Low-code development and model-driven engineering: Two sides of the same coin?

Davide Di Ruscio, Dimitris Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, Manuel Wimmer

► **To cite this version:**

Davide Di Ruscio, Dimitris Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, et al.. Low-code development and model-driven engineering: Two sides of the same coin?. Software and Systems Modeling, 2022, 21 (2), pp.437-446. 10.1007/s10270-021-00970-2 . hal-03916536

HAL Id: hal-03916536
https://hal.science/hal-03916536

Submitted on 30 Dec 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Low-code development and Model-driven engineering:

Two sides of the same coin?

Davide Di Ruscio · Dimitris Kolovos · Juan de Lara · Alfonso Pierantonio · Massimo Tisi · Manuel Wimmer

Received: date / Accepted: date

Abstract The last few years have witnessed a significant growth of so-called *low-code development platforms* (LCDPs) both in gaining traction on the market and attracting interest from academia. LCDPs are advertised as visual development platforms, typically running on the cloud, reducing the need for manual coding and targeting also non-professional programmers. Since LCDPs share many of the goals and features of model-driven engineering approaches, it is a common point of debate whether low-code is just a new buzzword for model-driven technologies, or whether the two terms refer to genuinely distinct approaches. To contribute to this discussion, in this expert-voice paper, we compare and contrast low-code and model-driven approaches, identifying their differences and commonalities, analysing their strong and weak points, and proposing directions for cross-pollination.

Keywords Low-code development · No-code development · Model-driven engineering

Davide Di Ruscio, Alfonso Pierantonio
University of L'Aquila
E-mail: firstname.lastname@univaq.it

Dimitris Kolovos
University of York
E-mail: dimitris.kolovos@york.ac.uk

Juan de Lara
Universidad Autónoma de Madrid
E-mail: juan.delara@uam.es

Massimo Tisi
IMT Atlantique, LS2N (UMR CNRS 6004)
E-mail: massimo.tisi@imt-atlantique.fr

Manuel Wimmer
Johannes Kepler University Linz
E-mail: manuel.wimmer@jku.at

1 Introduction

Low-code development platforms (LCDPs) are on the rise, with an increasing number of cloud vendors, such as Google, Microsoft, and Amazon, offering solutions for developing and operating complex software applications with little or no code. The main aims of LCDPs are to reduce the development and maintenance effort required to deliver and operate certain types of applications and to enable digital-savvy *citizen developers* who lack or have limited programming experience to contribute to the software development process directly.

As model-driven engineering (MDE) [3] has similar aims, there is an ongoing debate on how low-code software development is different from model-driven engineering and to what extent work carried out in the field of MDE is directly transferable to LCDPs [6]. In this paper, we aim at clarifying the commonalities and differences between both approaches. We argue that while the two approaches share similar high-level aspirations, for instance, not all model-driven techniques aim at reducing the amount of code needed to implement software solutions, and not all low-code approaches are model-driven.

The rest of the paper is organised as follows. In Section 2, we summarise the history of the low-code movement we have seen so far. In Section 3, we provide an overview of typical low-code development processes and tools that LCDPs offer. In Section 4, we contrast and compare the principles and practices of low-code development and MDE. In Section 5, we discuss possible reasons behind the increasing adoption of LCDPs. In Section 6, we identify lessons that the two communities can learn from each other. Finally, Section 7 summarises and concludes the paper.

2 The history of low-code development

The past decades have been marked by several industry trends aiming at reducing the amount of handcrafted code required to produce software such as 4GLs in the 80's [28], Rapid Application Development in the 90's [29], End-User Development in the 00's [26], and MDE in the last two decades [54].

The first use of the term *low-code* can be traced back to the market analysis firm Forrester in 2014 [43] (cf. Fig. 1), where *low-code development platforms* (LCDPs) were defined as "platforms that enable rapid delivery of business applications with a minimum of hand-coding and minimal upfront investment in setup, training, and deployment." It is interesting to note that this report identified the LCDP segment as specific to the production of *business applications*, such as software for accounting, customer relationship management, human resource management, outsourcing relationship management, field service, enterprise resource planning, enterprise content management, business process management, product lifecycle management, and other productivity-enhancing applications. In 2016, Forrester detailed the successful application domains for LCDPs in four specific application scenarios, i.e., database, request-handling, process, and mobile-first [49].

The definition has evolved, and in 2017, Forrester provided a more detailed version, characterizing LCDPs as "products and/or cloud services for application development that employ visual, declarative techniques instead of programming and are available to customers at low- or no-cost in money and training time to begin, with costs rising in proportion of the business value of the platforms" [45]. The focus here is on visual interfaces and declarative techniques, with Forrester especially emphasizing visual WYSIWYG development and model-driven development [17]. The focus on the *platform* is highlighted as a key differentiating aspect of these solutions with respect to the previous generation of declarative tools: LCDPs are platforms first, with features for application deployment and life-cycle management, as well as platform management [50].

Gartner identified a similar segment from 2016, called low-code application platform (LCAP) [58]. In particular, it introduced *enterprise LCAPs*, which aim at producing enterprise-class applications requiring high performance, scalability, high availability, disaster recovery, security, SLAs, resource use tracking, technical support from the provider, and API access to and from local and cloud services.

The year 2017 noted the start of a series of acquisitions for LCDP vendors as reported in [46]. Appian started an initial public offering (IPO) in May 2017 with a revenue of $150 million, and on August 1, 2018, Appian's market valuation nearly reached $2 billion. In July 2018, OutSystems received investments of $360 million and was rated with $100 million in annual revenue. In August 2018, Siemens announced the acquisition of Mendix for $730 million in cash or at least 7–8 times its annual revenue [46].

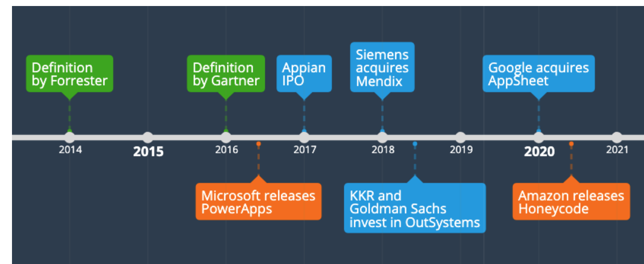

Fig. 1 Major events in low-code history

In 2017, Forrester estimated a market size for LCDPs of $3.8 billion[2]. Forrester also periodically surveys developers about LCDP usage[1]: In 2018, 23% of developers reported using low-code platforms, and another 22% planned to do so within a year [48]. In 2019, 37% of developers were using or planning to use low-code products [32].

In 2021, most large cloud providers offer LCDPs within their cloud-based solutions. Microsoft was among the first to embrace the trend by releasing its Power Apps LCDP in November 2016. In January 2020 Google acquired the LCDP provider AppSheet, and made it its flagship low-code solution. In June 2020 Amazon released Honeycode, an LCDP for web and mobile application development.

No-Code Development Platform (NCDP) is a related term used for platforms that eliminate the need for programming using visual languages, graphical user interfaces, and configuration. While the term is widely used in marketing, market analysis firms currently oppose using it to identify a clear market segment [47]. For the context of this paper, we consider NCDP and LCDP interchangeably, and consequently, hereafter, we use the term LCDP only.

3 Overview of low-code development platforms

In this section, we present an overview of the most significant LCDPs by considering the typical steps that

[1] The survey included more than 3 thousand developers in Australia, Canada, China, France, Germany, India, the UK, and the US. Developers participated with small material incentives.

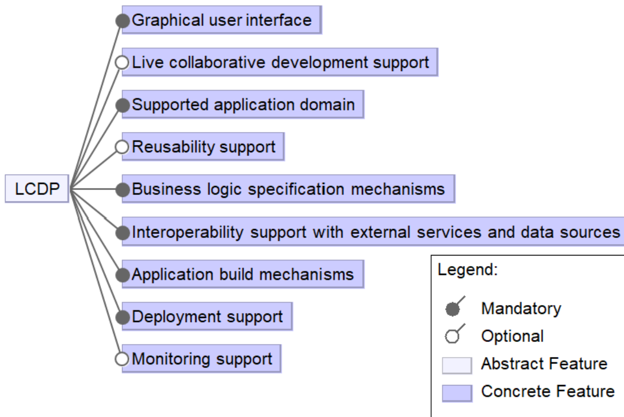

Fig. 2 Top-level features of LCDPs (refinement of [51])

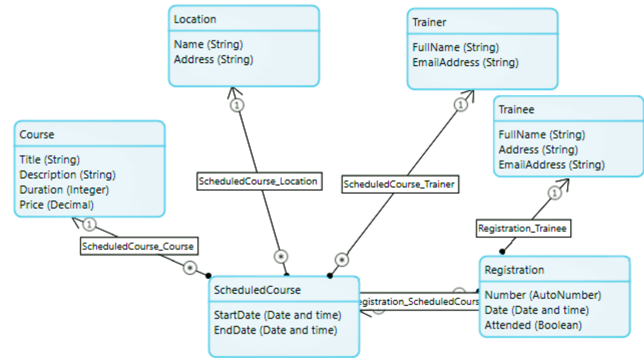

Fig. 3 A simple domain model specified in Mendix [51]

are performed when using them and by relying on a refined taxonomy originally presented in [51].

LCDPs support the development of applications that can be web-only or also native for the target deployment environments. Thus, they can natively support both desktop and mobile devices, and integrate with existing workflows developed with popular Software-as-a-Service (SaaS) applications, including Zapier, Amazon AppFlow, and Trello to mention just a few. Appian [2] is among the most long-lived LCDP, whereas Amazon Honeycode [1] and Google AppSheet [15] are among the most recent approaches.

Particular characteristics that distinguish existing LCDPs pertain to the user experience of advanced `Graphical user interfaces` (see Fig. 2) providing tools and widgets to enable citizen developers to conceive the desired applications. Drag-and-drop facilities, advanced reporting features, decision engines for modelling complex logic, and form builders are just examples of functionalities provided in the front-end of LCDPs. Moreover, LCDPs can give the users some `Live collaborative development support` to help developers that are geographically distributed and that want to work on the same applications collaboratively. Another distinguishing aspect of existing LCDPs is related to the `Supported application domain` intended to be the primary focus of interest. For instance, the main focus of Node-RED [34] is supporting the development of IoT applications. Other platforms support the development of chatbots [40], whereas the majority of existing LCDPs aim at being general-purpose supporting the development of any data-intensive application.

LCDPs can provide users with pre-defined artifacts, which can be used as starting points. This is reflected by the `Reusability support` feature shown in Fig. 2. For instance, *Salesforce App Cloud* [52] includes the extensive AppExchange marketplace [53] consisting of pre-built applications and components, reusable objects and elements, drag-and-drop process builder, and in-built Kanban boards.

As discussed in [51], realizing software systems with LCDPs encompasses several tool-supported steps, which are summarised below.

Domain modeling. In this phase, users are provided with modeling constructs to represent concepts and relationships underpinning the application being developed. Fig. 3 shows a simple domain model specified with the Mendix platform to describe training courses. Pre-built templates can be exploited as starting point, and interactive application analytics are provided out of the box. Other tools follow a similar approach. For example, Codebots [7] uses UML to specify domain models that are consumed to automatically generate target artifacts, including complete REST APIs, client libraries, Swagger API documentation, and a JSON Schema definition for each domain object.

User interface definition. Users define data forms and pages to create, edit, and visualize data that the application under development will manage. Fig. 4 shows a form-based screen in Microsoft Power Apps.

Business logic specification. Users define the control and data flows of the system under development through intuitive `Business logic specification mechanisms`. Graphical workflows and textual business rules are examples of business logic specifications that typically make use of one or more API call(s). Fig. 5 shows a simple Node-RED workflow. Node-RED implements a programming model that permits developing event-based applications, which can be specified by a wide range of node types available in an extensible palette. Workflow specification is also prominent in *Kissflow* [24], which mainly focuses on workflow automation for small businesses.

Integration with external services. LCDPs typically provide `Interoperability support with external services and data sources` to use

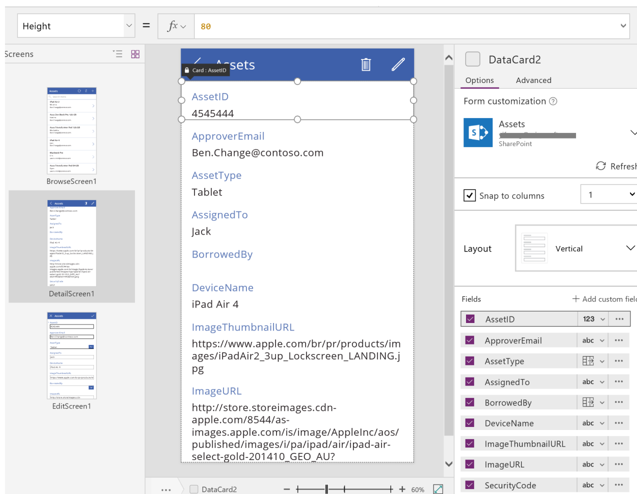

Fig. 4 User interface definition with Microsoft PowerApps [31]

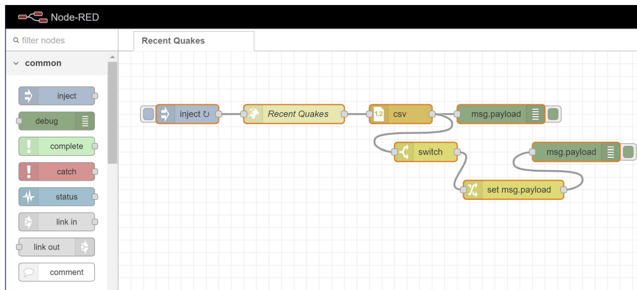

Fig. 5 Business logic specification with Node-RED [35]

services or consume data provided by third-party systems, e.g., using dedicated APIs. LCDPs can consume services provided by external providers such as Dropbox, Zapier, Office 365, and Google Drive. Thus, users might connect or integrate such services to build forms or to compile data reports. For instance, Fig. 6 shows the page in Zoho Creator [62] to configure the connection with Google Drive.

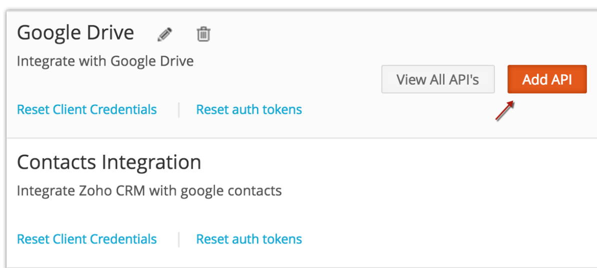

Fig. 6 Configuring the Google Drive connector in Zoho Creator [63]

Application generation and deployment. The next step of the process consists of generating and

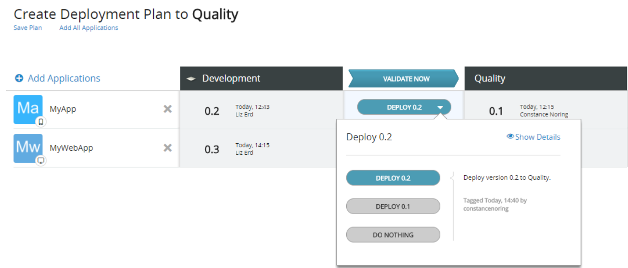

Fig. 7 Application deployment with OutSystem [38]

deploying the modeled application by means of provided `Application build mechanisms`. Several execution environments can be supported, as for instance, in the case of zAppDev [61], which provides users with different code generation facilities. Once the desired system has been specified and built, a dedicated `Deployment support` is available to deploy the system in private or public environments. Deployments are typically done on cloud infrastructures with a few clicks, as shown in Fig. 7. In particular, OutSystem [37] provides developers with quick mechanisms to publish developed applications, connect different services, and create real-time dashboards.

Application maintenance. The last step of the process is monitoring and maintaining the developed system by means of dedicated features, e.g., to react in case of unforeseen requirements that need to be addressed or fix issues that might occur during the operation of the system.

4 Low-code vs. Model-driven Engineering

Having discussed the main features of LCDPs, we compare them with MDE processes and technologies in this section.

MDE [3] encompasses software paradigms emphasizing the use of models as first-class artefacts during the development lifecycle. Hence, in MDE, models are used to specify, test, simulate, verify, modernize, maintain, understand, and generate code for the system, among many other activities. Still, not every MDE process ends with code generation but actively uses models. The goal of MDE is to increase productivity by automating different steps in software development employing models while augmenting the overall quality [19, 23]. For this purpose, MDE processes often rely on Domain-Specific Languages (DSLs), specially tailored for the domain at hand. Using domain-specific models makes descriptions more intentional and include less accidental detail than code written using general-purpose programming languages. Hence, those models

become easier to create, verify, and maintain than the corresponding low-level code.

In their turn, LCDPs promote the construction of applications using forms and graphical editors with little or no hand-crafted code. Since some of their target users are citizen developers, one of their key points is to reduce accidental complexity regarding the installation and operation of both the development environments and the developed applications. This way, they typically provide cloud-based development environments and manage the lifecycle of the designed applications (e.g., hosting, resource allocation and provisioning, usage analytics, etc.) Therefore, low-code development shares some of the goals of MDE, but there are some differences, too.

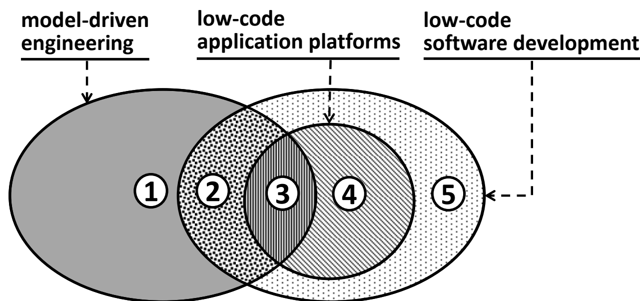

Fig. 8 Venn diagram showing commonalities and differences between model-driven approaches, low-code application platforms and low-code software development

Fig. 8 schematically illustrates the commonalities and differences between low-code and MDE approaches using a Venn diagram. The diagram represents the approaches following MDE, low-code development, and development based on low-code platforms in terms of sets. This leads to 5 regions of interest (marked as 1–5 in the figure). This way approaches termed "model-driven" by our community fall under regions 1, 2, and 3; with an overlap under 2 and 3 with low-code platforms and low-code development approaches. Instead, regions 4 and 5 are exclusively low-code, while region 1 is exclusively model-driven. The regions can be described as follows:

1. This region contains the model-driven approaches that use models as machine-processable artefacts but do not aim at reducing the amount of code required to implement the system. Instead, they focus on automating tasks like simulation, formal verification, software optimization, or reverse engineering. Examples of works in this category include the work of Cortellessa et al. [9] on analysing and refactoring UML design models for optimizing their performance; or reverse engineering tools like Modisco [5], which extracts models from code.
2. These are the approaches that use models as machine-processable artefacts and aim to reduce the amount of code required to implement a system (e.g., via code generation or interpretation) but without offering deployment or lifecycle management capabilities for the produced system. Examples of this class of approaches are JHipster [20] and its JDL [21] domain-specific language, Google Protocol Buffers [16], or the OlivaNova model execution system [39].
3. This region contains the platforms that use models to facilitate the development of software applications with reduced code and offer built-in deployment and lifecycle management facilities for the produced application. Examples include the Codebots [7] and Judo [22] low-code platforms, both of which are based on technologies of the Eclipse Modelling ecosystem [56].
4. This region, and the next one, contain approaches that cannot be considered model-driven. In particular, region 4 contains the platforms that facilitate the development of software applications with reduced code. Such approaches offer built-in deployment and lifecycle management facilities for the produced application. However, they do not use models that conform to explicitly defined languages/metamodels (e.g., they use data stored in a relational database or schema-less XML/JSON documents).
5. These approaches aim to reduce the amount of code required to implement a system without offering deployment or lifecycle management capabilities for the produced system, and – like region 4 – without using models that conform to explicitly defined languages/metamodels. Examples of this type of approach include database-schema-driven generators like Phreeze [41] and one-off generators such as those provided by Ruby on Rails [44].

Next, we elaborate on other aspects that differentiate model-based and low-code approaches, based on Fig. 8:

Platform. Low-code application platforms (regions 3 and 4 in the figure) are cloud-based: they can be used from the web browser and host the defined applications. This frees the user from both installing the development platform itself and from deploying the defined applications. This approach simplifies the adoption of low-code by newcomers. While MDE solutions can be cloud-based (falling in region 3) [8], this is not the norm today. Instead, many solutions are based on

the desktop, for example, those using the Eclipse Modelling Framework (EMF) [56], or meta-modelling tools like MetaEdit+ [30]. These approaches would fall under region 2 – and be considered low-code development approaches – if their aim is automating application development, otherwise they would fall in region 1.

Users. Low-code development platforms mainly target end-users, so-called citizen developers. Therefore low-code platforms tend to be easy to use for people with a non-technical background. This means that frequently (but not always), users of tools in regions 3 – 4 are citizen developers and non-professional programmers. For example, while low-code platforms like OutSystems target citizen developers, others like Judo target teams of business analysts, software architects, and programmers.
In their turn, MDE solutions can target end-users, but many of them are directed to professional software developers since they are expected to be used within development processes. Therefore, typically, users of approaches in regions 1 and 2 have a more technical background.

Domains. As mentioned in Section 2, the first wave of low-code targeted business applications. Recently, we are witnessing proposals for low-code tools in other domains, like IoT/event-driven applications (e.g., Node-RED [34]), chatbots (e.g., Google's Dialogflow [10], Amazon's Lex [25], IBM's Watson Assistant [59]) or Machine Learning (e.g., Google's AutoML [14] or RapidMiner [42]). MDE solutions (in regions 1–3 of the figure) can target those domains but frequently also target more technical areas, which require specialized engineers. These include domains like automotive [11], power engineering [13], or cyber-physical systems [33] in general, among many others.

5 Low-code development: why now?

In terms of their core ambition to expedite the delivery of software systems, low-code development platforms are not very dissimilar to previously tried approaches like 4GLs, case tools, etc., as already mentioned in Section 2. Essentially, they provide an environment for specifying the structure and behavior of a software system at a high level of abstraction. Such an environment shields developers from low-level concerns (e.g., specific databases, object-relational mappers, service, messaging, and security middleware). They then generate executable code that realizes the specified software system. Given the broad consensus that previous approaches were not wildly successful, why should low-code environments fare any better? There are multiple reasons why this may be the case, which we analyse next.

Cloud-Based Deployment. Beyond generating code, modern low-code platforms can also deploy the produced software systems on scalable cloud-based infrastructures and make them instantly available to users globally through web-based interfaces. This can dramatically shorten the time and effort required to release applications (and updates) to users and increase the appeal for low-code platforms as a medium for rapid application development and delivery.

Digital Native Workforce. Computer literacy has improved dramatically over the last 40 years. The basics of computer programming are taught in many countries as part of compulsory education, and the new generations of domain experts (e.g., accountants, medicinal practitioners, construction engineers) are digital natives. As a consequence, while most domain experts would require substantial training to master some part of the complexity of a CASE tool released 40 years ago, a growing number of contemporary domain experts have substantial experience with working with computers and non-trivial software, and arguably require a lot less training to use a low-code platform to implement bespoke applications.

Zero Setup. The fact that many low-code platforms are cloud-based and do not require installation of bespoke software significantly lowers the entry barrier for new users, who can evaluate such platforms and even develop and deliver small-scale applications at no cost from the familiar environment of their web browser.

Developer Shortfall. As software is becoming pervasive in all aspects of human activity, the demand for software developers has outgrown the supply of suitably skilled professionals, and the gap is constantly widening [4]. Moreover, highly skilled software developers are attracted to intellectually demanding (and financially rewarding) software systems instead of run-of-the-mill applications. This creates a growing gap for business applications that would be more effective than shared spreadsheets but are too expensive to implement and maintain manually.

Training Facilities. The media through which users learn have also changed considerably recently. A couple of decades ago, the primary learning media for application development environments were books written by technology experts. This landscape has changed dramatically with the growth of the web and, particularly, video sharing services such as YouTube, making it easier to deliver up-to-date training material aimed at different audiences. This enables citizen develop-

ers to develop and share their own training material (e.g., walk-throughs, screencasts) rather than acting as passive consumers.

6 What MDE can learn from low-code and vice versa

Based on the previously presented insights in low-code development and MDE, we will now discuss what the two approaches can learn from each other to tackle critical challenges for their future developments.

Generic vs. specific platforms. Many LCDPs attempt to cover a wide range of applications through an ever-growing library of highly configurable components. In the MDE community, it is widely accepted that in many cases, smaller domain-specific languages can be more beneficial for engagement with domain experts and automated reasoning and processing than large and complex all-encompassing languages such as UML. An open question is if the current generation of domain-agnostic LCDPs will increasingly struggle as they keep growing in complexity. This can give rise to domain-specific LCDPs in the future, which will target specific classes of systems and citizen developers. Here an opportunity is about reusing the rich technological infrastructure offered by MDE for building domain-specific platforms. Interestingly, while MDE is often referred to as an essential building block of low-code in the Forrester and Gartner reports, there is little evidence that existing LCDPs use technologies (predominately Eclipse-based) commonly used in the MDE community. Thus, it seems to be an opportunity to speed up the development of LCDPs with MDE technologies if the latter are ready to run on the web/cloud and can deal with the requirements of typical LCDP users.

Opening up Web/Cloud-based Platforms. A lesson that the MDE community can learn from the success of LCDPs is that web-based interfaces can significantly improve uptake and engagement with domain experts. A transition of core MDE technologies is underway with frameworks such as Xtext [60] and Sirius [55] providing web-based counterparts. However, significant effort is still required to realise the vision of zero-installation web-based MDE workbenches. Some efforts already started to reuse open source technologies for building up LCDPs [36]. As there is currently already a trend to migrate MDE technologies to the web/cloud, there may be an opportunity to develop the next generation of LCDPs with existing MDE technologies such as metamodeling frameworks for language engineering, code generators for producing the final applications, etc. [57]. This may be further supported by current initiatives for building open source cloud platforms such as GAIAX [12], which is especially important for long-living software systems.

Counteracting vendor lock-in. Since the introduction of CASE tools, one of the major concerns is the potential for vendor lock-in, i.e., application development and deployment are bound to a particular technology. While this may not be considered as a potential problem in the short term, it can become critical in the long term. For instance, consider migrating projects from CASE tools to MDE tools or projects developed with Rapid Application Development (RAD) approaches to modern cloud platforms. In the context of low-code such issues may also occur, e.g., imagine a LCDP produces applications that only work with a specific cloud provider's technology stack (cf. cloud vendor lock-in). Nevertheless, there are even more important aspects related to the development artefacts. First of all, is export of the development artefacts possible, and if it is, how can these artefacts be reused, imported, and interpreted in other platforms? The MDE community has invested substantial effort in this respect by providing dedicated standards for modeling languages (e.g., UML, BPMN), and even meta-modeling languages (e.g., MOF, Ecore), model exchange standards (e.g., XMI and HUTN), etc. It has to be explored if these approaches may also be reused for LCDPs or if other means are needed to prevent vendor lock-in.

Fostering ecosystems. Providing a LCDP is the first step, but then an ecosystem for this platform is required to ensure the continuous growth of a healthy user base. This may be even more important for LCDPs as professional developers as well as citizen developers may be targeted. Thus, the availability of documentation, support, consultancy, reusable components, etc., is of major importance. In MDE, such an ecosystem was triggered by Eclipse, i.e., a large and active ecosystem around the Eclipse Modeling Framework was established from the industrial and academic sides. It has to be further explored how such ecosystems will develop for LCDPs, as most current platforms are single vendor efforts. This issue also concerns the academic area, where scientific community efforts are required to stimulate research on topics related to low-code [57]. For instance, a current example is the low-code workshop [27] hosted with the MODELS conference since 2020, which provides a forum to discuss low-code development and MDE.

Managing software evolution. Notably, one of the most crucial stages of the software lifecycle is the maintenance of a software product after its release. Providing support to such activities requires the ability to grow in functionality and size without unwanted side effects satisfying new requirements emerging from the routine usage of the product. Managing software evolution processes in LCDPs is an interesting line of research since these platforms are managed and allow cloud-based monitoring of the developed applications. Consequently, the platform provider should offer as much support for evolution as possible. However, this may involve many different aspects. Considering the application level, we may need support for model/data co-evolution, e.g., the data model is changing and there are already running instances of the application in usage. Evolution also applies on the language level, which has been extensively researched in MDE, and is often referred to metamodel/model co-evolution [18]. Here, the problem applies both to low-code and MDE approaches. Low-code will only be successful if applications developed with low-code approaches can evolve for a longer time in combination with the LCDPs themselves.

7 Summary

This paper compared and positioned the relatively new low-code movement against the established model-driven engineering discipline. We summarised the history of low-code so far, provided an overview of typical low-code development processes and the tools that LCDPs offer to support them, and contrasted and compared the principles and practices of low-code and model-driven engineering.

While low-code and model-driven engineering both aspire to improve software development by raising abstraction and hiding implementation-level details, we argue that the two practices are not identical. Indeed, not all model-driven approaches aim at reducing the amount of code needed to implement software solutions, and not all low-code approaches are model-driven. However, being close conceptually creates substantial potential for applying existing knowledge and cross-pollination between the two disciplines.

Acknowledgements

This work has received funding from the Lowcomote project under European Union's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement n. 813884.

References

1. Amazon Honeycode. `https://www.honeycode.aws/`. Last access in Sept. 2021.
2. Appian. `https://appian.com/`. Last access in Sept. 2021.
3. M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice, Second Edition*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017.
4. T. Breaux and J. Moritz. The 2021 software developer shortage is coming. *Commun. ACM*, 64(7):39–41, June 2021.
5. H. Brunelière, J. Cabot, G. Dupé, and F. Madiot. Modisco: A model driven reverse engineering framework. *Inf. Softw. Technol.*, 56(8):1012–1032, 2014.
6. J. Cabot. Positioning of the low-code movement within the field of model-driven engineering. In E. Guerra and L. Iovino, editors, *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings*, pages 76:1–76:3. ACM, 2020.
7. Codebots. `https://codebots.com/`. Last access in Sept. 2021.
8. J. Corley, E. Syriani, and H. Ergin. Evaluating the Cloud Architecture of AToMPM. In *Proc. MODELSWARD*, pages 339–346. SciTePress, 2016.
9. V. Cortellessa, R. Eramo, and M. Tucci. From software architecture to analysis models and back: Model-driven refactoring aimed at availability improvement. *Information and Software Technology*, 127:106362, 2020.
10. Dialogflow. `https://dialogflow.com/`. Last access in Sept. 2021.
11. I. Drave, S. Hillemacher, T. Greifenberg, S. Kriebel, E. Kusmenko, M. Markthaler, P. Orth, K. S. Salman, J. Richenhagen, B. Rumpe, C. Schulze, M. von Wenckstern, and A. Wortmann. SMArDT modeling for automotive software testing. *Softw. Pract. Exp.*, 49(2):301–328, 2019.
12. GAIAX. `https://www.data-infrastructure.eu/GAIAX/Navigation/EN/Home/home.html`. Last access in Sept. 2021.
13. A. Gómez, X. Mendialdua, K. Barmpis, G. Bergmann, J. Cabot, X. D. Carlos, C. Debreceni, A. Garmendia, D. S. Kolovos, and J. de Lara. Scalable modeling technologies in the wild: an experience report on wind turbines control applications development. *Softw. Syst. Model.*, 19(5):1229–1261, 2020.
14. Google. AutoML. `https://cloud.google.com/automl/`. Last access in Sept. 2021.
15. Google AppSheet. `https://www.appsheet.com/`. Last access in Sept. 2021.
16. Google's protocol buffers. `https://developers.google.com/protocol-buffers`. Last access in Sept. 2021.
17. J. Hammond. The Forrester Wave: Mobile Low-Code Development Platforms, Q1 2017. *Forrester Research*, 2016.
18. R. Hebig, D. E. Khelladi, and R. Bendraou. Approaches to co-evolution of metamodels and models: A survey. *IEEE Trans. Software Eng.*, 43(5):396–414, 2017.
19. J. E. Hutchinson, J. Whittle, and M. Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Sci. Comput. Program.*, 89:144–161, 2014.
20. JHipster. `https://www.jhipster.tech`. Last access in Sept. 2021.

21. JHipster's JDL DSL. https://www.jhipster.tech/jdl. Last access in Sept. 2021.
22. Judo. https://judo.codes. Last access in Sept. 2021.
23. S. Kelly and J. Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008.
24. Kissflow. https://kissflow.com/workflow/process/. Last access in Sept. 2021.
25. Lex. https://aws.amazon.com/en/lex/. Last access in Sept. 2021.
26. H. Lieberman, F. Paternò, M. Klann, and V. Wulf. *End-User Development: An Emerging Paradigm*, pages 1–8. Springer Netherlands, Dordrecht, 2006.
27. LowCode Workshop at MODELS. https://lowcode-workshop.github.io/. Last access in Sept. 2021.
28. J. Martin. *Application Development without Programmers*. Prentice Hall PTR, USA, 1982.
29. J. Martin. *Rapid Application Development*. Macmillan Publishing Co., Inc., USA, 1991.
30. MetaEdit+ by Metacase. https://www.metacase.com/products.html. Last access in Sept. 2021.
31. Microsoft Power Apps. https://docs.microsoft.com/en-us/powerapps/maker/canvas-apps/working-with-forms. Last access in Sept. 2021.
32. C. Mines. Predictions 2020: More Changes For Software Development. *Forrester Research*, 2020.
33. M. A. Mohamed, G. Kardas, and M. Challenger. Model-driven engineering tools and languages for cyber-physical systems. a systematic literature review. *IEEE Access*, 9:48605–48630, 2021.
34. Node-RED. https://nodered.org/. Last access in 2021.
35. Node-RED (workflows). https://nodered.org/docs/tutorials/second-flow. Last access in Sept. 2021.
36. OSBP. https://www.eclipse.org/osbp/. Last access in Sept. 2021.
37. Outsystems. https://www.outsystems.com/. Last access in Sept. 2021.
38. Outsystems (deploying an application). https://success.outsystems.com/Documentation/11/Managing_the_Applications_Lifecycle/Deploy_Applications/Deploy_an_Application. Last access in Sept. 2021.
39. O. Pastor, J. Gómez, E. Insfrán, and V. Pelechano. The OO-method approach for information systems modeling: from object-oriented conceptual modeling to automated programming. *Inf. Syst.*, 26(7):507–534, 2001.
40. S. Pérez-Soler, S. Juarez-Puerta, E. Guerra, and J. de Lara. Choosing a chatbot development tool. *IEEE Softw.*, 38(4):94–103, 2021.
41. Phreeze. http://www.phreeze.com. Last access in Sept. 2021.
42. RapidMiner. https://rapidminer.com/. Last access in Sept. 2021.
43. C. Richardson and J. Rymer. New development platforms emerge for customer-facing applications. *Forrester Research*, 2014.
44. S. Ruby, D. Copeland, and D. Thomas. *Agile Web Development with Rails 6*. The Pragmatic Programmers, 2019. See also https://guides.rubyonrails.org/command_line.html#bin-rails-generate. Last access Sept. 2021.
45. J. Rymer. The Forrester Wave: Low-Code Development Platforms For AD&D Pros, Q4 2017. *Forrester Research*, 2016.
46. J. Rymer. Siemens Snaps Up Mendix; Low-Code Platforms Enter New Phase. *Forrester Research*, 2018.
47. J. Rymer and R. Koplowitz. Now Tech: Rapid App Delivery, Q1 2019. *Forrester Research*, 2019.
48. J. Rymer and R. Koplowitz. The Forrester Wave: Low-Code Development Platforms For AD&D Professionals, Q1 2019. *Forrester Research*, 2019.
49. J. Rymer and C. Richardson. The Forrester Wave: Low-Code Development Platforms, Q2 2016. *Forrester Research*, 2016.
50. J. Rymer and C. Richardson. Vendor Landscape: The Fractured, Fertile Terrain Of Low-Code Application Platforms. *Forrester Research*, 2016.
51. A. Sahay, A. Indamutsa, D. D. Ruscio, and A. Pierantonio. Supporting the understanding and comparison of low-code development platforms. In *Proc. 46th Euromicro Conference on Software Engineering and Advanced Applications SEAA*, pages 171–178. IEEE, 2020.
52. Salesforce. https://developer.salesforce.com/. Last access in Sept. 2021.
53. Salesforce (AppExchange marketplace). https://appexchange.salesforce.com/. Last access in Sept. 2021.
54. D. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39:25–31, 2006.
55. Sirius. https://www.eclipse.org/sirius/. Last access in Sept. 2021.
56. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, 2008. see also https://www.eclipse.org/modeling/.
57. M. Tisi, J. Mottu, D. S. Kolovos, J. de Lara, E. Guerra, D. D. Ruscio, A. Pierantonio, and M. Wimmer. Lowcomote: Training the next generation of experts in scalable low-code engineering platforms. In *STAF 2019 Co-Located Events Joint Proceedings*, volume 2405 of *CEUR Workshop Proceedings*, pages 73–78. CEUR-WS.org, 2019.
58. P. Vincent, K. Iijima, M. Driver, W. Jason, and Y. Natis. Magic Quadrant for Enterprise Low-Code Application Platforms. *Gartner*, 2016.
59. Watson. https://www.ibm.com/cloud/watson-assistant/. Last access in Sept. 2021.
60. Xtext. https://www.eclipse.org/Xtext/. Last access in Sept. 2021.
61. zAppDev. https://zappdev.com/. Last access in Sept. 2021.
62. Zoho Creator. https://www.zoho.com/creator/. Last access in Sept. 2021.
63. Zoho Creator (third-party integration). https://www.zoho.com/developer/help/extensions/connectors.html. Last access in Sept. 2021.