“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

D4.1 Low-Code Engineering Repository Architecture Specification

Project GA: 813884

Project Acronym: Lowcomote

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

Project officer: Dora Horváth

Work Package: WP4

Deliverable number: D4.1

Production date: 26 November 2020

Contractual date of delivery: 30 November 2020

Actual date of delivery: 30 November 2020

Dissemination level: Public

Lead beneficiary: University of L’Aquila

Authors: L. Berardinelli, A. Colantoni, D. Di Ruscio, F. Khorram, A. Indamutsa, I. Ibrahimi, A. Pierantonio, A. Sahay

Contributors: Lowcomote partners

Abstract

Low-code development platforms (LCPD) are software development platforms on the Cloud, provided through a Platform-as a-Service model, which allow 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.

Lowcomote will train a generation of experts that will upgrade the current trend of LCPDs to a new paradigm, Low-code Engineering Platforms (LCEPs). LCEPs will be open, allowing to integrate heterogeneous engineering tools, interoperable, allowing for cross-platform engineering, scalable, supporting very large engineering models and social networks of developers, smart, simplifying the development for citizen developers by machine learning and recommendation techniques. This will be achieved by injecting in LCDPs the theoretical and technical framework defined by recent research in Model Driven Engineering (MDE), augmented with Cloud Computing and Machine Learning techniques. This is possible today thanks to recent breakthroughs in scalability of MDE performed in the EC FP7 research project MONDO, led by Lowcomote partners.

The 48-month Lowcomote project will train the first European generation of skilled professionals in LCEPs. The 15 future scientists will benefit from an original training and research programme merging competencies and knowledge from 5 highly recognised academic institutions and 9 large and small industries of several domains. Co-supervision from both sectors is a promising process to facilitate agility of our future professionals between the academic and industrial world.

Executive summary

This document introduces an evolving architecture specification of a low-code engineering repository underpinning the design, analysis, and evolution of low-code artefacts. The proposed repository is intended to support a community of citizen developers and to enable the reusability of heterogeneous modelling artefacts. An extended *"4 + 1"* view model has been adopted to present the architecture of the repository that will be developed in the context of WP4 and that will support different services including model management, continuous software engineering, model recommendations, and quality assurance.

Contents

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 smart 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 very large number of low-code artefacts, and automatically learn from them.
- **RO3:** Producing advancements in *Scalable Low-Code Artefact Management*, as new algorithms and reusable components.

By referring to the architecture overview of the Lowcomote project shown in Fig. 1, this document presents initial results related to **RO2** developed in the context of WP4 and in particular of *ESR6 - Scalable and Extensible Cloud-based Low-Code Model Repository*. 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. [. . .]" .

In this respect, this document provides a high level overview of the evolving technical architecture of the Lowcomote repository. It describes the goals of the architecture, the use cases supported by the repository,

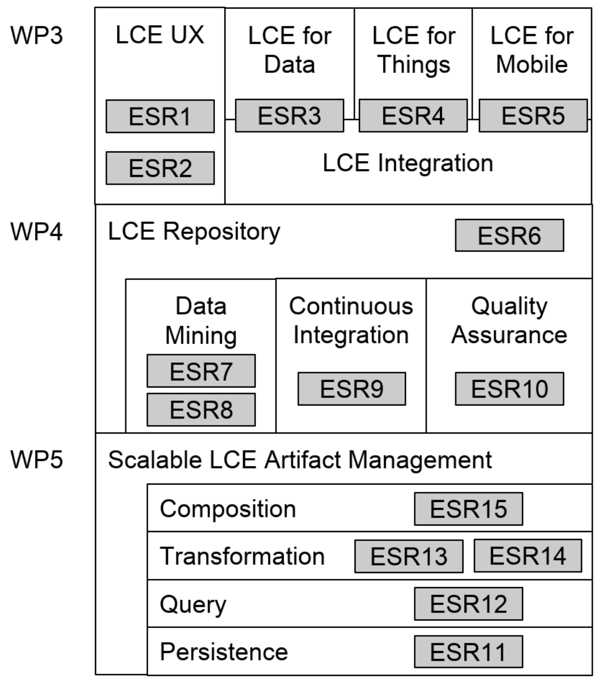


Figure 1: Overview of the Lowcomote project architecture

and the architectural components that will be implemented to best support the use cases. Moreover, it outlines the technologies that will be used for the implementation of the knowledge base. The development of the architectural description provided in this document 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 2 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 3 presents the architecture of the Lowcomote repository. In particular, the different architectural views used in this document are briefly presented, before presenting the architectural view decomposition of the envisioned Lowcomote repository.
- Chapter 4 concludes the document.

2 Low-Code Development Platforms

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 section, a technical survey is provided to distil 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 this 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.

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 [**?**] 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 [7].

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

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

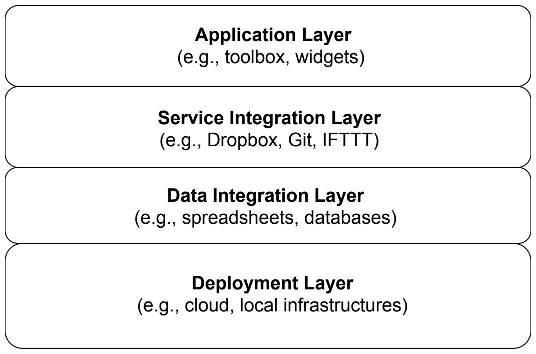

Application Layer
(e.g., toolbox, widgets)
Service Integration Layer
(e.g., Dropbox, Git, IFTTT)
Data Integration Layer
(e.g., spreadsheets, databases)
Deployment Layer
(e.g., cloud, local infrastructures)

Figure 2: Layered architecture of low-code development platforms

to specify their applications. The toolboxes and widgets used to build the user interface of the specified application are part of this layer. It also defines authentication and 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 [8] 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 [9].

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

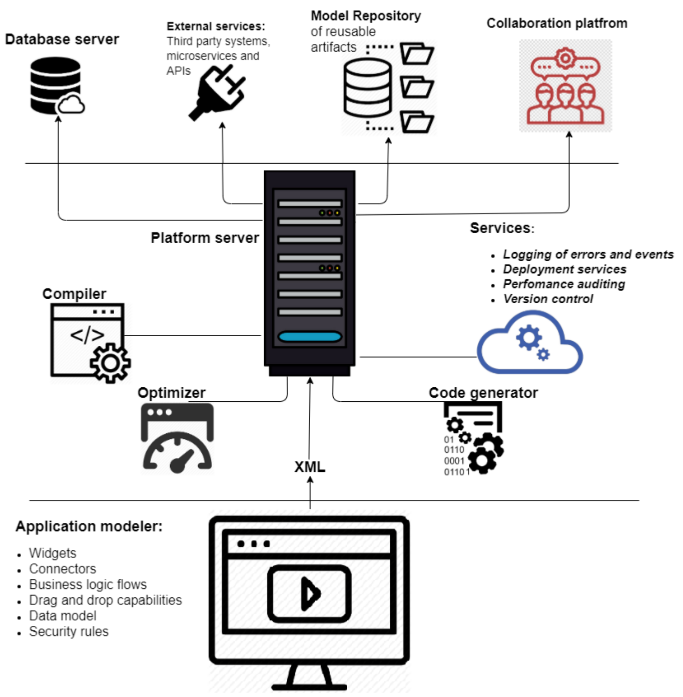


Figure 3: Main components of low-code development platforms

like authentication, load balance, business logic consistency, data integrity and security.

Low-code development platforms can also provide developers with repositories that can store reusable 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,

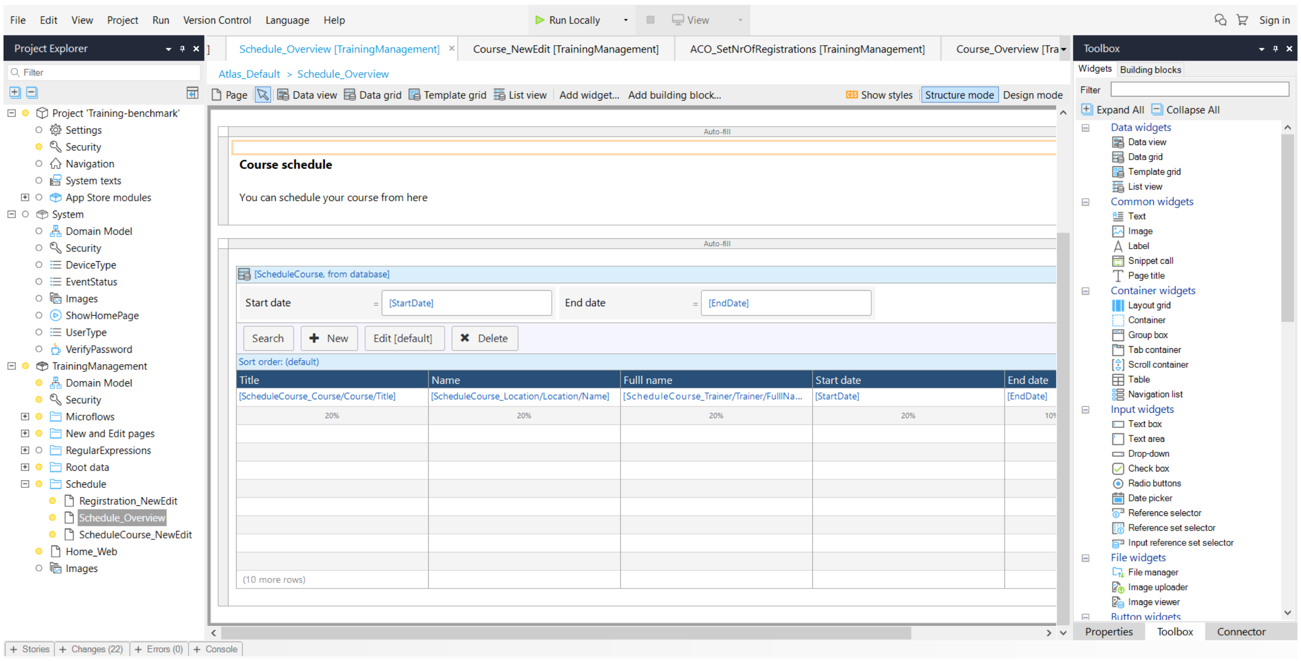


Figure 4: The application modeler of Mendix at work

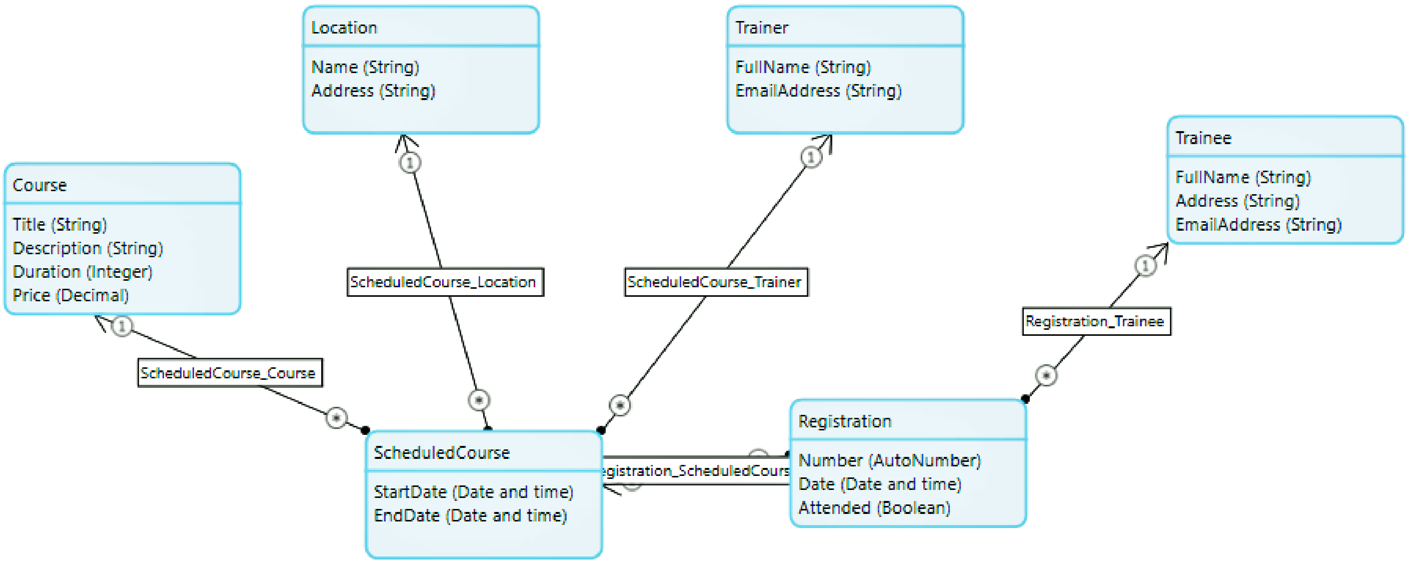


Figure 5: A simple data model defined in Mendix

defining constraints and dependencies generally through drag-and-drop facilities. A simple data model defined in Mendix is shown in Fig. 5.

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 [9] is a low-code development platform that allows developing desktop and mobile applications, 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 [8] 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

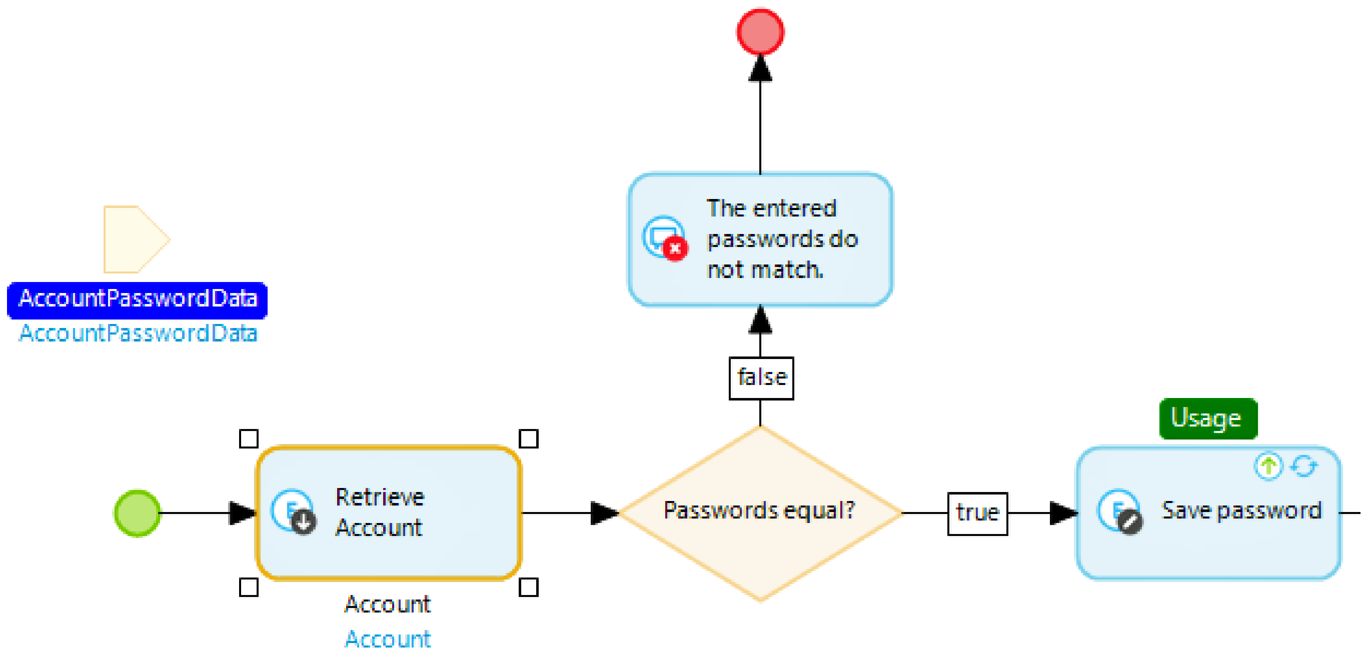


Figure 6: A simple logic defined in Mendix

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 [10] 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 [11] 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 [12] 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 [13] 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 [14] 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

[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

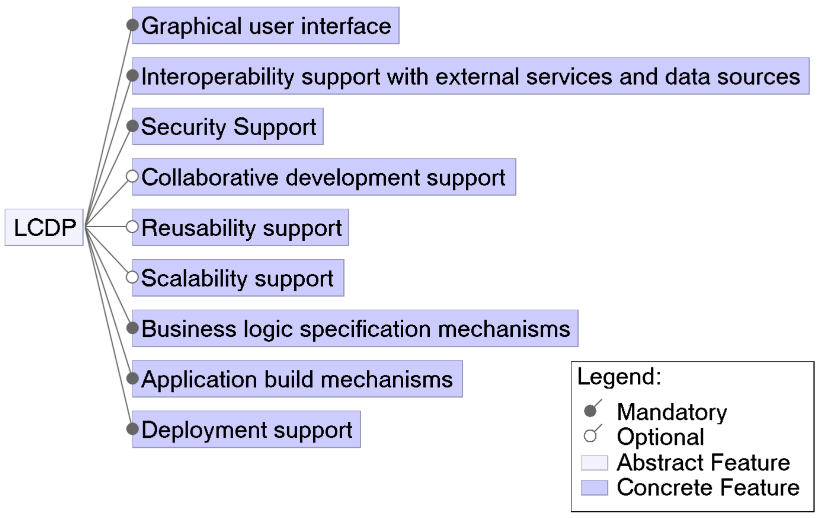

LCDP
Graphical user interface
Interoperability support with external services and data sources
Security Support
Collaborative development support
Reusability support
Scalability support
Business logic specification mechanisms
Application build mechanisms
Deployment support
Legend:
Mandatory
Optional
Abstract Feature
Concrete Feature

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

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 [15] 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.

2.3 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 [16], which are a common notation in domain analysis [17]. 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.

[12]https://appexchange.salesforce.com/

- *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 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 2 shows the outcome of the performed comparison by showing the corresponding supported features for each platform. The data shown in Table 2 are mainly obtained by considering the official resources of each platform as referenced by [9],[8], [10], [11], [12], [13], [14], [15], and by considering the experience we gained during the development of a benchmark application as discussed in the next section.

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,

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.

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

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		✓	✓	✓	✓	✓	✓	✓

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 [18] [19][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 [20]. 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.

3 Low-Code Engineering Repository Architecture

The development of the Lowcomote platform will rely on techniques and tools conceived by recent research in MDE (Model-Driven Engineering). MDE is characterized by the systematic use of models as primary abstraction entities all along the software development life-cycle to promote both abstraction and automation [7]. To this end, domain-specific languages are employed to shift the focus from third generation programming languages to the use of models that are close to the problem domain. Models are subsequently analysed and manipulated to generate the source code of the modeled systems. Despite the advantages related to the adoption of MDE, there are still some limiting factors that hamper a wider adoption of MDE including the following ones:

- The support for discovery and reuse of existing modelling artefacts is very limited. As a result, similar transformations and other model management tools often need to be developed from scratch, thus raising the upfront investment, and compromising the productivity benefits of model-based processes.
- Modelling and model management tools are commonly distributed as software packages that need to be downloaded and installed on client machines, often on top of complex software development IDEs (e.g. Eclipse).
- Integration mechanisms that enable developers to build applications and manage new artifacts and add additional functionalities.

To overcome these issues, in this chapter, we present the ongoing work related to the development of a cloud-based Low-Code engineering repository. This repository will expose its model management functionalities as-a-service to support remote access, manipulation and storage of different kinds of modelling artefacts. In addition, the repository will be tuned to support data analytics services to enable the application of machine learning models. The envisioned repository is expected to support several extensions and services from third-party entities, especially fellow researchers involved in this project. The architecture of this community-based model repository shall enable the reuse of heterogeneous modelling artefacts, support advanced search mechanisms, and be open to the addition of new functionalities while preserving robustness and scalability.

3.1 Related work

In this section we discuss state-of-the-art approaches for providing repositories of modeling artefacts, and outline outstanding research challenges for achieving a comprehensive solution to the problem of properly managing the persistence of models and the deployment and discovery of any kind of model management tools to enable their reuse and refinement.

AMOR - Adaptable Model Versioning [21]: it is an attempt to leverage version control systems in the area of MDE. AMOR supports model conflict detection, and focuses on intelligent conflict resolution by providing techniques for the representation of conflicting modifications as well as suggesting appropriate resolution strategies.

Bizycle [22]: it is a project aiming at supporting the automated integration of software components by means of model-driven techniques and tools. Among the different components of the project, a metadata repository is also provided in order to manage and store all the artefacts required for and generated during integration processes, i.e, external and internal documentation, models and metamodels, transformation rules, generated code, users and roles.

CDO[13]: it is a pure Java model repository for EMF models and meta models. CDO can also serve as a persistence and distribution framework for EMF-based application systems. CDO supports different kinds of deployments such as embedded repositories, offline clones and replicated clusters. However, the typical deployment scenario consists of a server managing the persistence of the models by exploiting all kinds of database backends (like major relational databases or NoSQL databases), and an EMF client application.

EMFStore [23]: it is a software configuration management system tailored to the specific requirements of versioning models. It is based on the Eclipse Modeling Framework and it is an implementation of a

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

	Managed Artefact	**Main purpose**	**Typical deployment scenario**
AMOR [21]	Model	Model versioning	Desktop application
Bizycle [22]	Model	Integration of software components	Desktop application
CDO	Model	Storage	Client-Server application
EMFStore [23]	Model	Model versioning	Client-Server application
GME [24]	Model	Storage	Client-Server application
ModelBus [25]	Model	Model versioning	Client-Server application
Morse [26]	Model	Model versioning	Software-as-a-service
ReMoDD [27]	Any	Documentation	Web-based interaction
MDEForge [28]	Model, Metamodel, Transformation	Storage, Added value services	Web-based interaction, Software-as-a-service

Table 3: Overview of existing MDE tools providing storage features

generic operation-based version control system. EMFStore implements, in the modeling domain, the typical operations implemented by SVN, CVS, Git for text-based artefacts, i.e., change tracking, conflict detection, merging and versioning. It consists of a server and a client component. The server runs standalone and provides a repository for models including versioning, persistence and access control. The client component is usually integrated into an application and is responsible for tracking changes on the model, and for committing, updating and merging.

GME - Generic Modeling Environment [24]: it is a set of tools supporting the creation of domain specific modeling languages and code generation environments. A repository layer is also provided to store the developed models. Currently, MS Repository (an object oriented layer on top of MS SQL Server or MS Access) and a proprietary binary file format are supported.

ModelBus [25]: it consists of a central bus-like communication infrastructure, a number of core services and a set of additional management tools. Depending on the usage scenario at hand, different development tools can be connected to the bus via tool adapters. Once a tool has been successfully plugged in, its functionality immediately becomes available to others as a service. Alternatively, it can make use of services already present on the ModelBus. Among the available services, ModelBus also includes a built-in model repository, which is able to version models, supports the partial check-out of models and coordinates the merging of model versions and model fragments;

Morse - Model-Aware Repository and Service Environment [26]: it is a service-based environment for the storage and retrieval of models and model-instances at both design- and run-time. Models, and model elements are identified by Universally Unique Identifiers (UUID) and 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 the models can be maintained in parallel;

ReMoDD - Repository for Model-Driven Development [27]: it is a repository of artefacts aiming at improving MDE research and industrial productivity, and learning experience of MDE students. By means of a Drupal Web application, users can contribute MDE case studies, examples of models, metamodels, model transformations, descriptions of modeling practices and experience, and modeling exercises and problems that can be used to develop classroom assignments and projects. Searching and browsing facilities 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 [28]: it is an extensible modeling framework consisting of a set of core services that permit to store and manage typical modeling artefacts and tools. Atop of such services it is possible to develop extensions adding new functionalities to the platform. All the services can be used by means of a Web access and by a REST API that permits to adopt the available model management tools by means of the software-as-a-service paradigm.

According to Table 3, 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

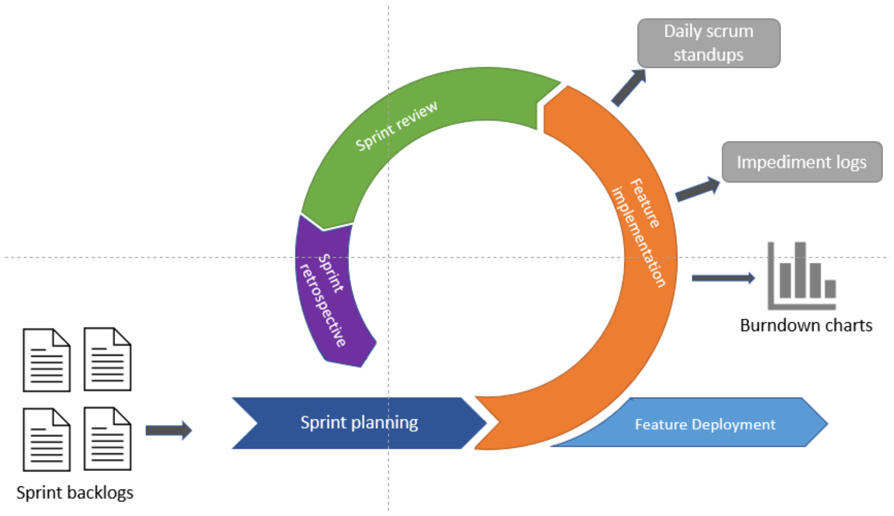


Figure 8: Scrum iterative sprints

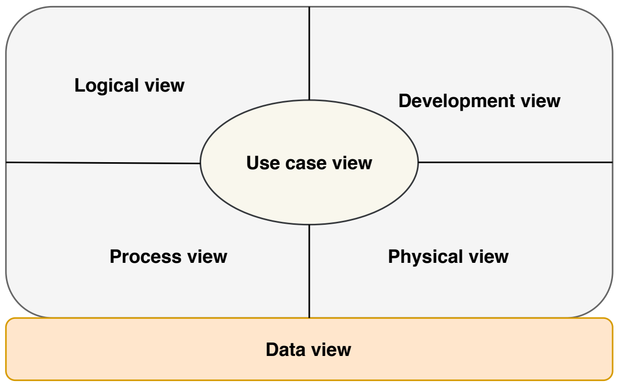


Figure 9: The extended *"4+1"* views of the Lowcomote repository architecture

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.

3.2 System views

In this section, the architecture of the Lowcomote repository is presented. Its development follows an iterative and incremental approach where activities consist of multiple sprints and feature deliveries as shown in Fig. 8. Iterations include analysis, design and implementation, testing and deployments organized in sprints and based on clearly defined features. To manage development, we will use several methodologies such as Scrum, Kanban and extreme programming. Scrum will help us organize our workflow in sprints to ensure independent fully functional features, and Kanban will help us visualize the workflow and better organize the backlogs. To enhance responsiveness and swift update of changes and requirements, extreme programming will be used.

To describe the architecture of the proposed system, we use 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. 9). Since the repository is supposed to store and manipulate modeling artifacts, we decided that adding such an extra view is appropriate to show also the structure of the data that will be managed by the repository. Thus, the six views that are used to describe the repository architecture are the following:

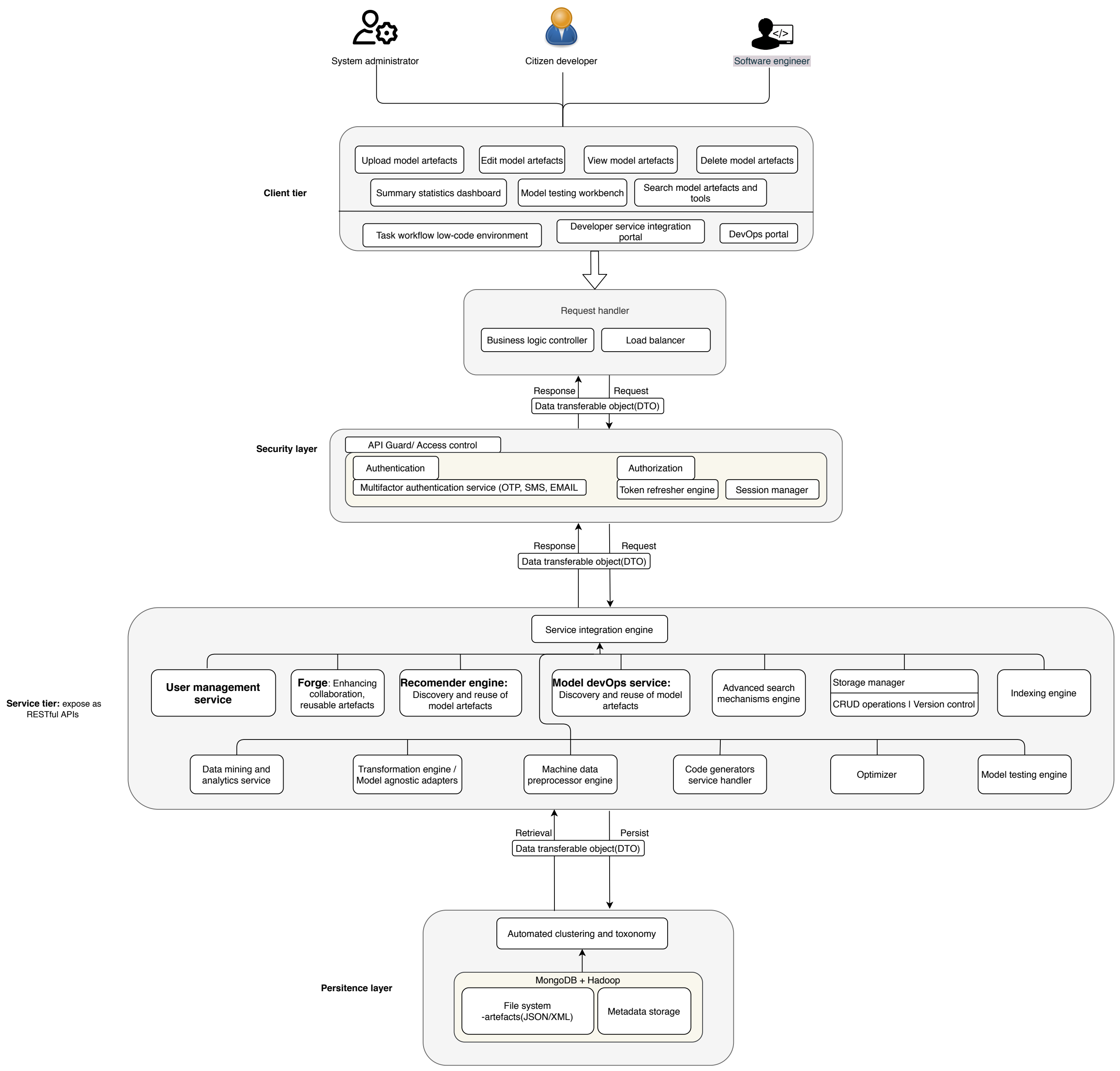

System administrator
Citizen developer
Software engineer
Client tier
Upload model artefacts
Edit model artefacts
View model artefacts
Delete model artefacts
Summary statistics dashboard
Model testing workbench
Search model artefacts and tools
Task workflow low-code environment
Developer service integration portal
DevOps portal
Request handler
Business logic controller
Load balancer
Response
Request
Data transferable object(DTO)
Security layer
API Guard/ Access control
Authentication
Authorization
Multifactor authentication service (OTP, SMS, EMAIL
Token refresher engine
Session manager
Response
Request
Data transferable object(DTO)
Service tier: expose as RESTful APIs
Service integration engine
User management service
Forge: Enhancing collaboration, reusable artefacts
Recomender engine: Discovery and reuse of model artefacts
Model devOps service: Discovery and reuse of model artefacts
Advanced search mechanisms engine
Storage manager
CRUD operations I Version control
Indexing engine
Data mining and analytics service
Transformation engine / Model agnostic adapters
Machine data preprocessor engine
Code generators service handler
Optimizer
Model testing engine
Retrieval
Persist
Data transferable object(DTO)
Persitence layer
Automated clustering and toxonomy
MongoDB + Hadoop
File system -artefacts(JSON/XML)
Metadata storage

Figure 10: High level architecture view

- **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 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 10 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 requested services. The high-level view of the system is detailed in the next sub-sections according to the augmented 4+1 model previously mentioned.

3.2.1 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. 10, 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.

3.2.1.1 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 11 shows the two main use cases involving software engineers as described below:

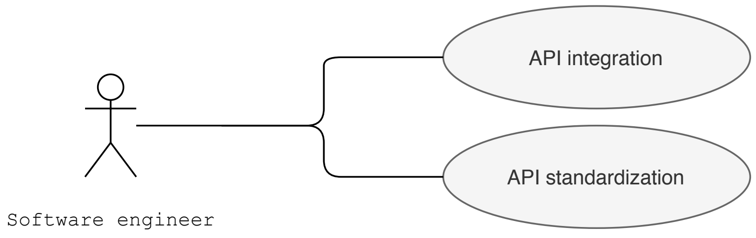


Figure 11: Software engineer use case diagram

- **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 [14] for API documentation and further testing.

3.2.1.2 System administrator System administrators will manage the repository infrastructure and the users that are allowed to exploit the repository functionalities. As shown in Fig. 12 the use cases that will be provided by the repository to system administrators are the following:

- **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;

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

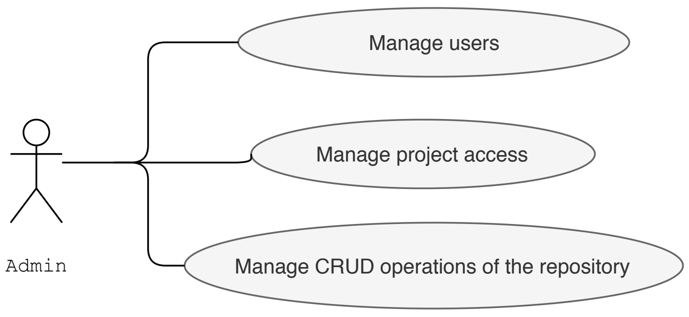


Figure 12: System administrator use case diagram

- **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.

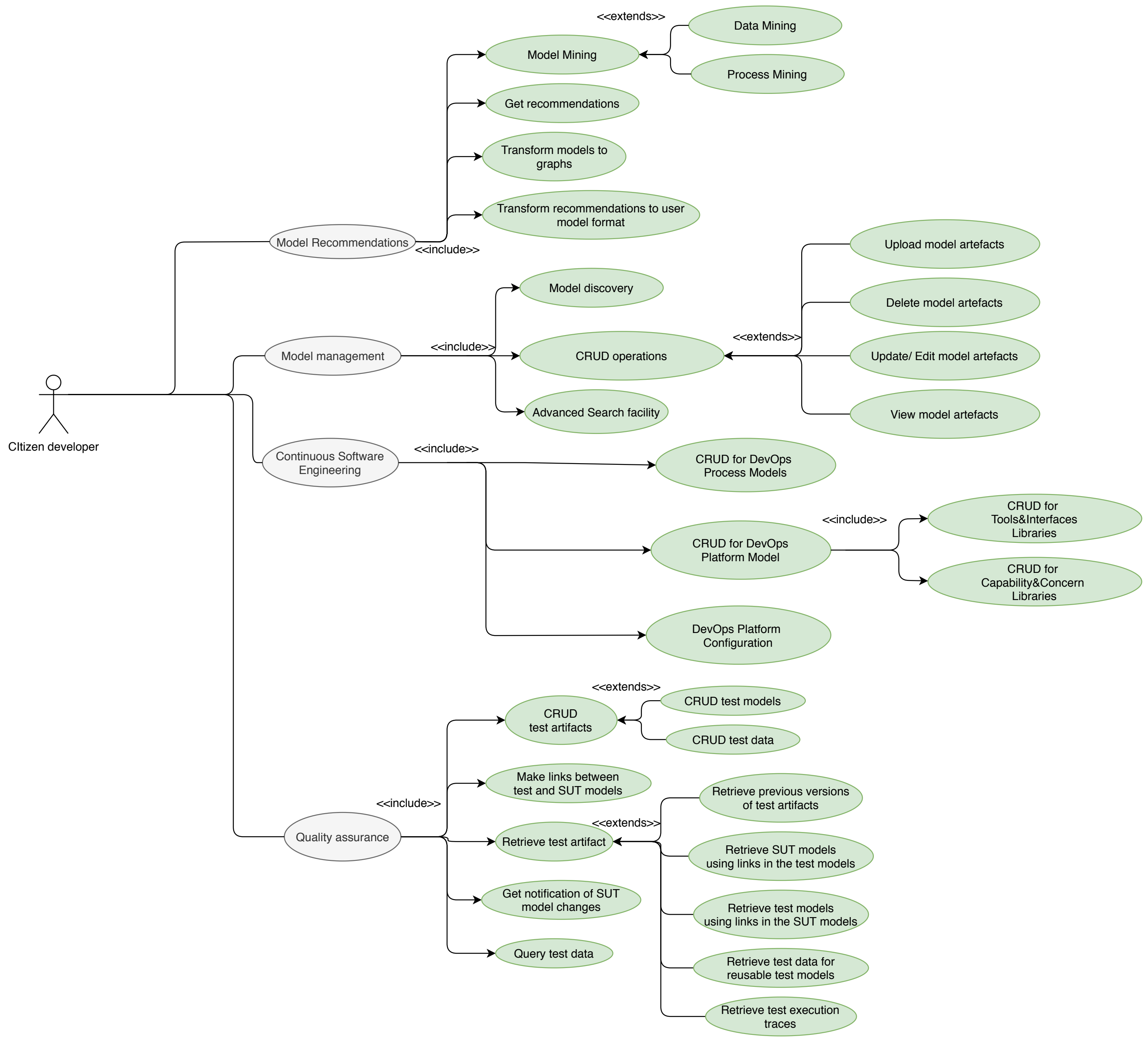
3.2.1.3 Citizen developer Citizen developers are the main target of the system and all the provided functionalities have been defined by considering the potential needs that inexpert developers expect from a Low-Code repository. As shown in Fig. 13, the functionalities provided by the envisioned Low-Code repository are grouped under four main categories, i.e., *Model management*, *Continuous Software Engineering*, *Model Recommendations*, and *Quality Assurance*, which are described in detail below.

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

- **CRUD Operations:** The system will implement several functionalities that will enable manipulations of modeling artefacts in the repository. The basic envisioned manipulations are:
  - The citizen developer can upload models to the repository.
  - The citizen developer can edit/update modeling artefacts stored in the repository using provided user interfaces.
  - The citizen developer can display selected modeling artefacts.
  - A citizen developer can delete modeling artefacts from the repository.
- **Model discovery:** This repository feature enables the user to discover relevant domain models. This feature permits the citizen developer to reuse any existing domain models and to avoid starting modeling processes from scratch.
- **Advanced Search facility:** In order to discover relevant models and related artifacts in the repository, advanced search facilities will be provided to the user.

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."* [29]. 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 comprehensive term to refer many continuous-* activities like Continuous Development, Continuous Integration, Continuous Deployment and Continuous Delivery to form complete DevOps processes [30]. The Low-Code repository will support model-driven CSE approaches by providing typical CRUD functionalities for DevOps process and platform models as discussed below. In particular, it will support the current and future extension of the DevOpsML conceptual framework presented in [31].

- **CRUD for DevOps Process Model**: The citizen developer can Create, Read, Update, Delete repository entries that correspond to Process Models (see [31]).



Citizen developer
Model Recommendations
<<include>>
Model Mining
Get recommendations
Transform models to graphs
Transform recommendations to user model format
<<extends>>
Data Mining
Process Mining
Model management
<<include>>
Model discovery
CRUD operations
Advanced Search facility
<<extends>>
Upload model artefacts
Delete model artefacts
Update/ Edit model artefacts
View model artefacts
Continuous Software Engineering
<<include>>
CRUD for DevOps Process Models
CRUD for DevOps Platform Model
<<include>>
CRUD for Tools&Interfaces Libraries
CRUD for Capability&Concern Libraries
DevOps Platform Configuration
Quality assurance
<<include>>
CRUD test artifacts
<<extends>>
CRUD test models
CRUD test data
Make links between test and SUT models
Retrieve test artifact
<<extends>>
Retrieve previous versions of test artifacts
Retrieve SUT models using links in the test models
Retrieve test models using links in the SUT models
Retrieve test data for reusable test models
Retrieve test execution traces
Get notification of SUT model changes
Query test data

Figure 13: Use case diagram

- **CRUD for DevOps Platform Model**: The citizen developer can Create, Read, Update, Delete repository entries that correspond to Platform Models (see [31]). This use case consists of supporting CRUD operations for reusable libraries of models representing tools with their interfaces, and required/provided capabilities to address CSE-related concerns.

- **DevOps Platform Configuration**: The citizen developer can query existing DevOps Process and Platform Models to combine them in Low-Code Platform configurations, thus combining tools, interfaces and provided capabilities to address given concerns [31].

It is worth mentioning that the Low-Code repository will provide typical storage functionalities as previously presented; advanced functionalities e.g., process/platform model-specific validation rules or analysis are primarily expected as capabilities of external tools.

<u>**Model Recommendations use cases:**</u> During the construction of new models, relevant model artifacts that are stored in the repository can be reused in order to facilitate the modeling process or to improve the state of an underdeveloped model. The possible reuse of modeling artifacts will be offered to the citizen developer in terms of recommendations.

- **Model mining:** Analyze by model mining techniques the existing MDE artifacts for potential reuse or other improvements. In general, we can distinguish two model mining techniques, *i)* data mining (classical check of artifacts' content) and *ii)* process mining (analysis of the processes the artifacts are involved in).

- **Get recommendations:** This use case consists of providing recommendations to the user during the modeling process. The recommendations can be triggered proactively (by the system) or re-actively (by the user). The recommendations occur to suggest the user e.g., the next modeling step, how to name a new model artifact, and what kinds of relationships can be set between newly created model artifacts. The recommendations can also provide useful suggestions on how to improve the current state of an underdeveloped model that has been just uploaded to the repository.
- **Transform recommendations to user model format:** After the user has selected any recommendation given from the repository side, the selected suggestion has to be converted into the user model format and merged with it.
- **Transform models to graphs:** Since the user models under construction might be of different formats, it is necessary that the models can be transformed from the user model format to the repository model format and vice versa. In particular, since the repository is a graph-based repository, the models to be persisted on the repository have to be converted into a corresponding graph-based representation. This feature is level and language agnostic.

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, which are stored and managed by the repository according to the use cases described below.

- **CRUD test artifacts**: The user can perform CRUD operations on two sets of test artifacts:
  - **CRUD test models**: The user can define test models (i.e., test cases) for performing functional testing of system models (i.e., models of the System-Under Test (SUT)). In addition of creating test models, she can read, update, and delete operations as well.
  - **CRUD test data**: The testing workbench allows definition of reusable test models by separating test data from test models. Therefore, the user can create new test data, and also read, update, or delete existing ones.
- **Make links between test and SUT models**: Each test model aims at testing and validating the correctness of specific SUT models. To provide traceability between these two sets of models, the user can explicitly define links between them.
- **Retrieve test artifact**: Various test artifacts, possibly in different sizes, will be stored in the repository for later reuse. Therefore, the system will permit users to perform the following retrieval operations:
  - **Retrieve previous versions of test artifacts**: Test case definition is an iterative process and test artifacts evolve along with system models. Consequently, different versions of test artifacts will be stored in the repository and the user can retrieve them.
  - **Retrieve SUT models using links in the test models**: The user can ask the repository to retrieve the system models that are associated with a specific test model (i.e., the system models that are tested through that specific test model).
  - **Retrieve test models using links in the SUT models**: The user can ask the repository to retrieve test models that are associated with a specific system model (i.e., the test models that are defined for testing that specific system model).
  - **Retrieve test data for reusable test models**: For reusable test models, the user can retrieve their associated test data from the repository.
  - **Retrieve test execution traces**: After execution of test models, execution traces will be generated and then stored in the repository. The user can retrieve them for troubleshooting of failed test models.

- **Get notification of SUT model changes**: Changes in SUT models can give place to inconsistencies with their associated test models. To prevent this, the repository will notify the user in case of SUT model changes.
- **Query test data**: The user can query stored test data to use specific sets of them. This capability is essential when the amount of stored test data for a test model is very large.

3.2.2 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. 14 and it is described in the following.

- **Artefact view control:** This component pertains to the client layer, and it is responsible for the basic functionalities of the repository, especially to show the repository contents such as available model artefacts, tools and services. It also presents several model manipulation operations such as upload, delete, update or upload of model artefacts on the repository.
- **Business logic controller:** This component is responsible for the main server application logic and APIs and thus ensuring the load balance of the system. It uses a security manager component, which also exposes public APIs. Therefore, any external communications will first go through from this component.
- **Security manager:** This component is responsible for ensuring 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 API guard and ensuring access control.
- **User manager:** This component shall store user data to ensure authentication and store tokens, sessions and other information collected from use cookies. Thus, user management tasks such as creating account, login and so on will be implemented in this component.
- **Service integration:** This component is responsible for integrating components (which are related to the management of different services such as testing, search, DevOps, etc.) and expose their APIs to the business logic controller in a homogeneous manner.
- **CRUD manager:** This component is responsible of upload, delete, update, edit and display of modeling artefacts. It is in constant synchronization with the persistence layer to ensure data integrity and information update.
- **Search engine:** This component implements advanced search mechanisms to ensure a swift search of model artefacts and tools regardless of the number of artefacts stored in the repository.
- **Indexer:** This component implements indexing facilities that will be useful for our search component and machine learning operations that require massive data processing.
- **Persistence component:** This component will interact with the databases used to store the managed artifacts. We will implement security guards that ensure data is secure and a security manager will continuously be working to ensure any request of data from repository's databases is authenticated and authorized e.g., with updated tokens.
- **Repository database :** This component will use the database to store data and metadata from the repositories components.
- **Model artifact recommender:** This component has the responsibility to provide recommendations to the user during the modeling process.
- **Query engine:** For a given underdeveloped or under construction model, the 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 and thus corresponding recommendations will be identified.

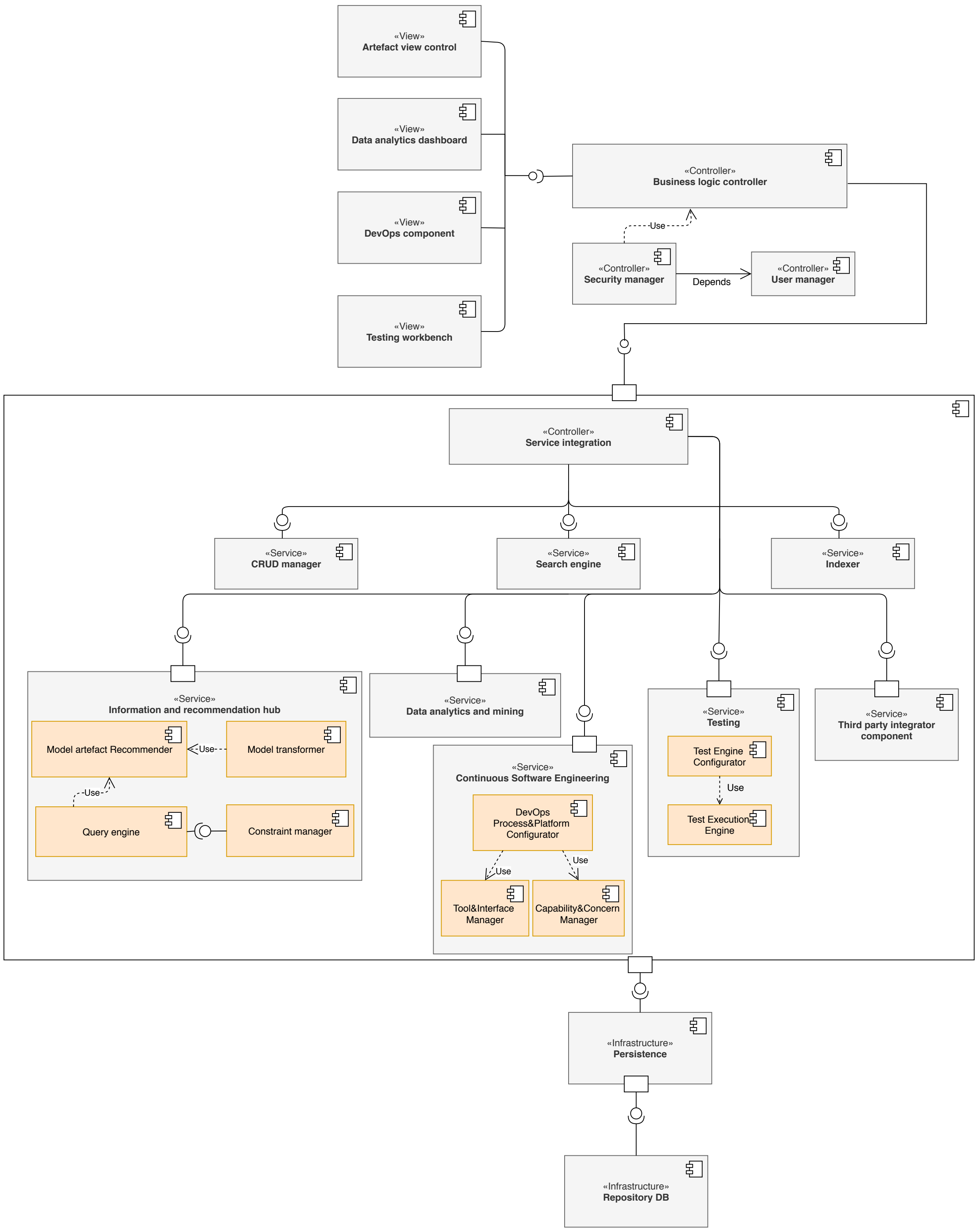


Figure 14: Logical view

- **Model transformer:** Once relevant recommendations are retrieved from the repository they are transformed to the format that is suitable for enabling their integration with the model under development.

- **Continuous Software Engineering**: this component acts as top level container of CSE-related services, acting as a facade for *i) citizen developers*, directly interacting with the Low-Code Engineering Repository, and for *ii) external tools* that can be integrated in a dedicated Low-Code platform configuration (see platform modeling in [31]) supporting given DevOps processes (e.g., continuous delivery [31] or testing functionalities as presented in Deliverable D4.3 [32]). In particular, the `DevOps Process and Platform Configurator` component will be in charge of two main activities: *i)* supporting configuration of DevOps processes and platforms [31] based on available libraries of tools (with their interfaces), according to given requirements (e.g., required capabilities to address given concerns [31]), and *ii)* collecting tool descriptions and their supported processes in shared libraries (e.g., tool descriptions created by tool providers [31]). In order to perform its functionalities, the `DevOps Process and Platform Configurator` will be supported by dedicated manager components (`Tool and Interface Manager`, and `Capability and Concern Manager`) with dedicated functionalities for existing libraries (see Fig.13), e.g., collection of predefined queries and recommendations for process and platform models.

- **Testing**: This service provides facilities for quality assurance of low-code systems. 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)`, which is a standardized language for abstract test case definition [33]. The citizen developer can design test cases by specifying TDL models. TDL is not executable by itself, but the engine makes TDL models executable. It also performs automatic configuration, execution, and evaluation of functional tests in the model level. The engine can be configured for various Domain-Specific Languages (DSLs) throughthe `Test Engine Configurator` component, so the testing service could be customized and reused by different LCDPs.

3.2.3 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. 20, and the main elements are described in Table 4.

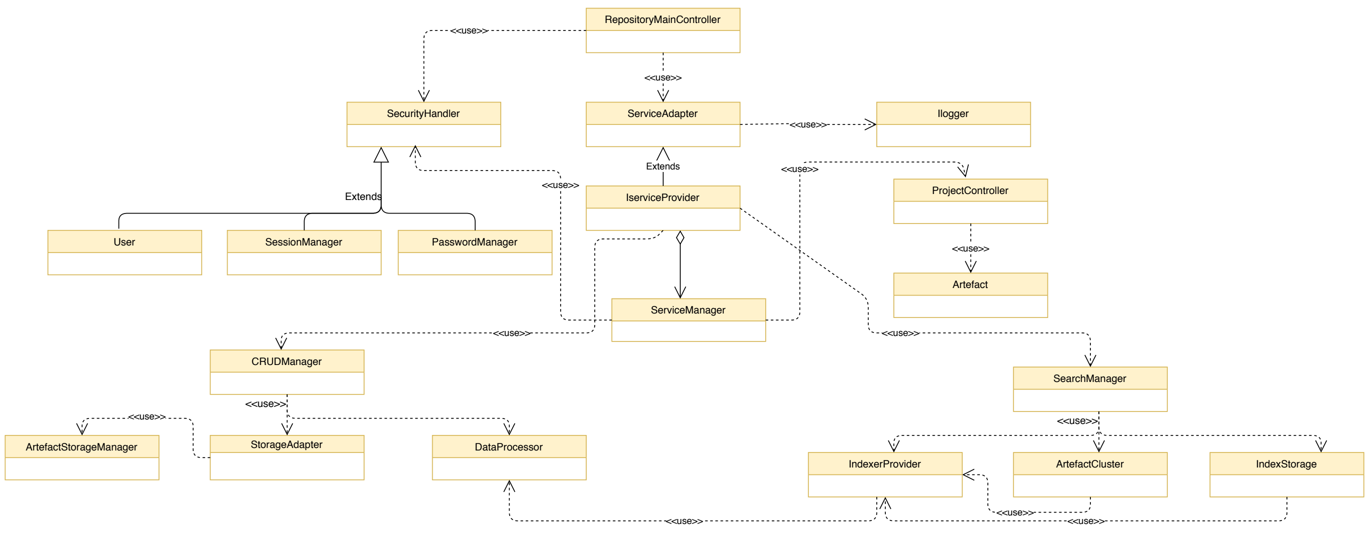


Figure 15: Development view

Table 4: 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.

3.2.4 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. In this section, representative processes will be presented concerning provided model management, continuous software engineering, model recommendation, and quality assurance services.

3.2.4.1 Model management services As explanatory services we shows the sequences of steps that are performed when searching modeling artifacts and when uploading new ones.

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. For each service against another service, authorization will be performed to validate sessions and tokens. As seen in the Fig 16, the service integration will be the access component because internal components will not be accessed out of 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.

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 authentication of the user. We will save the artefacts in a database and organize them accordingly as show in Fig 17. Metadata from the data transfer object will be saved at the persistence component, which captures the context of a given use case.

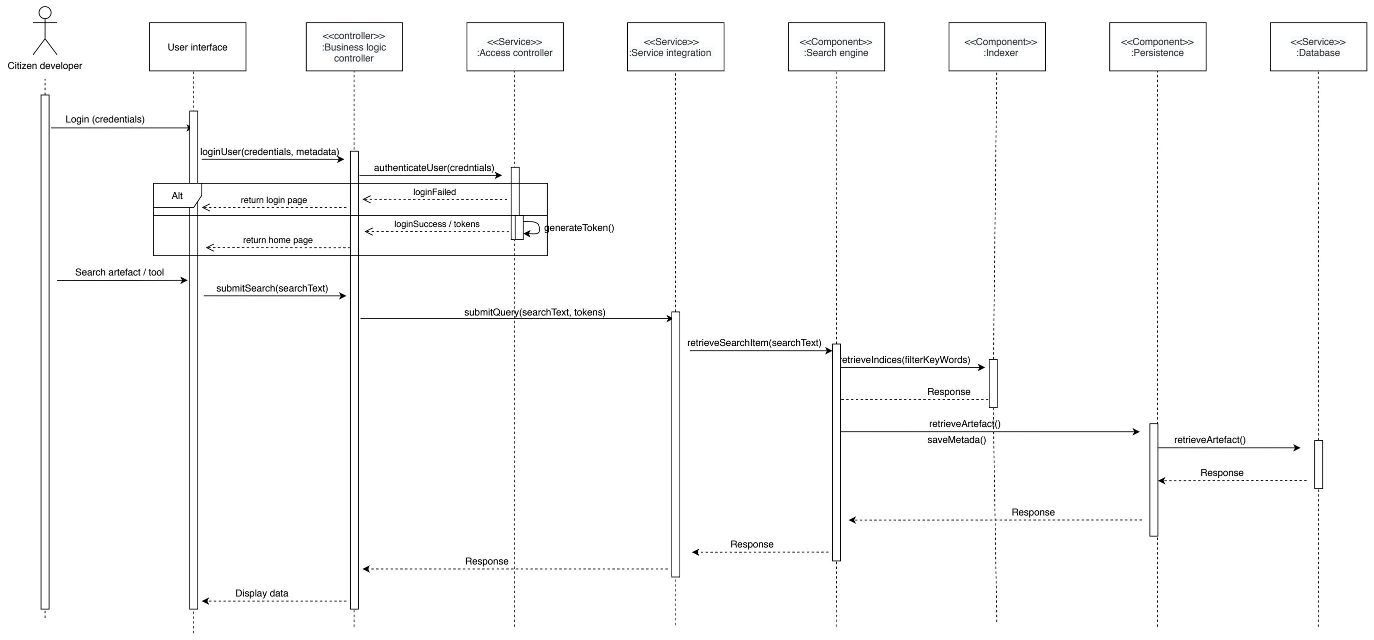


Figure 16: Search model artefact process view

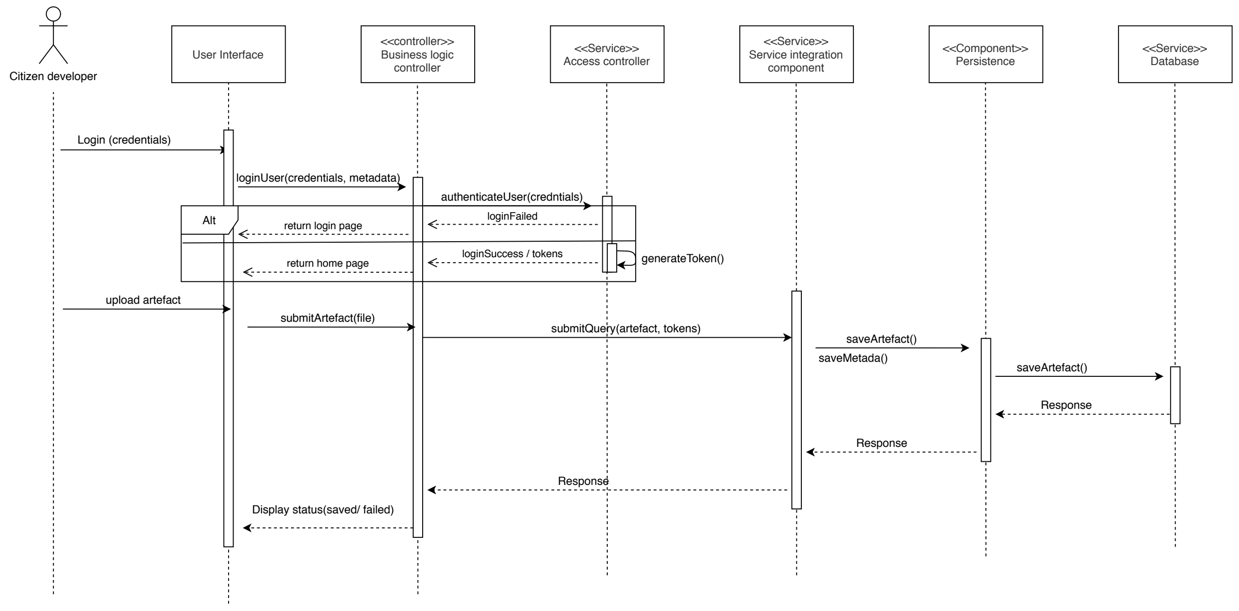

Citizen developer
User Interface
<<controller>> Business logic controller
<<Service>> Access controller
<<Service>> Service integration component
<<Component>> Persistence
<<Service>> Database
Login (credentials)
loginUser(credentials, metadata)
authenticateUser(credntials)
Alt
loginFailed
return login page
loginSuccess / tokens
generateToken()
return home page
upload artefact
submitArtefact(file)
submitQuery(artefact, tokens)
saveArtefact()
saveMetada()
saveArtefact()
Response
Response
Response
Display status(saved/ failed)

Figure 17: Upload model artefact process view

3.2.4.2 Continuous Software Engineering services The Low-Code Engineering Repository will provide CRUD functionalities for the model-driven artifacts of the DevOpsML framework presented in [31], and a prototypical implementation of the DevOpsML framework is available [34]. In [31] 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 18 and 19 provide an 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 an 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 [31], the process modeling activity is expected to be mostly supported by existing DSL and functionalities provided by LCPDs [35]. In [31], the OMG SPEM language [36] and compliant modeling tools (e.g., SPEM plugin for MagicDraw UML[15]) have been used. A Platform Model artifact is obtained at the end of the interaction depicted in Figure 18, 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

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

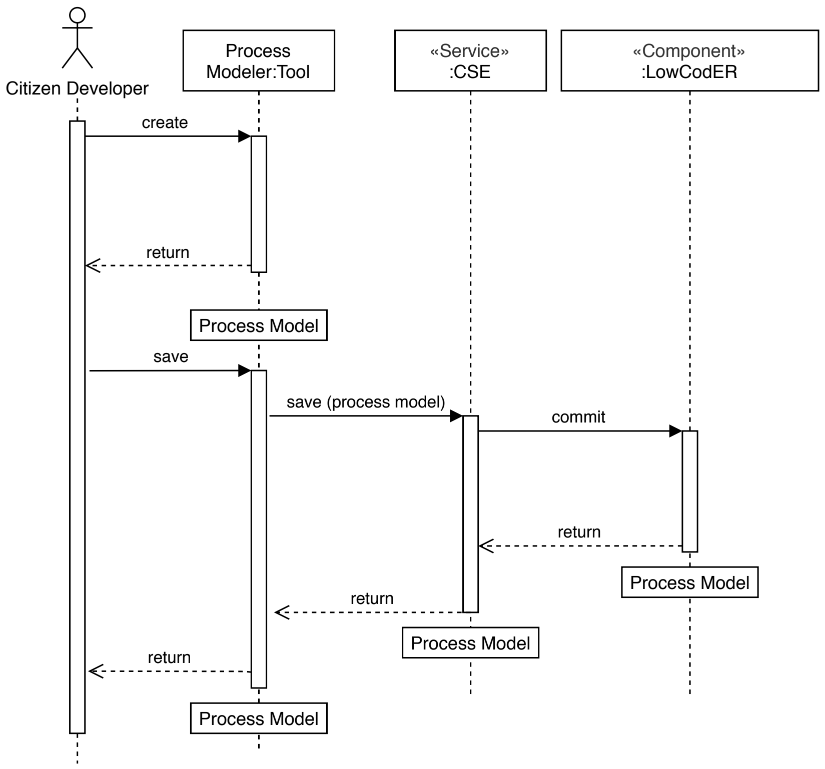


Figure 18: Creating a Process Model from an external Tool and storing it in the Low-Code Engineering Repository through the CSE service.

modeling tools. In [31], 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 separated libraries.

Figure 19 depicts a high-level interaction scenario with a citizen developer that creates library of platform elements. According to the DevOpsML framework [31], D4.3 [32] shows how the citizen developer can play different roles (see Deliverable D4.3) 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
- *as tool provider*, she can store the model of her preferred/known tools as a collection of provided interfaces and capabilities addressing given concerns[16].

The `Tool&Interface` and `Capability&Concern Manager` components (as shown in Figure 3) are in charge of providing repository-specific functionalities to support the DevOps Platform Configuration use case (Figure 13), like collecting statistics and suggesting recommendations of candidate platform elements for suitable configurations with respect to given requirements. For this purpose, if applicable, data and process mining techniques will be invoked on available MDE artifacts (i.e., the data) and CSE processes.

3.2.4.3 Model discovery and reuse services As shown in Fig. 20, after a citizen developer logs in into the repository, she can discover relevant domain models in order to reuse these models or at least to do not start modeling from scratch. The discovering process will be 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. If the user selects to use any given recommendation then the selected options will be transformed into the model under construction format and it will be merged with the model in the respective context.

3.2.4.4 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. 21), and to get notifications in

[16]Feature models of LCDPs [35] and existing classification of DevOps tools [37] can be used for this task.

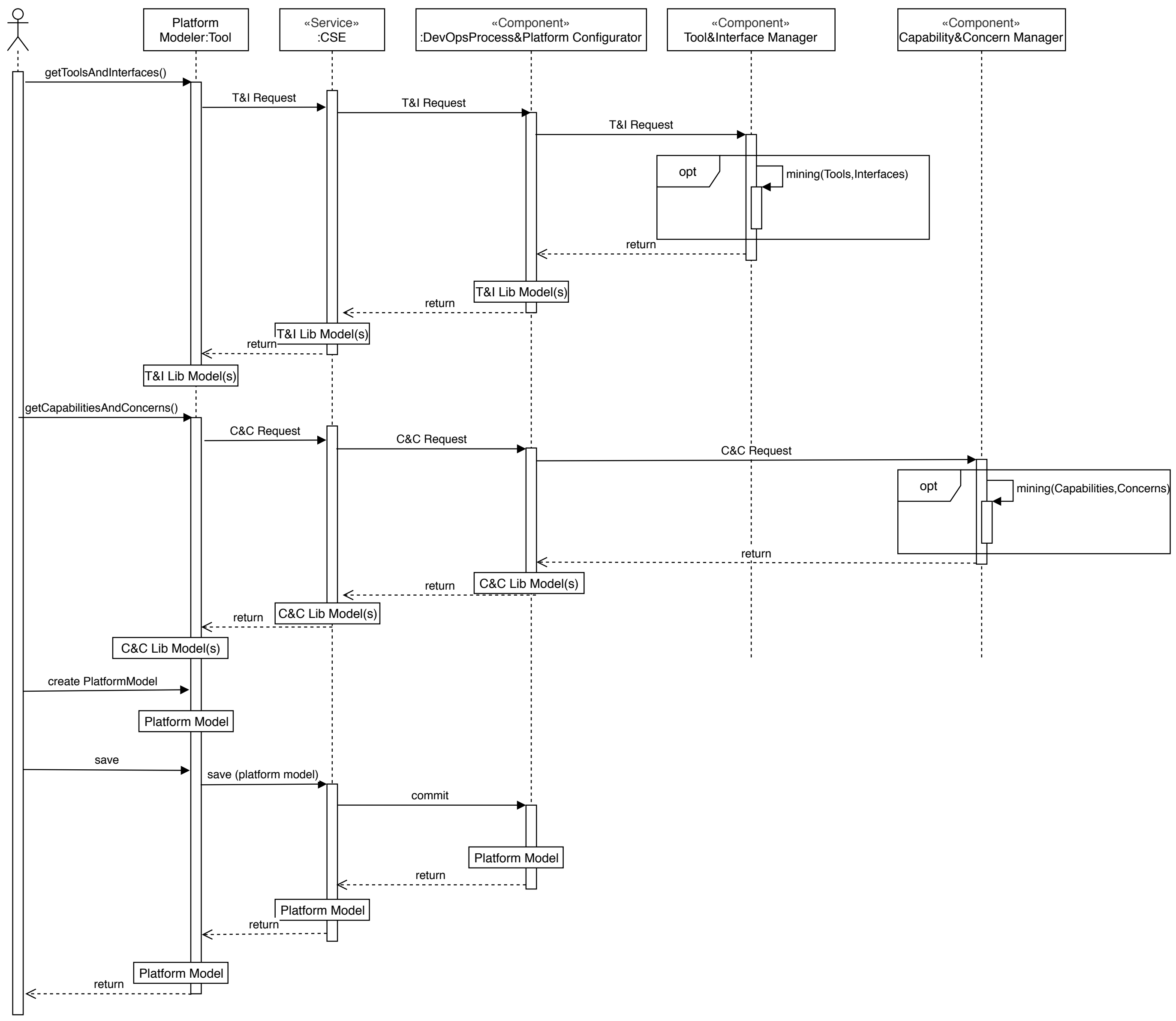


Figure 19: Creating a Platform Model from an external Tool via reusable libraries.

case of changes occurred in the model of the SUT (see Fig. 22). 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. 3.2.2), and the Low-Code repository (hereafter named *LowCodeER* for convenience).

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

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.

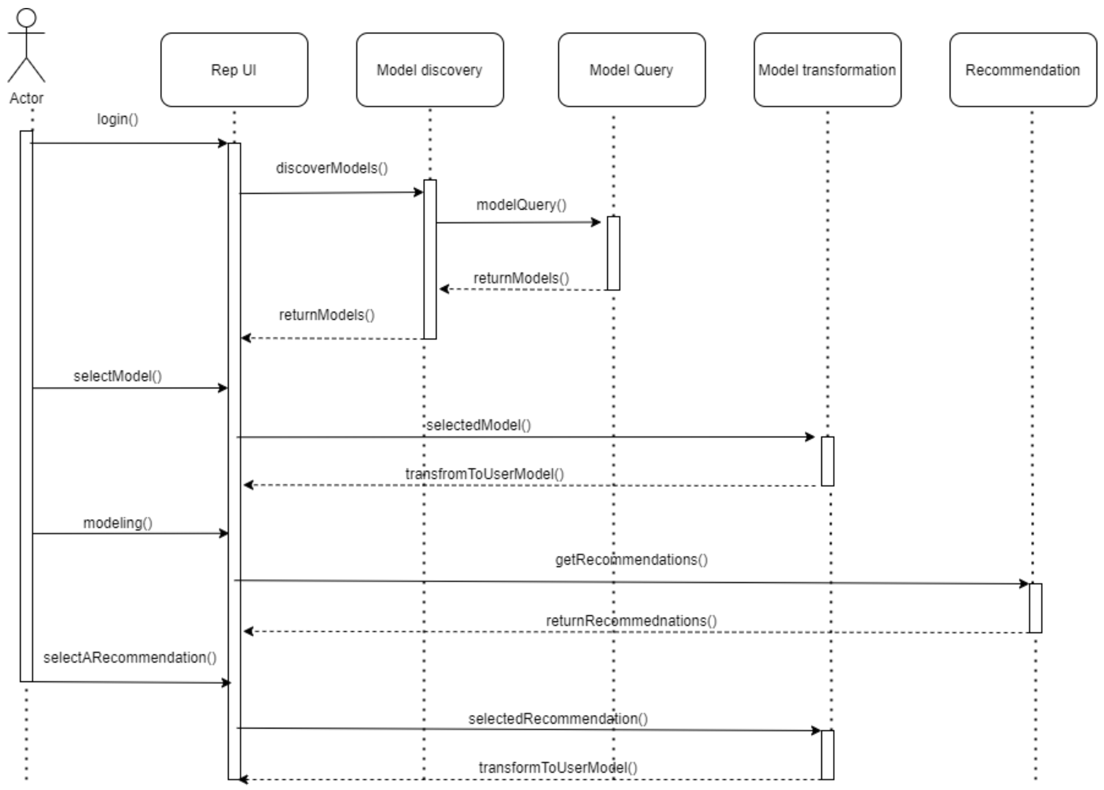


Figure 20: Model discovery and reuse

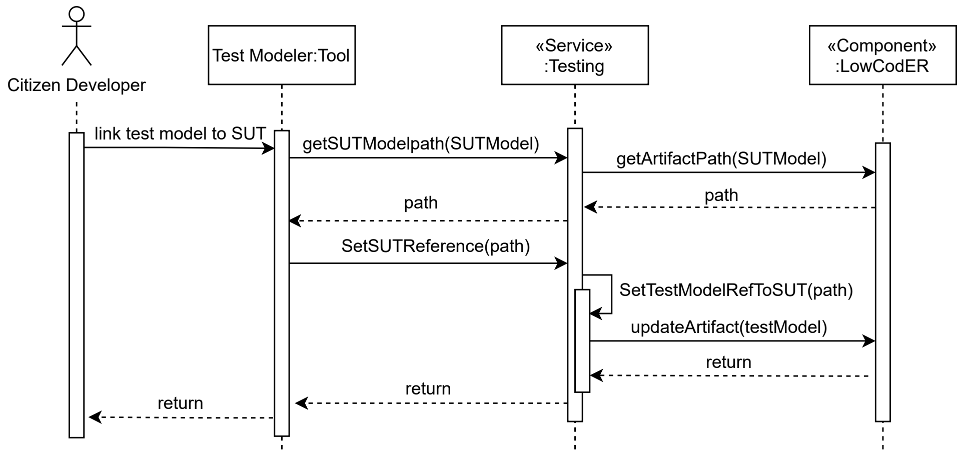


Figure 21: Setting the links between test models and their related system models (SUT models)

In both cases, appropriate notifications will be sent to the citizen developer through the modeler tool.

3.2.5 Data view

The data view, captures the essentials of the the data transfers that are exchanged between system processes. Since we intend to implement services on the repository as remote services, that implies numerous round-trip calls between client and the server that shall be performed by underlying processes.

Figure 23 shows the data transfer object (DTO) that aggregates the data repeatedly transferred by several calls between processes into a single object. This object will be stored, retrieved, serialized for transfer and deserialized for consumption. The manipulation of the DTO will be done at the persistence component that coordinates the persistence layer of the system. The response has two concepts, i.e. the Message and its Content. The Message is a JSON object with contained metadata and the content can be of any form depending on the involved service.

3.2.6 Physical view

The physical view describes the multi-tier architecture of the Lowcomote repository as shown in Fig. 24. According to the high level architecture diagram, Fig 10 shows different physical components of the

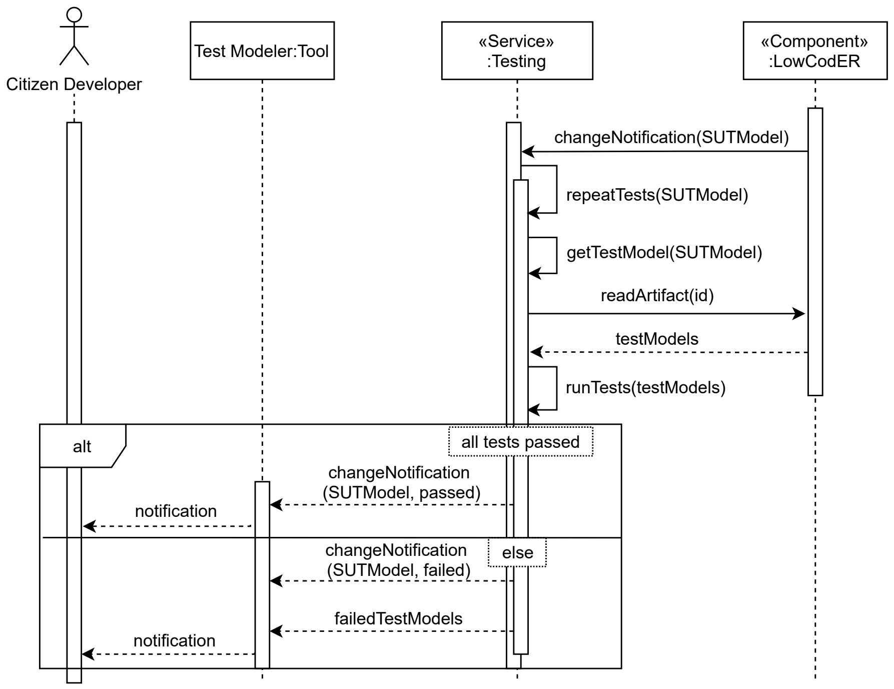


Figure 22: Getting notification of system model changes along with the related failed test models

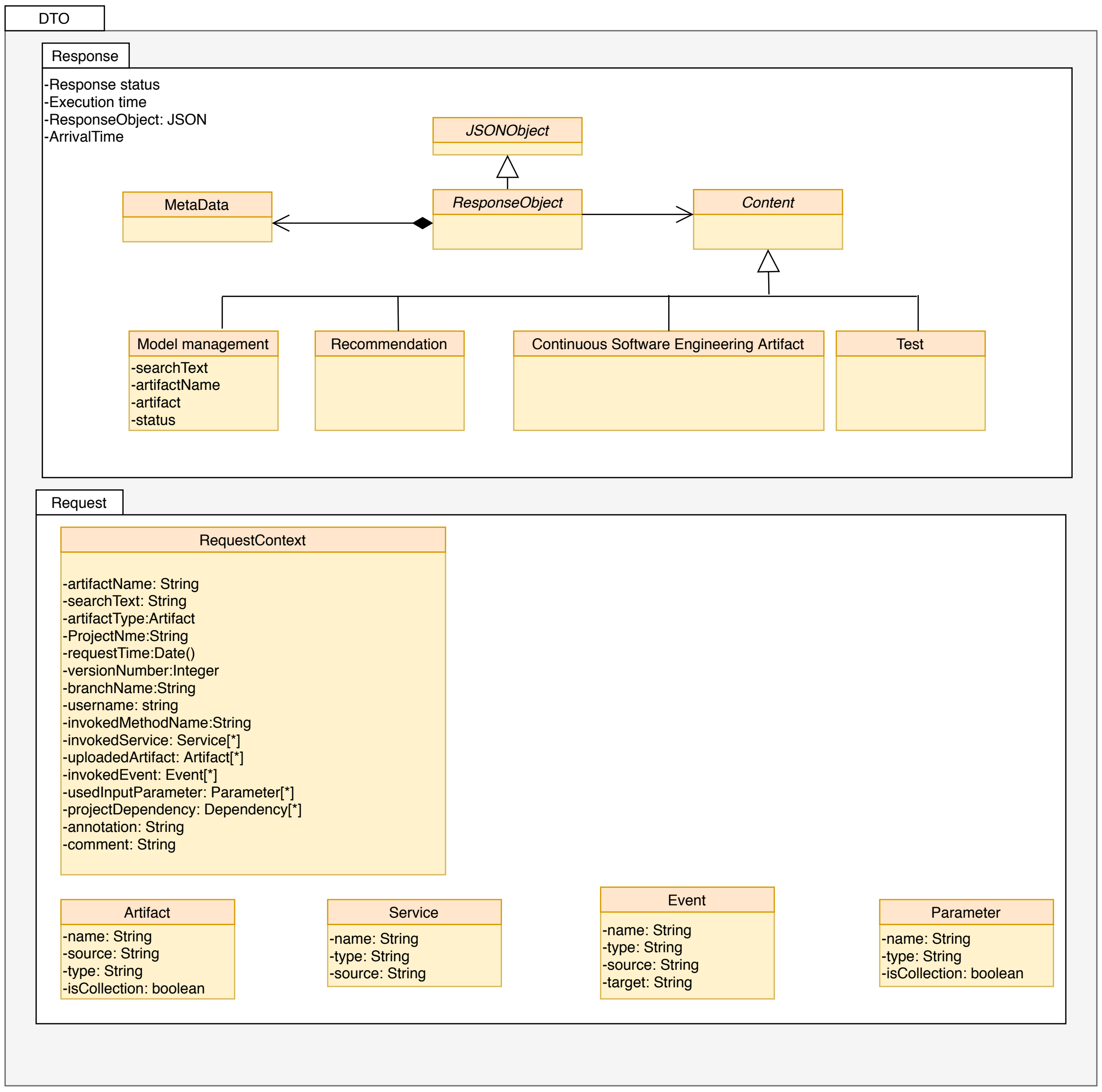


Figure 23: Data view

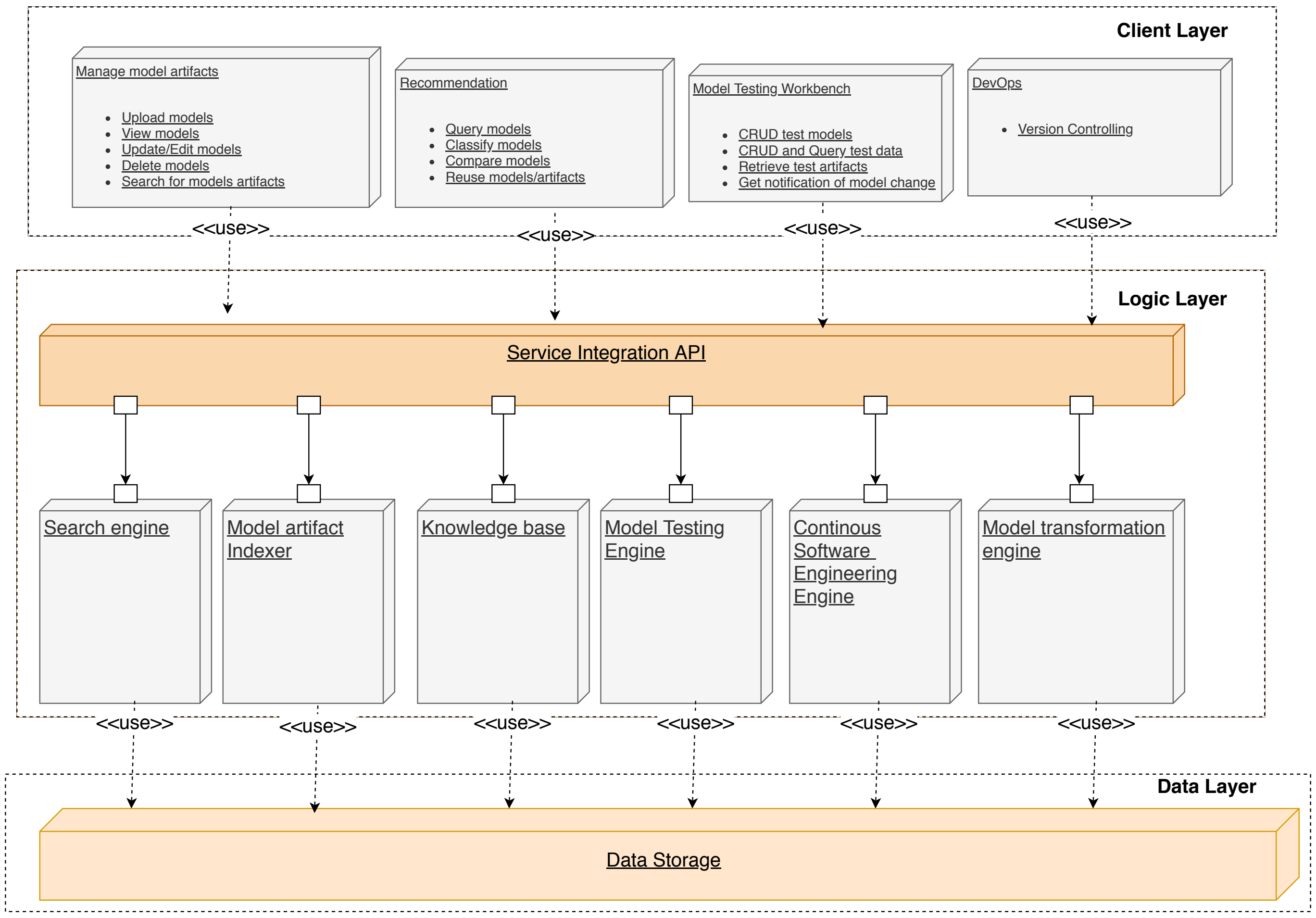


Figure 24: Physical view

system. It is important to remark that the physical view presented here is evolving, thus in the forthcoming iterations, it shall be updated accordingly.

The client will exploit the functionalities that will be developed by the different project partners especially those involved in the Work Package 4. The services that will be available in the *Client Layer* includes 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 will play the role of gateway for all the other services, which in turn will be able to access the data layer providing storage facilities.

4 Conclusion

Over the last years, the interest around LCDPs has significantly increased from both academia and industry. In this document, 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.

The performed analysis has been preparatory to understand the strengths and the limitations of the existing LCDPs. In particular, we took advantage of the experience gained in understanding the different LCDPs to define the architecture of the envisioned repository of the Lowcomote development platform. An extended *"4 + 1"* view model has been adopted to present the architecture of the repository that will be developed in the context of WP4 and that will support different services including model management, continuous software engineering, model recommendations, and quality assurance.

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

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

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

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

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

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

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

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

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

[16] 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.

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

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

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] Lowcomote EU Project Consortium. D4.3. concepts for testing in low-code engineering repositories. https://www.lowcomote.eu.

[33] 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.

[34] Linz BISE Institute, JKU. Devopsml, 2020. https://github.com/lowcomote/devopsml/tree/1.2.2, last accessed on 28/08/20.

[35] 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.

[36] OMG. Spem, 2008. https://www.omg.org/spec/SPEM/About-SPEM/, last accessed on 28/08/20.

[37] 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.