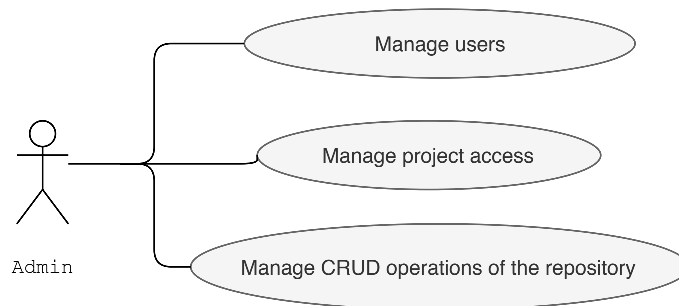

Figure 14: System administrator use case diagram

- **Manage users:** The system administrator will have the availability of typically user management functionalities, like addition and removal of users, password changes, and temporal suspension of user accounts.
- **Manage project access**: The administrator ensures the authorization aspects that permit or forbid users to have access to the items stored in the repository;
- **Manage CRUD operations of the repository**: The administrator have 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 certain service or other operations that she only has access and permission to perform.

[28]https://swagger.io/specification/

4.2.1.3 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 repository. As shown in Fig. 15, the functionalities provided by the Low-Code repository are grouped under four main categories, i.e., *Model Repository*, *DevOps Model Framework*, *Model Recommendations*, *Model Mining* and *Model testing framework*, which are described in detail below.

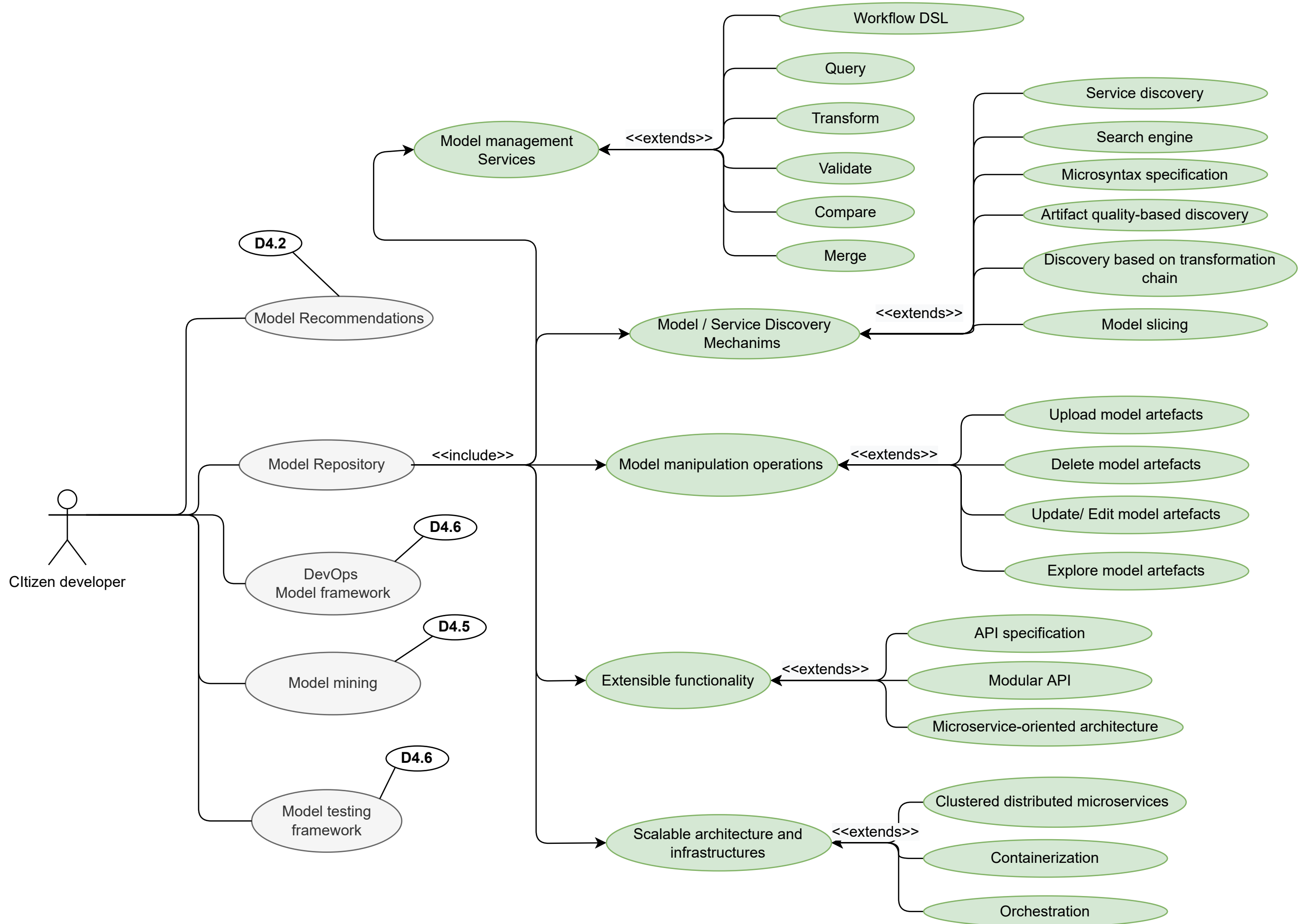

Figure 15: Use case diagram

Model Repository use cases: They represent the functionalities pertaining to the management of any kinds of modeling artifacts to be stored and searched in the repository. Thus, the user has the availability of the model management use cases discussed below.

- **Model Manipulation Operations:** The system implements several functionalities that enables manipulations of modeling artefacts 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 stored in the repository via provided user interfaces.
  - The citizen developer can explore and view selected modeling artifacts using the tree view.
  - A citizen developer can delete modeling artifacts from the repository.

  These operations are modular and represent the smallest logically isolated functionality. Therefore, the operations can be used to extend the repository by reusing them in further higher level abstraction activities that can build on these operations.
- **Model / Service Discovery Mechanisms:** This repository feature allows the user to find relevant domain models and model management services. It is intended to allow developers to reuse existing domain models and thus avoid having to reinvent the wheel and start modeling processes from scratch.

  The discovery mechanisms on the repository include discovery that target model artifacts but also we enable the discovery of services available at the repository:
  - *Service Discovery*: This feature allows the user to discover model management services in the repository. A dedicated Service registry is responsible of recording available services, their status

and logs.
  - *Search Engine*: This is an integrated, industry-grade 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 is capable of filtering 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 in 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.
  - *Model slicing:* It enables the discovery and reuse of different domain model classes and their related UI components as well as their related DSL code. All these cross-related artifacts are provided as a model slice.
- **Extensible functionality:** The architecture and design of functionalities at the repository strive to facilitate the extensibility, reuse and integration of services built upon 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, notable API specifications are used to facilitate this endeavor through the use of SWAGGER 2.0 OpenAPI[29] and GraphQL[30] 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 the codebase.
  - *Microservice oriented architecture*: A microservice is a highly cohesive, single-purpose, decentralized service. It should have only one purpose and be self-sufficient. This type of architecture can easily track down and fix bugs in an isolated service. Furthermore, we can run the API remotely via execution on-demand as needed. This makes managing and maintaining the code base and adding new functionality much more straightforward. Plus, there is the advantage of holistic resource management.
- **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 management and reusability.
  - *Clustered distributed microservices*: By having microservices in a cluster, we can maintain the services in a pool of services and benefit from several advantages. By clustering distributed microservices, we enabled load balancing and strategic monitoring of resources and traffic across multiple microservices. We can also manage workloads by reserving nodes for some intensive workloads that have special requirements. In short, clustering services improve the scalability of services and ensure they can handle the desired traffic.
  - *Containerization*: By containerizing services using Docker technology, we isolated the model management services from the deployed environments. This ensures an improvement in the efficiency and security of our repository. In addition, this feature simplifies the management of the services.
  - *Orchestration*: By using Kubernetes, we have orchestrated the services in our cluster. By orchestration, we mean organizing multiple clusters to function as a single unit. This allows resources to be used more efficiently, and tasks are scheduled on the most appropriate machines based 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 assembly of model management services. This

[29] https://swagger.io/specification/v2/
[30] https://graphql.org/

allows the services to be executed in workflows.

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."* [82]. 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. It is used as a comprehensive term to refer to many continuous-* activities like Continuous Development, Continuous Integration, Continuous Deployment and Continuous Delivery to form complete DevOps processes [83]. The Low-Code repository supports model-driven CSE approaches by providing specific CRUD functionalities for DevOps process and platform models as discussed below. In addition, it aims at supporting the current, and future extension of the DevOpsML conceptual framework presented in [84]. For more information, please refer to deliverable **D4.6**.

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 reuses of modeling artifacts are offered to the citizen developer in terms of recommendations. For more information, please refer to deliverable **D4.2**.

Quality Assurance use cases: In any Low-Code development platforms, citizen developers should be involved in all the phases of the application development process, including testing. Therefore, the envisioned platform provides specific functionalities to test the Low-Code system under development. Testing operations are specified in terms of models as detailed in the deliverable **D4.6**.

4.2.2 Logical view

In the Logical View, the Lowcomote repository is decomposed into a set of collaborating components. The main focus of this view is on the functional architecture of the repository. A component diagram, which shows the aforementioned decomposition is illustrated in Fig. 16 and described in the following.

- **Artefact view control:** This component pertains to the client layer. This component is responsible for the repository's basic functionality, especially showing repository contents such as available model artefacts, tools and services. It also presents several model manipulations such as upload, delete, and update of models in the repository.
- **Business logic controller:** This component is responsible for the main server application logic and APIs, thus ensuring the system's load balance. In addition, it uses a security manager component. Finally, this component is the base where we expose public APIs using openAPI 3.0. and thus, any external communications will first go through this component.
- **Security manager:** This component ensures authentication and authorization of users to services or services to services. It will contain the logic necessary to shield our repository contents against attacks and malware, thus acting as an API guard and ensuring access control. It also includes a component for managing sessions and access tokens.
- **User manager:** This component stores user data to ensure authentication and manage user sessions.
- **Service integration:** This component is responsible for integrating components together and exposing their APIs to the business logic controller components.
- **CRUD manager:** This component is responsible for uploading, deleting, updating, editing, and displaying model artefacts. It constantly synchronises with the persistence layer to ensure data integrity and information update.
- **Search engine:** This component is responsible for searching model artefacts and tools on the repository. It is implemented using advanced search mechanisms techniques to ensure a swift search of model artefacts and tools regardless of the number of artefacts present in the repository.
- **Indexer:** This component uses indexing tools to underpin the search component and machine learning operations that require massive data processing.
- **Persistence:** This component combines several database APIs that store data from the various components and ensures a single output of the data from the repository. In addition, we implemented security guards that ensure data is secure and that any request of data from the repository's databases is authenticated and authorized with updated tokens.
- **Repository DB:** This component uses the database to store data and metadata from the repositories components. It implements the MapReduce programming model at this level to better process and

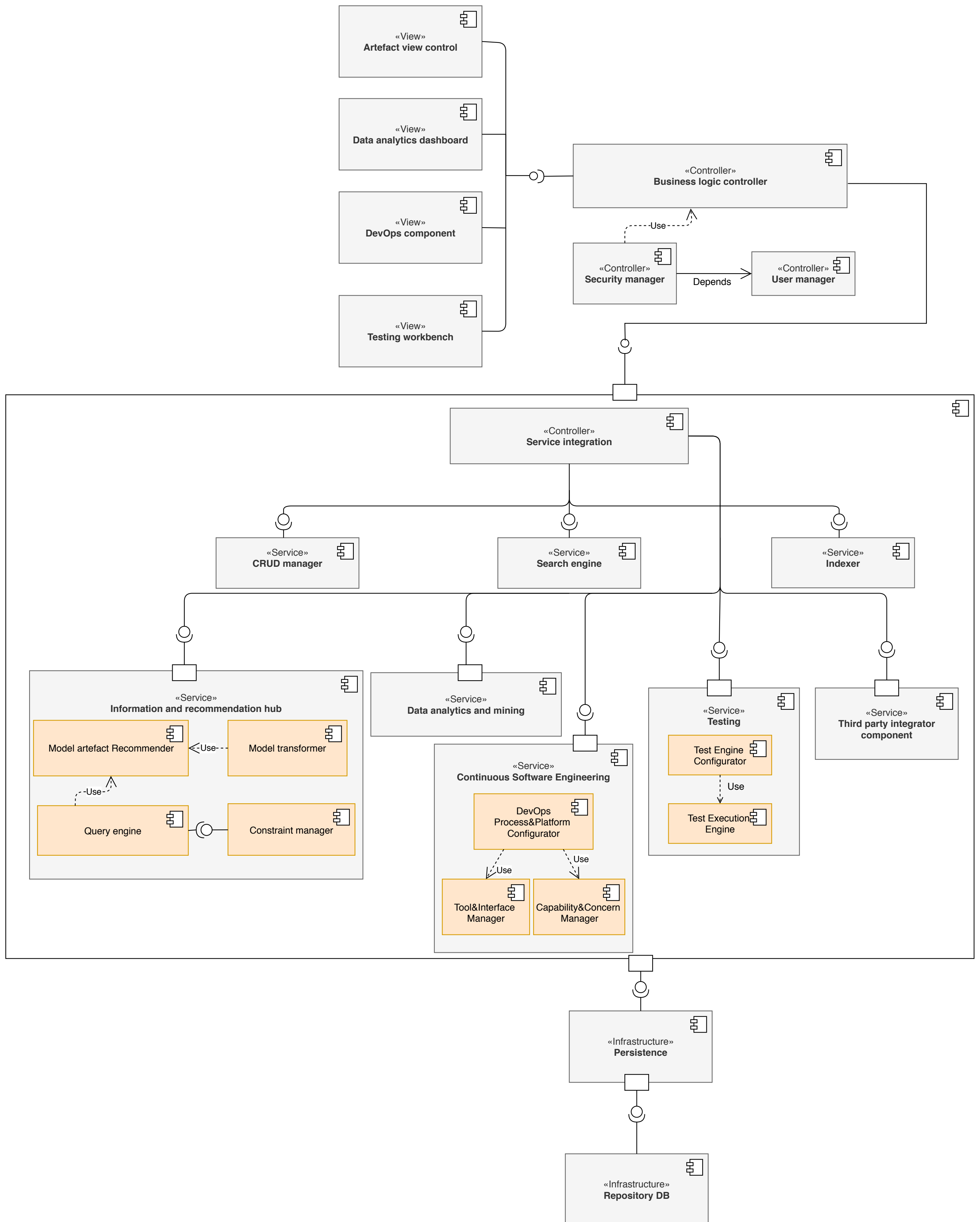

Figure 16: Logical view

store big data.

- **Model artifact recommender:** This component is responsible for providing recommendations to the user during the modeling process. The recommendations can be triggered proactively (by the system) or re-actively (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 kind of relationship can be set between newly created model artifacts. The recommendations can also provide useful suggestions

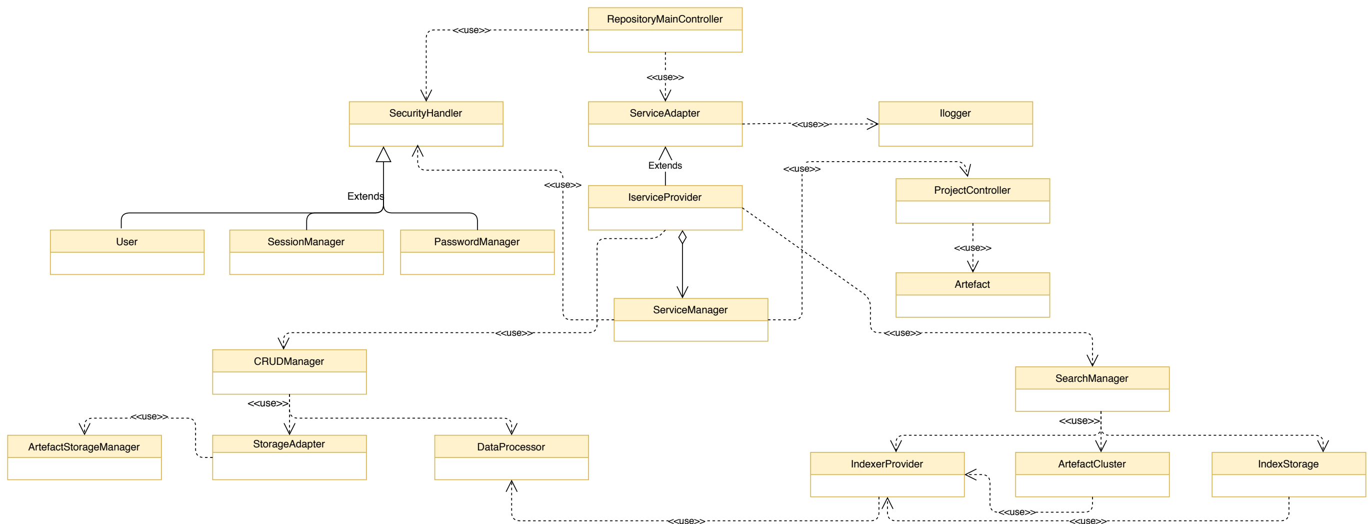

Figure 17: Development view

for improving the current state of an underdeveloped model just uploaded to the repository.

- **Query engine:** For a given underdeveloped or under-construction model, recommendations will be realized through the query engine, which gets as input the model details and queries the repository to get the same or similar models. The retrieved models from the query result will be compared with the model under construction. For more information, please refer to deliverable **D4.2**.
- **Model transformer:** After the query engine gets similar models, and the corresponding recommendations have been provided to the user, the model transformer component is involved in applying the selected recommendation to the model under construction.
- **Continuous Software Engineering (DevOps)**: This component acts as a top-level container of CSE-related services, acting as a facade for i) citizen developers, directly interacting with the Low-Code Engineering Repository, and ii) external tools that can be integrated into a dedicated Low-Code platform configuration (see platform modeling in [84]) supporting given DevOps process(es) (e.g., continuous delivery [84] or testing functionalities (see Deliverable D4.3 [85]). In particular, a DevOps Process and Platform Configurator is in charge of two main activities: i) supporting configuration of DevOps processes and platforms [84] based on available libraries of tools (with their interfaces), according to given requirements (e.g., required capabilities to address given concerns [84]), ii) collecting tool descriptions and their supported processes in shared libraries (e.g., tool descriptions created by tool providers [84]). Furthermore, to perform its functionalities, the DevOps Process and Platform Configurator is supported by dedicated manager components (Tool and Interface Manager, Capability and Concern Manager) with dedicated functionalities for existing libraries (see Fig.15) (e.g., collection of predefined queries and recommendation for process and platform models). For more information, please refer to deliverable **D4.6**.
- **Testing**: 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 various LCDPs. The *Test Execution Engine* component is a dedicated and configurable engine that conforms to Test Description Language (TDL); it is a standardized language for abstract test case definitions [86]. Citizen developers can design test cases using the TDL language (i.e., creating TDL models). TDL is not executable, but the engine makes TDL models executable. It also performs automatic configuration, execution, and evaluation of functional tests at the model level. The engine can be configured for various Domain-Specific Languages (DSLs) through the 'Test Engine Configurator' component so that the testing service can be customized and reused by different LCDPs. For more information, please refer to deliverable **D4.6**.

4.2.3 Development view

The development view focuses on the actual software design of the Lowcomote repository. The static structure of the system is illustrated in Fig. 17, and the main elements are described in Table 7.

Table 7: Description of the main Lowcomote repository static structure elements

Element Type	Name	Description
Class	**RepositoryMainController**	This class is the main controller of the system, thus it is the point of consumption for external APIs and exposure of internal functionalities. It coordinates all internal components of the repository. Internal functionalities will be hidden to the external entities of the system.
Interface	**ServiceProvider**	To extend the functionalities of the system it will be necessary to implement this interface.
Class	**ServiceAdapter**	This class will implement the adapter pattern to facilitate interoperability between interfaces without modifying the source code.
Class	**SecurityHandler**	This class will manage security features of the repository. Security APIs will be accessed from this class and thus internal features of internal security components will be hidden, and only chosen APIs will be exposed from this class.
Interface	**Ilogger**	This interface will be implemented by logging classes in different components of the system.
Class	**User**	It implements user management operations. It is used by *SecurityHandler* as parent class.
Class	**SessionManager**	This class will manage 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 will be managed starting from this class.
Class	**PasswordManager**	This class will manage the passwords of the users, hashing, and other password strengthening techniques will be implemented in this class.
Class	**ServiceManager**	This class will manage internal services. All load balancing, logging, and performance metrics will be calculated in this class. Services will be 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 will manage projects that are stored in the repository. A project is related with different artifacts and users having different roles. It will use the *SecurityHandler* class.
Class	**Artefact**	This is the main entity class that manages the artefacts of the repository. It captures the overall data context of incoming and stored artefacts.
Class	**CRUDManager**	This class will manage the CRUD operations of the different kinds of artifacts stored or to be stored in the repository.
Class	**ArtefactStorageManager**	Implementation of CRUD operations will be done in this class.
Class	**StorageAdapter**	This class will expose the CRUD operations to the external world, including *ServiceManager* class. Several other external classes shall be able to use this facility without modifying their internal structure.
Class	**DataProcessor**	Inherited from the CRUD manager, this class will extract needed metadata and organize data into a predefined data structure for later consumption by data analytics and machine learning tasks.
Class	**SearchManager**	This class coordinates the searching of artefacts and tools managed by the repository. This class will have several utility classes that prepare the query strings to be used for searching processes. Search texts will be sliced into chunk of single words to facilitate filtering. The resulted words will be matched to related clusters and indices before retrieval.
Interface	**IndexProvider**	This interface will expose indices of artefacts stored in the repository to services such as search and recommendations that use them. Thus, anyone who wants to use indices will implement this interface.
Class	**IndexStorage**	This class will save the indices in a predefined data structure and manage them.

4.2.4 Process view

The process view deals with the dynamic aspects of the Lowcomote repository. It shows at a high level of abstraction how the main functionalities of the repository are executed. This section will present representative processes concerning provided model management, continuous software engineering, model recommendation, and quality assurance services.

4.2.4.1 Model management services As explanatory services, we show the sequences of steps that are performed when searching modeling artifacts and when uploading new ones. We selected two main use

cases below for a demonstration:

Search modeling artefacts: This use case retrieves modeling artifacts according to user requests. Files will be indexed to ensure search speed. For any use case, the process will start with authentication. Then, authorisation will be performed for each service against another service to validate sessions and tokens. As seen in Fig 18, the service integration will be the access component because internal components will not be accessed from the service integration component. A single point of access will help us ensure and evaluate service performance and data integrity. The persistence component will manage the storage of modeling artefacts.

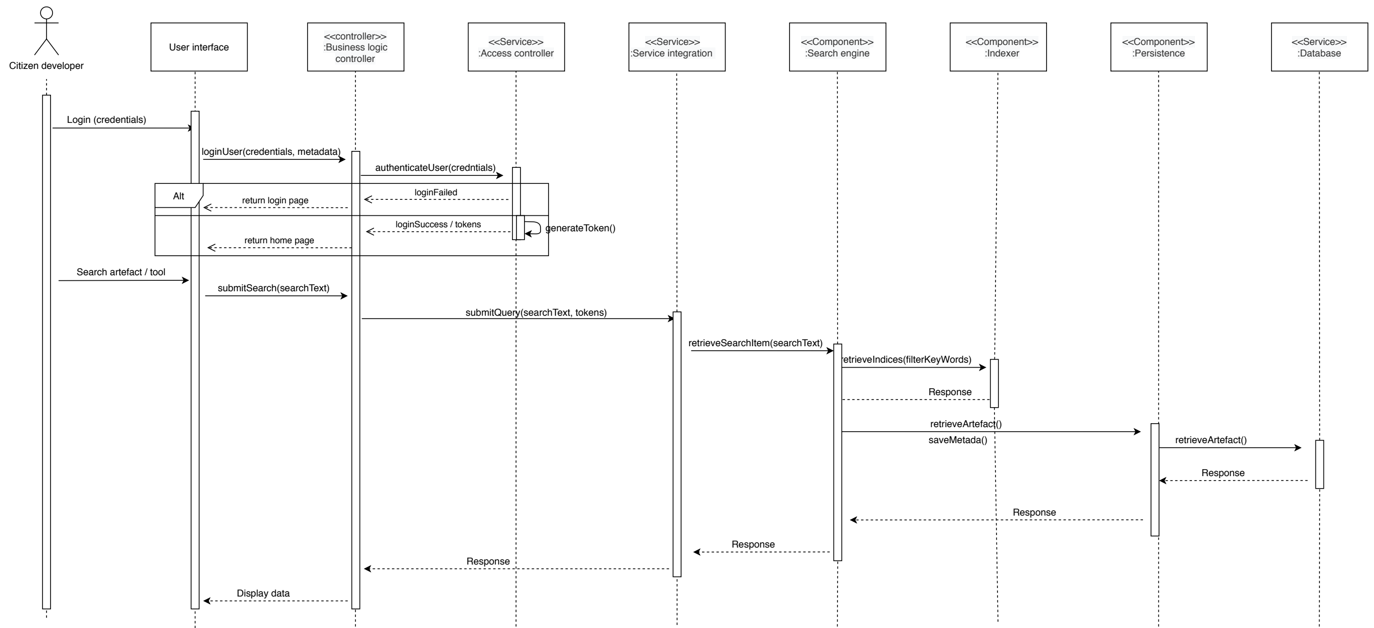

Figure 18: Search model artefact process view

Upload of modeling artefacts: The upload of modeling artefacts is one of the basic functionalities provided by the repository. Similarly to all the other use cases of the repository, the first step is related to the user's authentication. We will save the artefacts in a database and organize them accordingly, as shown in Fig 19. Metadata from the data transfer object will be saved in the persistence component, which captures the context of a given use case.

4.2.4.2 Continuous Software Engineering services

The Low-Code Engineering Repository provides the means for CRUD functionalities for the model-driven artifacts of the DevOpsML framework presented in [84], and a prototypical implementation of the DevOpsML framework is available [87]. In [84] 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 20 and 21 provide a high-level view of possible interactions among the Low-Code repository and external tools dealing with process and platform models, respectively. 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 [84], the process modeling activity is expected to be mostly supported by existing DSL and functionalities provided by LCPDs [88]. In [84], the OMG SPEM language and compliant modeling tools (e.g., SPEM plugin for MagicDraw UML[31]) have been used. A Platform Model artifact is obtained at the end of the interaction depicted in Figure 20, which is editable by the citizen developer and stored in the repository. Library and platform modeling activities are as well intended to be supported by external modeling tools. In [84], 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.

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

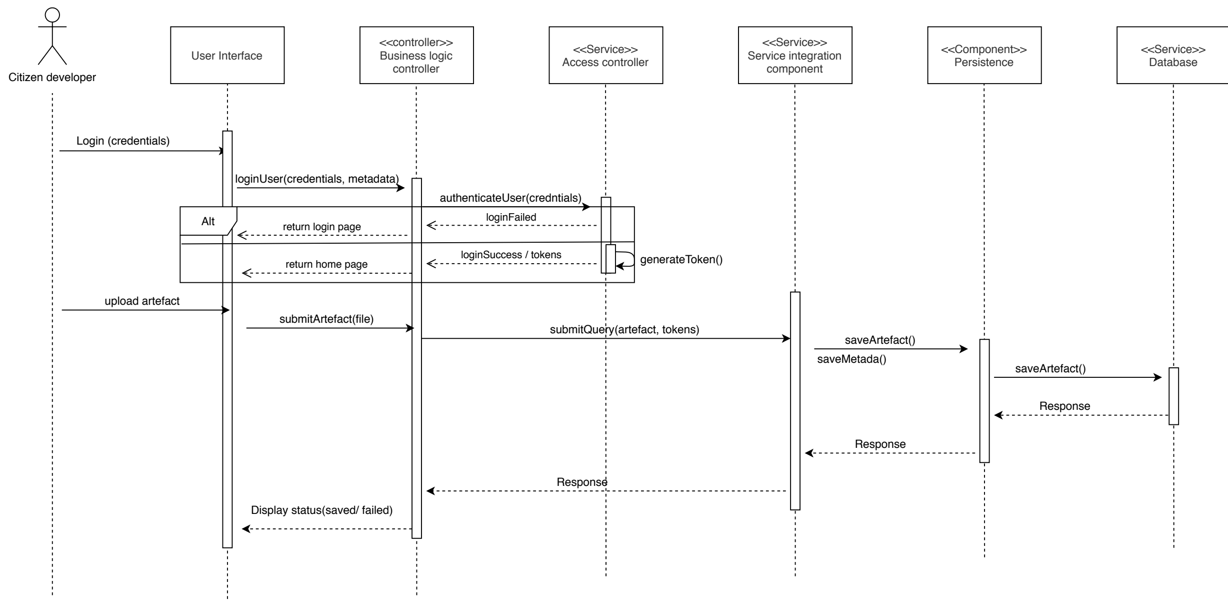

Figure 19: Upload model artefact process view

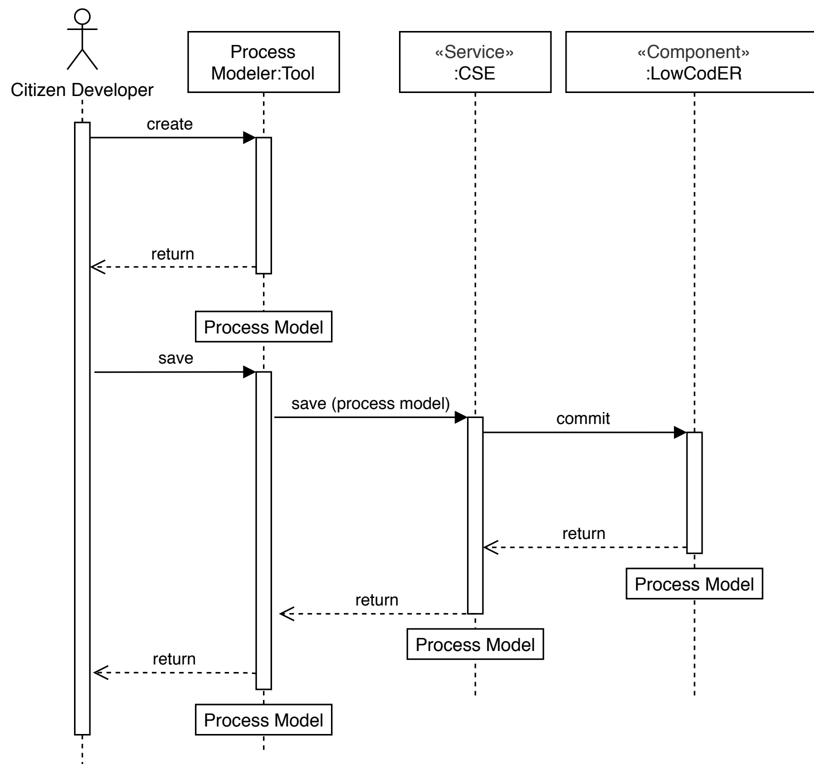

Figure 20: Creating a Process Model from an external Tool and storing it in the Low-Code Engineering Repository through the CSE service.

Figure 21 depicts a high-level interaction scenario with a citizen developer that creates a library of platform elements. According to the DevOpsML framework [84], D4.3 [85] shows how the citizen developer can play different roles depending on her backgrounds. In particular,

- *as requirement engineers*, she can express requirements by creating libraries of *required* tools, interfaces, capabilities and concerns.
- *as tool provider*, she can store the model of her preferred/available tools as a collection of provided interfaces and capabilities addressing given concerns[32].

The `Tool&Interface` and `Capability&Concern Manager` components (as shown in Figure 16) are in charge of providing repository-specific functionalities to support the DevOps Platform Configuration use case (Figure 15), 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

[32]Feature models of LCDPs [88] and existing classification of DevOps tools [89] can be used for this task.

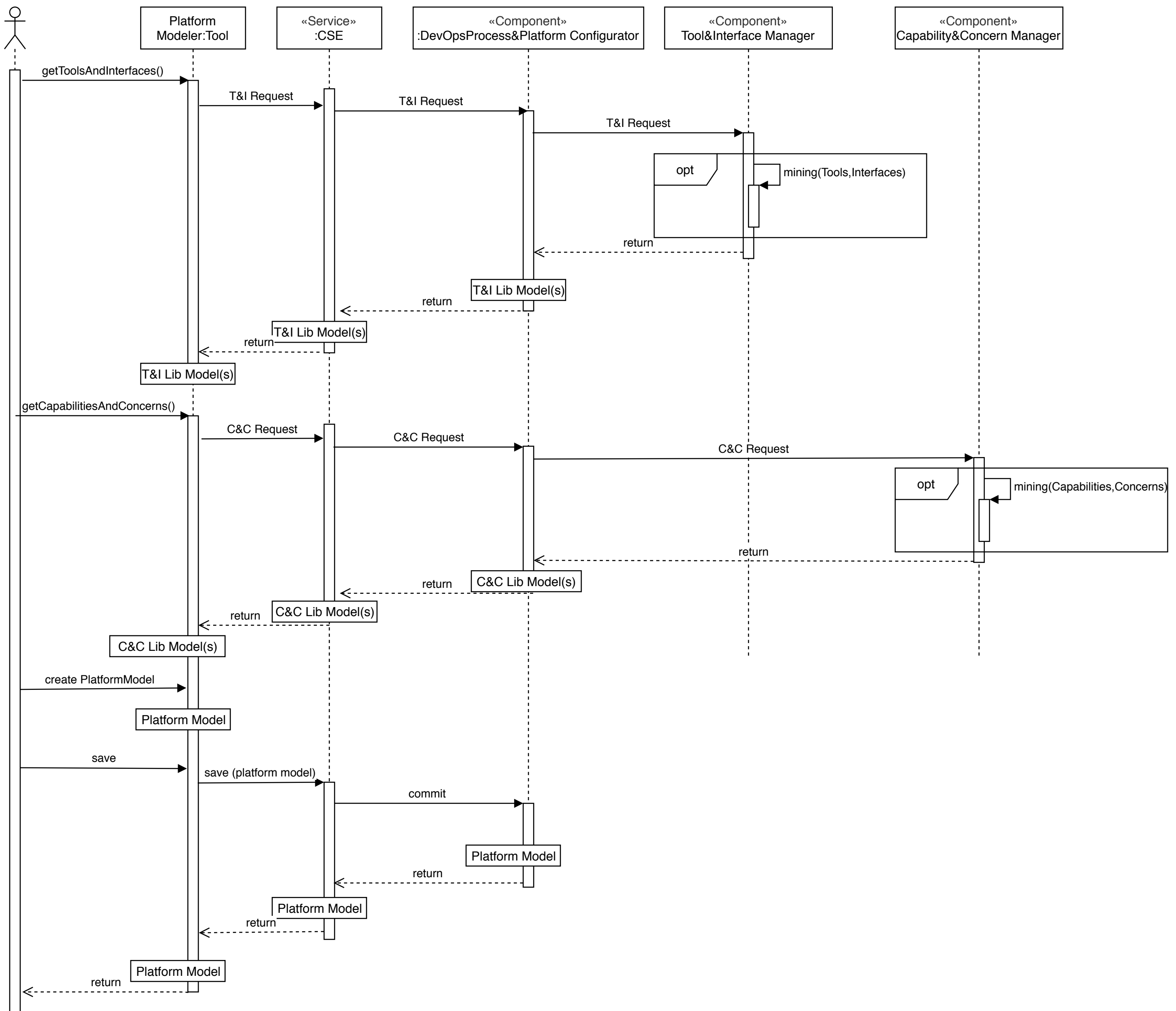

Figure 21: Creating a Platform Model from an external Tool via reusable libraries.

mining techniques will be invoked on available MDE artifacts (i.e., the data) and CSE processes.

4.2.4.3 Model discovery and reuse services As shown in Fig. 22, 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 will is done by using an advanced query facility from the repository. If the developer decides to reuse any given model, then this model will be loaded to the UI, and the user can customize it.

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

4.2.4.4 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. 23) and to get notifications in case of changes occurring in the model of the SUT (see Fig. 24). 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

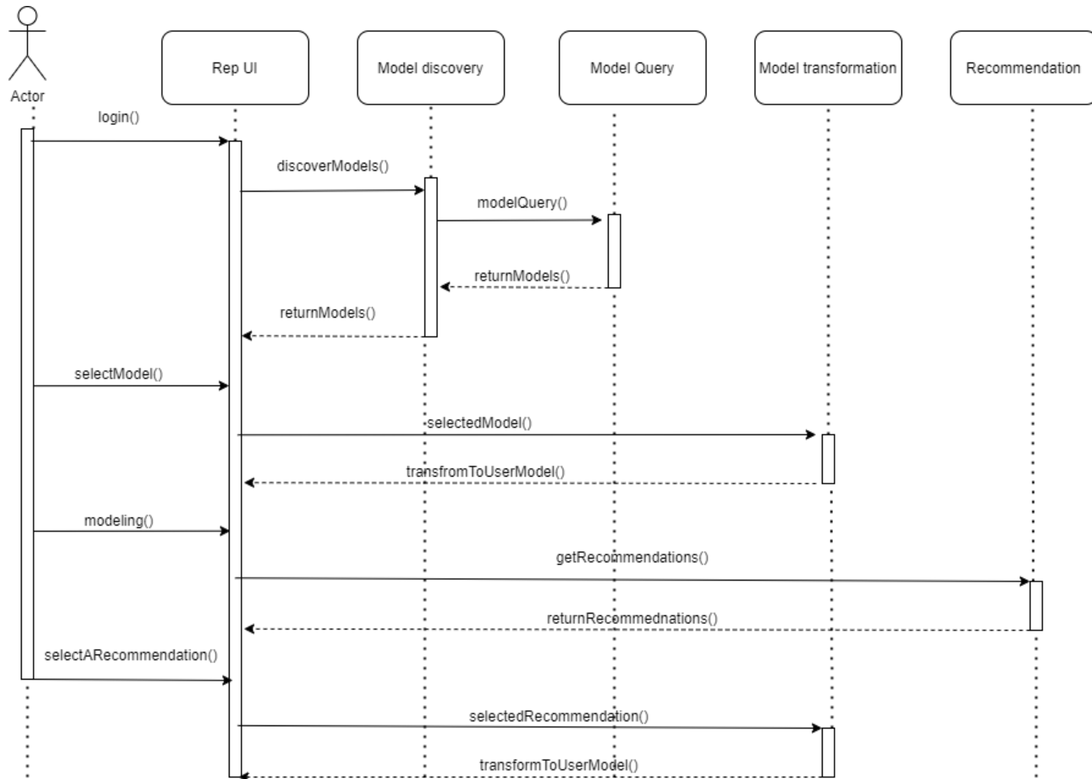

Figure 22: Model discovery and reuse

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.

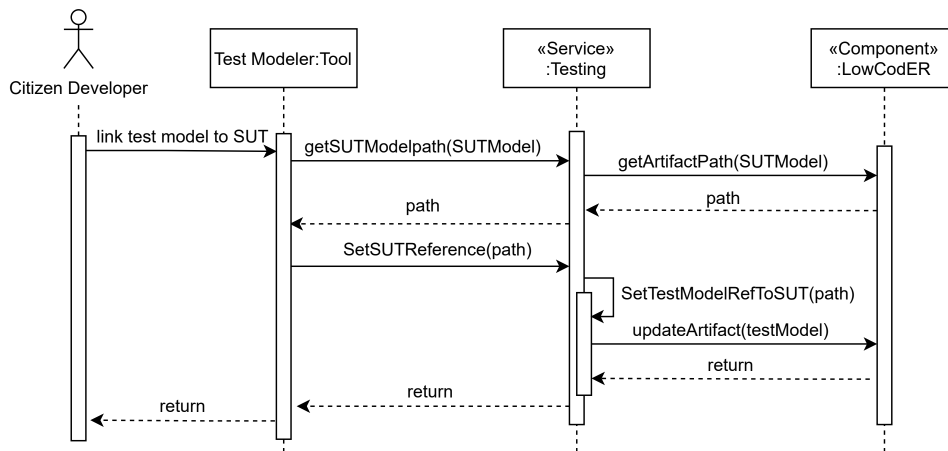

Figure 23: Setting the links between test models and their related system models (SUT models)

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 will be sent to the citizen developer through the modeler tool.

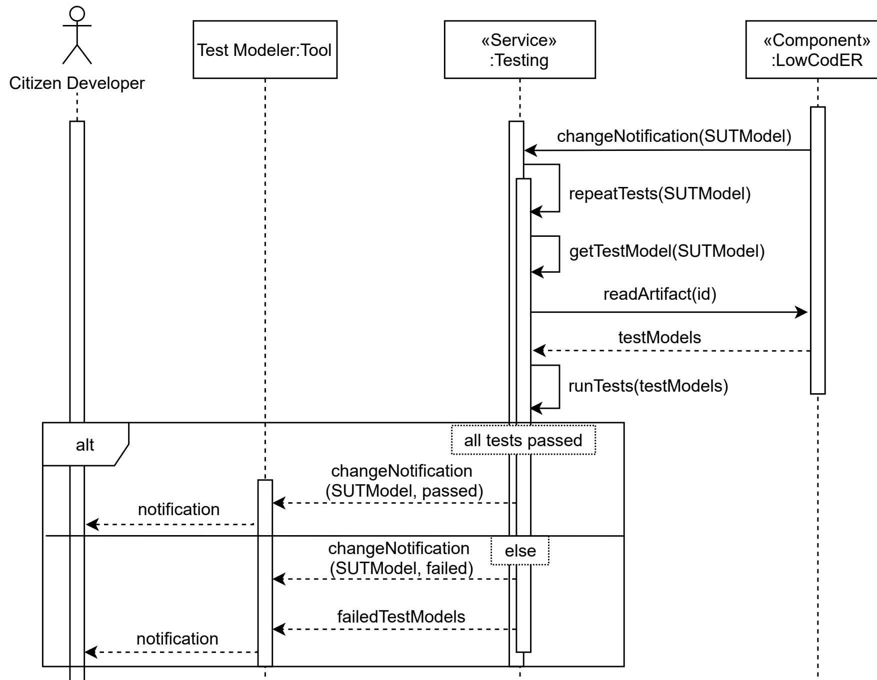

Figure 24: Getting notification of system model changes along with the related failed test models

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 numerous round-trip calls between the client and the server that underlying processes shall perform.
Figure 25 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 the Lowcomote repository as shown in Fig. 26. According to the high-level architecture shown in Fig 12, the system includes different physical components. The client exploits the functionalities developed by the different project partners, especially those involved in Work Package 4. The services are made available in the *Client Layer* and include model management services, recommendations, model testing workbench and continuous integration services. Such services communicate with the *Logic Layer* by 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.

4.3 Conclusion

This chapter presented the architecture of the repository developed in the project to store and mine different low-code artifacts. In particular, the developed repository provides model management capabilities made available as a service to support remote access, manipulation, and storage of various modeling artifacts. In the following chapters, we detail the components of the repositories that have been the focus of ESR6 and ESR8.

DTO

Response

-Response status
-Execution time
-Response: JSON
-ArrivalTime

Request

RequestContext

-artifactName: String
-searchText: String
-artifactType:Artifact
-ProjectNme:String
-requestTime:Date()
-versionNumber:Integer
-branchName:String
-username: string
-invokedMethodName:String
-invokedService: Service[*]
-uploadedArtifact: Artifact[*]
-invokedEvent: Event[*]
-usedInputParameter: Parameter[*]
-projectDependency: Dependency[*]
-annotation: String
-comment: String

Artifact
-name: String
-type: String
-isCollection: boolean

Service
-name: String
-type: String
-source: String

Artifact
-name: String
-source: String

Event
-name: String
-type: String
-source: String
-target: String

Parameter
-name: String
-type: String
-isCollection: boolean

ResponseObject

Model management
-searchText
-artifactName
-artifact
-status

Recommendation

Continuous integration

Test

MetaData

Figure 25: Data view

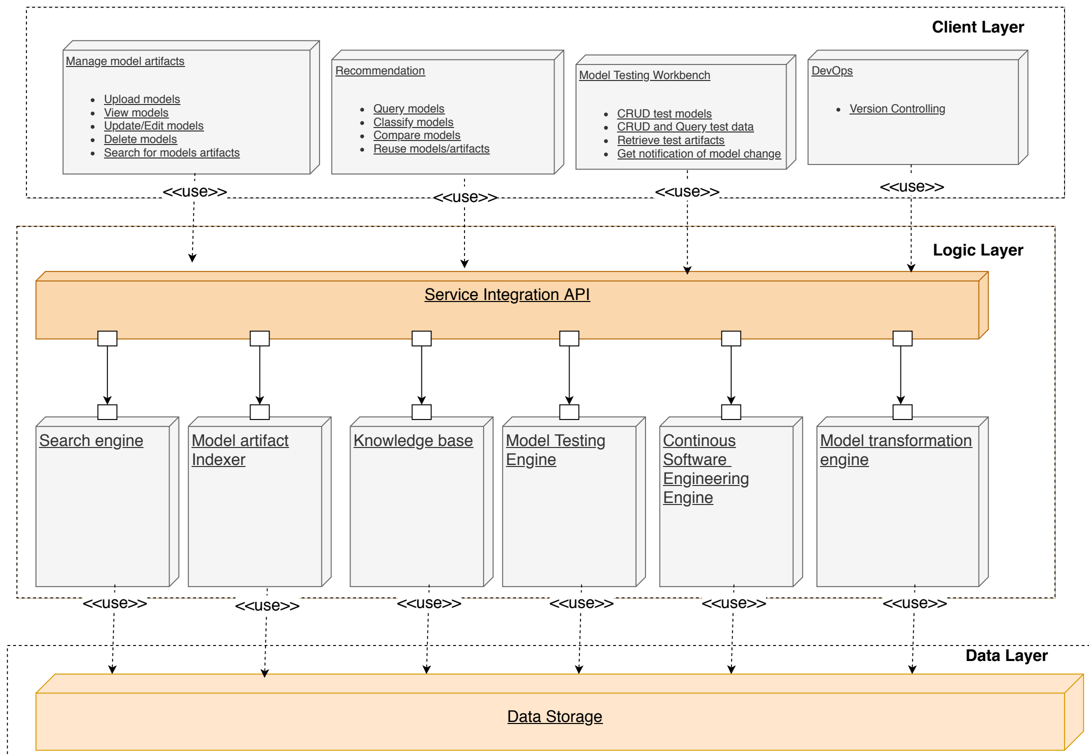

Figure 26: Physical view

5 MDEForgeWL: cloud-based discovery and composition of model management services

The Modeling as a Service (MaaS) initiative [90] fostered the adoption of model management operations as services over the Internet. To this end, as presented in the previous chapters, over the last years, several repositories have been proposed by academia and industry to enable the reuse of modeling artefacts and their remote execution as services [91]. Thus, by employing the MaaS paradigm, the development of complex systems requires the composition of several atomic services, which must be properly discovered, and orchestrated. However, currently available model persistence services do not facilitate such operations mainly because they do not expose remote APIs [92] and do not provide the means to register and discover the services to be composed. Moreover, the current main challenges that make the composition of model management operations a strenuous 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[33]);
- 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.

In this chapter, we propose the adoption of a low-code development platform (LCDP) to develop complex model management processes. In particular, LCDPs provide intuitive visual environments to citizen developers to build fully operational applications, which do not require a strong programming background [93]. The considered context is characterized by atomic model management operations provided as services by (potentially) different providers. The envisioned LCDP supports the discovery and the orchestration of the services needed to develop the wanted composed process. The objective is to develop an event-driven approach based on trigger-action programming as done by LCDPs like IFTTT and Zapier among popular services [94]. In such platforms, users can connect various independent services, organize and customize them in a specific flow to achieve their goal [95]. Similarly to such services, the proposed platform will support high-level abstraction and automation to compose model management services provided by different repositories.

This chapter presents the MDEForgeWL that enables efficient composition and discovery of model management services and modeling artefacts by means of a dedicated low-code development platform. Moreover, by using a dedicated DSL, the user can specify complex workflows to orchestrate underneath model management services. The language is based on a *trigger-action* paradigm where services can trigger the execution of other ones. Users can plan, organize, customize and execute an arbitrary model-driven task workflow by involving independent model management services in a specific flow to achieve their defined goals [92].

The MDEForgeWL engine is implemented using a microservice-oriented architecture by exploiting the Kubernetes technology.[34] Kubernetes offers out-of-box benefits such as auto-scalability, extensibility, and dynamic selection of services based on the load [92]. Moreover, Kubernetes permits the discovery of model management services, and their usage via remote APIs. The code repository of MDEForgeWL is available online.[35] Thus, the main contributions of this chapter are the following:

- Support service and model artefacts' 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 on a cloud-based model repository

The chapter is organized as follows: Section 5.1 discusses the background, and Section 5.2 makes an overview of related works and a comparison of existing approaches to compose model management operations. Section 5.3 introduces the architecture of MDEForgeWL and presents the MDEForgeWL language at work by means of an illustrative example, whereas Section 5.4 concludes the chapter.

[33] https://www.eclipse.org/epsilon/doc/workflow/
[34] https://kubernetes.io
[35] https://github.com/Indamutsa/model-management-services.git

5.1 Background

In this section we discuss the background of this work by focusing on aspects related to *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 [96]. This new paradigm uses cloud computing to encapsulate heterogeneous and autonomous services into a service pool that exhibits various functional and non-functional features [97]. 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.*" [97]
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 [97]. Moreover, such migration is a pillar in supporting features such as collaboration, remote reuse, high availability, extensibility of model artefacts, and their management services [98]. 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 [97]. Besides, it fosters inter-organizational interaction by enabling service discovery, composition, and execution of their business logic [97]. 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 [99].
Generally, a service is defined as an invokable network, independent high-abstracted and self-contained remote operation that executes low-level functionalities and might return some data [100]. 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 modeling artefacts 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 [101]. To enable these features, service preconditions, effects, inputs, and outputs are encoded in a computer-interpretable form such as a DSL [96].
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 [102]. 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 artefacts reusability, and from the user's perspective, she has seamless access to a variety of complex services [103].

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 [104]. 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 [105]. Due to the arduous effort involved in developing a domain-specific language (DSL), MDE techniques are wielded during their design, and implementation [106].
In this context, MDE techniques express solutions at the same problem domain level. 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 [104]. The former offers projectional editing that facilitates parsing the text, thus overcoming the limits of developing DSL editors [102]. As for Xtext[36], 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 [106].

[36]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 (see Sec. 5.2.1). Different criteria are also presented to elaborate a comparative table of the analyzed approaches (see Sec. 5.2.2).

5.2.1 Overview of related work

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 [107, 108]. 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 [109]
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 artefacts and tools [110]. 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 [110]. We intend to implement self-healing mechanisms within our DSL that do 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[111], 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 [112] is a more advanced language towards reactive model management workflows. 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 [113] 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 [114] 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 [115] 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[110] 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) [27] paradigm. Moreover, as shown in Table 8, existing approaches can be analyzed concerning the features described in the following.

- *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

Table 8: 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							
Artefact sharing capabilities	✓						
Sharing configuration	✓						
Reusability							
Code reuse	✓	✓			✓		
artefacts' 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	✓						

open-source modeling framework, and it is no surprise that most of the tools make use of it. Several initiatives have been migrating that infrastructure to the cloud[37] and enable 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 artefacts or enable collaboration between different stakeholders.
- *Reusability*: It concerns available means to reuse already specified artefacts.
- *Scalability*: This feature is related to the architecture used during tool implementation. We evaluate

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

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 performing model transformations using ATL[38] rather than ETL[39] 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 8 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 27 shows an overview of the proposed MDEForgeWL architecture, which has been designed to support the definition and execution of scalable workflows consisting of cloud-based compositions of model management services. The architecture relies upon and extends MDEForge [91] by adding discovery mechanisms to identify available services involved in the workflows being executed. The architecture is organized into four tiers: the *front-end*, the *execution engine*, the *cluster of model management services*, and the *persistence layer*. In the following, the four tiers of the proposed architecture are singularly described.

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

Figure 28 shows a mock-up of the proposed environment providing users with the ability to create and automate workflows on cloud-based repositories using graphical environments with drag and drop capabilities, and custom scripting by the use of a domain specific language as referred to in figure 30. The custom scripting is enabled by an editor where the user can programmatically express complex expressions of the workflow. The services and extensions on the repositories are organized in decoupled and distributed microservices to emphasize the separation of concerns and foster individual service maintainability, scalability, and extensibility [116].

According to the explanatory workflow shown in Fig. 28, the citizen developer might want to upload a Performance Model Interchange Format (PIMF) model [117] and generate a corresponding SySML model out of it. Then, she can validate the model, calculate dedicated metrics, extract some metadata, and once done, merge the obtained information into another SySML model. The obtained model can be stored in the repository, and the user can be notified together with the complete execution logs. The services used in the above scenario are remotely accessed as services through APIs, and the storage systems are distributed services consisting of several network nodes.

[38] https://www.eclipse.org/atl/
[39] https://www.eclipse.org/epsilon/doc/etl/

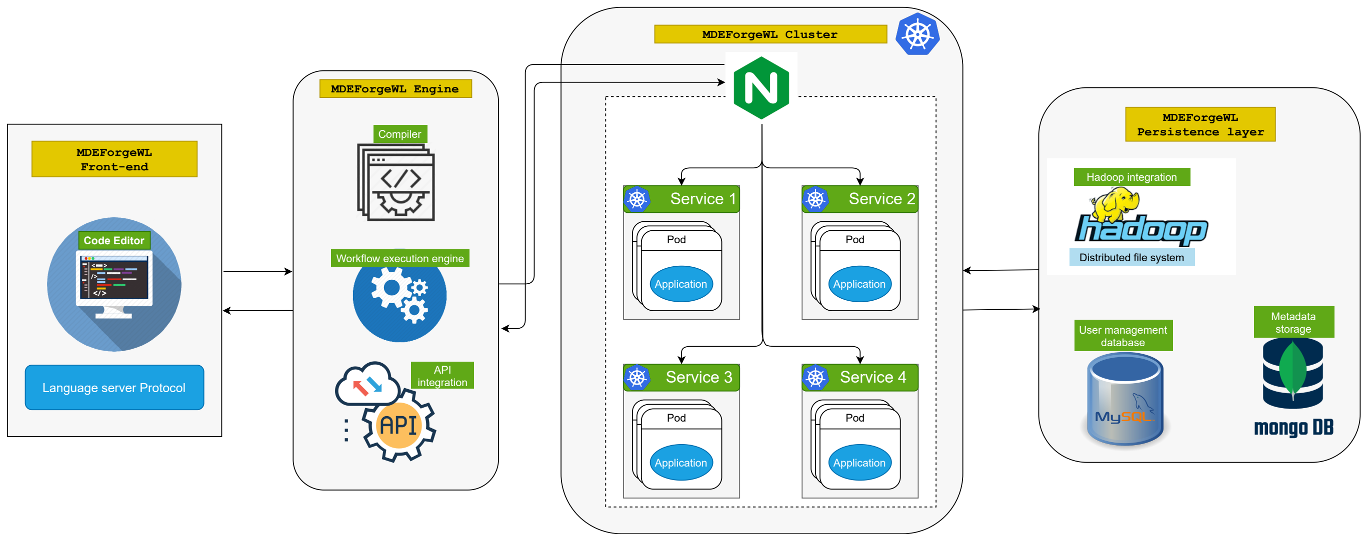

Figure 27: Overview of the MDEForgeWL architecture

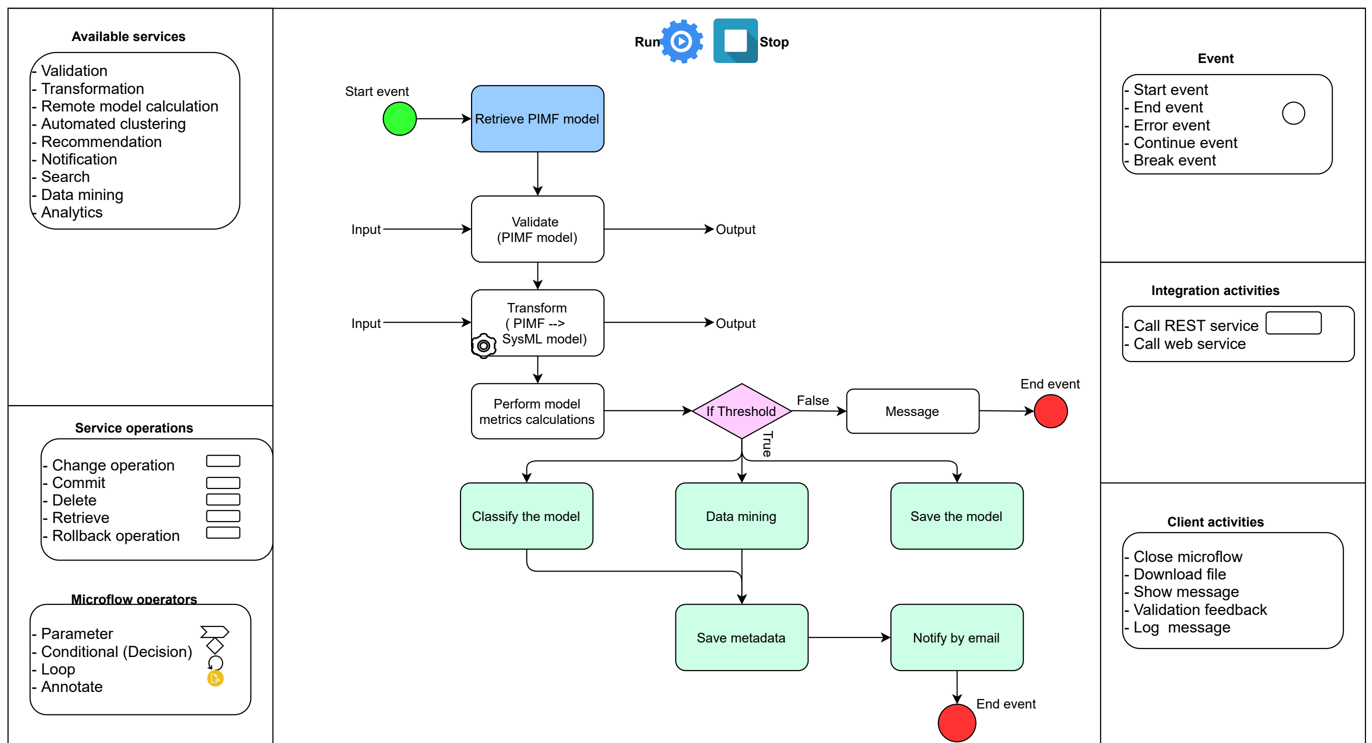

Figure 28: Example mock-up of graphical task workflow environment

Developing and execute model management workflows like the one shown in Fig. 28 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. 29. According to the shown metamodel fragment, workflows consists 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 modeling artefacts and variables. Events of interest and their sources are defined as seen in Fig 28, and they result from different providers that trigger specific actions as instructed. Nodes represent decoupled, and independent micro-services orchestrated when the specified workflow is executed.

Figure 30 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 analyse, test, and deploy

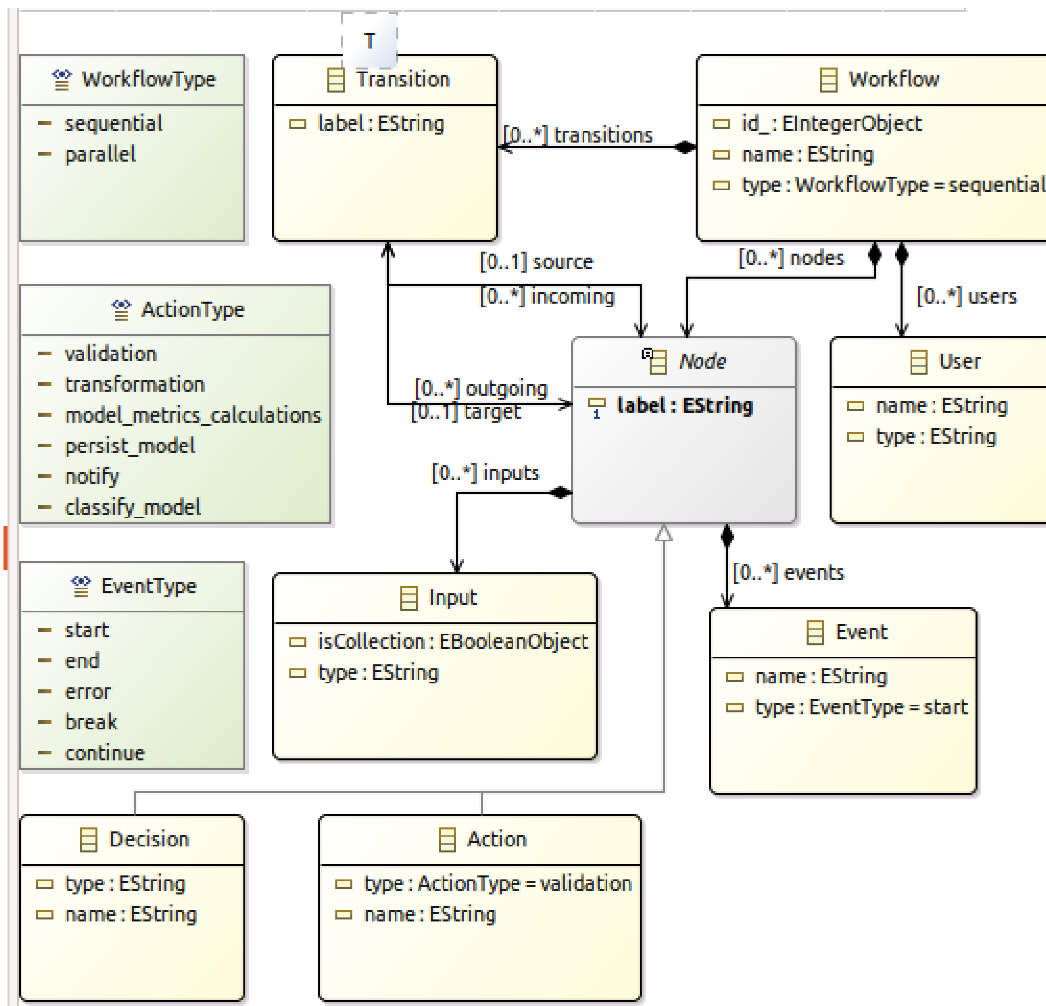

Figure 29: Fragment of the proposed workflow metamodel

models by means of a dedicated DevOps support. Such support is provided by the workflow definition and analysis component. The service integration component ensures seamless integration of external and internal services. Once the modeled workflow is ready, the incoming model (task workflow) is transformed and executed by the engine as presented in the next 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 reduced effort for the user. To achieve this objective, the language is declarative: you specify what you want, and we deliver the results according to the given specification. As previously mentioned, MDEForgeWL has been developed in Xtext, and a fragment of the corresponding metamodel is shown in Fig.31. Each specification has `WorkflowProgramModel` as root model element. Each workflow

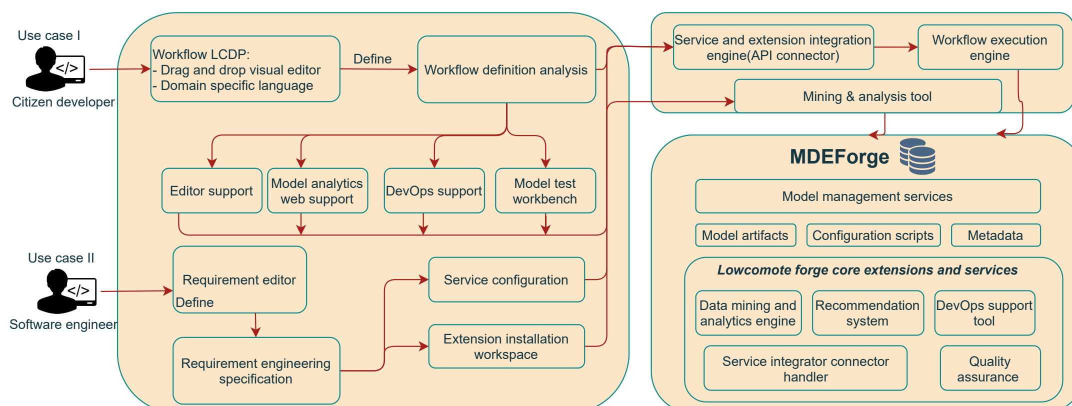

Figure 30: Logical view describing the backend and frontend aspects of the system

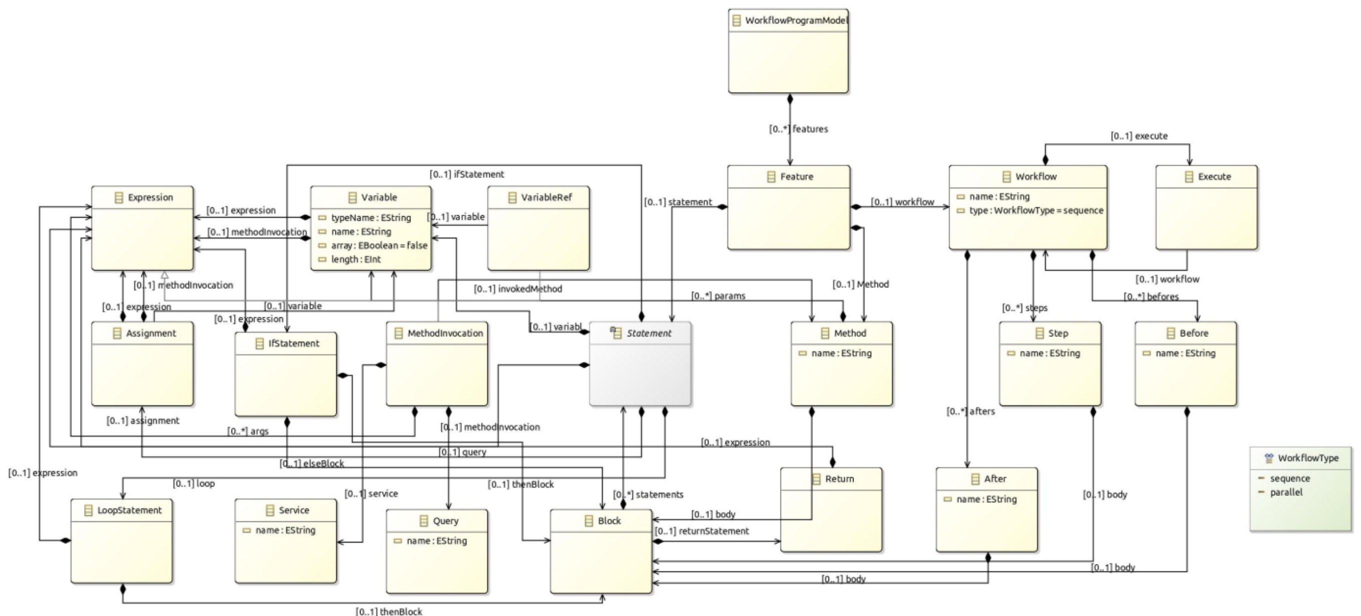

Figure 31: Fragment of the MDEForgeWL metamodel

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 statements, we can have expressions and conditional and loop statements that assist the control flow during the execution of the specification composition. A workflow block is made up 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.
Within our language, services are an abstraction of model management operations (e.g., model transformations, validations, model query, and model comparison operations). These are the functionalities that are containerized and deployed individually in the cluster. Executing model management operations like model transformations via traditional techniques could be complex, time-consuming, and error-prone, and it usually requires specific frameworks that need to be installed locally. Instead, our approach permits calling specific services with corresponding arguments, 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 1: Service call example

The argument can be a variable or the identifier of the artifact stored in the repository. As shown in Listing 2, you 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 because it permits users to specify query predicates to find the artifacts satisfying given properties. In particular, as shown in line 2 of Listing 2 2, the language permits the declaration of variables evaluated and used later in 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 carrying on with 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 artefact(id: 1, type: model, ext: xmi)
3 var sourceMetamodel = query artefact(
4         type: metamodel, ext: ecore, hasModel: sourceModel)
5
6 var targetMetamodel = query artefact(
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 artefact( name: "catalogue.xmi", conformsTo: "catalogues.ecore",
12        sharedUsername: ["john"], sharedUserNumber: < 3 )
13 var emlscript = query artefact(id: "23")
```

```
14 var eclscript = query artefact(id: "44")
15 var eolscript = query artefact(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 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[40], Grafana[41] and Zipkin[42]. We have also implemented distributed logging mechanisms within the workflow engine to monitor the workflow executions' progress and facilitate troubleshooting.

5.3.4 The MDEForgeWL cluster

This DSL can be used as a plugin in Eclipse platform, but our endeavours 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 the traditional fashion of modeling. The cluster is built using Kubernetes, an open-source container-orchestration platform [118]. 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 [118]. 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

[40]https://prometheus.io
[41]https://grafana.com
[42]https://zipkin.io

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 [118]. 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, which is referred to as a service, thus all communication are made through the service and the service load-balance the workload among the bootstrapped pods [118]. Since the MDEForgeWL cluster is deployed using Google Kubernetes Engine[43], 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. Based on current usage, desired target, and user demands, auto scalers scale up or down the number of running pods or replicas, perform dynamic management of CPU/memory utilization of machines inside the cluster and increase or decrease the number of nodes where pods are running [119].
Self-contained, fully-fledged model management services are organized in a distributed microservice architecture that ensures their resilience, security, loose coupling, flexibility, fault tolerance, extensibility, and scalability. These microservices 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 [120]. 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 [121].
The proposed microservice architecture also includes a service registry, a service API gateway, authorization & authentication server (it implements the OAuth2.0 protocol[44]) and a resource server as shown in Fig 32. 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 do 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 artefacts such as models, metamodels, and transformations. MDEForge, our cloud-based model repository, consists of model management services that allow persistence and management of typical modeling artifacts and tools. Services are accessed and used through RESTful Web APIs [91]. 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

[43]https://cloud.google.com/kubernetes-engine
[44]https://oauth.net/2/

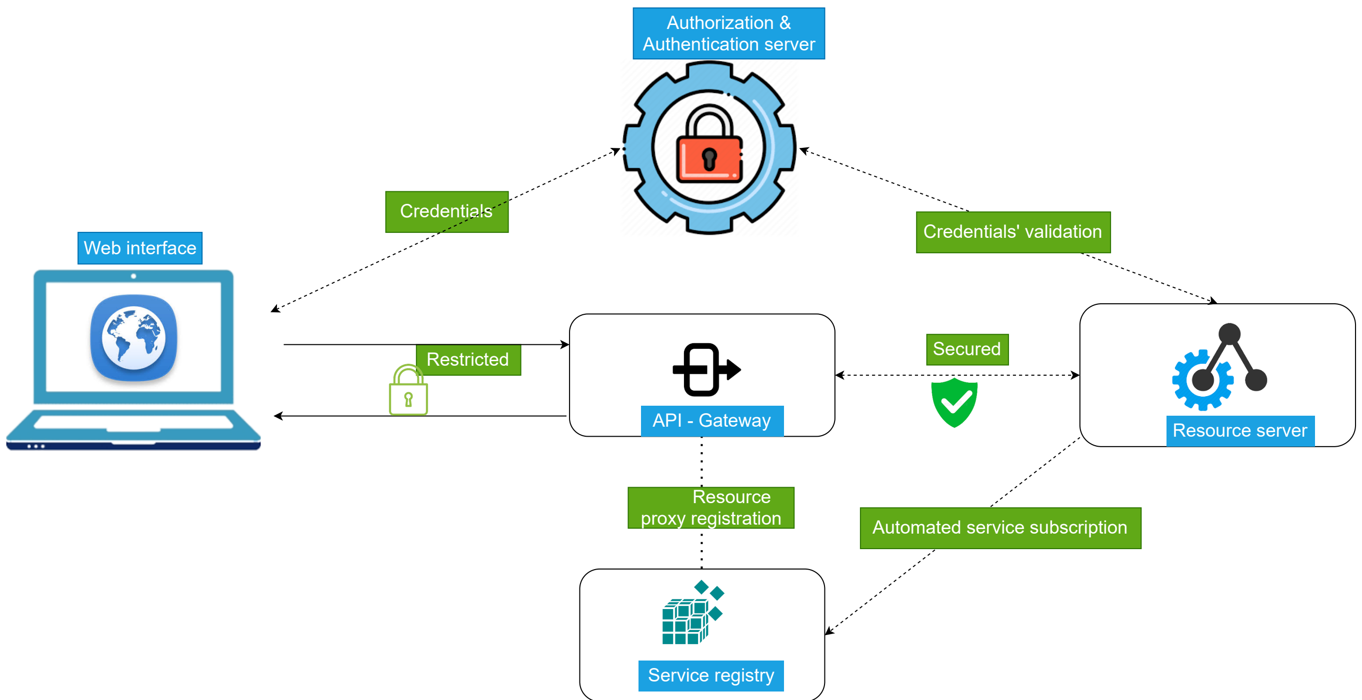

Figure 32: Detailed view of the MDEForgeWL cluster

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 presented MDEForgeWL, a novel approach to support the development of complex model management operations. In particular, a low-code development platform is presented, resembling the functionalities offered by currently available LCDPs like IFTTT and Zapier. Such platforms permit the development of complex processes by integrating and executing different services. The proposed approach proposes adopting a microservice-based architecture to integrate and execute model management services, which are orchestrated on the cloud according to specifications given by the user employing a BPMN-like modeling language.

The proposed approach aims to overcome current challenges faced by traditional modelling environments that heavily rely on locally downloaded resources. Such environments are limited in their scalability and extensibility, and their services exhibit high coupling with the local environment. In the next chapter, we show advanced query mechanisms that are provided by the proposed repository.

6 Enabling service-oriented 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 [27, 123, 124, 125]. Several initiatives in MDE have proposed a variety of technologies and facilities to simplify and automate MDE processes [98, 126]. Model management activities in particular (e.g. model transformation, validation, merging, model object queries) have made MDE a practical reality in the Software Development LifeCycle (SDLC) [127, 92] (c.f. Figure 33). These activities consist of operations that turn models into programs rather than mere sketches that are relevant only in the design and planning phases [128]. SDLC outlines each task to reduce waste and increase efficiency during the development process [129, 130]. In addition, SDLC ensures that projects stay on track and maintain a viable investment [129].

As shown in Figure 33, the stage *1* in software development is made of planning and performing requirement engineering over a software product. Right after this stage, stage *2* involves technical tasks involving MDE practices in part or whole. The stage *2* follows agile methodologies and consists of design and prototyping, software development, testing, deployment and maintenance phases. Stage *3* and *4* ensure the discovery and reuse of persisted artifacts respectively. Therefore, models are employed in whole or part to create a full-fledged application as concrete executable software artifacts throughout the SDLC [27, 128].

Empirical studies have shown that barriers to broader adoption of MDE practices need to be overcome, particularly regarding the discovery and reuse of MDE artifacts and tools [131]. Limited efficient discovery and reuse of existing modeling artifacts and tools often result in reinventing the wheel [132]. As a result, model artifacts and tools are re-developed from scratch, inflicting unnecessary upfront investment and compromising the productivity benefits of MDE-based processes [132, 133]. However, discovering and reusing artifacts require efficient mechanisms at different granular dimensions of model artifacts. In addition, the need for efficient persistence and retrieval of artifacts is primal in this quest [91]. In this context, model repositories have been conceived to tackle the issues related to the discovery and reuse of modeling artifacts and tools [134]. Because MDE can be used at various stages of SDLC, 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 (c.f. Figure 33). Furthermore, MDE relies on model repositories to enable collaborative modeling activities [98, 125].

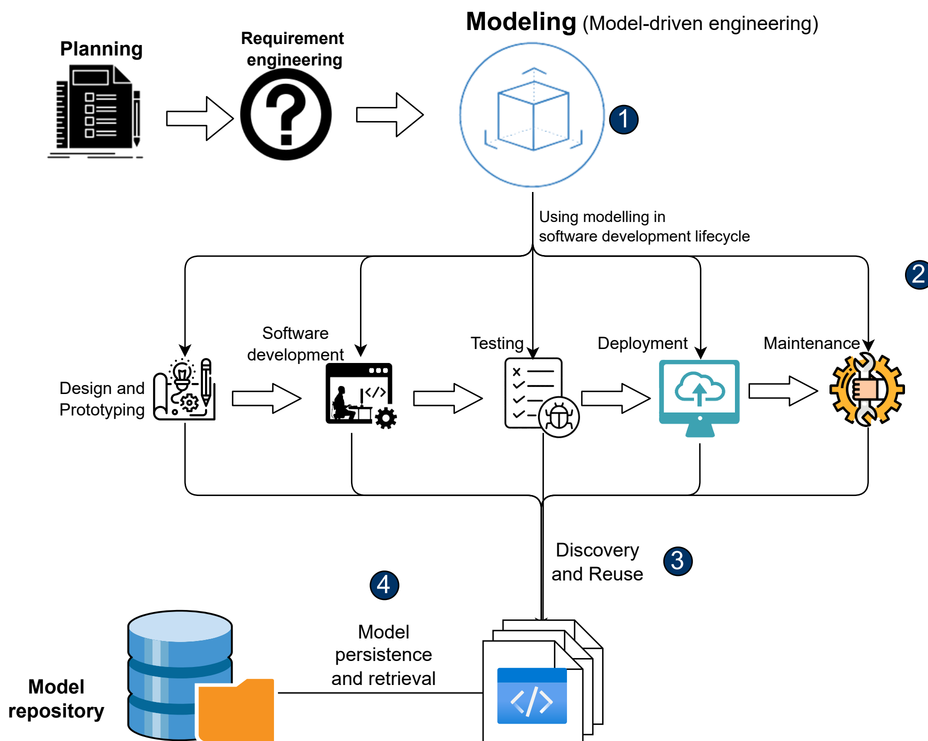

Figure 33: Modeling in Software development lifecycle

Despite significant efforts in artifact discovery and reuse, there is still a lack of efficient advanced discovery mechanisms to support model discovery, accessibility, and retrieval [124, 91]. Platform dependency of discovery tools in MDE does not provide generic and contextual MDE mechanisms for searching and discovering MDE artifacts without compromising core modeling principles. Ideally, discovery techniques

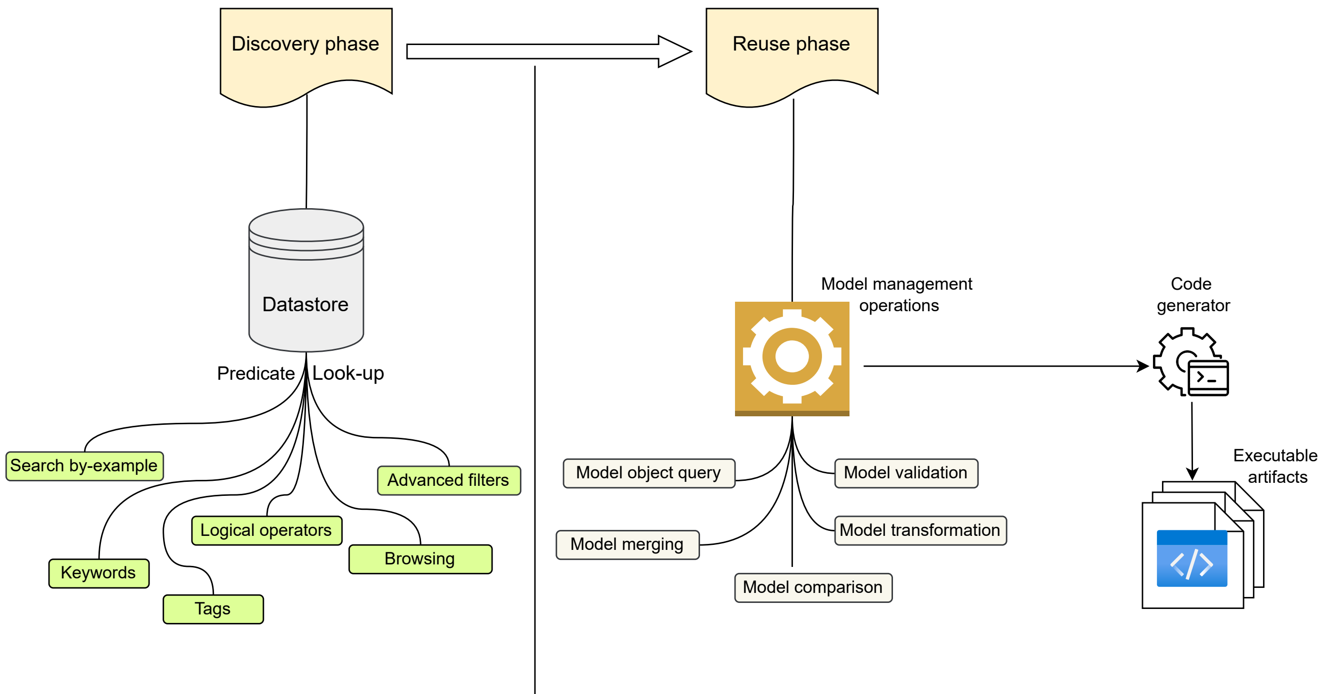

Figure 34: Discovery and reuse process of model artifacts

should be able to start from basic queries and gradually get complex to filter out the exact artifacts persisted in a distributed environment [135]. In addition, filters should not be restricted to modeling 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 [136, 137]. 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 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. The 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 [127]. 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, *iii)* and with 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 predicate. In addition, retrieved artifacts can be reused on the same platform along with the deployed MMOs.

The provided approach is built upon a cloud-based and distributed infrastructure using cutting-edge technologies [138, 139]. They play a vital role in developing and managing micro-services and their orchestrations in cloud-based distributed systems [138, 139, 127].

The main contributions of this chapter are the following:

- Provide an overview of existing mechanisms to discover modeling artifacts in model repositories;
- Identify significant challenges to achieving the practical discovery of modeling 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.

6.1 Background and Motivation

Modern modeling techniques in data-intensive applications require the discovery and reuse of relevant model artifacts [135]. In this context, there are two phases in the reuse process of modeling artifacts (c.f. Figure 34). 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 [140]. MDE offers the concept of a megamodel as a building block for large-scale modeling [141, 140, 142]. A megamodel is used to create and leverage global relationships and metadata on basic macroscopic entities as models and metamodels [141, 140, 142, 143]. A megamodel conceals fine-grained details that impede comprehension of the system in its global perspective [141, 143]. The global picture includes consideration of system architecture, interactions between model artifacts, relationships between artifacts, results of model transformations, etc [143].
By taking into account the megamodel-structure in the predicate constructs, predicates are used to perform an exhaustive look-up of stored data to retrieve model artifacts based on internal element composition [144]. Implementing such discovery mechanisms can be error-prone and time-consuming if not adequately supported by a cloud-based distributed environment [144]. However, these mechanisms enable flexible and fast retrieval of relevant artifacts using *a combination of various criteria* such as megamodel structure, tag mechanisms, search keywords, logical operators, search by-example and browsing [134, 98]. The returned data at the end of the first phase (discovery) are model artifacts, not model artifact composition elements(c.f. Figure 34). Retrieved artifacts are then reused in the reuse phase, where they can be manipulated, merged, compared, queried and transformed using MMOs before they are used to generate code that directly participates in SDLC (c.f. Figure 33, 34).
Modern model repositories exhibit the following features:

- Huge number of large inter-related heterogeneous model artifacts of models, transformations, data files, source code, file descriptors
- 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 such as model transformation, model object query, model validation, model merging, and model comparing

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 certain size greater than a given size. 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

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 technological agnostic. Unfortunately, most of the current solutions in this area are still platform-dependent [145, 146]. 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 mechanisms that allow query specification at low-level granularity, requiring a high learning curve and technological expertise [146, 145].

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 6.2.3). We will also discuss existing tools and approaches in light of the selected features (c.f. Section 6.2.2).

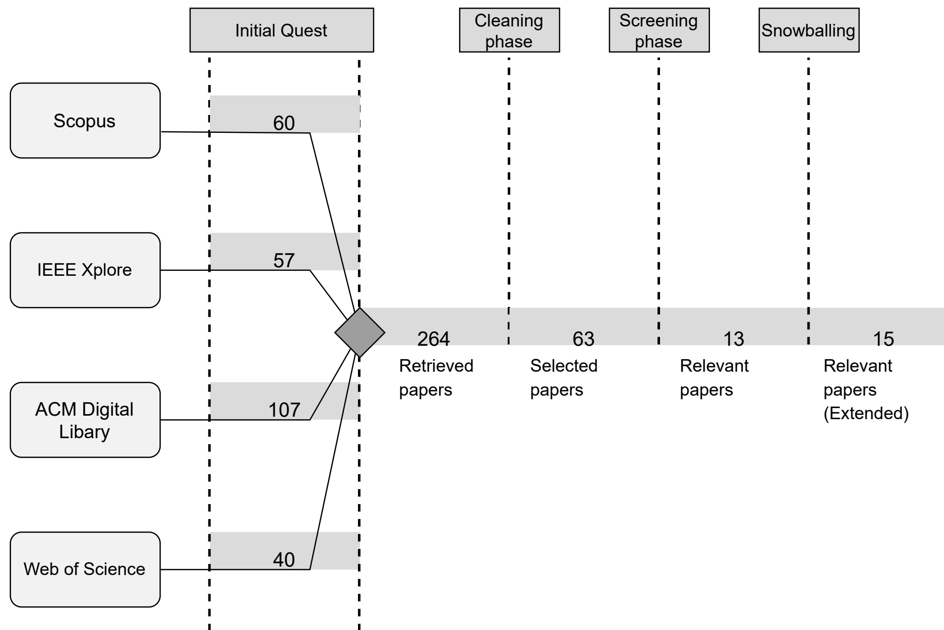

Figure 35: Publication selection process

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 modelling artefacts?*

We performed a systematic investigation to identify the main challenges of artifact discovery on model repositories. Following accepted guidelines [147, 148], we conducted a search, selection, and mapping process. We screened and identified relevant published literature as shown in Figure 35 and Table 9.

Table 9: 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 [45], Web of Science [46], the ACM Digital Library [47] and IEEE Xplore [48] (c.f. Table 9). Although each database has its query string specifications and search fields, we tried to find publications that contained keywords from each column of Table 10. These keywords should be found directly in the title, abstract, or keywords.
We managed to retrieve 264 documents: Scopus (Elsevier)[45] provided 22.6% of the total documents. ACM Digital Library[47] responded to the query with the highest number of 40.2% of the total documents retrieved, as shown in Figure 36. The query of IEEE Xplore[48] and Web of Science[46] yielded 21.4% and 15% of the total retrieved documents, respectively.
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.

[45]https://www.scopus.com/home.uri
[46]https://clarivate.com/webofsciencegroup/solutions/web-of-science/
[47]https://dl.acm.org/
[48]https://ieeexplore.ieee.org/Xplore/home.jsp

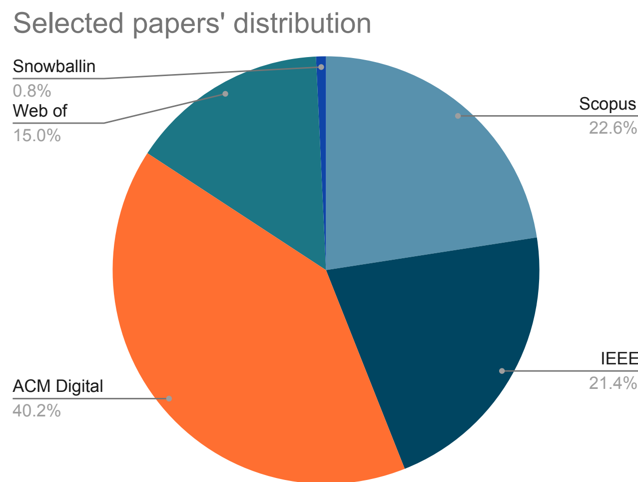

Figure 36: Distribution of queried papers.

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 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, 13 articles were selected. The tools are further analyzed and compared in Table 11 and Table 12.
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 15 papers: [149], [150], [135], [151], [32], [152], [144], [153], [154], [155], [156], [157], [158], [159], [160]. In the next section we reviewed the approaches proposed in the selected papers.

Table 10: 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

6.2.2 Results

MAR [149] 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 6.2.3). It uses HBase[49] to create an index that is later used to enable fast query response. Recent updates have integrated the Lucene search engine[56] to support indexing and keyword-based search [150]. 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 [135, 151] 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.

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

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 [32]. 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 [32].

EFinder [152] is an Eclipse plugin developed to support model search in EMF ecosystem. It is designed to implement scenarios such as model consistency, example generation, partial solution and scrolling. Model artifacts are filtered based on Object Constraint Language (OCL) constraints. However, the tool has technological dependencies that could hamper model discovery of heterogeneous model artifacts from environments other than the EMF ecosystem. The syntax used in model discovery is complex and presents a significant learning curve, precluding citizen developers. Scalability and extensibility are not considered.

Basciani et al. [144] 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 modeling 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.

Moogle [153] 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[50]. It has EMF parsers that read and parse EMF model artifacts and index them later. Moogle can index different model artefacts 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 [154] 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 [155] 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 [124].

In [156], the authors use a keyword-based ontology called WordNet[51] 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 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 modeling artifacts.

[50] https://solr.apache.org/

[51] https://wordnet.princeton.edu/

Bislimovska et al. proposed MultiModGraph [157], 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 [158] 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.[159] 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.[160] 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.

6.2.3 Designed features for productive discovery mechanisms

To answer the research question ***RQ1***, we analyzed the collected papers previously overviewed and identified important features that meet current needs related to the discovery and reuse of model artifacts and tools. Each feature is explained in the following, along with the respective roles in a discovery facility. The overall features are used to compare the selected tools as shown in Table 11 and Table 12.

Management of megamodel relations

Megamodeling [161] provides a way to define different types of relationships between modeling 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 the below features to come into play in discovering 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 creating subsequent models. Since metamodels can be extended while maintaining the original conformance, this enables 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 11 and Table 12.

▹ *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 involved artifacts 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 does not support 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 [91, 127].

Model-as-a-service (MaaS)
This cloud computing model enables the provision of model management operations in the form of services [162]. 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. Furthermore, enabling MaaS on a discovery platform is essential to populate the index with heterogeneous artifacts from different sources and improve reusability factors.

▹ *Model Management Services (MMS)*: This feature is intended for tools that use a data store and MMSs. This scheme facilitates the reuse of artifacts managed on the same platform as the discovery mechanisms. In addition, 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. Therefore, 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 group reflects the discovery mechanisms' support of different model artifacts and tools. 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. For example, 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. Therefore, filtering these artifacts by their type, e.g., models, metamodels, or scripts, is essential for discovery mechanisms. Moreover, their relationships are the basis of the megamodel, which is also important for the discovery mechanisms of model artifacts.

Access interface
This group of features refers to the access options that allow interaction between the 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 essential to opt for facilities (e.g., web facilities) that involve citizen developers in discovery processes.

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

▹ *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
This features group refer to means facilitating quality-aware searching for artifacts. In particular, search tools should be able to retrieve artifacts based on their quality score to filter out low-quality elements.

▹ *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, this feature allows tools to trigger the calculation of metrics based on events, which are, in turn, triggered when a specific artifact is created or updated. The derived metrics are kept along with the metadata of the artifact.

Indexing support

Features in such group allow 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 automatically indexes newly created artifacts or when they are updated or deleted. 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. This way, files are automatically indexed when they persist without manually managing or performing indexing of specific artifacts.

▹ *Full-text search support*: This feature 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[52] [144] or a custom-developed search engine such as MAR[149].

Tool interoperability
Features in this aspect allow integration with other tools and facilitate reuse in third-party applications.
▹ *Generated REST API*: This feature allows developers to convert queries made from the front end 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 essential to develop modular APIs to facilitate the extensibility of the search/retrieval mechanisms.

Query mechanism
Discovery mechanisms support several types of queries. We have investigated and identified tools and related mechanisms, as shown in Table 11 and Table 12. Below are some of the key instruments used in the search/discovery of modeling 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. In addition, the retrieved collections are usually ranked based on a 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. In addition, artifact can be retrieved based on their internal features, such as class names or attributes.

▹ *Query by-example query*: This feature allows users to search for and discover artifacts using examples and a simple template. The user specifies an example model for artifacts describing the expected results in the template.

▹ *DSL-based*: Tools that use domain-specific query languages or extensions of existing languages.

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

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

▹ *Advanced filters*: Framing queries to return results that 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 categories. 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 artifacts.

Scalability support

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

Table 11: Comparison table of various discovery tools (1/2)

Feature	**MAR** [149]	**IncQuery** [135]	**EFinder** [152]	**MDEForge** [144]	**Bislimovska et al.** [159]	**Gomes et al.** [156]
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			✓			

The services involved in the discovery mechanisms must be scalable and extensible to support the workload of a potentially large community of users.

▹ *Service orchestration*: This feature allows the deployed services involved in advanced discovery mechanisms to be managed according to the system load. Furthermore, their execution is done seamlessly, so 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 many services that need to be strongly decoupled to facilitate their reusability.

▹ *Cloud deployment*: Tools that can be deployed in the cloud, enabling execution over the Internet.

6.2.4 Comparing model search approaches

We selected works with supporting tools from the existing approaches identified in Section 6.1 and as shown in Table 11 and Table 12.

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. [144] is one of the 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 [149], MDEForge [144], Hawk [155], Moogle [153], and Bislimovska at al. [159] have managed to integrate a search engine into their mechanisms. Using a search engine is important to

Table 12: Comparison table of various discovery tools (2/2)

Feature	**Moscript** [154]	**Hawk** [155]	**MultiModGraph** [157]	**MORSE** [158]	**Kotopoulos et al.** [160]	**Moogle** [153]
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	✓					

ensure fast retrieval of relevant artifacts. Thanks to this feature, they implemented tag-based, keyword-based, and query-by-example mechanisms.
Table 11 and Table 12 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 [154], IncQuery [135], Hawk [155], Kotopoulos et al. [160], and EFinder [152] use external domain-specific languages (DSLs) to search for modeling artifacts. 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 11 and Table 12 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 [144], MAR [149], MORSE [158], Moogle [153], and IncQuery [135] 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[149] and IncQuery[135] are among the most popular tools that can be used as cloud-based services. These tools have implemented service mechanisms that enable on-demand execution to assist users during model search and discovery sessions.

6.2.5 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 [163, 164]. Due to the amount of data that both systems and users generate regularly, traditional data processing methods have proven inadequate for the tasks expected [165, 166]. 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 [167, 168]. 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 modeling 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 [146]. The volume of data received from systems and users is too large for traditional computing paradigms [169]. Local infrastructures are not designed for Big Data, so cloud computing is chosen [170]. 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 [149], Moogle [153], MDEFoge [144] and IncQuery [135] 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 11 and Table 12 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 11 and Table 12.

▹ *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. 34. 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 presents MDEForge-Search, an approach able to address the limitations presented in Section 6.2.5. MDEForge-Search is designed to support discovery and reuse of model artifacts and tools. In the following, we present architectural design and logical layers of the approach.

6.3.1 Architectural design

In this section, we describe the MDEForge-Search high-level architecture. MDEForge-Search is an extension of the MDEForge infrastructure [91, 127]. Figure 37 shows a high-level view of the building blocks of the proposed platform. They are deployed using a distributed micro-service architecture based on Kubernetes[53] [127]. In the remainder of this section, we dive deeper into its architectural design as shown in Figure 37.

6.3.1.1 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[54]. For a durable data store that can handle highly intensive computational jobs and transactions, it is advantageous to have MongoDB[54] as the single primary source of truth for writing operations, rapid data ingestion, and ultimately index data in Elasticsearch[55] [171]. We thus offload search and analytics' activities to Elasticsearch[55]. Elasticsearch[55] is a distributed, open-source, and highly scalable search and analytics engine. It is built on Apache Lucene[56] 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[172]. 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 backup [173] 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.

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. Figure 39), and model management services as shown in Figure 30. 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 [127]. MaaS enables developers to design and maintain modelling resources and tools that are subsequently made available to end-users as software as a service (SaaS) templates [174]. In this cloud computing tier, web services are the foundation for building distributed applications (c.f. Figure 39). Business logic and underlying technology are abstracted into packages, and high-level on-demand capabilities are outsourced via the Internet [174, 134]. Its usage enables on-demand execution of services across the network [134, 162, 175]. Moreover, it facilitates collaboration among stakeholders, resource optimization and interoperability regardless of underlying technologies [134]. With MaaS, local configuration and infrastructure setup are replaced with cloud-based

[53]https://kubernetes.io/
[54]https://www.mongodb.com/
[55]https://www.elastic.co/
[56]https://lucene.apache.org/

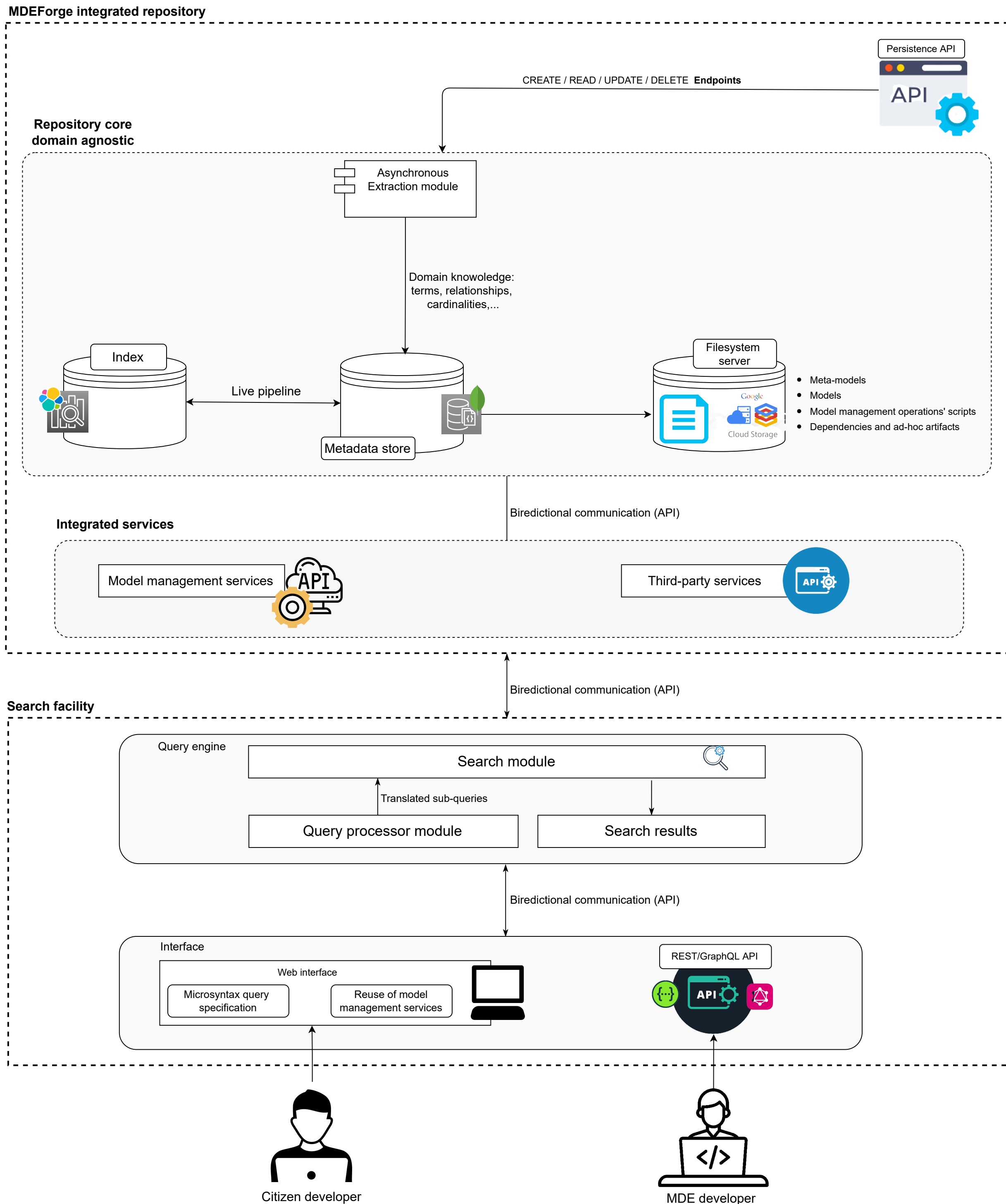

Figure 37: MDEForge-Search high level architecture

infrastructure, thus drastically lowering time-to-market [98]. 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 [127].

6.3.1.2 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 6.3.2.

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

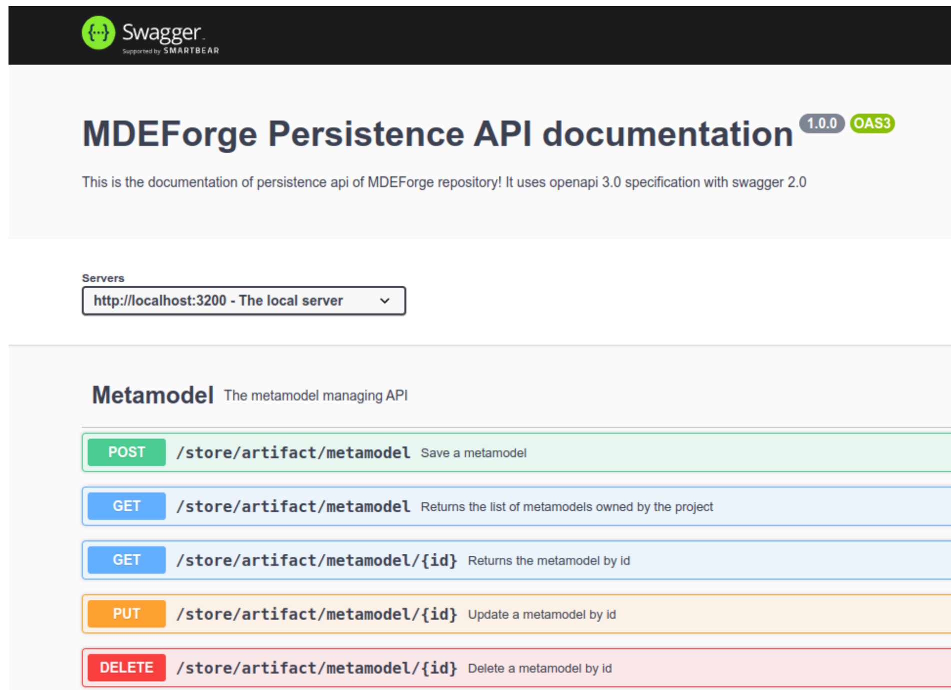

Figure 38: MDEForge-Search Persistence API

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 6.4.2.1). 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 6.4.4).
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 [176] (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 deployed on the same platform. She can also perform CRUD operations on the retrieved artifacts.

6.3.2 Logical layers

Figure 39 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.

6.3.2.1 Data layer The journey of data persistence in a cloud-based model repository begins with an API built on the repository warehouse. It is the part of the repository that houses structured data pulled and pre-processed from the data lake. The data lake is not organized, and data is collected in their varied sets of raw data in their native format. Before further processing, unprocessed data are dumped in the data lake. 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. Collected heterogeneous modelling artifacts are organized as models, meta-models, or DSL scripts. The actual files, however, are stored at Google cloud storage, as shown in Fig. 37. The link

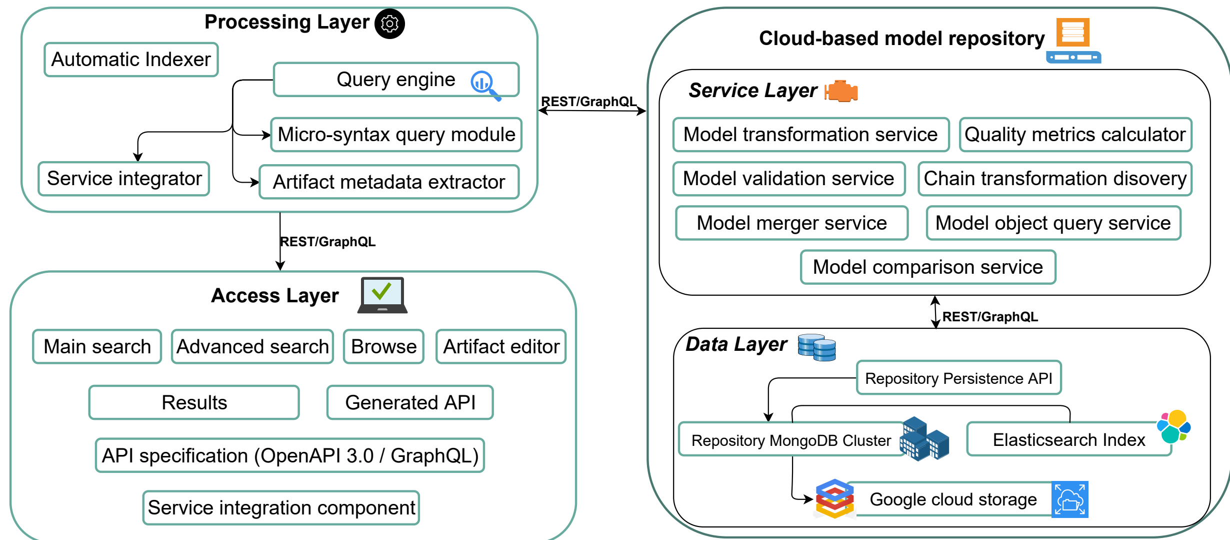

Figure 39: MDEForge-Search Persistence API

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.

6.3.2.2 Service layer It is a cluster that provides a pool of model management services deployed natively in the cloud-based environment [127]. Each service wraps a corresponding engine responsible for carrying out model operations previously deployed on local infrastructure, as shown in Figure 39. Following the MaaS paradigm, these model operations are exposed externally and can be executed on-demand over the Internet [177]. 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.

6.3.2.3 Processing layer It processes the query from the application layer (c.f. Figure 39). 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).

6.3.2.4 Access layer It consists of graphical interfaces and APIs that are used to explore the repository via the query engine (c.f. Figure 37 and Figure 39). 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 search results are displayed based on the relevancy score respectively [172]. 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.

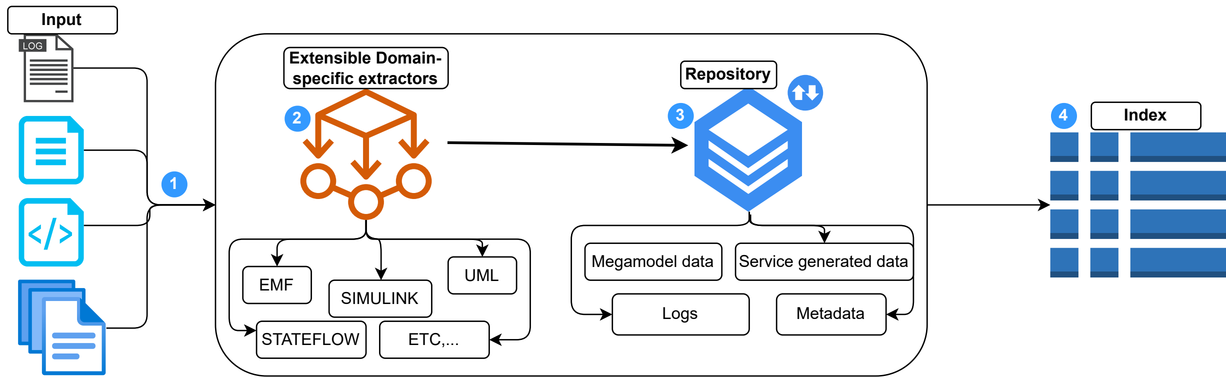

Figure 40: Data flow in MDEForge-Search

6.4.1 Data ingestion and processing

In a cloud-based model repository with heterogeneous artifacts, ingesting and processing data is critical. As shown in Fig. 40, 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 Figure 40, *(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 *(i4)* (c.f Figure 40) involves indexing structured data for rapid discovery and exploration.
Data in Elasticsearch is indexed in a cluster of nodes that are replicated and sharded [178]. Internally, Elasticsearch employs Apache Lucene[56] to index data using an inverted index. An inverted index is a data structure to enable fast retrieval of matched terms [179, 178].
Elasticsearch employs the boolean model to find matching documents. It also uses function called *practical scoring function* to retrieve relevant artifacts [178].

Practical scoring function [179]

```
score(q,d) =                          [1]
        .queryNorm(q)                 [2]
        .coord(q,d)                   [3]
        . ∑ (                         [5]
            tf(t in d)                [6]
           . idf(t)^2                 [7]
           . t.getBoost()             [8]
           . norm(t,d)                [9]
           ) (t in q)                 [4]
```

The above function [1] `score(q,d)` returns the relevance score of a document `d` for query `q`. [2] `queryNorm(q)` is the query normalization factor [179] represented mathematically as such:

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

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.

[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 [179]:

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

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

[4] `sums` the weight for each term t in query q [5] for document d. [6] `tf(t in d)` assesses term frequency for term t in document d. [7] `idf(t)` calculates the inverse document frequency for term t. It responsible of punished repetitive terms such as like, or, and, so,... thus a lower weight is assigned to such terms making less frequency terms relevant to zoom in the relevant document. [8] `t.getBoost()` The boost applied to the query to make one query clause more important than the other. [9] `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 [179, 178].

6.4.2 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.

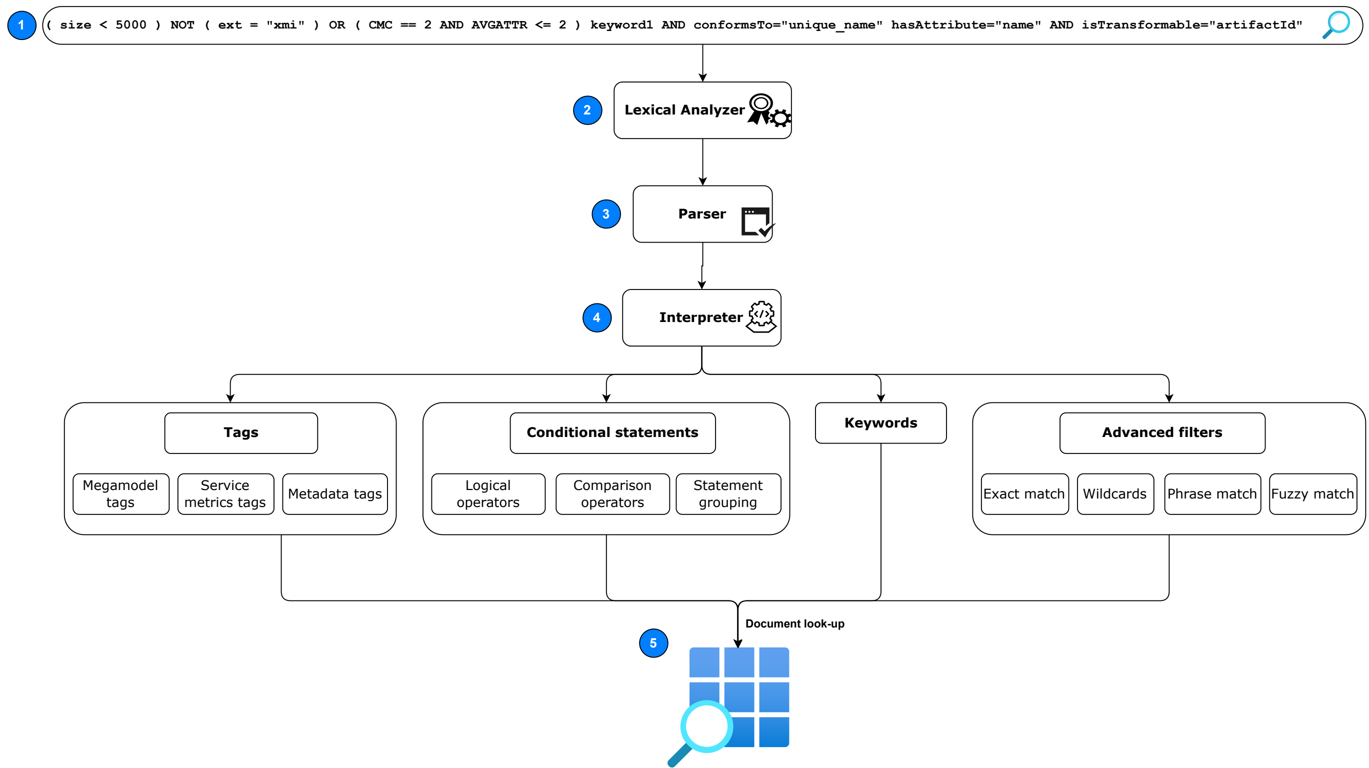

Figure 41: Microsyntax query-specification information flow

6.4.2.1 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 [180]. 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 [181]. In this regard, our domain-specific microsyntax query specification is designed to facilitate the discovery and exploration of the repository using an advanced query mechanism. Figure 41 shows the flow of information inside the infrastructures that underpin the microsyntax-based query specification mechanism.

As shown in Fig. 41, (i) the user formulates a query in the form of one query line of text. Next, the text is consumed by (ii) a lexical analyzer that breaks down the text into tokens. This process involves scanning the input for defined patterns and dividing it into meaningful units used by the parser later on [180]. (iii) Once the input has been tokenized, it is used to build an abstract syntax tree (AST). The AST is a tree-like structure that represents the code in a way that is easier for the machine to digest [180]. (iv) After the AST is constructed, the AST is traversed and converted into machine code that is executed to produce the desired results. In this way, the computer interprets the query text to generate an executable artifact. As shown in Fig. 41, the (iv) phase of this process considers essential constructs that compose the microsyntax query specification. These constructs include tags, conditional statements, keywords

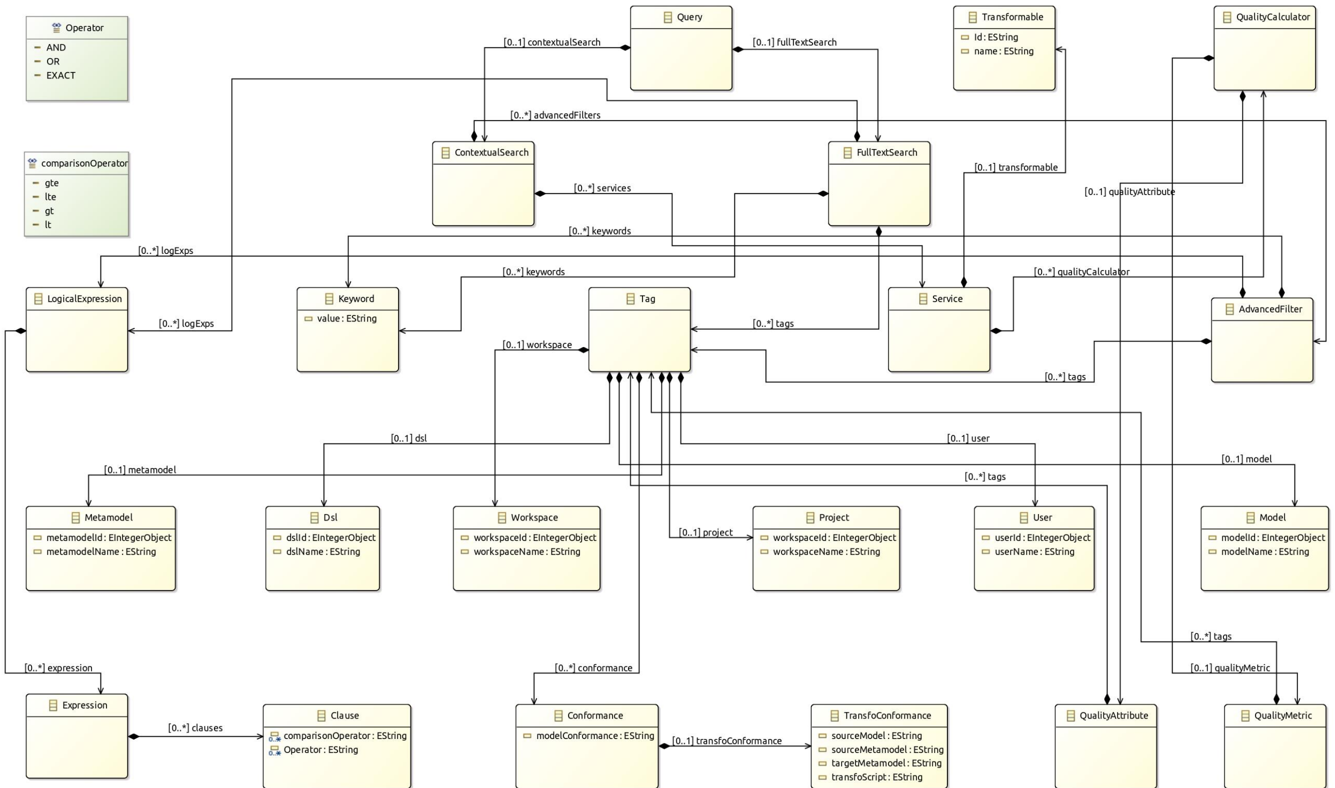

Figure 42: Simplified metamodel of the query specification

and advanced filters, discussed in the next paragraph. The final phase (v) is the document look-up in the repository index, which we explained earlier in the previous section 6.4.1.

The query metamodel

As shown in the simplified metamodel in Figure 42, at the top of this microsyntax query specification, we have two entities: *Full-text search* and *Contextual search*. *Full-text search* performs an exhaustive lookup using predicates across the persisted artifacts, related logs and metadata. Contextual search provides search in a precise search space such as artifact content, creation timestamp, name, type and so on. The full-Text search consists of search keywords, tags, and logical expressions, which are conditional statements that can be expressed into nested clauses. The conditional statements are made of logical and comparison operators, as shown in Figure 42. The boolean logic allows the user to combine terms with AND, OR, and NOT operators to efficiently narrow and filter relevant documents containing the desired information. Search keywords are words that are uniquely tokenized to facilitate indexing in an inverted index data structure.

Tags are critical in our query specification because they can direct you to the relevant artifact. Tags are composed of repository data organizational structure, which is user-oriented and hierarchical to support users, workspace and projects along with related access permissions. They target artifacts by properties or metadata fields. Tags consist of conformance features, where the user can retrieve models that conform to a given metamodel. We also support transformation conformance, where the user can retrieve artifacts that participated in a given transformation operation. Contextual search uses the microsyntax query specification to a given field of an artifact (c.f. Figure 44).

Figure 43 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 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.40 (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 Figure 43, 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 search 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. Figure 42). 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 (Contextual search) in these circumstances. More tips are provided on the web interface on

retrieving exact matches, special characters and how to escape them. We also instruct how to perform fuzzy searches or the usage of wildcards.

Figure 43: Search results

6.4.2.2 Advanced searches Figure 44 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 advanced search integration employs a combination of the microsyntax query specification, the quality assessment service, and the optimal transformation chain of a given metamodel to find relevant artifacts. The microsyntax query specification can be used against an artifact field 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. In addition, documents can be retrieved based on their quality attributes and metrics or whether they participated in a given chain transformation (c.f. Figure 44).

Figure 44: Advanced search

6.4.2.3 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 Figure 45). 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. Figure 46). 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. Figure 47).

MDEFORGE
Search Facility
Browse Services Log In Register

Browse model artifacts

Select artifact type: Metamodel

Alphabetical

A B C D E F G I J K L M N O P Q R S T U V W Y Z

- Academia.ecore
- Accident.ecore
- Accounting.ecore
- Admission.ecore
- Advertisement.ecore
- Agriculture.ecore
- Animal.ecore
- Apartment.ecore
- Apple.ecore
- Arizona.ecore
- Army.ecore
- Arrival.ecore
- Article.ecore
- Artifact.ecore
- Artist.ecore
- Assistance.ecore
- Association.ecore
- Astronaut.ecore
- Athlete.ecore
- Atlas.ecore
- Attack.ecore
- Attorney.ecore
- Auction.ecore
- Audience.ecore
- Author.ecore
- Automobile.ecore
- Aviation.ecore

1 2 3 4 5

© MDEForge. All right reserved

Figure 45: Browsing page

MDEFORGE
Search Facility
Browse Services Log In Register

Name: SimpleOOP.ecore

Name: SimpleOOP.ecore
Description: "We are trying to save the metamodel using the api"
Type: METAMODEL
StorageUrl: http://178.238.238.209:3200/files/metamodels/SimpleOOP-1649864593972-32.ecore
Project: Project_id
Workspace: Workspace_id
Access control: PUBLIC
CreatedAt: TimeStamp
Last modifiedAt: TimeStamp

Edit content

© MDEForge. All right reserved

Figure 46: 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 47.

MDEFORGE Search Facility

Browse Services Log In Register

Name: SimpleOOP.ecore

This XML file does not appear to have any style information associated with it. The document tree is shown below.


```xml
<ecore:EPackage xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
  <eClassifiers xsi:type="ecore:EClass" name="Program">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="pName" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="classes" ordered="false" lowerBound="1" upperBound="-1" eType="#//Klass" containment="true">
      <eAnnotations source="http://www.eclipse.org/OCL/Collection">
        <details key="nullFree" value="false"/>
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Statement" abstract="true"/>
  <eClassifiers xsi:type="ecore:EClass" name="MethodInvocation" eSuperTypes="#//Statement">
    <eStructuralFeatures xsi:type="ecore:EReference" name="source" eType="#//Objekt" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="target" eType="#//MethodDeclaration"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Assignment" eSuperTypes="#//Statement"/>
  <eClassifiers xsi:type="ecore:EClass" name="Loop" eSuperTypes="#//Statement"/>
  <eClassifiers xsi:type="ecore:EClass" name="Objekt">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="type" eType="#//Klass"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Parameter">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" lowerBound="1" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="type" lowerBound="1" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
</ecore:EPackage>
```


Cancel Update

© MDEForge. All right reserved

Figure 47: Artifact editor

6.4.3 Platform reuse mechanisms

MDEForge-Search incorporates model management operations (MMO) as services. Hence as shown in Figure 34, We have deployed these MMO engines to facilitate the reuse of artifacts persisted in the repository [127]. 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. Figure 48).

Advanced management services

Choose a service...

Source model | Input type | Enter selected field..

Source metamodel | Input type | Enter selected field..

Target metamodel | Input type | Enter selected field..

Script | Input type | Enter selected field..

Console

Figure 48: Model management services available on the platform

6.4.4 Integrated services in discovery mechanisms

In this subsection, we present two integrated services we have added atop of the core services of the platform and that have been successfully applied to include quality attributes (c.f. Section 6.4.4.1) and megamodels relationships (c.f. Section 6.4.4.2) in the proposed query mechanism.

6.4.4.1 Quality assessment service Software Quality Engineering [182] 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 by Basciani et al. in [183]. 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 modelling artifacts enabling automated quality assessment. In our work, a quality assessment service for each supported artefact is provided that takes in a modelling 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.

6.4.4.2 Transformation chain service There are a number of sub-tasks that can be engaged with the model transformation composition. The first task is to identify the possible model transformation chains (available in the repository/folder) between the source and the target model. Leading to the identification of possible chains, the best transformation chain is estimated based on multiple criteria such as transformation coverage, information loss, and the number of transformation hops [184]. Also, one of the main objectives of the model transformation composition service is to optimize the execution of the best-selected chain [185]. This work is done by rewriting the transformation query and estimating the dependency between the meta-class and the structural features of the involved meta-model in the model transformation followed by chaining them up.

6.5 Integration of MDEForge with the Droid recommender framework

This section shows the integration of MDEForge into the Droid [176] framework. Droid is a textual DSL that automates the configuration, evaluation and synthesis of RS for particular modelling languages.
RSs are information filtering systems that guides users in selecting items among a potentially large set [186]. 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.
To create a Droid project, the system requires a set of information to be defined. First, the RS developer needs to specify the recommender's name and the technology that the RS will serve. It has support to create RSs for meta-modelling (e.g. Ecore) and modelling (e.g. UML, XMI). Furthermore, the RS developer can also select the default recommendation setting option to create a pre-filled template with a set of recommendation settings. Additionally, the data for training and testing the RSs needs to be provided. 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. With this, all the RSs specified in the recommendation methods can be trained, evaluated and compared simultaneously to be thereupon deployed on a REST service.
Droid allows via extension point to extend the data collection sources. For this integration, we extended Droid with the advanced query mechanisms exposed here. Figure 49 shows a scheme of this integration. The RS developer can create a Droid project *1* providing the name and the technology for the new RS. For the integration, the data collection sources *2* were extended to query the MDEForge API. The RS developer queries the Rest API *3* via a JSON POST request that sends in return a list of artifacts. With this information, a Droid project is created and the DSL *4* with the basic configuration is provided. It includes a textual editor with code compilation, a validator and a code generator that synthesizes Java code *5* from the project specification. Ultimately, Droid can train, test and compare the RSs defined *6*.
Figure 50 shows an example of data collection using the advanced query mechanism engine of MDEForge. In the example, the RS developer is creating an RS for Ecore meta-modelling. 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. By pressing the search button, a JSON request is sent to the MDEForge API. The returning lists of artifacts are shown in the table viewer. The table presents each artefact's name, extension, and size and the 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 RS developer can finish the wizard, and a Droid project is created with the collected data.

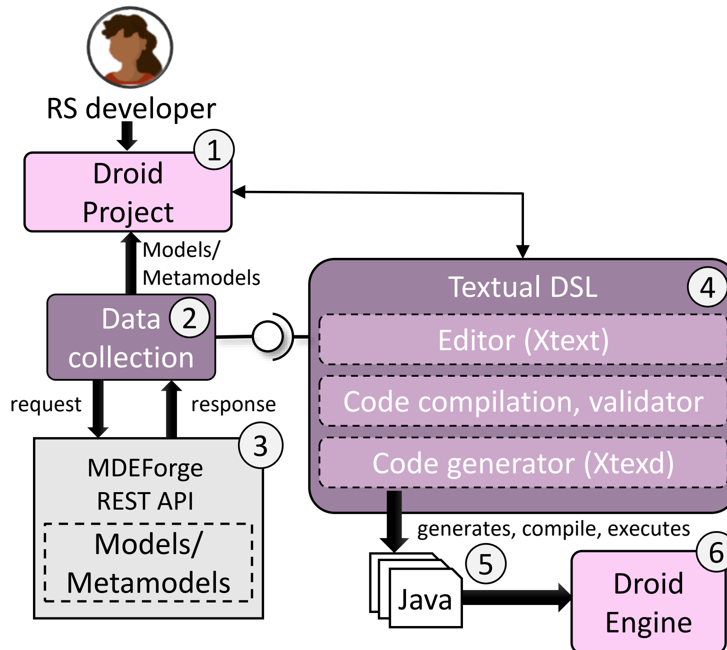

Figure 49: Architecture of Droid with MDEForge

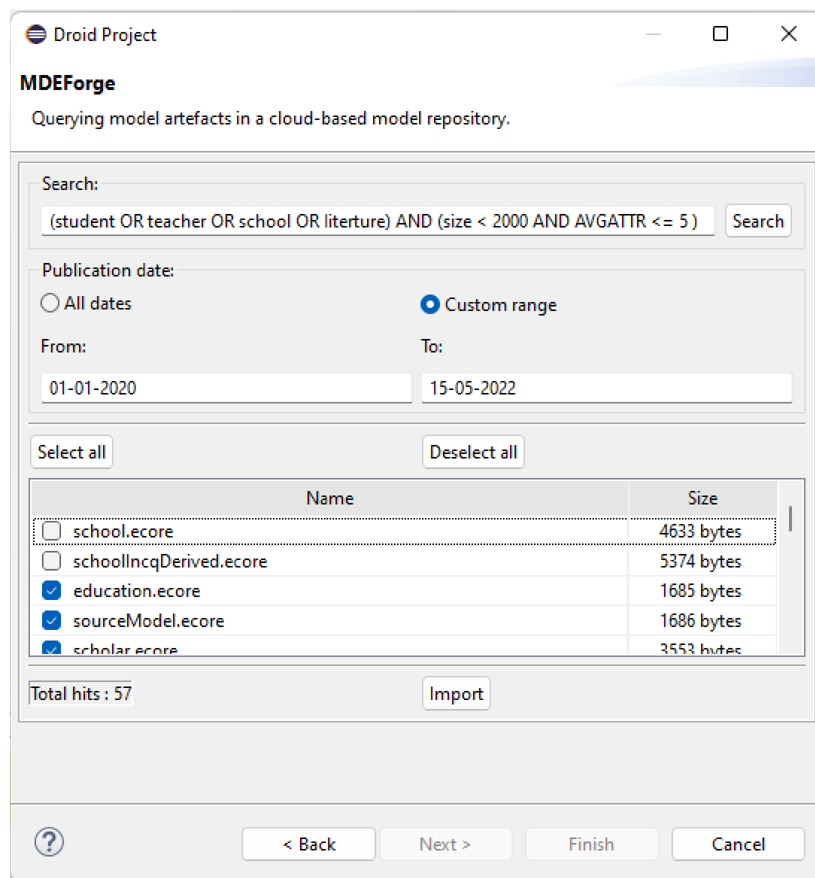

Figure 50: Droid wizard page for the advanced query mechanism.

6.6 Conclusion

This chapter 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. 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 is already using the platform API to train and evaluate machine learning models.

7 Model slicing on low-code engineering platforms

This chapter presents an approach for model reuse through model slicing on Low-code platforms (LCP). To get the information on heterogeneous models, our approach converts all the heterogeneous models to a homogeneous graph which will serve as a knowledge graph (KG). And to cope with the different levels of models, we created two repositories, one for data models (DMs) and another for the Form models. The two repositories persist the KG for the DM and the Forms, respectively. The model slicing approach gets a DM class as input and queries both repositories to get related entities to it. The required entities within the DM repository, like base class, composition, etc., will constitute the horizontal slice since they belong to the same model level (i.e., DM) as the input class. And the related entities from the Form repository will constitute the vertical slice since they belong to a different level than the input class. Finally, both slices will be merged as a single model slice and provided to the developer[57] in a JSON format.
As a proof of concept, we have tested our approach on the zAppDev[58] LCP by using 5 different zAppDev DM (represented in XML) with a total of 27 different domain classes, and 47 different Form models related to these domain models. The evaluation revealed that 77.78% of the given DM classes had any kind of cross-model relation i.e. we could extract successfully 21 distinct cross-language and cross-level model slices from these zAppDev models. The approach has been developed on Spring boot and is provided as a REST API.

7.1 Related Work on Model Slicing

Although to the best of our knowledge this is the first approach towards model ruse through slicing on cross-level and cross-language LCP models, inspired by program slicing approaches [187, 188], we will show some related work to model slicing.
Salay et al. [189] present an algorithm for megamodels slicing. The algorithm gets as input the megamodel and by using the traceability relation among the entities of the models that construct the megamodel it extracts the model slice. Our approach is search-based and not static based, i.e., it doesn't iterate through the cross-level model entities of a megamodel, it queries different repositories to find relevant matches to the input class.
Taenzer et al. [190] present a formal framework for creating model slicers capable of incrementally changing a model slice after performing any change on it. Our approach is search-based and generates model slices from a single input class to support the LCP users during the modeling process. Furthermore, we are not required to implement the model slicing update since it can be updated directly by the LCP users based on their needs after being integrated.
Compared to the approaches that enable model slicing for a specific model type [191, 192, 193, 194, 195, 196] our approach checks for related entities among different models of different types and extract them as a model slice.
In [192] Sagar et al. present a UML metamodel pruning algorithm. The algorithm takes as input a large metamodel and some required properties and classes that shall not be removed; it prunes the large input metamodel by removing its unnecessary entities based on a given criterion and thus extracts a smaller running metamodel that will serve as a smaller instance of the large metamodel. This approach aims to compute the smallest functional version of a large UML model, while our approach tends to get cross-model related entities.
Bergmayr et al. [193] presents a metamodel shrinking approach that uses refactoring techniques to detect noised metamodel classifiers and features and then package the remaining ones as a type-safe metamodel itself. While this approach tends to shrink metamodel elements, it differs from our approach, which tends to get related entities among different models.
In [194], Kagdi et al. present a UML slicing concept that explains how to compute model slices from a given UML class model. The concept explains that a model slice can be computed by starting from a given class and extracting related classes to it based on a set of given predicates. This approach is different from ours since it extracts UML model slices, and our approach extracts model slices from different and cross-level models.
Kelsen et al. [195] explain how a model based on lattice theory can be decomposed into several smaller models that are easier to understand. It splits a model into several smaller models considered model slices, whereas our approach extracts model slices from different models.
Blouin et al. [196] presents Kompren, a model slicer modeling language. The concepts and relations of this language are defined in an internal metamodel which will be instantiated by an internal model slicer model based on the input metamodel information. A mode slicer function will be generated when compiling the model slicer. Although, it differs from our approach in that our approach extracts related cross-model entities.
Burgueno et al. [197] use model slicing to provide recommendations during the modeling process. Their approach will slice only the relevant entities for the recommendations to avoid computing the whole model.
Our work is also inspired by cross-language program slicing approaches [187], where Nguyen et al. present a web-slicing technique for dynamic Web applications. It extracts code slices from different coding languages at hand by tracing the data flow through the program's execution. On the other hand, it checks if any function of any coding language affects this data.
Weiser in [188] presents an algorithm for program slicing based on criteria defined by a data-flow analysis on specific variables expressed on specific statements.
To clarify the concepts used throughout this chapter, we will initially provide some background information.

[57]In this chapter, we use the terms developer interchangeably for citizen developer
[58]https://zappdev.io/

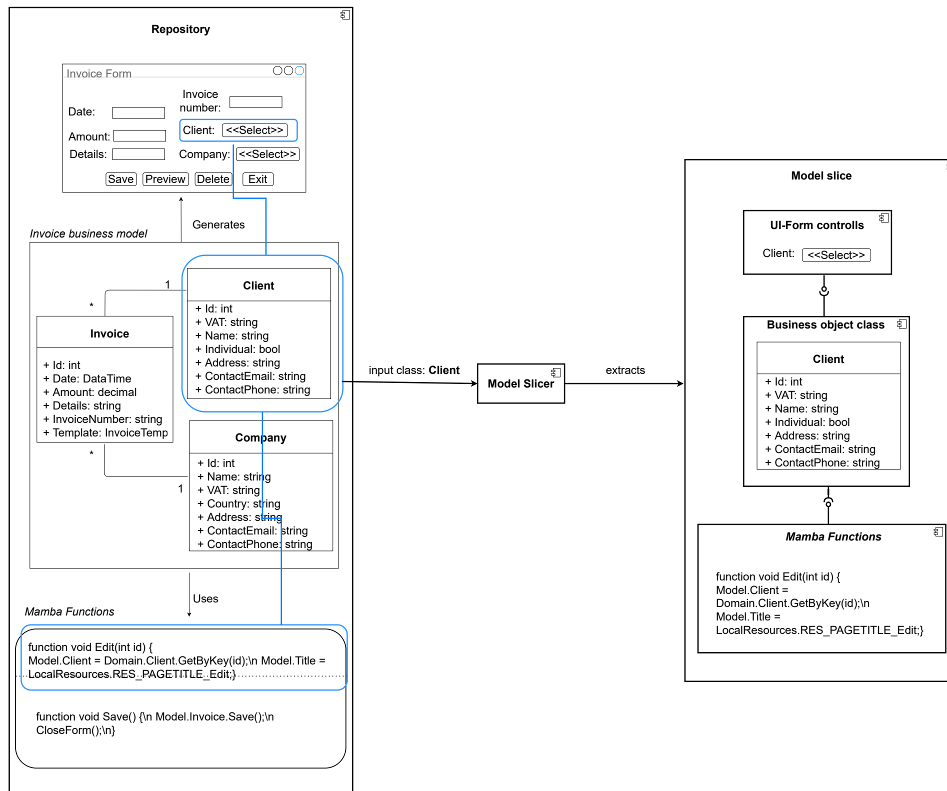

Figure 52: Model reuse through model slicing - running example

7.2 Background Information about the zAppDev LCDP

zAppDev is a web-based, model-driven development environment, allowing developers of any technology and proficiency level to easily create, edit and reuse models of software artifacts (e.g. database models, business logic models, user interface models, and more), covering the complete application development lifecycle while having total control of the process. As explained in [88], an LCDP typically consists of 4 different layers, which are included as well on zAppDev and are comprised of: *(1)* The Form models represent the application layer; *(2)* The service integration layer by API adapters; *(3)* The data integration layer is represented by data models, the service models, API Adapters; and finally *(4)* The Cloud represents the deployment layer.

The layers of interest for us are the application and data integration layers and their corresponding models since these are the only models the developers have direct contact with. Concretely, in this work, we will work with zAppDev data models a.k.a **business object models (BOs)**, and Form models a.k.a **UI-Form models**. The BO and UI-Form models belong to the zAppDev L1 level of the 3-level meta-modeling architecture as depicted in Fig. 51.

Both, the BOs and the UI-Form models are instances of the Business Object metamodel and Web Form UI metamodel respectively (L2 level). Whereas both metamodels conform to the zAppDev metalanguage written in the C# programming language (L3 level). The metamodels and the metalanguage are embedded on zAppDev and the developers have no access to them, they start the work directly by creating BO models as instances of the BO metamodel and auto-generate the UI-Form models from the BO models or they can initially design the UI-Form model and connect it afterwards to the relevant BO model. Lastly, the developers define the business logic of any UI component within the UI-Form model by using the Mamba language.

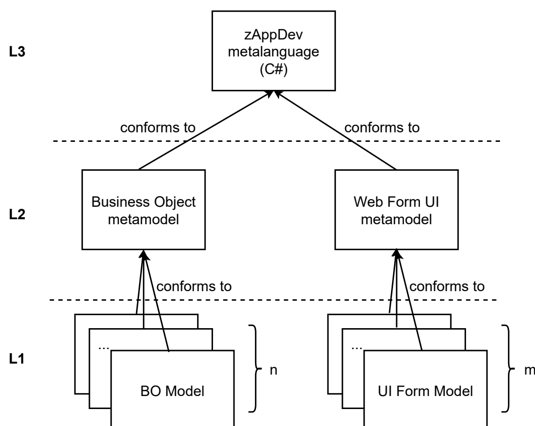

Figure 51: Metamodeling layers on the zAppDev LCDP

By explaining a running example we clarify how this approach can extract model slices from LCDP models.

7.2.1 Running example

Assume that in an LCDP there is an `Invoice` software containing a BO, a UI-Form model named "Invoice Form" auto-generated from the BO classes or manually constructed by the developer, and the business logic functions related to

the BO classes. The architecture of the `Invoice` software is depicted in Fig. 52. In the zAppDev LCDP, the BOs are presented in XML format, the UI-Forms as JSON files and the DSL functions are written in the Mamba[59] language. As we can see in Fig. 52 on the left part, the BO is constructed from the classes `Invoice`, `Client`, and `Company`. We aim to get only the related cross-level and cross-language artifacts to the `Client` BO class. As depicted in Fig. 52, the `Client` BO class is related to the Invoice Form with a label with the text `Client` on it and a combo box. Further, we can see that the `Client` class has an Edit function related to it. To emphasize the connection of all the `Client` related cross-level and cross-language entities we rounded and connected them with a blue cycle and blue lines.
The idea of a model slicer approach on an LCDP, as shown in Fig. 52, would be to provide only the `Client` BO class as input to the approach. It would be capable of computing and extracting all the cross-related entities to the `Client` BO class and integrating those on an LCDP. As shown on the right side of Fig. 52, the extracted model slice from the approach is a model that contains only the `Client` relevant entities across the LCDP models.
The model slicer finds connected entities to the BO input classes on cross-level and cross-language models. Then, it presents these as a model slice to the developers so everything connected within the entire production line on any LCDP can be reused and integrated automatically on an LCDP. Inspired by this running example, we have created a model-slicing approach that is explained in more detail in Section 7.3.3.
These below Sections will be presenting in more detail how the model slicer extracts model slices from cross-level models. The overview of the model slicer is presented in Fig. 53.

7.3 Repositories

The first step toward model slicing is persisting the models in a repository so they can be reused for different business needs afterwards. Since all models in zAppDev are graph-based, the repository of our approach is also graph-based. We selected the Resource Description Framework (RDF) [198] as our model format since it is a graph-based model and the standard format of W3C[60], this is relevant to LCPs which are cloud-based. Thus, various models will be converted and merged into a single RDF graph. We will use and refer to this RDF graph as our approach's knowledge graph (KG).
Since developing any software on an LCP ones needs to create its' DM, its' UI in a Form model, and the business logic which is persisted in any of these two (especially in zAppDev), we have created two different repositories for the model slicer, one which persists the knowledge graph for the DM, and another for the Form models. Thus, as explained in Fig. 53, *step 1* of our approach creates the repositories that persist the knowledge graph for the DM and Form models by converting them to RDF and merging them to their respective knowledge graphs.

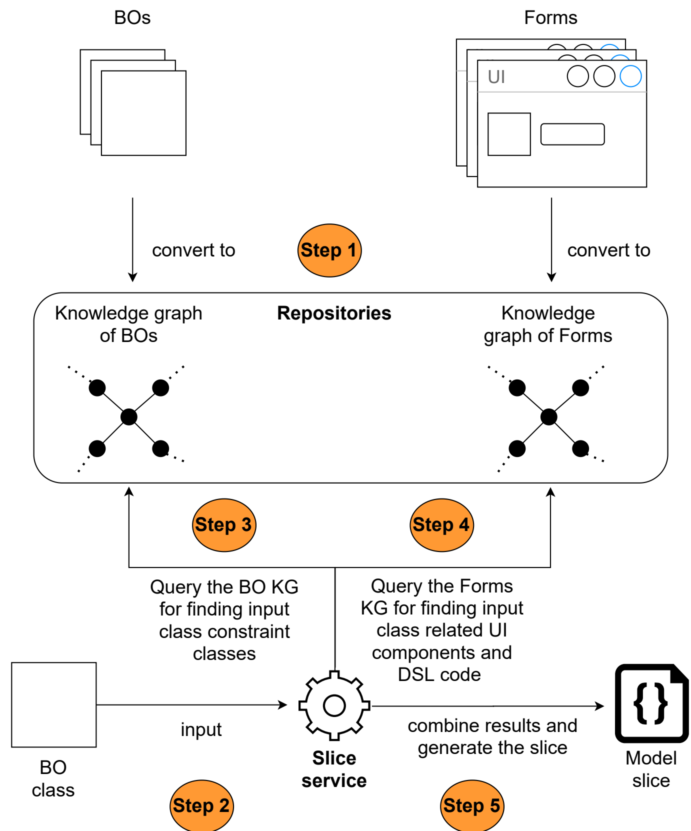

Figure 53: Model slicing approach overview

[59]https://docs.zappdev.com/MambaLanguage/About/
[60]https://www.w3.org/TR/2004/REC-rdf-concepts-20040210

7.3.1 Input Class

Step 2 of our approach is getting the input class. The input class is the core entity of any model slice because any other entity of the model slice has to be related in a specific form to it. Hence, the input class defines the slicing criterion for our approach. After having the input class, the slice service - responsible for the business logic part of the model slicer - will query the repositories to find relevant connections between the existing entities within the repositories and the input class.

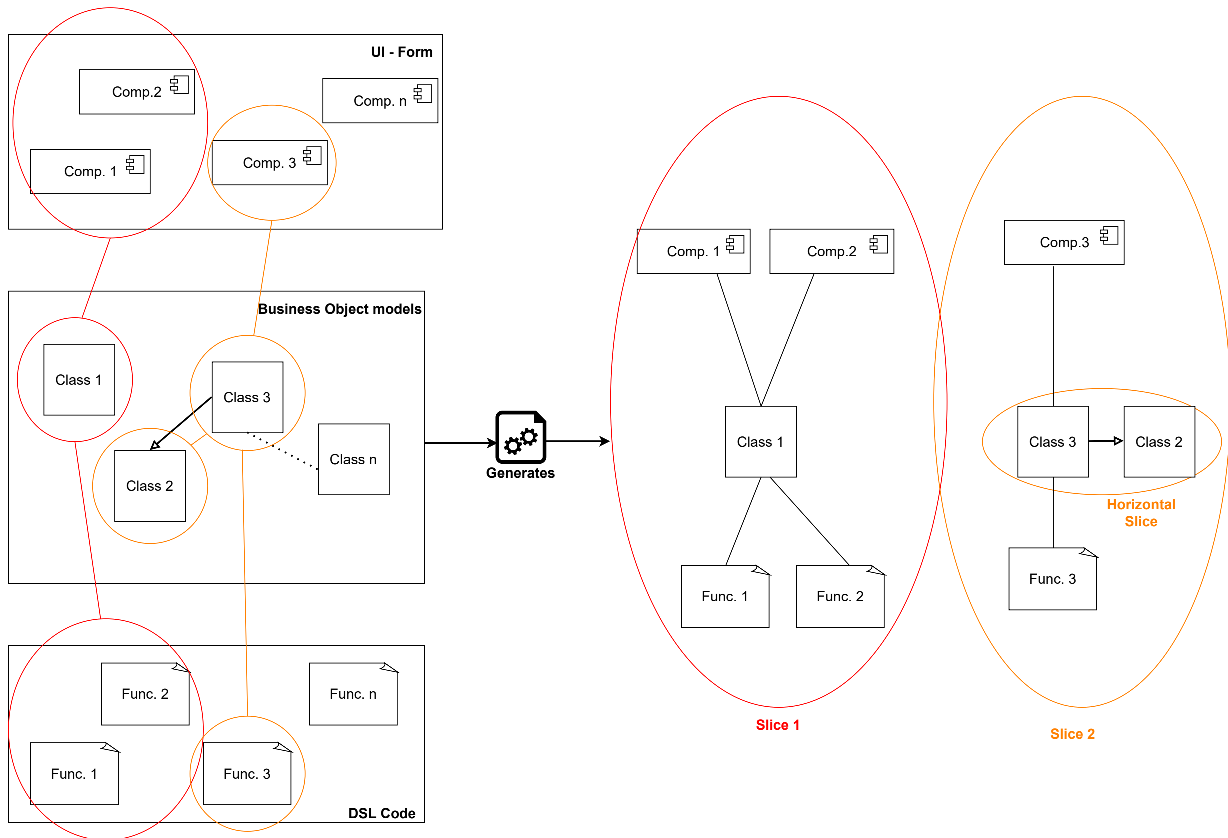

Figure 54: Model reuse through model slicing

7.3.2 Horizontal Slice

The first check the input class will go through is if it has any **constraints class** that has to be integrated with the input class. For instance, if the input class inherits a class within the BO classes or has a composition class, then the base/composition class has to be integrated into the LCP to avoid model validation errors. Thus in *step 3*, our approach checks in the DT (BO) repository if there is any such constraints class related to the input class. If any such class can be found, it will be provided to the developers with the input class. Since the input and constraint classes are on the same level (within the business object), we call this kind of relation horizontal slicing. However, since we are slicing only zAppDev models so far, the horizontal slice is defined only by checking if the input class has any base class within the BO repository.

7.3.3 Vertical Slice

Next, the model slicer service will check if any related entity is within the Form KG to the input class. In step 4, the model slicer will return all the relevant information about the related entities to the input class. In our case, we query the information about UI components, i.e., the UI component name, the UI component data source - which shows to which specific attribute of the input class the UI component is related, and the type of the UI component, e.g. TextBoxControll, CheckBox, etc. Although a lot of other relevant information can be retrieved, e.g. the cascade style sheet (CSS) information about the UI components, UI components layout information, etc. Since in zAppDev, the DSL functions, a.k.a. Mamba functions, are persisted within the Form models, in step 4, the model slicer also queries related Mamba functions to the input class. Hence the UI components and DSL functions are not in the same model level as the input class; we call this relation of connected cross-level models a vertical slice.
Finally, the extracted horizontal and vertical slices will be merged as a single model slice and integrated into the LCP. An example of model slicing is presented in Fig. 54. We see that the meta-class `Class 1` is connected to the UI components `Comp. 1` and `Comp. 2` and also to the DSL functions *Func. 1* and *Func. 2*; thus, the connection of all these entities would give a single slice (Slice 1). Further in Fig. 54, we can see that the meta-class `Class 3` has a constraint

Model slices on zAppDev models				
BO models	BO classes	UI-Form models	Horizontal Slices	Vertical Slices
1 CoreBO	10		0	9
2 DTOs	2		0	2
3 Expenses	5	47	0	4
4 ProjectBO	3		0	3
5 TaskBO	17		1	8
Total	48 (27 Distinct)	47	1	26 (21 Distinct)

Table 13: Model slices on zAppDev models

class `Class 2` within the BO, it is also related to the component `Comp 3` on the UI - Form model, and it has also a related DSL function *Func. 3*. The connection of these related entities to `Class 2` would produce another modeling slice (Slice 2).

7.3.4 The model slicing approach at work

This section demonstrates how the model slicer extracts model slices on real LCDP models. As a proof of concept,

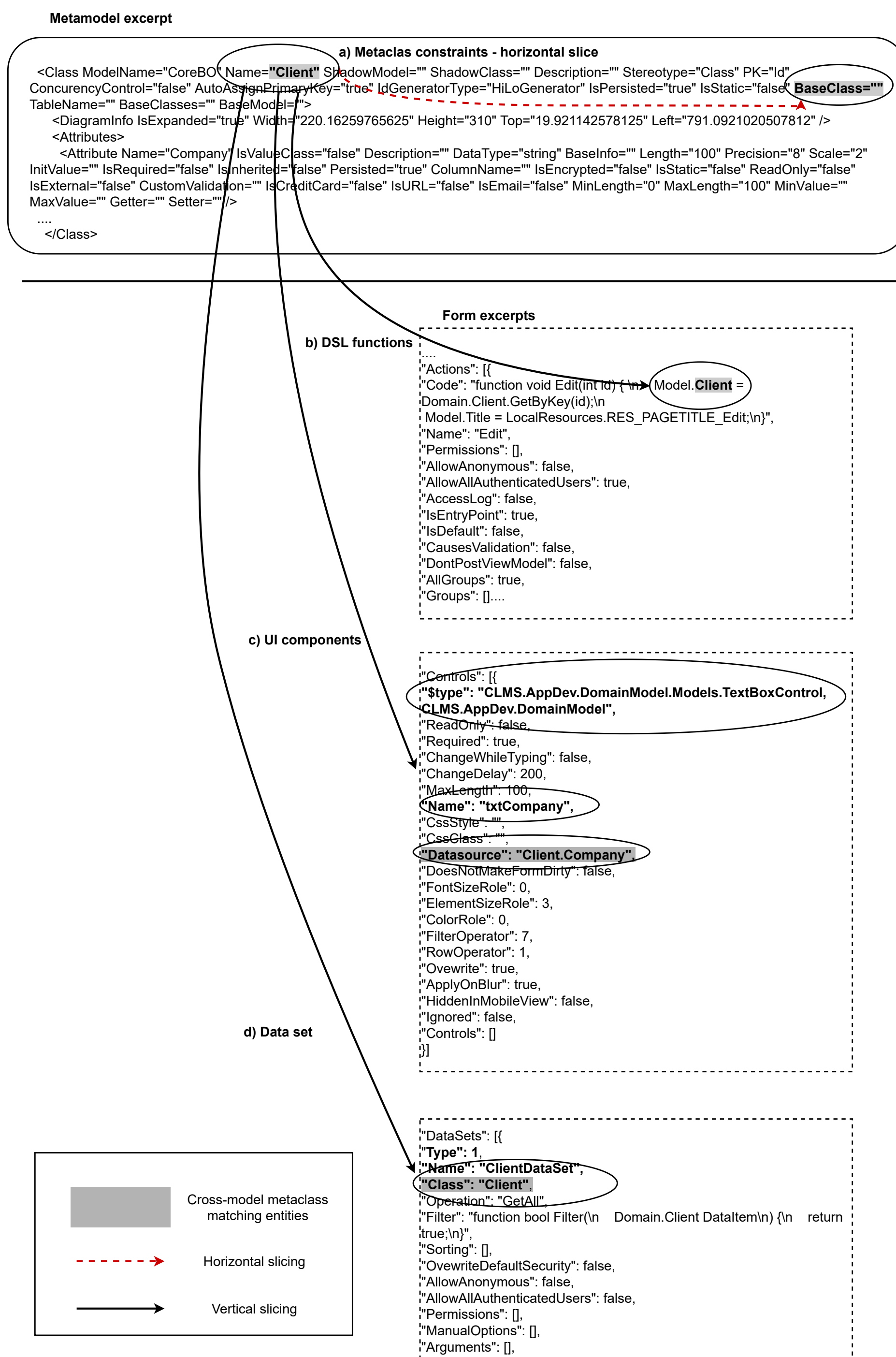

Figure 55: Model slicing in zAppDev

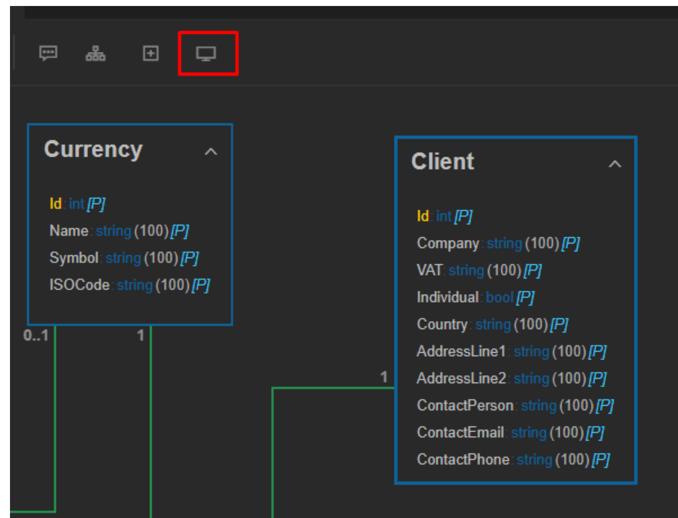

Figure 56: Triggering the model slicer service for the Client class

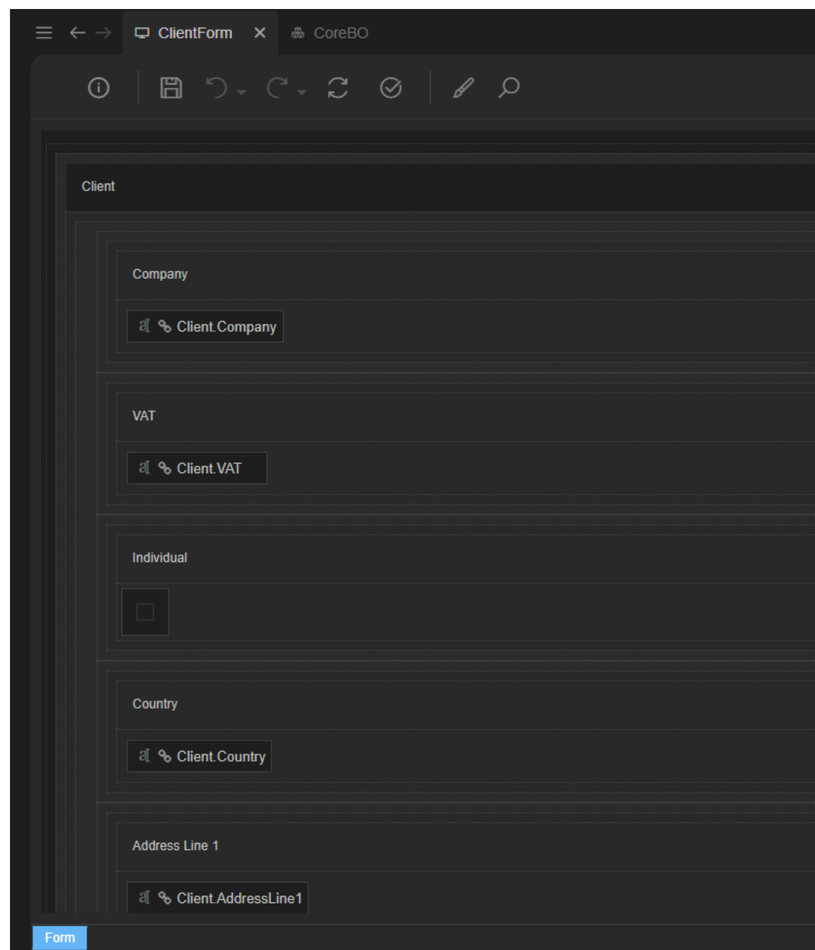

Figure 57: Integration of the Client model slice in zAppDev

we got five different BOs and 47 different UI-Form models generated by these business objects. We have created two knowledge graphs containing all the information about the BOs and the Form models, respectively. Then we selected each class iteratively from the BOs, gave this as an input class to the model slicer, and checked the extracted model slices. For example, in Fig. 55, we have presented the information that the model slicer will extract. The model slicer will try to extract a model slice related to the `Client` class.

Initially, the model slicer will check within the BOs at the *baseClass* element to find any base class so it can create the horizontal slice (a)). This check is presented with the red dashed arrow. In this demo, the `Client` class hasn't any base class and since, at the time of writing this deliverable, this is the only checked constraint for BO classes, the model slicer will not define any horizontal slice. Next, the model slicer will check for extracting the relevant Mamba functions from the zAppDev Form models (b). The model slicer will check within the *Code* notation to find any related function to the `Client` class. The found functions will be returned as part of the vertical slice.

Next in (c), the model slicer will extract the required information about the relevant UI components to the `Client`

class. First, the model slicer will check if there is any *Datasource* notation that is related to the `Client` class, if there is any, then it gets the information about that Datasource related *Name* and *$type* notation. These three notations: *Name, Datasource* and *$type* will be returned also as part of the vertical slice. Finally, the model slicer checks for any related *dataset* to the `Client` class (d)). It checks the notation *Class* within the *Dataset* notation if it is Client. If there is a match, then will the model slicer get the respective *Name, Operation,* and *Filter* information. All this information will also be returned as part of the vertical slice.
In the end, all the information about the horizontal and vertical slices will be merged in a single JSON file and integrated into the zAppDev LCDP. This model slice is extracted after selecting the `Client` BO class and clicking the "Create forms from Slice" button located in the menu bar as shown in Fig. 56. The model slice is integrated on the zAppDev LCDP as a UI-Form model including the extracted information for the `Client` BO class. A snapshot of the integrated model slice on zAppDev is depicted in Fig. 57.
In Table 13 we have outlined the results of how many model slices could be extracted from the zAppDev models. For the study, we have used 5 different business object models and 47 different UI-Form models. We listed all the BO classes from all the 5 BO models and counted 27 distinct classes. We iterated through each of these BO classes and set each of them successively as an input class to the model slicer. Of all these BO classes only one class had a base class (PMOUser had as base class ApplicationUser) i.e., a horizontal slice. Also from the set of 27 distinct BO classes, the model slicer could extract 21 vertical slices which means that 21 BO classes have at least one related entity on the UI-Form models. From this amount of data we got from zAppDev, we could conclude that **77.78%** of the BO classes are related at least to a base class or at least to one UI-Form model entity. This fact reveals the emerging need for a cross-language and cross-level model reuse approach on LCDP that can be facilitated through the model slicer provided in this work.
The model slicer has been developed using Spring boot and is provided as a REST API for use in the zAppDev LCDP. The source and the repositories containing the KG used for the evaluation are available on GitHub[61]

7.4 Conclusion

In this chapter, we have presented an approach that enables the reuse of cross-related models on an LCP through model slicing. The model slicing approach gets as input a data model class and queries the data model for any constraint-related class and the Form model repositories to get UI components and DSL functions. The current approach enables model slicing of zAppDev models, but conceptually it can be used for any LCP. We plan to fine-grain the model slicer by providing UI component layout information, slicing the class attributes, etc. We also aim to integrate the model slicer on a model recommendation approach so that all its cross-related model entities will be integrated automatically after selecting a suggested data model class.

[61] https://github.com/iliriani/Model-slicer

8 Conclusion

This document presents the work done to conceive and develop a scalable and extensible cloud-based low-code model repository. First, we discussed the contextual background of Low-code development platforms. Then, we compared the prominent LCDPs along with their taxonomy. We also shared the challenges and user experience of these platforms as they seek mainstream adoption as new software development platforms. Then, we discussed the open challenges and opportunities of migrating modeling platforms to the cloud. More than 600 papers were analyzed, and we shared our insights and the built-in benefits of modeling on the cloud. The conceived Cloud-based Low-Code Model Repository system architecture was presented. We also presented its detailed system views along with the related work done by the research community in MDE. Subsequently, we discussed the milestones achieved toward a scalable and extensible Low-Code Model Repository. The repository has been extended from a storage facility to a hub of modeling tools and services that remove the need for local installations and configurations. The artifacts are stored in scalable facilities to ensure their integrity and discovery. Utilities that enable service orchestration and composition were developed to allow the execution of model management workflows. The last chapter presented the discovery and reuse capabilities developed atop the repository. The repository houses advanced discovery mechanisms that ensure the discovery and reuse of the artifacts on a single platform. We also conceived an approach that slices model artifacts toward their reuse.

References

[1] C. Richardson and J. R. Rymer. The forrester wave: Low-code development platforms, q2 2016. tech. rep. *Forrester Research*, 2016.

[2] John R. Rymer. The forrester wave: Low-code platforms for business developers, q2 2019. *Forrester Research*, 2019.

[3] Paul Vincent, Kimihiko Iijima, Mark Driver, Jason Wong, and Yefim Natis. Magic quadrant for enterprise low-code application platforms. *Gartner report*, 2019.

[4] Ieee recommended practice for architectural description for software-intensive systems. *IEEE Std 1471-2000*, pages 1–30, 2000.

[5] Robert Waszkowski. Low-code platform for automating business processes in manufacturing. *IFAC-PapersOnLine*, 52(10):376–381, 2019.

[6] An introduction to low-code platform. https://www.mendix.com/low-code-guide/. Accessed: 2020-03-23.

[7] Francesco Basciani, Juri Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio. Mdeforge: An extensible web-based modeling platform. volume 1242, 09 2014.

[8] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*, volume 1. 09 2012.

[9] Mendix platform features. https://www.mendix.com/platform/. Accessed: 2020-03-23.

[10] Outsystem platform features. https://www.outsystems.com/platform/. Accessed: 2020-03-23.

[11] Zoho creator platform features. https://www.zoho.com/creator/features.html. Accessed: 2020-03-23.

[12] Microsoft powerapps platform overview. https://docs.microsoft.com/en-us/powerapps/maker/. Accessed: 2020-03-23.

[13] Google app maker platform guide. https://developers.google.com/appmaker/overview. Accessed: 2020-03-23.

[14] Kissflow platform overview. https://kissflow.com/process-management/. Accessed: 2020-03-23.

[15] Salesforce app cloud platform overview. https://developer.salesforce.com/platform. Accessed: 2020-03-23.

[16] Appian platform overview. https://www.appian.com/. Accessed: 2020-03-23.

[17] Krzysztof Czarnecki. *Generative programming - principles and techniques of software engineering based on automated configuration and fragment-based component models*. PhD thesis, Technische Universität Illmenau, Germany, 1999.

[18] Krzysztof Czarnecki. *Domain Engineering*, pages 433–444. American Cancer Society, 2002.

[19] Justice Opara-Martins, R. Sahandi, and Feng Tian. Implications of integration and interoperability for enterprise cloud-based applications. pages 213–223, 10 2015.

[20] Amandeep Singh, Pardeep Mittal, and Neetu Jha. Foss: A challenge to proprietary software. *IJCST*, 4, 2013.

[21] Juri Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Collaborative repositories in model-driven engineering [software technology]. *IEEE Software*, 32:28–34, 05 2015.

[22] Best low-code development platforms software. https://www.g2.com/categories/low-code-development-platforms. Accessed: 2020-05-26.

[23] Aymen J Salman, Mohammed Al-Jawad, and Wisam Al Tameemi. Domain-Specific Languages for IoT: Challenges and Opportunities. *IOP Conference Series: Materials Science and Engineering*, 1067(1):012133, 2021.

[24] A. Bucchiarone, F. Ciccozzi, L. Lambers, A. Pierantonio, M. Tichy, M. Tisi, A. Wortmann, and V. Zaytsev. What is the future of modeling? *IEEE Software*, 38(02):119–127, mar 2021.

[25] Mohsen Marjani, Fariza Nasaruddin, Abdullah Gani, Ahmad Karim, Ibrahim Abaker Targio Hashem, Aisha Siddiqa, and Ibrar Yaqoob. Big IoT Data Analytics: Architecture, Opportunities, and Open Research Challenges. *IEEE Access*, 5:5247–5261, 2017.

[26] Laith Farhan, Sinan T. Shukur, Ali E. Alissa, Mohmad Alrweg, Umar Raza, and Rupak Kharel. 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, 2017.

[27] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. 2012.

[28] Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. A Low-Code Development Environment to Orchestrate Model Management Services. *Advances in Production Management Systems*, 2021.

[29] Luca Berardinelli, Alexandra Mazak, Oliver Alt, and Manuel Wimmer. *Model-Driven Systems Engineering: Principles and Application in the CPPS Domain*, pages 261–299. 05 2017.

[30] Davide Di Ruscio, Mirco Franzago, Ivano Malavolta, and Henry Muccini. Envisioning the future of collaborative model-driven software engineering. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*, (May):219–221, 2017.

[31] Shanzhi Chen, Hui Xu, Dake Liu, Bo Hu, and Hucheng Wang. A vision of IoT: Applications, challenges, and opportunities with China Perspective. *IEEE Internet of Things Journal*, 1(4):349–359, 2014.

[32] Ábel Hegedüs, Gábor Bergmann, Csaba Debreceni, Ákos Horváth, Péter Lunk, Ákos Menyhért, István Papp, Dániel Varró, Tomas Vileiniskis, and István Ráth. 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, 2018.

[33] Jorge Biolchini, Paula Gomes Mian, Ana Candida Cruz Natali, and Guilherme Horta Travassos. Systematic review in software engineering. *System Engineering and Computer Science Department COPPE/UFRJ, Technical Report ES*, 679(05):45, 2005.

[34] A. Salihbegovic, T. Eterovic, E. Kaljic, and S. Ribic. 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, 2015.

[35] Fabrizio F. Borelli, Gabriela O. Biondi, and Carlos A. Kamienski. Biota: A buildout iot application language. *IEEE Access*, 8:126443–126459, 2020.

[36] Node-RED. Node-red: Low-code programming for event-driven applications. https://nodered.org/, 2020. Last accessed May 2020.

[37] Thiago Nepomuceno, Tiago Carneiro, Tiago Carneiro, Clemens Korn, and Alexander Martin. 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, 2018.

[38] Thiago Nepomuceno, Tiago Carneiro, Paulo Henrique Maia, Muhammad Adnan, Thalyson Nepomuceno, and Alexander Martin. 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, 2020. Association for Computing Machinery.

[39] Jussi Kiljander, Janne Takalo-Mattila, Matti Etelapera, Juha-Pekka Soininen, and Kari Keinanen. Enabling end-users to configure smart environments. In *2011 IEEE/IPSJ International Symposium on Applications and the Internet*, pages 303–308, 2011.

[40] AtmosphereIoT. Fast time to first data. https://atmosphereiot.com/, 2020. Last accessed May 2020.

[41] Jagni Dasa Horta Bezerra and Cidcley Teixeira de Souza. 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, 2019. Association for Computing Machinery.

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

[43] Flavio Corradini, Arianna Fedeli, Fabrizio Fornari, Andrea Polini, and Barbara Re. FloWare: An Approach for IoT Support and Application Development. In Adriano Augusto, Asif Gill, Selmin Nurcan, Iris Reinhartz-Berger, Rainer Schmidt, and Jelena Zdravkovic, editors, *Enterprise, Business-Process and Information Systems Modeling*, pages 350–365, Cham, 2021. Springer International Publishing.

[44] Guillermo Cueva-Fernandez, Jordán Pascual Espada, Vicente García-Díaz, Cristian González García, and Nestor Garcia-Fernandez. Vitruvius: An expert system for vehicle sensor tracking and managing application generation. *Journal of Network and Computer Applications*, 42:178–188, 2014.

[45] Cristian González García, B. Cristina Pelayo G-Bustelo, Jordán Pascual Espada, and Guillermo Cueva-Fernandez. Midgar: Generation of heterogeneous objects interconnecting applications. a domain specific language proposal for internet of things scenarios. *Computer Networks*, 64:143–158, 2014.

[46] Wajid Rafique, Xuan Zhao, Shui Yu, Ibrar Yaqoob, Muhammad Imran, and Wanchun Dou. An application development framework for internet-of-things service orchestration. *IEEE Internet of Things Journal*, 7(5):4543–4556, 2020.

[47] Manfred Sneps-Sneppe and Dmitry Namiot. 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, 2015.

[48] Robert Kleinfeld, Stephan Steglich, Lukasz Radziwonowicz, and Charalampos Doukas. 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, 2014. Association for Computing Machinery.

[49] Amir Taherkordi and Frank Eliassen. 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, 2016.

[50] Yannis Valsamakis and Anthony Savidis. Personal Applications in the Internet of Things Through Visual End-User Programming. In Claudia Linnhoff-Popien, Ralf Schneider, and Michael Zaddach, editors, *Digital Marketplaces Unleashed*, pages 809–821, Berlin, Heidelberg, 2018. Springer Berlin Heidelberg.

[51] Yi Xu and Abdelsalam Helal. Scalable cloud–sensor architecture for the internet of things. *IEEE Internet of Things Journal*, 3(3):285–298, 2016.

[52] Simon Mayer, Ruben Verborgh, Matthias Kovatsch, and Friedemann Mattern. Smart configuration of smart environments. *IEEE Transactions on Automation Science and Engineering*, 13(3):1247–1255, 2016.

[53] Badr El Khalyly, Mouad Banane, Allae Erraissi, and Abdessamad Belangour. Interoevery: Microservice based interoperable system. In *2020 International Conference on Decision Aid Sciences and Application (DASA)*, pages 320–325, 2020.

[54] Behailu Negash, Tomi Westerlund, Amir M Rahmani, Pasi Liljeberg, and Hannu Tenhunen. DoS-IL: A Domain Specific Internet of Things Language for Resource Constrained Devices. *Procedia Computer Science*, 109:416–423, 2017.

[55] Behailu Negash, Tomi Westerlund, Pasi Liljeberg, and Hannu Tenhunen. Rethinking 'Things' - Fog Layer Interplay in IoT: A Mobile Code Approach. In *11th International Conference on Research and Practical Issues of Enterprise Information Systems (CONFENIS)*, volume LNBIP-310, pages 159–167, Shanghai, China, October 2017. Springer International Publishing.

[56] Fei Li, Michael Vögler, Markus Claeßens, and Schahram Dustdar. Towards automated iot application deploy-

ment by a cloud-based approach. In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, pages 61–68, 2013.

[57] Nicolas Ferry, Phu Nguyen, Hui Song, Pierre-Emmanuel Novac, Stéphane Lavirotte, Jean-Yves Tigli, and Arnor Solberg. 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, 2019.

[58] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. 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, 2016. Association for Computing Machinery.

[59] Mahda Noura, Mohammed Atiquzzaman, and Martin Gaedke. Interoperability in Internet of Things: Taxonomies and Open Challenges. *Mobile Networks and Applications*, 24(3):796–809, 2019.

[60] Mauro Conti, Ali Dehghantanha, Katrin Franke, and Steve Watson. Internet of Things security and forensics: Challenges and opportunities. *Future Generation Computer Systems*, 78:544–546, 2018.

[61] Eclipse Foundation. 2020 annual eclipse foundation community report, 2020. Last accessed July 2021.

[62] Melanie Bats and Stephane Begaudeau. Sirius web: 100Last accessed July 2021.

[63] André Restivo, Hugo Sereno Ferreira, João Pedro Dias, and Margarida Silva. Visually-defined real-time orchestration of iot systems. 2020.

[64] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C.M. Leung. Developing iot applications in the fog: A distributed dataflow approach. In *2015 5th International Conference on the Internet of Things (IOT)*, pages 155–162, 2015.

[65] João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. Empowering visual internet-of-things mashups with self-healing capabilities. *arXiv preprint arXiv:2103.07395*, 2021.

[66] Léa Brunschwig, Esther Guerra, and Juan de Lara. 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, 2020. Association for Computing Machinery.

[67] Yun Mi Antorini and Albert M Muñiz. The Benefits and Challenges of Collaborating with User Communities. *Research-Technology Management*, 56(3):21–28, 2013.

[68] Olaf David, Wes Lloyd, Ken Rojas, Mazdak Arabi, Frank Geter, James Ascough, Tim Green, G. Leavesley, and Jack Carlson. 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, 2014.

[69] Giancarlo Fortino, Claudio Savaglio, Giandomenico Spezzano, and MengChu Zhou. 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, 2021.

[70] Andreas Wortmann, Benoit Combemale, and Olivier Barais. A systematic mapping study on modeling for industry 4.0. In *Proceedings of the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*, MODELS '17, page 281–291. IEEE Press, 2017.

[71] Sergio Teixeira, Bruno Alves Agrizzi, José Gonçalves Pereira Filho, Silvana Rossetto, and Roquemar de Lima Baldam. 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, 2017.

[72] Felicien Ihirwe, Davide Di Ruscio, Silvia Mazzini, Pierluigi Pierini, and Alfonso Pierantonio. Low-code engineering for internet of things: a state of research. In Esther Guerra and Ludovico Iovino, 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, 2020.

[73] Partha Pratim Ray. A Survey on Visual Programming Languages in Internet of Things. *Scientific Programming*, 2017:1231430, 2017.

[74] Christian Prehofer and Ilias Gerostathopoulos. Chapter 3 - Modeling RESTful Web of Things Services: Concepts and Tools. In Quan Z Sheng, Yongrui Qin, Lina Yao, and Boualem Benatallah, editors, *Managing the Web of Things*, pages 73–104. Morgan Kaufmann, Boston, 2017.

[75] Petra Brosch, Philip Langer, Martina Seidl, and Manuel Wimmer. Towards end-user adaptable model versioning: The by-example operation recorder. In *Procs.of CVSM '09*, pages 55–60, Washington, DC, USA, 2009. IEEE Computer Society.

[76] R. Kutsche, N. Milanovic, G. Bauhoff, T. Baum, M. Cartsburg, D. Kumpe, and J. Widiker. BIZYCLE: Model-based Interoperability Platform for Software and Data Integration. In *Procs.of the MDTPI at ECMDA*, 2008.

[77] M. Koegel and J. Helming. Emfstore: a model repository for emf models. In *Software Engineering, 2010 ACM/IEEE 32nd Int. Conf. on*, volume 2, pages 307–308, May 2010.

[78] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing*, 2001.

[79] Christian Hein, Tom Ritter, and Michael Wagner. Model-driven tool integration with ModelBus. *Workshop Future Trends of Model-Driven \ldots*, 2009.

[80] Ta'id Holmes, Uwe Zdun, and Schahram Dustdar. Automating the Management and Versioning of Service Models at Runtime to Support Service Monitoring. In *EDOC*, pages 211–218, September 2012.

[81] Robert France, Jim Bieman, and BettyH.C. Cheng. Repository for model driven development (remodd). In *Models in Software Engineering*, volume 4364 of *LNCS*, pages 311–317. Springer Berlin Heidelberg, 2007.
[82] Ramtin Jabbari, Nauman bin Ali, Kai Petersen, and Binish Tanveer. 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, 2016. Association for Computing Machinery.
[83] Jokin Garcia and Jordi Cabot. Stepwise adoption of continuous delivery in model-driven engineering. In Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer, editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 19–32, Cham, 2019. Springer International Publishing.
[84] Alessandro Colantoni, Luca Berardinelli, and Manuel Wimmer. 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, 2020. Association for Computing Machinery.
[85] Lowcomote EU Project Consortium. D4.3. concepts for testing in low-code engineering repositories. https://www.lowcomote.eu.
[86] Philip Makedonski, Gusztáv Adamis, Martti Käärik, Finn Kristoffersen, Michele Carignani, Andreas Ulrich, and Jens Grabowski. Test descriptions with etsi tdl. *Software Quality Journal*, 27(2):885–917, 2019.
[87] Linz BISE Institute, JKU. Devopsml, 2020. https://github.com/lowcomote/devopsml/tree/1.2.2, last accessed on 28/08/20.
[88] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio. Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 171–178, 2020.
[89] Necco Ceresani. The periodic table of devops tools v.2 is here, June 2016. https://blog.xebialabs.com/2016/06/14/periodic-table-devops-tools-v-2/, last accessed on 28/08/20.
[90] Hugo Brunelière, Jordi Cabot, and Frédéric Jouault. Combining Model-Driven Engineering and Cloud Computing. In *MDA4ServiceCloud'10 Workshop co-located with ECMFA*, June 2010.
[91] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio. MDEForge: An extensible Web-based modeling platform. *CEUR Workshop Proceedings*, 1242(September):66–75, 2014.
[92] Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. 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, 2021. Springer International Publishing.
[93] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. Supporting the understanding and comparison of low-code development platforms.
[94] Supachai Vorapojpisut. A Lightweight Framework of Home Automation Systems Based on the IFTTT Model. *Journal of Software*, 10(12):1343–1350, 2015.
[95] Amir Rahmati, Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. IFTTT vs. Zapier: A Comparative Study of Trigger-Action Programming Frameworks. 2017.
[96] Shanchen Pang, Qian Gao, Ting Liu, Hua He, Guangquan Xu, and Kaitai Liang. A Behavior Based Trustworthy Service Composition Discovery Approach in Cloud Environment. *IEEE Access*, 7:56492–56503, 2019.
[97] Amin Jula, Elankovan Sundararajan, and Zalinda Othman. Cloud computing service composition: A systematic literature review. *Expert Systems with Applications*, 41(8):3809–3824, 2014.
[98] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Collaborative repositories in model-driven engineering. *IEEE Software*, 32(3):28–34, 2015.
[99] Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. Web service composition: A survey of techniques and tools. *ACM Computing Surveys*, 48(3), 2015.
[100] Pablo Rodriguez-Mier, Carlos Pedrinaci, Manuel Lama, and Manuel Mucientes. An integrated semantic web service discovery and composition framework. *IEEE Transactions on Services Computing*, 9(4):537–550, 2016.
[101] David Martin, Massimo Paolucci, Sheila McIlraith, Mark Burstein, Drew McDermott, Deborah McGuinness, Bijan Parsia, Terry Payne, Marta Sabou, Monika Solanki, Naveen Srinivasan, and Katia Sycara. Bringing semantics to web services: The owls approach. In Jorge Cardoso and Amit Sheth, editors, *Semantic Web Services and Web Process Composition*, pages 26–42, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
[102] Gennaro Cordasco, Matteo D'Auria, Alberto Negro, Vittorio Scarano, and Carmine Spagnuolo. Toward a domain-specific language for scientific workflow-based applications on multicloud system. *Concurrency Computation*, (February), 2020.
[103] G. Stürmer, J. Mangler, and E. Schikuta. 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, 2009.
[104] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.
[105] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.
[106] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.
[107] Mario Cortes-Cornax, Sophie Dupuy-Chessa, and Dominique Rieu. Choreographies in bpmn 2.0: new chal-

lenges and open questions. In *Proceedings of the 4th Central-European Workshop on Services and their Composition, ZEUS*, volume 847, pages 50–57. Citeseer, 2012.
[108] Mark Von Rosing, Stephen A. White, Fred Cummins, and Henk De Man. Business process model and notation-BPMN. *The Complete Business Process Handbook: Body of Knowledge from Process Modeling to BPM*, 1(January):429–453, 2014.
[109] Maria Hjorth. Strengths and weaknesses of a visual programming language in a learning context with children. 2017.
[110] A Groovy-based Model Operation Orchestration. A Groovy-based Model Operation Orchestration Language. 2017.
[111] Camilo Alvarez and Rubby Casallas. MTC Flow: A tool to design, develop and deploy model transformation chains. *ACadeMics Tooling with Eclipse, ACME 2013 - A Joint ECMFA/ECSA/ECOOP Workshop*, 2013.
[112] Beatriz Sanchez, Dimitris S. Kolovos, and Richard Paige. 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, 2019.
[113] Wolfgang Kling, Frédéric Jouault, Dennis Wagelaar, Marco Brambilla, and Jordi Cabot. MoScript: A DSL for querying and manipulating model repositories. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6940 LNCS:180–200, 2012.
[114] José E Rivera, Daniel Ruiz-Gonzalez, Fernando Lopez-Romero, José Bautista, and Antonio Vallecillo. Orchestrating ATL Model Transformations. *Proc. of MtATL 2009*, (June):34–46, 2009.
[115] Alessio Di Sandro, Rick Salay, Michalis Famelis, Sahar Kokaly, and Marsha Chechik. MMINT: A graphical tool for interactive model management. *CEUR Workshop Proceedings*, 1554:16–19, 2015.
[116] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017.
[117] C. M. Llad and C. U. Smith. Performance model interchange format (pmif 2.0): Xml definition and implementation. In *Quantitative Evaluation of Systems, International Conference on*, Los Alamitos, CA, USA, sep 2004. IEEE Computer Society.
[118] John Arundel and Justin Domingus. *Cloud Native DevOps with Kubernetes*. 2019.
[119] kubernetes.io. Kubernetes documentation, 2021.
[120] W3C. Web services description language (wsdl) version 2.0 part 1: Core language, 2007.
[121] Shrabani Mallick, Rajender Pandey, Sanjeev Neupane, Shakti Mishra, and D. S. Kushwaha. Simplifying Web service discovery & validating service composition. *Proceedings - 2011 IEEE World Congress on Services, SERVICES 2011*, pages 288–294, 2011.
[122] Hrishikesh Vijay Karambelkar. *Scaling Big Data with Hadoop and Solr Second Edition*. 2015.
[123] Alberto Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.
[124] Konstantinos Barmpis and Dimitrios S. Kolovos. Towards scalable querying of large-scale models. In *Modelling Foundations and Applications*, pages 35–50, Cham, 2014. Springer International Publishing.
[125] Konstantinos Barmpis. Towards scalable model indexing. (March), 2016.
[126] Ana I. Molina, William J. Giraldo, Jesús Gallardo, Miguel A. Redondo, Manuel Ortega, and Guillermo García. Ciat-gui: A mde-compliant environment for developing graphical user interfaces of information systems. *Advances in Engineering Software*, 52:10–29, 2012.
[127] Arsene Indamutsa, Juri Di Rocco, Davide Di Ruscio, and Alfonso Pierantonio. 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, 2021.
[128] Luca Berardinelli, Alexandra Mazak, Oliver Alt, Manuel Wimmer, and Gerti Kappel. *Model-Driven Systems Engineering: Principles and Application in the CPPS Domain*, pages 261–299. Springer International Publishing, Cham, 2017.
[129] Yu Beng Leau, Wooi Khong Loo, Wai Yip Tham, and Soo Fun Tan. Software development life cycle agile vs traditional approaches. In *International Conference on Information and Network Technology*, volume 37, pages 162–167, 2012.
[130] Mohankumar Muthu, K Banuroopa, and S Arunadevi. Green and sustainability in software development lifecycle process. *Sustainability Assessment at the 21st century*, 27(63), 2019.
[131] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2013.
[132] Massimo Tisi, Jean-Marie Mottu, Dimitrios S. Kolovos, Juan de Lara, Esther Guerra, Davide Di Ruscio, Alfonso Pierantonio, and Manuel Wimmer. Lowcomote: Training the next generation of experts in scalable low-code engineering platforms. In *STAF (Co-Located Events)*, volume 2405 of *CEUR Workshop Proceedings*, pages 73–78. CEUR-WS.org, 2019.
[133] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 171–178, 2020.
[134] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Model repositories: Will they become reality? *CloudMDE@MoDELS*, 1563:37–42, 2015.

[135] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. Incquery-d: A distributed incremental model query framework in the cloud. In *Model-Driven Engineering Languages and Systems*, pages 653–669, Cham, 2014. Springer International Publishing.

[136] Regina Hebig, Andreas Seibel, and Holger Giese. On the unification of megamodels. *Electronic Communications of the EASST*, 42, 2012.

[137] Dragan Gašević, Nima Kaviani, and Marek Hatala. 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, 2007.

[138] Peng Zhao, Peizhe Wang, Xinyu Yang, and Jie Lin. Towards cost-efficient edge intelligent computing with elastic deployment of container-based microservices. *IEEE Access*, 8:102947–102957, 2020.

[139] Lin Gu, Deze Zeng, Jie Hu, Hai Jin, Song Guo, and Albert Y. Zomaya. Exploring layered container structure for cost efficient microservice deployment. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–9, 2021.

[140] Jean Béziv. On the need for megamodels. pages 1–9, 2004.

[141] El Hadji Bassirou Toure, Ibrahima Fall, Alassane Bah, Mamadou Samba Camara, and Mandicou Ba. Consistency preserving for evolving megamodels through axiomatic semantics. In *2017 Intelligent Systems and Computer Vision (ISCV)*, pages 1–8, 2017.

[142] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 4:171–188, 5 2005.

[143] Juri Di Rocco, Davide Di Ruscio, Johannes Härtel, Ludovico Iovino, Ralf Lämmel, and Alfonso Pierantonio. Understanding MDE Projects: Megamodels to the Rescue for Architecture Recovery. *Software and Systems Modeling*, 19(2):401–423, 2019.

[144] Francesco Basciani, Davide Di Ruscio, Juri Di Rocco, Ludovico Iovino, and Alfonso Pierantonio. Exploring model repositories by means of megamodel-aware search operators. *CEUR Workshop Proceedings*, 2245:793–798, 2018.

[145] Antonio Bucchiarone, Jordi Cabot, Richard Paige, and Alfonso Pierantonio. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, 19:1–9, 01 2020.

[146] Felicien Ihirwe, Arsene Indamutsa, Davide Di Ruscio, Silvia Mazzini, and Alfonso Pierantonio. 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, 2021.

[147] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64, 08 2015.

[148] Claes Wohlin, Per Runeson, Paulo Neto, Emelie Engström, Ivan Machado, and Eduardo Almeida. On the reliability of mapping studies in software engineering. *Journal of Systems and Software*, 86:2594–2610, 10 2013.

[149] José Antonio Hernández López and Jesús Sánchez Cuadrado. 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, 2020. Association for Computing Machinery.

[150] José Antonio Hernández López and Jesús Sánchez Cuadrado. An efficient and scalable search engine for models. *Software and Systems Modeling*, 2021.

[151] Benedek Izsó, Gábor Szárnyas, István Ráth, and Dániel Varró. Incquery-d: Incremental queries in the cloud. pages 1–4, 2013.

[152] Jesús Sánchez Cuadrado and Martin Gogolla. Model finding in the emf ecosystem. *Journal of Object Technology*, 19:1–21, 2020.

[153] Daniel Lucrédio, Renata P de M Fortes, and Jon Whittle. Moogle: a metamodel-based model search engine. *Software & Systems Modeling*, 11(2):183–208, 2012.

[154] Wolfgang Kling, Frédéric Jouault, Dennis Wagelaar, Marco Brambilla, and Jordi Cabot. Moscript: A dsl for querying and manipulating model repositories. In Anthony Sloane and Uwe Aßmann, editors, *Software Language Engineering*, pages 180–200, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[155] Konstantinos Barmpis and Dimitris Kolovos. 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, 2013. ACM.

[156] Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, and Carlos Bento. Using wordnet for case-based retrieval of uml models. *AI Communications*, 17(1):13–23, 2004.

[157] Bojana Bislimovska, Güneş Aluç, M. Tamer Özsu, and Piero Fraternali. Graph search of software models using multidimensional scaling. *CEUR Workshop Proceedings*, 1330:163–170, 2015.

[158] Ta'id Holmes, Uwe Zdun, and Schahram Dustdar. Morse: A model-aware service environment. In *2009 IEEE Asia-Pacific Services Computing Conference*, pages 470–477. IEEE, 2009.

[159] Bojana Bislimovska, Alessandro Bozzon, Marco Brambilla, and Piero Fraternali. 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, 2012.

[160] Giorgos Kotopoulos, Fotis Kazasis, and Stavros Christodoulakis. 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, 2007.

[161] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the large and modeling in

the small. In *Model Driven Architecture*, pages 33–46. Springer, 2004.

[162] Dumitru Roman, Sven Schade, Arne-Jørgen Berre, Nils Rune Bodsberg, and J Langlois. Model as a service (maas). 01 2009.

[163] Muhammad Nouman Zafar, Farooque Azam, Saad Rehman, and Muhammad Waseem Anwar. A systematic review of big data analytics using model riven engineering. *ACM International Conference Proceeding Series*, pages 1–5, 2017.

[164] Michele Guerriero, Saeed Tajfar, Damian A. Tamburri, and Elisabetta Di Nitto. 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, 2016. Association for Computing Machinery.

[165] Gurpreet S. Sachdeva. *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, 2017.

[166] Belachew Regane. Model-driven engineering for big data. 08 2019.

[167] Sabrina Boubiche, Djallel Eddine Boubiche, Azeddine Bilami, and Homero Toral-Cruz. Big data challenges and data aggregation strategies in wireless sensor networks. *IEEE Access*, 6:20558–20571, 2018.

[168] Qingsong Guo, Jiaheng Lu, Chao Zhang, Calvin Sun, and Steven Yuan. 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, 2020. Association for Computing Machinery.

[169] Rodrigo Laigner, Marcos Kalinowski, Pedro Diniz, Leonardo Barros, Carlos Cassino, Melissa Lemos, Darlan Arruda, Sergio Lifschitz, and Yongluan Zhou. 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, 2020.

[170] Izumi V. Hinkson, Tanja M. Davidsen, Juli D. Klemm, Anthony R. Kerlavage, and Warren A. Kibbe. A comprehensive infrastructure for big data in cancer research: Accelerating cancer research and precision medicine. *Frontiers in Cell and Developmental Biology*, 5, 9 2017.

[171] Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2013.

[172] Bharvi Dixit, Rafal Kuc, Marek Rogozinski, and Saurabh Chhajed. *Elasticsearch: A Complete Guide*. Packt Publishing, Birmigham, Mumbai, 2017.

[173] Liberios Vokorokos, Matúš Uchnár, and Lubor Leščišin. Performance optimization of applications based on non-relational databases. In *2016 International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pages 371–376, 2016.

[174] Guobing Zou, Bofeng Zhang, Jianxing Zheng, Yinsheng Li, and Jianhua Ma. 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, 2012.

[175] Fábio Paulo Basso, Toacy Cavalcante Oliveira, Cláudia M.L. Werner, and Leandro Buss Becker. Building the foundations for 'mde as service'. *IET Software*, 11(4):195–206, 2017.

[176] Lissette Almonte, Sara Pérez-Soler, Esther Guerra, Iván Cantador, and Juan de Lara. *Automating the Synthesis of Recommender Systems for Modelling Languages*, pages 22—-35. Association for Computing Machinery, New York, NY, USA, 2021.

[177] Hugo Bruneliere, Jordi Cabot, and Frédéric Jouault. 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)*, Paris, France, June 2010.

[178] Clinton Gormley and Zachary Tong. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine*. " O'Reilly Media, Inc.", 2015.

[179] Lucene's practical scoring function: Elasticsearch: The definitive guide [2.x].

[180] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[181] Steve Cook, Gareth Jones, Stuart Kent, and Alan Cameron Wills. *Domain-specific development with visual studio dsl tools*. Pearson Education, 2007.

[182] Stephen H Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Professional, 2003.

[183] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. A customizable approach for the automated quality assessment of modelling artifacts. In *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 88–93. IEEE, 2016.

[184] Francesco Basciani, Mattia D'Emidio, Davide Di Ruscio, Daniele Frigioni, Ludovico Iovino, and Alfonso Pierantonio. Automated selection of optimal model transformation chains via shortest-path algorithms. *IEEE Transactions on Software Engineering*, 46(3):251–279, 2018.

[185] Jesús Sánchez Cuadrado, Loli Burgueno, Manuel Wimmer, and Antonio Vallecillo. Efficient execution of atl model transformations using static analysis and parallelism. *IEEE Transactions on Software Engineering*, 2020.

[186] Gediminas Adomavicius and Alexander Tuzhilin. 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, 2005.

[187] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Cross-language program slicing for dynamic web applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 369–380, New York, NY, USA, 2015. Association for Computing Machinery.

[188] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press, 1981.

[189] Rick Salay, Sahar Kokaly, Marsha Chechik, and T. S. E. Maibaum. Heterogeneous megamodel slicing for model evolution. In *ME@MoDELS*, 2016.

[190] Gabriele Taentzer, Timo Kehrer, Christopher Pietsch, and Udo Kelter. A formal framework for incremental model slicing. In *FASE*, 2018.

[191] Reza Ahmadi, Juergen Dingel, and Ernesto Posse. Slicing uml-based models of real-time embedded systems. 07 2018.

[192] Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc Jézéquel. Meta-model Pruning. In *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, Denver, Colorado, USA, United States, 2009.

[193] Alexander Bergmayr, Manuel Wimmer, Werner Retschitzegger, and Uwe Zdun. Taking the pick out of the bunch - type-safe shrinking of metamodels. In Stefan Kowalewski and Bernhard Rumpe, editors, *Software Engineering 2013*, pages 85–98, Bonn, 2013. Gesellschaft für Informatik e.V.

[194] Huzefa Kagdi, Jonathan I. Maletic, and Andrew Sutton. Context-free slicing of uml class models. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, page 635–638, USA, 2005. IEEE Computer Society.

[195] Pierre Kelsen, Qin Ma, and Christian Glodt. Models within models: Taming model complexity using the sub-model lattice. In Dimitra Giannakopoulou and Fernando Orejas, editors, *Fundamental Approaches to Software Engineering*, pages 171–185, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[196] Arnaud Blouin, Benoît Combemale, Benoit Baudry, and Olivier Beaudoux. Modeling model slicers. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems*, pages 62–76, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[197] Loli Burgueño, Robert Clarisó, Shuai Li, Sébastien Gérard, Jordi Cabot, Loli Burgueño, Robert Clarisó, Shuai Li, Sébastien Gérard, and Jordi Cabot An Nlp-based. An NLP-based architecture for the autocompletion of partial domain models To cite this version : HAL Id : hal-03010872 A NLP-based architecture for the autocompletion of partial domain models. 2021.

[198] Ora Lassila and Ralph R Swick. Resource description framework (RDF) model and syntax specification. World Wide Web Consortium Recommendation. (October), 1999.