In Search of the Essence of Low-Code: An Exploratory Study of Seven Development Platforms. InMODELS '21. Proceedings of the 24th ACM/IEEE International Conference on Model Driven E...

Conference Paper · November 2021

CITATIONS
0

READS
192

2 authors:

Alexander C. Bock
University of Duisburg-Essen
19 PUBLICATIONS **312** CITATIONS
SEE PROFILE

Ulrich Frank
University of Duisburg-Essen
180 PUBLICATIONS **2,629** CITATIONS
SEE PROFILE

In Search of the Essence of Low-Code: An Exploratory Study of Seven Development Platforms

Alexander C. Bock
Research Group Enterprise Modeling and Information Systems, University of Duisburg-Essen
Essen, Germany
alexander.bock@uni-due.de

Ulrich Frank
Research Group Enterprise Modeling and Information Systems, University of Duisburg-Essen
Essen, Germany
ulrich.frank@uni-due.de

***Abstract*—Rapidly growing attention has been directed in recent years toward a type of software development and execution environment now passing under the name of 'low-code development platforms.' The fundamental claim is that limiting traditional coding mechanisms in favor of a variety of alternative means of design and specification yields substantial efficiency gains in professional and private software development. But although much stir at present surrounds low-code development platforms, it is by no means clear what, if any, features are distinctive of these systems, and whether any of these features mark out a technology which can be considered original. This paper presents an exploratory study of seven low-code development platforms, with the aim of discovering their essence and assessing them critically in the light of research in information systems development. An analysis framework covering a number of criteria regarding professional information systems development is used to characterize the selected platforms, and to point out features commonly, occasionally, and rarely possessed by them. The study reveals that hardly any features of low-code development are innovative in and of themselves, with novelty primarily consisting in their combination and integration. Still, we argue in conclusion, a number of research opportunities can be made out with an eye on the leitmotif of low-code development.**

***Index Terms*—Low-Code, Information Systems Development, Integration, Reuse, Abstraction.**

I. Introduction

'Low-code development platforms' (LCDP) are said to offer a number of advantages over classical approaches to software systems development. One dominant claim is that low-development platforms can contribute to the *efficiency* of software development, indeed, that they hold the prospect of undercutting the costs of traditional software development projects by a significant margin. Another recurring claim is that low-code development platforms can be of utility both to *professional* software developers and *private* users unfamiliar with information technologies (IT), with the latter often being portrayed as 'citizen developers.' For example, Gartner has, in characteristic tone, touted that "Enterprise low-code application platforms deliver high-productivity and multifunction capabilities across central, departmental and citizen IT functions" [1, p. 1]. For some years, the discussion of low-code systems has been limited largely to the practical sector, before it caught the attention of researchers. Notably, the tone and the unfettered optimism of some authors in this nascent literature are not unlike those of IT vendors and market research companies (see, e.g., [2] [3] [4]).

Notwithstanding the ebullient rhetoric, there can be no doubt that the goals intended to be served by low-code development platforms are important. Indeed, these goals—among them, *increasing productivity*, *reducing costs*, promoting system *adaptability*, and *empowering users*—are classical goals of professional software development, and they have been at the center of software engineering and information systems research since the founding of the disciplines. Moreover, although vendors rarely state so in explicit terms, it would appear that low-code platforms substantially draw on principles from *model-driven development*, a topic which has occupied the attention of software engineering researchers for decades.

But despite the stir over the low-code trend, there is a glaring scarcity of information and consensus regarding the actual *technical distinguishing features* of low-code development platforms. As soon as one descends from the level of broad slogans to that of concrete technical capacities, it is surprisingly difficult to find consistent and substantive descriptions of what constitutes these systems. This paper is intended as a contribution to the search for the essence of low-code development platforms.

More precisely, this paper has two goals. The first is to present the results of an *exploratory study* of the features of seven low-code development platforms available on the current market. The aim of this study is provide an overview and comparison of the technical capacities and functionalities of these products now sold under the heading of 'low-code.' The investigation is governed by an analysis framework covering several major segments of software development. The second goal is to assess the revealed scope of low-code development platforms in relation to *research on software development productivity* and *end-user computing*. The findings reported here form part of a larger continuing study of ours on the scope and limits of LCDPs [5]. Taken together, our work is intended to furnish insights on the essence of low-code, and to critically appreciate the trend in the larger context of software engineering and information systems development.

The paper proceeds as follows. In order to provide the appropriate context for our study, we begin, in section VI, with a brief overview of related research. In section III, we explain

the method, the selection process, and the analysis framework informing the exploratory study. Section IV presents the findings of the study, discussing the historical background, a basic classification, and the functional features of the chosen platforms. In section V, we discuss the key findings and the limitations of our study, and in section VI, we propose routes for future research. The paper concludes in section VII.

II. Brief Overview of Related Research

The search for the essence of low-code platforms is closely linked to the question of their relation to prior research. Are LCDPs designed on the basis of existing research on software design and engineering? Are their components innovative in comparison with available methods, tools, and instruments? The following overview is intended to provide a foundation for discussing these questions after the presentation our exploratory study. Of course, space forbids a comprehensive survey, so focus will be placed on a few salient approaches.

A. Focus on Productivity

The essential driver of software development productivity is *reuse*. Artifacts that support reuse require abstractions, hence, *models* representing invariances among a range of systems. In addition, reuse also requires concepts for cost-efficient and safe *adaptation* to individual requirements. Among the prominent approaches addressing these aspects are the following.

Conceptual models are a pivotal means for the representation of domain knowledge in software development. However, the effort involved in constructing conceptual models from scratch can be considerable. There are two approaches to meet this challenge, both of which are based on reuse. *Reference models* are supposed to describe the states of affairs in a larger domain, such that they can be adapted into models of a whole range of particular systems. The adaptability of reference models to situation-specific requirements depends on the abstractions involved. In the case of data or object models, generalization is an attractive option, because specialization, as a monotonic extension, does not compromise the original reference models. Process models, however, bulk against generalization [6]. For this reason, other approaches have been proposed to express commonalities among process models, such as 'behavioral profiles' [7] and 'families' of process variants [8]. Apart from controlled adaptation, any modification is conceivable that can be expressed in the underlying modeling language – at the risk of jeopardizing the integrity of a model.

Domain-specific modeling languages (DSMLs) are modeling languages whose concepts are intended as reconstructions of the technical terminologies used in certain domains. Thus, they free designers from defining domain concepts themselves. In addition, they support the construction of consistent models, because they restrict the scope of possible models or systems in accordance with domain constraints integrated into the DSML. It is possible to couple the use of reference models and DSMLs so as to yield an especially powerful foundation for reuse and adaptation.

Software development productivity is also promoted by a variety of facilities for the *automatic transformation* of design artifacts into executable representations. Among such transformations are mappings of object models to code, mappings of workflow models to schemata for workflow management systems (WFMS), and mappings of data/object models to data base schemata. Research on *model-driven development* has similar but wider aims, seeking to help generate entire software systems from models [9] [10]. The objective is to drastically reduce the need for coding. As another example, *domain-specific frameworks* are similar to reference models in that they aim at enabling reuse in multiple specific cases through adaptation. Different from conceptual models, however, a frameworks are incomplete software systems [11] [12]. Prominent examples of domain-specific frameworks include skeleton enterprise systems for the financial industry [13], and frameworks which can be tailored into individual webshops.

B. Focus on Empowerment

Enabling non-programmers to develop software requires the provision of accessible representations which abstract away from the technical details of programming languages and other software development facilities. Typical examples of such technical details are the dichotomy between types and instances and GUI implementation patterns. Two main approaches have evolved. The first seeks to provide users with a general, but *simplified model of computing* and the accessible, often graphical description of individual programs. The field of 'end-user computing' or 'end-user development' [14], which aims at "empowering end-users to develop and adapt systems themselves" [15, p. 1], subsumes several lines of research following this strategy. Prominent among these are 'programming by example' [16], 'model-based development,' where the hope is that a user "just provides a conceptual description of the intended activity to be supported and the system generates the corresponding interactive application" [15, p. 4], as well as various strategies relying on customization. Visual programming, too, aims at abstracting away from the formalistic appearance of classical program code [17] [18].

The second principal approach, which may be combined with the first, aims at providing users with concepts mirroring their familiar *technical terminology*, as well as with functionality tailored to the class of domain-specific tasks for which they are responsible (e.g., accounting, procurement, or sales). Apart from domain-specific application systems for user-driven adaptation, the most important realization of this general strategy are the already mentioned DSMLs.

We return to some of the described approaches after the presentation of the main results of our study.

III. Method and Analysis Framework

The purpose of this section is to describe the research approach of the present study. We first describe, in general terms, the *method*, and the *assumptions* underpinning the work reported here (subsection III-A). Afterwards, we present the *analysis framework* by which we have characterized the

chosen LCDPs (subsection III-B), and we will explain the considerations which have guided our *selection* of platforms for investigation (subsection III-C).

A. Method and Assumptions

The essential aim of the following exploratory study, as stated in the introduction, is to obtain an overview of the technical capacities and functionalities of a number of products marketed with reference to the label 'low-code,' and to reveal the degree to which these features are common to the different platforms. The principal method of investigation has been to examine and assess the chosen LCDPs with regard to a set of *technical criteria* recorded in an analysis framework. The sources contributing to the design of the framework will be explained in subsection III-B.

As also stated at the outset, the findings reported in this paper form part of a larger, ongoing study on the scope and limitations of low-code environments in the current market [5]. In this paper, we present the results of an analysis of *seven LCDPs*, whose selection will be explained in subsection III-C. Our complete project covers some additional platforms, and it also involves a more detailed presentation of each of them than is possible in the present paper. We are aware of only one other published study of low-code platforms [19]. The aims and results of that study, in part, coincide with ours. But our concern is less with the comparison of individual LCDPs, and more with a critical appraisal of their main features and their place in the context of software engineering research.

The technical review of all platforms has been conducted by the authors of this paper. All platforms were assessed in practical terms by accessing, and working in, the live development and execution environments, to which access was obtained in the form of evaluation licenses. In no case, to the best of our knowledge, did the licenses impose significant restrictions on the availability of features relevant for our study. Among the steps involved in the technical examination were the open exploration of the environments, the search for specific types of functionality, the development of small-scale example software solutions, as well as the consultation of the official documentations and additional educational materials provided by the platform vendors. It is our assumption that it has been possible to ascertain the essential capacities and characteristics of the studied platforms. However, no doubt some details of the platforms as well as some modalities of their practical application must remain unconsidered due to our mode and scope of investigation. These limitations will be discussed in subsection V-B. Furthermore, some of our analysis criteria do not lend themselves to easy assessment using a simple n-ary evaluation scale. Cautionary remarks will be given in the remainder of this section where relevant.

B. Analysis Framework

The investigation of the platforms has been governed by an analysis framework defining a set of criteria in several categories. As is common practice, this was with the intention of ensuring that all analysis objects are assessed consistently with respect to the same aspects of a domain of interest. In the present study, the domain is that of the design, implementation, deployment, and maintenance of software systems.

TABLE I
ANALYSIS FRAMEWORK

Category	Criterion
Static Perspective	Mechanisms for data structure definitions
	Data modeling component
	Internal databases and persistence mechanisms
	Access to external data sources (APIs)
	Data reference models
	Adaptation mechanisms for data (reference) models
Dynamic Perspective	Mechanisms for program flow specifications
	Process modeling component
	Integration with static/functional components and artifacts
	Process reference models
	Adaptation mechanisms for process (reference) models
Functional Perspective	Mechanisms for functional specifications
	Functional modeling component
	Generic functional reference specifications
	Domain-specific functional reference specifications
	Adaptation mech. for functional (reference) specifications
GUI Design	GUI design component
	Graphical GUI editor
	Automatic generation of GUIs from data structures
	GUI reference models
Roles and Users	Specification mechanisms for roles and users
	Modeling component for roles and users
Artificial Intelligence	Internal AI components
	Integrability of external AI services

The specific analysis framework developed for, and used in, our study is shown in table I. As is apparent, the framework contains 24 criteria in six categories. The design of the framework had two major kinds of sources, which might be described as deductive and inductive in nature. First, one part—indeed, the majority—of the categories and criteria of the framework were defined, deductively, in orientation to *established perspectives* and principles in software development. For example, the first three categories of the framework—*static perspective*, *dynamic perspective*, and *functional perspective*—are three well-known, traditional perspectives on systems design (see, e.g., [20, sect. 103.3]). As another example, the first four categories all include criteria concerning the availability of *reference models*, or of similar types of reference specifications. These criteria have been defined in view of the striking importance of reference artifacts in software productivity research, as explained in section II. Second, some other categories and criteria were defined, inductively, on the basis of the actual features and components offered by the platforms. For example, the last category, *artificial intelligence* (AI), was defined primarily because some of the products make it possible to use AI services of varying characters.

While most criteria of the framework are self-explanatory, some additional clarifying remarks are in order. As is seen in table I, we have sometimes defined separate criteria for the availability of 'mechanisms' and 'modeling components' for the same purpose. By the former, we mean *any* kind of specification mechanism, including simple dialog forms or

table structures. By the latter, we mean *graphical modeling components*, based either on proprietary or common modeling languages, for example, the Entity-Relation Model (ERM). This distinction is made because sometimes, platforms offer the former, but not the latter.

It is also seen that we consider, in several categories, the availability of *adaptation mechanisms* for self-defined or vendor-supplied (reference) artifacts and models. The point of this criterion is that adaptation mechanisms may be provided with different degrees of sophistication. For example, in a simple case, it might only be possible to apply certain basic modifications to an existing artifact (e.g., adding or deleting activities in an existing process model). In more sophisticated cases, this process might governed and supported by a variety of additional constraints and mechanism (e.g., domain-specific constraints which prevent the definition of nonsensical associations). Finally, it will be noticed that the category 'dynamic perspective' contains the criterion 'integration with static and functional components and artifacts,' whereas no analogous criterion appears in the other two main categories. The reason is that the integration with other perspectives is of special importance in the dynamic realm. For example, process models almost always have to make to reference to the data structures or functions used in the process modeled, whereas the converse is not true, for example, of data models.

C. Platform Selection

Depending on how one counts, there are now between about twenty and several dozen vendors offering products and services in the rubric of 'low-code.' It is, of course, far beyond our scope, both in the present paper and in our larger project, to undertake an exhaustive review of all these solutions. Our study must, therefore, of necessity remain exploratory, and a limited number of low-code environments have to be selected for study. In the present paper, as already anticipated, we present the analysis of *seven* such platforms. Furthermore, it has to be underlined that in view of the extraordinary heterogeneity of products and services sold today under the name of 'low-code,' no claim to representativeness or generalizability can be justified. What one vendor proffers as a low-code solution need not bear any connection to what the next vendor has on offer. However, having noted these epistemic reservations, we can say that our selection of platforms has, of course, been guided by a coordinated procedure.

Table II shows the seven platforms selected for study. The list has emerged as follows. First of all, at the beginning of our project, we conducted a general survey of the market, and we also consulted some existing market reports (such as [1]). This yielded a list of close to 40 environments and services. All of them were explicitly branded as 'low-code' solutions, although, as we will point out below, most products are simultaneously marketed under multiple other labels. Our primary intention in selecting for study platforms from this sizable list, then, was to cover a *spectrum as broad as possible* with regard to several criteria. One criterion was *vendor size* and *market influence*. For example, as is easily seen, our selection contains

TABLE II
SELECTED LOW-CODE DEVELOPMENT PLATFORMS

Label	Platform
LC_1	Microsoft PowerApps
LC_2	Mendix
LC_3	Appian
LC_4	Wavemaker
LC_5	Pega
LC_6	Quickbase
LC_7	Bonita Platform

products of such large vendors as Microsoft, Mendix, Appian, and Pega, but also the solutions of such smaller companies as Wavemaker, Quickbase, and Bonita. Another criterion was the *intended user population*. For example, the target groups of Microsoft PowerApps, Mendix, and Appian include, in large measure, semi-professional to professional developers, whereas Quickbase is mainly marketed towards lay persons, or so-called 'citizen developers.' Another criterion was the *general purpose* or character of the platform. Although existing low-code platforms vary considerably, it turned out that, within the subset considered, it was possible to roughly distinguish four prototypical forms. The characteristics of these prototypical forms will be outlined in subsection IV-B. For example, Microsoft, Mendix, Appian, and Pega are complex, multi-use platforms, whereas Quickbase is essentially a basic data management platform, and Bonita is, indeed, a classical WfMS. In the selection process, we made sure that at least one example of each identified form was picked out. Lastly, it deserves mention that in one case, we had to exclude a platform initially considered an appropriate candidate, namely Google AppMaker. The reason is that Google recently decided to discontinue that project, replacing it with another platform. The replacement, in turn, falls in the category 'no code,' which is not the subject of this study.

In the service of brevity, we will henceforth use uniform shorthands to refer to the seven platforms, taking the form LC_1 through LC_7. The assignment of labels is shown in table II.

IV. An Exploratory Study of Selected Low-Code Platforms

We now present the findings of our study. This will be done in three steps. First, in order to provide a general context, we will make some observations about the *historical background* and *product positioning* of the seven studied platforms (subsection IV-A). Following this, in order to give an initial characterization, we will explain four aforementioned *prototypical forms*, into which the platforms may be grouped (subsection IV-B). The main part of this section, then, reports on the analysis of the platforms in relation to the established *analysis framework* (subsection IV-C).

A. Historical Background and Product Positioning

A striking fact about all considered low-code products is that they have existed, in one form or another, before the label 'low-code' was invented. The companies offering the

chosen systems have all been active in the areas of software development, business information systems, and automation for many years, if not decades. With the exception of LC_1, the solutions now marketed as low-code environments have been the single or primary products of the company for most of that time.

An analysis of archived versions of the vendors' homepages revealed that most products have been further developed and repositioned over the years, being portrayed with a changing variety of catchwords familiar in the IT industry. Sometimes, the term 'low-code' has, in fact, replaced earlier labels and become the dominant marketing metaphor. For example, LC_2 and LC_4 have been previously offered mainly as solutions for 'Rapid Application Development' (RAD), 'Platform as a Service' (PaaS), and, in the case of LC_2, as a 'Model-Driven Application Platform.' LC_3 has a history as a tool for 'Business Process Management' (BPM) in general, and as a tool for 'Mobile' BPM and 'Human-Centric' BPM in particular. LC_6 has previously been called an 'Online Database Software,' building on 'Software as a Service' (SaaS). Now these products are marketed chiefly as low-code environments.

It must be noted, however, that the significance attached to the label 'low-code' varies considerably. For example, whereas the term figures prominently in the marketing of the products just mentioned, it is only of secondary importance in that of LC_1, LC_5 and LC_7. At any rate, the bottom-line is that present-day low-code solutions are rarely new products developed from scratch. More often than not, they either extend or rebrand previously existing products, some of which have a long history on the market.

Another closely related observation is that many solutions are not marketed solely as platforms for low-code development. To the contrary, products in this segment are often said to be solutions for multiple primary purposes, which are typically expressed in the form of various well-known catchwords. For example, LC_3 and LC_5 are also said to be solution for 'Automation' in general and 'Robot Process Automation' (RPA) in particular, as well as for 'Case Management' and 'Business Process Management.' Similarly, LC_7 is also, and indeed preeminently, offered as a solution for 'Business Process' and 'Workflow' Management.

Besides being presented as solutions for 'low-code development,' LCDPs are routinely said to offer components for a variety of secondary purposes, and to contribute to a host of more general aims, all of which are explained in terms of still further catch-words. For example, many products are advertised as bringing components for the use of 'Artificial Intelligence' (e.g., LC_1, LC_3, and LC_5) and 'Microservices' (e.g., LC_2 and LC_4). Also, it is often asserted that products assist in achieving such general aims as empowering 'Citizen Developers' (e.g., LC_2, LC_5, LC_6, LC_7), supporting 'Agile' development (e.g., LC_2 and LC_5), and following principles of 'DevOps' and its varieties (e.g., LC_2, LC_4, and LC_7).

Thus, low-code solutions are often supposed to contribute to a number of purposes of varying clarity and technical specificity. Some of them are reflected in distinct technical components mentioned below, whereas others are given no more than lip-service.

B. Initial Characterization: Prototypical Forms

As already suggested by the foregoing observations, products sold as solutions for low-code development are remarkably heterogeneous. They therefore do not lend themselves to definite classification. Nonetheless, it is possible to identify certain prototypical forms of environments. These are not without overlap, and some products realize features of more than one of them. But distinguishing between four prototypical forms of low-code solutions gives a serviceable indication of the general character of the analyzed products.

1) Basic Data Management Platform: One form of low-code environment is a relatively simple hybrid between database management system, GUI designer, and surrounding architecture into which individual solutions are deployed (exemplified by LC_6). A product of this form allows users to configure simple application-like environments where users can manage data for a limited area of concern, using traditional GUIs which largely abstract from the details of the underlying database technologies. It is sometimes implied that the prime purpose of such products is to replace the use of spreadsheets. For example, a user may generate a simple application-like environment to manage items in a warehouse or customer inquiries. Platforms of this kind can largely rely on drag-and-drop mechanisms, graphical data models, and user dialogs, almost completely avoiding the need for traditional code. This is the only identified form of LCDP which can reasonably be expected to be readily accessible to the lay person.

2) Workflow Management Systems: Another prototypical form of low-code environment is, in effect, a classical component of information systems, namely a workflow management system (exemplified by LC_7 and, in part, LC_3). A product of this form allows users, in familiar fashion, to construct conceptual workflow models, to define GUIs, data structures, connections to external systems, and user roles, and to execute instances of that workflow type in a local or online environment. As it has always been the case with systems of this kind, most (but not all) specifications can be made in user dialogs and graphical representations of models. For this reason, some WfMS vendors may have conveniently taken up the label 'low-code' to profit from an existing market trend.

3) Extended, GUI- and data-centric integrated development environments: A third prototypical form of low-code platform is an extended type of integrated development environment that com-bines components for GUI design, data modeling, data persistence management, process (application logic) modeling, external API use, and deployment support (exemplified by LC_2 and LC_4). A low-code environment of this form allows users to build, within the confines of a generic and usually implicit application architecture, applications of low to moderate complexity. The focus is mainly on GUI-centric, web or mobile applications to manage and present data. In the simplest case, the components of such solutions can be applied without the use of classical programming languages

and similar technical specification mechanisms, largely relying on conceptual models and UI-based configuration. But a basic understanding of static, functional, and dynamic abstractions in software development is required in any case. Also, as soon as advanced requirements need to be implemented, familiarity with conventional technologies, or proprietary look-alikes, is usually required, for example, Java, JavaScript, SQL, HTML, CSS, SOAP, and a variety of common APIs.

Overall, products of this form are directed at an intermediate to (semi)professional audience. They are intended to reduce the efforts of routine tasks in the development of web or business applications, such as designing GUIs, implementing the Model-View-Controller (MVC) pattern, defining Object-Relational mappings, managing dependencies, and deploying the solutions in different target environments.

4) Multi-use platforms for business application configuration, integration, and development: The fourth prototypical form of low-code platform is the most varied and eclectic one. It is a multi-use platform where applications (or application-like environments) can be configured and developed, using any of a variety of components, building blocks, and existing systems, within the framework of an integration-oriented, business-centric system architecture (exemplified by LC_1, LC_3, and LC_5). These are invariably the solutions of large vendors. Platforms of this kind involve all or most components of the previous prototypical forms and expand on them in two ways. First, they have, at their core, some integration-oriented, business-centric system architecture. For example, LC_3 and LC_5 have an architectural core akin to that of classical Workflow Management systems or Case Management systems, whereas LC_1 is built as a more general business application integration suite. Second, the platforms are designed so that a variety of customizable, building block-like application units can be inserted into different places in an individual solution. These units are intended to address aspects of such areas as artificial intelligence, business intelligence, and advanced means for static and dynamic integration.

In effect, platforms of this type can be applied in widely divergent ways. In some cases, they may be used like platforms of simpler forms. In more complex cases, they may be used to build solutions for the operations of complete business units, integrating a range of existing systems and data sources.

C. Technical Features

We now turn to the central part of our study, the analysis of the seven low-code developments platforms on the basis of the analysis framework explained in subsection III-B. Since the primary aim of this paper is to contribute to the clarification of the essence of LCDPs, we will divide the presentation according to the prevalence of the identified features. We will discuss, in turn, features found to be *commonly*, *occasionally*, and *rarely* exhibited by the studied platforms. Table III gives a condensed summary of the results. As already cautioned in subsection III-B, no single ordinal scale can adequately represent the many shades and nuances of capacities present—or absent—in the different platforms. The table is, therefore, to be consulted together with the following explanations and the discussion of limitations in subsection V-B.

1) Common Features: Several features are offered by all or almost all of the seven analyzed solutions.

To begin with, the static perspective is usually addressed quite extensively. Every considered product features a component for the definition of *data structures*. In the vast majority of cases, this component is offered in the form of a conceptual modeling tool, implementing either a variant of the ERM or some proprietary language. Sometimes, data structures can only be defined in UI-based configuration dialogs or lists.

Another common feature falling into the static category is the capacity to access *external data sources* using a variety of APIs. For example, the platforms all permit to access relational database and other persistence technologies using standard APIs like JDBC; and each platform contains an individual list of so-called connectors to other types of files and systems, such as CSV files, ordinary office document types, Google documents, and the like. The platforms are generally designed so that data may be stored either in an internal database system, or in existing external systems, where the import and export is accomplished according to user-defined patterns.

Another feature possessed by every studied low-code platform is a *GUI designer*. Without exception, the platforms incorporate a component to develop graphical user interfaces and to couple them, in various ways, with other implementation artifacts defined elsewhere. All GUI designers support the interactive design of GUIs, building on a palette of pre-defined widgets. The vast majority of platforms provide drag-and-drop mechanisms for that purpose. The scope of the provided widgets varies, with some products only covering basic elements like buttons and text boxes, and others supplying a long list of interactive and representational means, including diagrams and pre-structured item galleries.

The coupling of GUIs and data structures is fairly convenient in most environments. The user does not have to implement the MVC pattern on their own, relying instead on a pre-implemented version of it. Furthermore, all systems, with the partial exception of LC_6 and LC_7, provide distinct support in adapting the GUI to different target environments (e.g., personal computers, tablets, and smartphones). The exact scope of such functionalities varies, but, as a rule, the developer is largely freed from implementing GUI adaptation routines.

The above features are among the most prominent ones in all systems. A couple of other features are equally common, although they are sometimes not as readily apparent. This is particularly true of features falling into the functional perspective, which are often much less obvious and less sophisticated than those in the static perspective. One common functional capacity is the ability to make *basic functional specifications*. Usual approaches include simple expression languages for decision rules or business rules, dialog-based ways of specifying conditions, and even simple drag-and-drop systems (as in LC_6). Besides this, each solution provides a library of *standard operations* for general purposes, such as mathematical functions. Moreover, all solutions enable, albeit

TABLE III
FEATURES OF THE STUDIED LOW-CODE DEVELOPMENT PLATFORMS

Criterion	LC_1	LC_2	LC_3	LC_4	LC_5	LC_6	LC_7	Overall
Static Perspective								
Mechanisms for data structure definitions	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●
Data modeling component	●●●	●●●	●●●	●●●	●●●	●●○	○○○	●●○
Internal databases and persistence mechanisms	●●●	●●●	●●●	●●●	●●●	●●●	○○○	●●●
Access to external data sources (APIs)	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●
Data reference models	●●○	●○○	●○○	○○○	●○○	●○○	○○○	●○○
Adaptation mechanisms for data (reference) models	●○○	●○○	●○○	○○○	●○○	●○○	○○○	●○○
Dynamic Perspective								
Mechanisms for program flow specifications	●●○	●●○	●●○	●●○	●●○	●○○	●●●	●●○
Process modeling component	●●○	●●○	●●○	○○○	●●○	●○○	●●●	●●○
Integration with static and functional components and artifacts	●●●	●○○	●●○	●●○	●●○	○○○	●●●	○○○
Process reference models	●○○	●○○	○○○	○○○	●○○	○○○	○○○	●○○
Adaptation mechanisms for process (reference) models	●○○	●○○	●○○	●○○	●○○	●○○	●○○	●○○
Functional Perspective								
Mechanisms for functional specifications	●●○	●●○	●●○	●●○	●●○	○○○	●●○	●●○
Functional modeling component	○○○	○○○	○○○	○○○	○○○	○○○	○○○	○○○
Generic functional reference specifications	●●●	●●●	●●●	●●○	●●●	●○○	●●○	●●○
Domain-specific functional reference specifications	●●○	●○○	●○○	○○○	●○○	○○○	○○○	●○○
Adaptation mechanisms for functional (reference) specifications	●○○	●○○	●○○	●○○	●○○	●○○	●○○	●○○
GUI Design								
GUI design component	●●●	●●●	●●●	●●●	●●●	●○○	●●●	●●●
Graphical GUI editor	●●●	●●●	●●●	●●●	●●●	●○○	●●●	●●●
Automatic generation of GUIs from data structures	○○○	●○○	●●○	○○○	●●●	●○○	●●○	●●○
GUI reference models	●●○	●○○	●○○	●○○	●○○	●○○	●○○	●○○
Roles and Users								
Specification mechanisms for roles and users	●●●	●●●	●●●	●○○	●●●	●●○	●●○	●●○
Modeling component for roles and users	○○○	●○○	○○○	○○○	○○○	○○○	○○○	○○○
Artificial Intelligence								
Internal artificial intelligence components	●●○	●○○	●●○	○○○	●●○	○○○	○○○	●○○
Integrability of external artificial intelligence services	●●○	●○○	●●○	○○○	●○○	○○○	○○○	●○○

Explanation: ○○○ = not or weakly addressed; ●○○ = partly addressed; ●●○ = well addressed; ●●● = extensively addressed.

to varying degrees, to invoke and integrate *external functions* via APIs. For example, almost every system allows to use standards such as Web Services and RESTful services; many systems make it possible to use a long list of APIs of individual providers, such as Google APIs and Salesforce APIs; and some systems, like LC_1 and LC_3, also permit to use specialized APIs for Artificial Intelligence services.

Moving on to a different area, another common feature is that all solutions, with the partial exception of LC_4, come with a component to define *roles and user rights*. The roles and user system is usually contained in the governing architecture of the low-code platform itself, which will be deployed together with the custom application.

Finally, almost all considered solutions offer advanced support in the *deployment* of the applications, although this takes quite different forms. For example, LC_3, LC_5, and LC_6 require to install the environment of the low-code platform on a web server, and then individual applications are deployed into that environment. LC_4, in contrast, allows to deploy the developed solutions as self-contained applications for various devices and machines.

2) Occasional Features: Certain features are found only in some of the reviewed systems. Many of them fall into the dynamic perspective, where quite different strategies are taken. One occasional feature, as already suggested, is the availability of a *work-flow modeling component* and a *workflow engine*. It may either be the case that a system, plainly, *is* a classical WfMS to begin with (LC_7) or that it is a more complex multi-use platform which incorporates a WfMS at its architectural core (LC_1, LC_3, LC_5). The functionalities offered in this connection are familiar. The workflow modeling component is usually based on a conceptual modeling language, such as Business Process Model and Notation (BPMN) (LC_3 and LC_7), or it is a simplified proprietary representational structure (LC_1, LC_2, and LC_5). In platforms incorporating workflow components, the workflow engine usually governs a large part (but not necessarily all) of the interaction between users and the system at runtime.

A somewhat different approach to dynamic specifications is taken by LC_2, LC_4, and LC_6. Here, the perspective is less business-centric, and more generic and technical. These platforms mainly provide components to define the interaction with, and transition between, user-defined UI forms (see above). For example, LC_2, features a proprietary process modeling component to define the transitions between UI pages. LC_4 and LC_6, in turn, do not offer modeling components for state transitions of the application, instead using basic menus and, sometimes, event-catching scripts.

Another feature we detected occasionally is the availability of advanced or *traditional coding components*. Some sys-

tems, such as LC_3, LC_4, and LC_7, involve one or several explicit components where procedural specifications can be made using traditional programming languages and related technologies. Most frequently, the systems use Java and JavaScript; other languages are integrated depending on the focus of the systems. For example, LC_4, which is directed at web development, permits to use CSS. It must be said, however, that at some level of the architecture, almost all low-code platforms grant recourse to traditional programming code, either by providing quasi-hidden coding components accessible only to advanced developers, or by allowing to invoke self-programmed applications or libraries using APIs.

A more extensive category of features is largely exclusive to the large-scope multi-use platforms of major vendors (LC_1, LC_3, and LC_5). These environments usually offer a variety of configurable, building block-like *application units* which can be integrated at different places in an individual solution. For example, these units cover the area of business intelligence (such as components for data analysis and reporting), the area of artificial intelligence (such as pre-built components for text recognition and sentiment analysis), and the area of static and dynamic integration, such as components for data extraction, transformation, and loading (ETL), and components for what is now called robotic process automation. There is considerable variance in the functional scope and the maturity of all these pre-developed application units.

3) Rare Features: Finally, there are several features which are present only in a few systems on the market, and often only to a limited degree. Interestingly, among these are key capacities which would be expected given earlier research on software development productivity, namely, domain-specific reference models or other reusable implementation artifacts (see section II). When it comes to domain-specific *reference data models*, only the large-scale platform LC_1 provided a sophisticated library of reusable data structures, addressing primarily common business and communication concepts, such as 'customer,' 'address,' and 'email.' Some other systems, like LC_2 and LC_6, offered some smaller data reference structures for the general domain of software development (involving concepts like 'user' and 'session'), or for certain application domains, like facility management or customer services.

Turning to domain-specific *reference functions* or functional implementations, still fewer artifacts are provided. LC_1 does make it possible to reuse a considerable number of functions implemented in various business applications of the same vendor. Most other systems, like LC_2, LC_3, and LC_5, offer catalogs of reusable functions, but these are mostly generic in nature, and they do not take on a dominant role in the intended use of the systems. Lastly, as far as reusable artifacts of a dynamic nature, such as *reference process models*, are concerned, we did not identify any noteworthy deliverables in the studied platforms. There are a couple of examples including pre-defined process models here and there, but these were very limited in scope and depth.

Finally, of course, it needs to be noted that besides the various features discussed above, every solution on the market is characterized by a more or less extensive range of individual features, which cannot be enumerated here. For example, LC_2 provides an AI-based component to predict and recommend next steps in the modeling process (a feature whose utility is relatively limited). All such features vary drastically in scope and maturity and would need to be assessed individually.

V. Discussion

The foregoing study permits a number of observations to be made. We will now discuss, in turn, the key findings (subsection V-A) and the limitations of our study (subsection V-B).

A. Key Findings

No new technology. We may begin by highlighting what is, depending on one's expectations, either the most surprising or the least surprising fact about the studied low-code development platforms. In and of themselves, the different individual tools and components of which these environments are comprised are neither radically new nor innovative in any way. To the contrary, most of these tools and components are well-known and have been used in professional software development for many decades. For example, data modeling components, process modeling components, database management systems, data source APIs, and graphical GUI editors all belong to the standard arsenal of presumably the largest share of professional software developments projects.

Low-code platforms integrate various classical development components in one environment. Notwithstanding the previous point, one of the respects in which low-code platforms deviate from classical development infrastructures is that they incorporate all or most tools and components required for a limited class of software development projects in one environment. When using a classical infrastructure in professional software development, one routinely has to deal with a sizeable array of separate tools, such as IDEs, modeling tools, DBMS, Object-Relational (O-R)-mapping frameworks, GUI libraries and editors, external libraries, dependency managers, deployment and compilation assistants, and so on. In low-code platforms, these tools are integrated in one environment, so that there is less need to switch between different systems and, more importantly, less need to maintain and integrate the implementation artifacts produced by these separate technologies.

Productivity gains mainly ensue from reducing the efforts of routine tasks. As is implied by several of the foregoing findings, low-code platforms produce productivity gains primarily by reducing the efforts of routine tasks in software development projects of low to moderate complexity. This applies in several ways. One is that through the provision of a pre-defined, integrated environment, there is less effort in synchronizing the artifacts produced by previously separate development components. Productivity is also promoted by supplying reusable reference implementations for such generic tasks as GUI design, O-R mapping, MVC implementation, and deployment in different environments. As a result, these standard parts of an application need not be implemented manually in each and every project.

Conceptual modeling is at the core of the platforms, but not at the core of marketing. Another key finding concerns the way in low-code platforms are sold. As our analysis has indicated, conceptual modeling components, whether for data models or for workflow models, are among the most important components of LCDPs, and one of the principled ways in which they are able to decrease the need for traditional coding. This is the case although the modeling languages provided tend to be rather simplistic, lagging behind the state of the art in research on conceptual modeling. To a degree, modeling components also involve ideas from Visual Programming (cf. section VI). Interestingly, however, conceptual modeling is not highlighted in the marketing of low-code products. Few vendors present themselves as offering solutions for model-driven software engineering; rather, a variety of more recent catchwords are employed. This is another indication that what has changed are often labels, not technologies.

Reuse is addressed at a generic architectural level, not at a domain-specific level. As our analysis has demonstrated, present-day low-code platforms rarely intend to offer reusable artifacts at a domain-specific levels. They neither offer domain-specific reference models of noteworthy sophistication, nor do they provide DSMLs. Rather, what is reused by these platforms are fairly generic, and often implicit, architectural frameworks for specific classes of application systems. This involves reference implementations for such areas as GUI design, OR mappings, MVC operations, access to external data sources, and so on. Within these frameworks, individual solutions can be configured and developed. For example, some systems hold generic frameworks for certain types of web or mobile application, whereas others build on architectural frameworks akin to those of WfMS.

B. Limitations

A number of limitations apply to our study. One principal limitation has already been stressed in subsection III-C. Because we have only investigated a small sample of low-code development platforms, no pretense can be made of representativeness or generalizability of results. Although we have intended to cover a spectrum as broad as possible, it is by no means possible to assert that we have revealed all significant capacities of products found on the current market, or, conversely, that the capacities which we have identified are really characteristic of most low-code environments. Because of the free use of the label 'low-code,' it is entirely possible that products and services on offer under that label, now or in the future, possess or will possess features altogether different from what has been shown above.

Several other limitations of this study have to do with criteria not taken into consideration. Two such criteria are *scalability* and *performance*. Our uses and tests of the platforms were invariably restricted to the design of simple, small-scale software systems. We can make no statement about the behavior of the systems in larger and more complex projects. Another cluster of criteria ignored here concerns the area of *live deployment, operations, and maintenance*. Although we have pointed out that almost all studied tools offer support in deployment, it was beyond our means to assess the rolling-out and operation of solutions in realistic organizational settings. Closely related to this point are means for supporting *collaboration* and *project management*. We have concentrated on the technical features essential to software development, rather than on the management of surrounding activities. Some, but not all, LCDPs are also claimed to offer support for the latter.

A different criterion whose evaluation could not be undertaken here is the *usability* or *accessibility* to lay persons ('citizen developers'). In order to properly evaluate low-code environments on this criterion, it would be necessary to conduct a study with subjects who are truly non-proficient in software development. As both authors of this paper have a background in computer science, it was not possible to judge how the platforms are perceived by non-IT experts.

A completely different class of criteria with respect to which our analysis is limited are *economic* criteria. It was beyond our scope to compare, for example, price/performance measures of any kind. As an aside, however, it may be mentioned that the price models for most services are not made public anyway. Moreover, it was our impression that some of the vendors, for example, LC_5, closely tie their price models to additional consulting and educational services.

Finally, another limitation pertains to the analysis which we have, in fact, undertaken. Despite our best efforts, no guarantee can be made that we have not overlooked one feature or another in the systems. One reason is that most, though not all, of the investigated platforms are complex and extensive environments. Exploring each and every component within them would be a project whose scope far exceeds our capacities. Another reason is that despite—and, sometimes, *because* of—the orientation toward lay users, it is often surprisingly difficult to find the places where even some of the most basic configurations can be made. For example, it is often opaque where, if at all, functional specifications can be made. All the same, while certain details may or may not have escaped our attention, we are confident that we have adequately captured the general character of the examined development platforms.

VI. Opportunities for Future Research

Although our conclusions substantially deflate the claims popularly expressed about low-code development, it is not advisable to ignore this trend. Instead, we agree with Cabot [21] that the generated momentum may (re-)kindle interest in a number of research themes of great theoretical and practical importance. Striking examples, we think, include these:

Domain-specific abstractions, like reference models and DSMLs, promise to contribute both to reuse in software development and to the support of non-programmers. As we saw, however, existing LCDPs rarely offer domain-level implementation artifacts. One route for future research may, therefore, consist in reviving research on domain-specific abstractions, and to study how these can be effectively integrated with other components of current development suites, including, but not

limited to, LCDPs. Research on this matter is likely to profit from collaboration with the disciplines dealing with the fields in question, for example, business and organization studies.

Mitigating the conflict between range and productivity of reuse: A fundamental design conflict concerning reuse is that the more specific a reusable artifact is (e.g., a data model), the better is its contribution to reuse, but the lower is the range of cases covered, and, hence, the possible economies of scale. Various approaches exist to reduce this conflict, such as generalization/specialization and multi-level language architectures. But more research is needed for the further reduction of the conflict under all perspectives on software development and in the context of larger integrated development environments, including, but not limited to, LCDPs.

Synchronization of models and code. To support the model-driven construction and adaption of software systems, it is essential to keep models and code synchronized. Especially promising, though challenging, are approaches featuring a common representation of models and programs (e.g., [22]).

Adaptable software architectures: In large and complex projects, the adaptation of an existing systems can be more efficient than building a new system from reusable artifacts. This strategy requires architectures based on powerful abstractions. Enriching the idea of frameworks with DSMLs and the integration of predefined artifacts such as components or services may be a promising approach.

VII. Conclusions

Our investigation of seven low-code development platforms does not lend support to the hypothesis that the systems sold under that heading substantially advance the state of the art. In many ways, they lag behind the frontiers of research on software design, implementation, and maintenance. What distinguishes these platforms is that they integrate, in one environment, multiple well-known and traditional system design components so as to reduce the efforts of routine tasks in implementing business applications within the confines of certain, more or less restrictive frameworks.

Nevertheless, LCDPs deserve the attention of researchers. These platforms are intended to address goals of great social and economic importance in a world permeated more and more by software. Also, because existing LCDPs in large measure build on modeling components, they may help increase the public awareness of conceptual modeling, and (re-)spark research on advanced modeling languages and architectures, among other themes in software design and implementation.

But at the same time, the low-code trend, having its source primarily in marketing, gives occasion for critical reflection on the terminology in professional software development. As our study has demonstrated, platforms sold under the label 'low-code' do not make up a well-defined class of technological environments with a uniform set of features. While we, as academics, are not in a position to prescribe to any practitioner what terms to use, we believe that maintaining a consistent terminology is among our core responsibilities. At present, it is questionable to consider 'low-code' a proper scientific term.

References

[1] P. Vincent, Y. Natis, and et al., "Magic Quadrant for Enterprise Low-Code Application Platforms," Gartner, Gartner Report Sep. 2020, 2020.

[2] F. Ihirwe, D. Di Ruscio, S. Mazzini, P. Pierini, and A. Pierantonio, "Low-code engineering for internet of things," in *Proc. of the 23rd ACM/IEEE International Conference on Model Driven Eng. Lang. and Syst.: Companion Proc.*, E. Guerra and L. Iovino, Eds. New York: ACM, 2020, pp. 522–529.

[3] R. Sanchis, O. García-Perales, F. Fraile, and R. Poler, "Low-Code as Enabler of Digital Transformation in Manufacturing Industry," *Applied Sciences*, vol. 10, no. 1, p. 12, 2020.

[4] R. Waszkowski, "Low-code platform for automating business processes in manufacturing," *IFAC-PapersOnLine*, vol. 52, no. 10, pp. 376–381, 2019.

[5] U. Frank, P. Maier, and A. Bock, "Low Code Platforms: Promises, Concepts and Prospects. A Comparative Study of Ten Systems," University of Duisburg-Essen, Essen, ICB Research Report 70, 2021.

[6] U. Frank, "Specialisation in Business Process Modelling: Motivation, Approaches and Limitations," University of Duisburg-Essen, Essen, ICB Research Report 51, 2012.

[7] S. Smirnov, M. Weidlich, and J. Mendling, "Business Process Model Abstraction Based on Behavioral Profiles," in *Service-Oriented Computing - ICSOC 2007*, ser. LNCS, B. J. Krämer, K.-J. Lin, and P. Narasimhan, Eds. Berlin: Springer, 2007, vol. 4749, pp. 1–16.

[8] F. Milani, M. Dumas, N. Ahmed, and R. Matulevičius, "Modelling families of business process variants: A decomposition driven method," *Information Systems*, vol. 56, pp. 55–72, 2016.

[9] S. W. Liddle, "Model-Driven Software Development," in *Handbook of Conceptual Modeling*, D. W. Embley and B. Thalheim, Eds. Berlin and Heidelberg: Springer, 2011, pp. 17–54.

[10] M. Brambilla, J. Cabot, M. Wimmer, and L. Baresi, *Model-Driven Software Engineering in Practice: Second Edition*, ser. Synthesis Lectures on Software Engineering. San Rafael: Morgan & Claypool, 2017.

[11] W. Codenie, K. de Hondt, P. Steyaert, and A. Vercammen, "From custom applications to domain-specific frameworks," *Communications of the ACM*, vol. 40, no. 10, pp. 70–77, 1997.

[12] M. E. Fayad and R. E. Johnson, Eds., *Domain-specific application frameworks: Frameworks experience by industry*. NY: Wiley, 2000.

[13] K. A. Bohrer, "Architecture of the San Francisco frameworks," *IBM Systems Journal*, vol. 37, no. 2, pp. 156–169, 1998.

[14] B. A. Nardi, *A small matter of programming: Perspectives on end user computing*, 2nd ed. Cambridge, Mass.: MIT Press, 1995.

[15] H. Lieberman, F. Paternò, M. Klann, and V. Wulf, "End-User Development: An Emerging Paradigm," in *End User Development*, ser. Human-Computer Interaction Series, H. Lieberman, F. Paternò, and V. Wulf, Eds. Dordrecht: Springer, 2006, vol. 9, pp. 1–8.

[16] R. St. Amant, H. Lieberman, R. Potter, and L. Zettlemoyer, "Programming by example: visual generalization in programming by example," *Commun. ACM*, vol. 43, no. 3, pp. 107–114, 2000.

[17] G. Costagliola, V. Deufemia, and G. Polese, "A framework for modeling and implementing visual notations with applications to software engineering," *ACM Transactions of Software Engineering Methodologies*, vol. 13, no. 4, pp. 431–487, 2004.

[18] D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph, and K. Doyle, "Fabrik: a visual programming environment," in *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, N. Meyrowitz, Ed. New York: ACM, 1988, pp. 176–190.

[19] 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)*, 2020, pp. 171–178.

[20] S. A. Demurjian, Sr., "Traditional Software Design," in *Computer Science Handbook*, A. B. Tucker, Ed. Boca Raton: Chapman & Hall/CRC and ACM and Taylor & Francis, 2004, chapter 103.

[21] J. Cabot, "Positioning of the low-code movement within the field of model-driven engineering," in *Proc. of the 23rd ACM/IEEE International Conference on Model Driven Eng. Lang. and Syst.: Companion Proc.*, E. Guerra and L. Iovino, Eds. New York: ACM, 2020, pp. 535–538.

[22] T. Clark, P. Sammut, and J. Willans, "Super-Languages: Developing Languages and Applications with XMF (2nd Edition)," ArXiv e-prints 1506.03363, 2015. [Online]. Available: https://arxiv.org/abs/1506.03363