Navigating the Low-Code Landscape: A Comparison of Development Platforms

Jörg Christian Kirchhof, Nico Jansen, Bernhard Rumpe
Software Engineering, RWTH Aachen University
acronio GmbH
Aachen, Germany
https://se-rwth.de https://acronio.de

Andreas Wortmann
Institute for Control Engineering of
Machine Tools and Manufacturing Units (ISW)
University of Stuttgart
Stuttgart, Germany
wortmann@isw.uni-stuttgart.de

***Abstract*—Low-code development is a software-development paradigm that can reduce the conceptual gap between application domain problem expertise and software engineering solution expertise. Hence, it might be a vital means to reduce the impact of the world-wide scarcity of software-developers. Consequently, many different low-code development platforms (LCDPs) have surfaced in the last 5-10 years, which domain experts must understand to chose the best-possible LCDP for their applications. This choice is not made restricting to front-end features of these LCDPs, but also to back-end features, which the domain experts are usually less familiar with. To support domain experts in making this decision, we have investigated popular LCDPs for features relevant to domain experts. Our findings aim to guide these domain experts, but also researchers in better understanding the landscape of LCDPs.**

***Index Terms*—Software Engineering, Low-Code, No-Code, Model-Driven Software Engineering**

I. Introduction

In our cyber-physical society, software is vital in advancing progress and generating added-value in virtually every domain. However, the scarcity of software developers hinders innovation, efficiency, and competitiveness. Consequently, increasingly more and complex software is contributed by experts from these domains, which gives rise to the conceptual gap [1] between their expertise of *what* needs to be solved in the problem domain (*e.g.,* avionics, biology, medicine, production, ...) and their lack of expertise on *how* to solve that in the solution domain of software. Instead of making domain experts professional software engineers, the abstraction of software engineering needs to increase further to ease their contribution to adding software-based value. This is the story of model-driven development (MDD) [2].

Where research in MDD has produced a wealth of successful and widely applicable methods, solutions, and tools (*e.g.,* UML, EMF, ATL, ...), it did only rarely produce tightly coupled platforms to support domain experts in contributing software-based value. low-code tools (LCTs)[1] [3], [4], [5] aim to reduce this gap by combining selected modeling languages, transformations, editors, and further tooling to provide a tightly integrated software to produce a specific kind of application (*e.g.,* B2B, mobile, web) using MDD. Hence, low-code is a success story of MDD [6].

[1]Not every low-code tool is a "platform".

Since 2018, LCTs proliferated into many domains, to produce different kinds of applications, using various modeling techniques, and development support. While low-code and LCTs are becoming ubiquitous, they support different featues, deployment modes, and target platforms. To guide researchers and practitioners in selecting an appropriate LCT, we have conducted a survey on their capabilities that investigates thier support for different database technologies, deployment, target platforms, and more.

Hence, our contributions are

- an investigation of relevant features of LCTs;
- the analysis and comparison of selected LCTs regarding these features; and
- a discussion of challenges in the use of the investigated LCTs.

In the following, Section II describes the features we investigated for each LCDP, before Section III introduces the analyzed LCTs, and Section IV compares these with another regarding those features. Afterward, Section V discusses observations, Section VI highlights related studies, and Section VII concludes.

II. LCDP Features

With the increasing incorporation of MDD in the engineering of software systems, the number of low-code applications is also growing. [7] elaborates on their interrelations and discusses that these notions, while not the same and partially for different end-users, are tightly interwoven and can benefit vastly from each other. Furthermore, [7] highlights key characteristics of low-code tools. Notable are the largely cloud-based development platforms, which require no setup and can be easily used by citizen developers, the majority of the targeted users. For this reason, aspects of usability, collaboration, types of editors, target platforms, and supported APIs are essential. Therefore, our analysis according to these characteristics focuses on the following features:

A. Development Features

Language describes the language that the developers can use to specify their application or to add handwritten code. In case there is no specific domain-specific language (DSL), it is usually the language of the generated code. This

feature does *not* refer to the language that the LCT itself is implemented in.

Hand-Written Code Support describes the ability to integrate handwritten code. The code generated by low-code tools cannot always cover all use cases due to the diversity of projects. This feature ensures that developers can meet their requirements even if they are outside the tool's normal case.

Hot Reload describes the ability of the tool to automatically apply changes to the data model / graphical user interface (GUI) without manually rebuilding and reloading the application. Especially in early development phases, there may be frequent changes. Hot reload makes it easier to test changes.

3rd Party Plugins describe the ability to integrate third party services. Such third-party services can be, for example, Dropbox, SAP, or Salesforce. Third-party services can be used to connect the generated applications to any existing infrastructure.

Visual Editor describes the feature that models and/or GUIs can be edited graphically. Usually this is done via drag-and-drop and, in the case of GUIs, in *What you see is what you get*-mode.

B. Database Technologies

SQL Database describes the property of holding data in SQL databases. This property is also fulfilled if external SQL databases can be integrated. SQL is widely used in industry. Fulfilling this criterion facilitates integration with existing applications that are based on SQL.

Graph Database describes the property of holding data in graph databases. This property is also fulfilled if external graph databases can be integrated. Graph databases are used when there are highly complex relations between data (*e.g.,* in social network applications).

Key-Value / Document-oriented Database describes the property of holding data in key-value or document-oriented databases. This property is also fulfilled if external databases can be integrated. Such databases are used especially for unstructured data and when horizontal scalability is required. Examples include Google Firestore, Amazon DynamoDB, and MongoDB.

C. Type of Target App

Web is fulfilled when the tool creates an application that runs in the browser.

iOS is fulfilled when the tool creates a native iOS application, *i.e.,* one that can (potentially) be downloaded from the app store. It does not include responsive web applications that adapt their interface to the size of mobile devices without actually being installed on the device.

Android is fulfilled when the tool creates a native Android application, *i.e.,* one that can (potentially) be downloaded from the app store. It does not include responsive web applications.

Desktop is fulfilled when the tool creates a native application for desktop computers (*i.e.,* Windows, macOS, and/or Linux). It does not include web applications unless they are installed as separate applications independent of the browser.

D. Features of Generated App

Live Collaboration describes the feature that users can work on the data in the generated application simultaneously and see live changes made by other users. Examples of this are Google Docs or Overleaf, where users of the word processor can see the cursors and changes of other users working on the same document live. Live collaboration can make it easier to synchronize users' work when there are a larger number of users.

REST API describes the property that the generated application can also be operated via a REST application programming interface (API), *e.g.,* new data can be entered. REST APIs enable users to automate their tasks.

E. Deployment & Operation

Serverless / Cloud-native describes the characteristic that applications can be run in a cloud in a way that no servers need to be administered. The administration of applications can be very complex. It is therefore desirable, especially for non-technically trained users, to keep this effort low. This property is also considered to be fulfilled if the provider of the tool itself acts as the hoster of the application and the administration effort is thus eliminated.

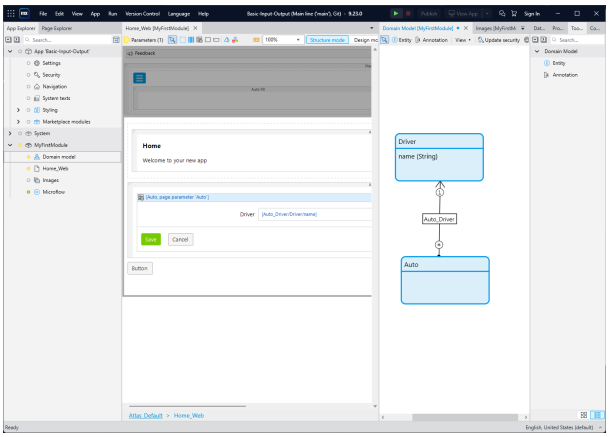
Monitoring describes the feature to technically monitor the execution of the generated application, *e.g.,* to view traffic or CPU load. In particular, this does not include services such as Google Analytics, which do not monitor the execution of the application itself, but the behavior of the user. Through such monitoring functions, problems can be detected and corrected.

III. Low-Code Tools

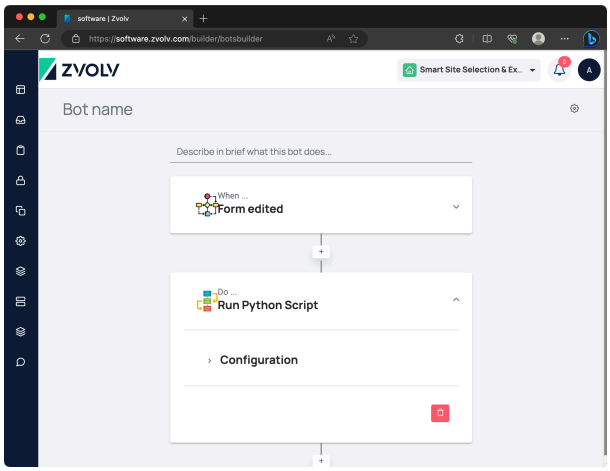
In recent years, a large number of tools for rapid application development have been introduced. Here we present the tools that we will compare in this paper. This list includes mostly low-code tools, but not exclusively. Low-code tools compete in practice with well-known web frameworks that also promise rapid development of applications. For comparison, our evaluation therefore also includes some tools that cannot be classified as low-code tools. In the following, we will now briefly introduce each tool (in alphabetical order, ignoring company names):

A12 [8] is a low-code framework for developing web applications. The most prominent application developed using A12 is the federal German tax report software *Elster*.

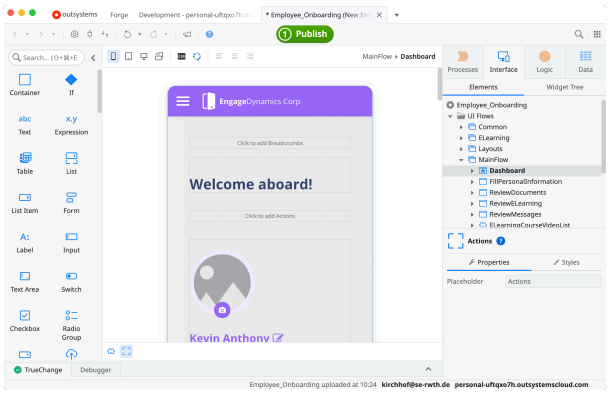
AWS Amplify Studio is a low-code tool for creating mobile and web applications. The Amplify CLI enables developers to integrate their apps with other AWS services. The



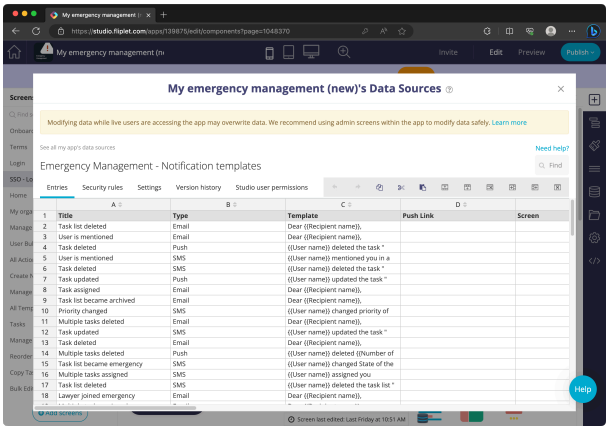
(a) Mendix. Left: GUI. Right: Data model.



(b) Zvolv's bot editor. Bots are defined in as configurable step-by-step workflows.



(c) Outsystems' GUI editor. GUIs can be composed using a drag-and-drop editor using the elements on the left.



(d) Fliplet's data editor. Data is presented in Excel-like spreadsheets.

Figure 1. Examplary screenshots of some of the examined tools.

offered GUI components are customizable using Figma, an interface design tool.

Oracle APEX is a low-code tool for creating web and mobile applications focussed on enterprise applications. It offers a drag-and-drop visual editor for GUIs and integrates with other Oracle technologies like Oracle's SQL databases.

Appian is a low-code tool focussed on automating processes. Besides enabling developers to create dashboards (using a drag-and-drop visual editor), data (using forms), and processes (using activity diagrams), it also features process mining capabilities to discover inefficiencies in processes.

Google AppSheet is a no-code tool for automating workflows. Using an activity-diagram-like workflow editor, users can create automations. AppSheets can be integrated with third party apps and data like Google Sheets, SQL databases, or Dropbox.

BettyBlocks is a no-code platform for building web applications. Dashboards can be defined using a drag-and-drop visual editor. The data model is defined using online forms and actions can be defined using activity diagrams. It offers integrations for many third party services such as Twitter, Google Maps and Slack.

Caspio is a no-code tool for creating business applications. The applications are focussed on the database whose data can be visualized on dashboards. A visual drag-and-drop editor is used for modeling data structures using class diagrams.

Electron is a framework for building cross-platform desktop applications using web development languages like CSS and JavaScript. Electron is not a low-code tool, but a popular choice for creating cross-platform applications. Prominent examples include the collaboration app *Microsoft Teams*, the music streaming application *TIDAL*, or the software development tool *GitHub Desktop*.

Fliplet is a low-code tool for mobile and web applications. GUIs can be created using a drag-and-drop visual editor, while data can be edited using spreadsheets (*cf.* Fig. 1(d)). It offers a wide range of integrations and features like notifications (push, email, SMS), analytics, or single sign on.

Flutter is a UI framework for cross-platform applications. **Serverpod** is an open-source project that provides a server-side backend for Flutter applications. Applications are developed using the Dart language that follows a similar syntax as Apples SwiftUI DSL for defining GUIs. Third parties extend Flutter with low-code editors.

Ionic is a framework for developing cross-platform mobile applications (iOS and Android) using web technologies like JavaScript. It offers UI elements looking similar to those provided by operating systems. In contrast to React Native, Ionic creates regular web applications, that interface with the hardware of the device (*e.g.,* the camera) only when necessary.

Mendix is a low-code application platform for web and mobile applications. Developers can define the GUI using a drag-and-drop visual editor and the backend using class diagrams (called *Domain Models*) (*cf.* Fig. 1(a)) and activity diagrams (called *Microflows*). Through various third party extensions, other systems like SAP can be integrated, *e.g.,* into the class diagrams.

OutSystems (or Service Studio) is a low-code development tool for building mobile and web applications. OutSystems offers drag-and-drop visual editors (*cf.* Fig. 1(c)). The data structures are defined using class diagrams. The GUI can be (partly) generated from the data structures and adapted manually. The behavior can be specified using activity diagrams.

Pega is a low-code tool focussed on workflow automation. Developers define processes using a drag-and-drop visual editor for business process models. Processes can be visualized and analyzed using artificial intelligence.

Microsoft Powerapps is a low-code tool for enterprise applications. Being a Microsoft product, it offers many integrations with other Microsoft products that are widely used in business scenarios. Overall, over 800 APIs can be integrated, using wrappers called *connectors*. Powerapps is often combined with other services from **Microsoft Azure**, Microsoft's cloud platform.

React Native is a framework for developing cross-platform mobile applications (iOS and Android) using JavaScript and React. The resulting applications are not like regular web applications but require mobile devices (or their emulators) to be executed. React Native cannot be considered a low-code tool, however, it is known to be capable of enabling rapid application development.

Vaadin is a framework for building Java-based web applications. The Vaadin Designer enables developers to also define GUIs using a drag-and-drop visual editor. Vaadin provides over 45 GUIs components, depending on the pricing tier.

Wavemaker is a low-code tool for creating mobile and web applications. It offers a drag-and-drop visual editor for GUIs. Under the hood, Wavemaker is a code generator for React Native. This code is not bound to the Wavemaker platform, preventing vendor lock-in.

Xamarin is a framework for developing cross-platform applications (both desktop and mobile) using C# and .NET. The framework's goal is to create apps that "look and feel [like] native" apps[2]. According to Microsoft, over 75 % of the code can be reused between all mobile platforms.

Zvolv is a low-code tool focussed on automating processes. After modeling a process using activity diagrams, the processes can be partly automated using so-called *bots* (*cf.* Fig. 1(b)), that either instantiate and configure predefined actions or use hand-written Python code. Data about the processes can be shown in dashboards. Zvolv offers integrations with third party tools like Dropbox, SAP, or Slack.

IV. Comparison

We have examined the previously mentioned tools to see if they have certain features. We rely on the freely available information about the tool, *i.e.,* websites, (YouTube) videos, documentation and, if available for free, trying out the tools ourselves. Not all features of the evaluated tools are available open-source. For example, Vaadin's *Observability Kit* is only available in an *Ultimate* subscription with unknown pricing. In such cases, we still recognize the feature as fulfilled if we can find information about the feature existing. Table I shows our feature evaluation.

We would like to highlight some of the observations from this comparison:

Observation 1: *Handwritten code is an important addition to visual editors.* Almost all of the tools examined offer the possibility of using handwritten code in addition to the generated code. While the automatically generated code covers some standard cases, it is not sufficient, especially when describing the behavior of the application (as opposed to the look).

Observation 2: *SQL is the most widely used database technology for low-code tools.* We suspect that this is the case because class diagrams for data modeling translate naturally into SQL schemas. In addition, SQL is widely used industrially. Some of the vendors therefore offer a way to integrate existing SQL databases into the low-code applications. In these cases, it makes sense to use the same technology for newly modelled databases as well.

Observation 3: *Low-Code tools prefer creating web applications over native desktop applications.* Low-code vendors do not seem to have a greater interest in developing native desktop applications. For the most part, it is expected that desktop users will use a web app.

Observation 4: *Low-code solutions often act as glue code between third party services.* For many of the examined tools, the real value of the tool is not in completely new applications, but in enabling non-computer scientists to combine existing third-party tools. Many of the tools examined offer third-party plug-ins. How many plugins are supported in each case varies.

[2]https://visualstudio.microsoft.com/xamarin/

Table I
COMPARISON OF FEATURES OF TOOLS FOR RAPID APPLICATION DEVELOPMENT (● = FULFILLED, ◐ = PARTLY FULFILLED, ○ = NOT FULFILLED, **?** = UNKNOWN)

	Development Features					**Database Technologies**			**Type of Target App**				**Features of Generated App**		**Deployment & Operation**	
Tool	**Language**	**Hand-Written Code Support**	**Hot Reload**	**3rd Party Plugins**	**Visual Editor**	**SQL Database**	**Graph Database**	**Key-Value / Doc. Database**	**Web**	**iOS**	**Android**	**Desktop**	**Live Collaboration**	**REST API**	**Serverless / Cloudnative**	**Monitoring**
A12	TypeScript, Java	●	●	○	●	●	○	○	●	○	○	○	○	●	●	○
AWS Amplify Studio	Target-dependent (Java, Swift, Dart, . . .)	●	○	●	●	●	●	●	●	●	●	○	○	◐ [1]	●	●
Oracle APEX	JavaScript	●	○	◐ [10]	●	●	○	○	●	○	○	○	○	●	●	●
Appian	Proprietary DSL	●	**?**	●	●	●	○	○	●	●	●	○	● [5]	●	●	●
Google AppSheet	N/A (No-Code Tool)	○	○	○	○	●	○	○	●	●	●	○	○	●	●	●
BettyBlocks	N/A (No-Code Tool)	◐ [11]	**?**	●	●	**?** [4]	**?** [4]	**?** [4]	●	○	○	○	**?**	○	●	**?**
Caspio	JavaScript	●	**?**	●	●	●	○	○	●	○	○	○	● [5]	●	●	○
Electron	JavaScript	●	○	○	○	○	○	○	○	○	○	●	○	○	○	○
Fliplet	JavaScript	●	○ [7]	●	●	●	○	○	●	◐ [9]	◐ [9]	○	**?**	●	●	○
Flutter + Serverpod	Dart	●	●	●	◐ [3]	●	○	◐ [1]	●	●	●	●	◐ [2]	○	○ [8]	●
Ionic	JavaScript	●	●	○	○	●	○	○	●	●	●	○	○	○	○	●
Mendix	Java	●	○	●	●	●	○	○	●	●	●	○	○	●	●	●
OutSystems	C#, JavaScript	●	○	●	●	●	●	●	●	●	●	○	○	○	●	●
Pega(systems)	Java	●	**?**	●	●	●	○	○	●	●	●	○	● [5]	○	●	●
Powerapps + Azure	N/A[6], PowerFx	●	●	●	●	●	●	○	●	○	○	○	●	●	●	●
React Native	JavaScript	●	●	●	◐ [1]	◐ [1]	◐ [1]	◐ [1]	●	●	●	●	○	○	◐ [1]	◐ [1]
Vaadin	Java	●	●	●	●	●	○	○	●	○	○	○	●	○	○	●
Wavemaker	JavaScript	●	○	●	●	●	○	●	●	●	●	○	○	●	●	●
Xamarin	C#	●	●	●	○	◐ [1]	○	◐ [1]	○	●	●	●	○	○	●	○
Zvolv	No-Code (Process), JSON, Python (Bots)	●	○	●	○	**?** [4]	**?** [4]	**?** [4]	●	●	●	○	○	●	●	○

[1] Offered by third party
[2] Serverpod offers software for live collaboration, but it requires manual programming of the synchronization logic
[3] Offered by a third party vendor, https://flutterflow.io/
[4] The service uses databases but it is unknown which type of database it uses. We suspect it is SQL since parts of the GUI use related vocabulary (*e.g., tables*).
[5] Developers can define the *refresh behavior* / *refreshing conditions* that specify when to reload data.
[6] By integrating Powerapps with other Azure services, it is possible to use almost any programming language for hand-written code.
[7] While there is no hot reload, users can be notified about updates or even forced to update
[8] Serverpod can generate Terraform files to deploy to AWS EC2. The management of the virtual machines then lies with the developer.
[9] It is possible to publish mobile apps via the regular app stores. However, Fliplet says they can update the app without going through the app store's review process again. Therefore, the app is likely just a shallow wrapper around a web view.
[10] While it is possible to integrate plugins, Oracle does not provide a rich library of plugins like other vendors do.
[11] BettyBlocks is marketed as a no-code tool. It is possible to add custom JavaScript components and action steps using the *enablement toolkit*.

The established providers can usually offer significantly more integrations than new providers.

Observation 5: *Many low-code tools try to act as one-stop-shop solutions that also host applications.* Many providers of low-code tools want to sell their users not only the application itself, but also act as a hosting provider. This, of course, leads to a certain vendor lock-in. We expect that the low-code providers in the background are themselves customers of a cloud. They therefore have their own interest in generating their applications in a cloudnative and scalable manner.

Based on our practical experiments with the tools that go beyond comparing the features, we also came to the following findings:

Observation 6: *Low-Code tools often rely on one-shot generation.* If the low-code tools freely release the code they generate, sometimes this code is not suitable to be generated multiple times. Sometimes the concept of the tool is that manual changes are made to the generated code, which are of course overwritten by running the generator again. In other situations, the tool does not expect the generation process to be run multiple times and crashes and/or destroys parts of the project. If one wants to develop long-term maintainable applications, this approach has corresponding problems. It effectively undermines the low-code approach, since only handwritten code can be used after initial generation.

Observation 7: *There is sometimes no clear separation between generated and handwritten code.* Related to the first finding, we found that it is not always made clear which parts of the code were generated by a code generator and which parts are handwritten. Such marking of generated code can be done, e.g., by a comment at the top of the file warning against adjusting the code manually or also by a suitable folder structure. For example, in Java projects generated code is often collected in a folder called `target/generated-sources` or similar and thus clearly separated from handwritten code.

Observation 8: *Although small example projects are often easy to understand, developing real applications remains hard.* Many of the projects studied offer sample projects. We welcome this in principle, as it makes it easier to get started with a new tool. Understandably (from a marketing perspective), however, these sample projects are designed precisely to demonstrate only the very easy-to-use features of the respective tool. The development of new applications that differ significantly from the sample projects, on the other hand, often requires a significant training period. Given that rather little information exists about many of the tools examined, familiarization with a low-code tool can therefore be more difficult than familiarization with an established framework for application development.

Observation 9: *Visual editors struggle to offer an appropriate level of abstraction.* Many of the tools examined offer visual editors for drag-and-drop editing of models. In our experience, such editors make it easier to get started because they show the user their options. However, if the language is sufficiently complex, such editors can also quickly overload the user with information by displaying significantly too many parameters whose effect is not directly understandable. At the other extreme, such editors offer only very limited possibilities and then quickly require handwritten code again to achieve the desired goal.

Observation 10: *Many low-code tools are mainly shallow GUIs for existing general purpose programming languages (GPLs), DSLs, or database technologies.* Often tools are advertised under the term "low-code" whose main purpose is a thin wrapper for existing DSLs in the form of a visual editor. For example, many of the tools examined rely on SQL tables. This makes it necessary to model corresponding data structures. Some tools use forms in which the user (unknowingly) enters essentially the same information as in a SQL `CREATE TABLE` statement. While these tools may be less intimidating to non-computer-scientists due to an often visually appealing interface, we suspect that in the long run, learning the underlying (textual) language may be more efficient.

Overall, when examining the tools, we noticed that many of them have a similar structure. Fig. 2 provides an overview of this architecture. Most low-code solutions seem to expect their developers to model three aspects of the application: Data structures, behavior, and GUI. The data is modeled either in the form of class diagrams, forms, or spreadsheets. Spreadsheets are used in combination with real data where the first column or row represents the structure and the other cells are filled with actual data. Behavior is normally modeled using activity diagrams or business process models. GUIs are usually modeled using visual drag-and-drop *what you see is what you get*-editors. Deviations from this structure exist. For example, UMLP uses a textual GUI description language.

From a technical perspective, three types of artifacts are commonly used: 1) SQL databases, whose schemas were generated from the data models, are mostly used as databases. 2) The behavior of the application is handled via (for the most part invisible to developers) code of a GPL or a model interpreter. 3) Frontend definitions. The concrete technology for the frontend depends on the target platform. Usually, the code for the behavior and the code for the frontend can be extended or (partially) replaced by handwritten code.

The code created in this way is then deployed on a cloud infrastructure. In many cases, the providers of the low-code tool also act as hosters and also take care of the provision of the generated application. We assume that in most cases no own infrastructure is provided here, but infrastructure from hyperscalers is resold. For example, during our tests, we found that the IPs that were being used to deliver our web applications by Outsystems and Fliplet belonged to the IP range of Amazon CloudFront. Additionally, many low-code tools utilize 3rd party services to integrate with other popular services such as Dropbox or Google Sheets.

V. Discussion

Based on our findings, we can observe features favorable for most low-code tools. While full compliance with all features is not necessarily required for each platform, some emerge as

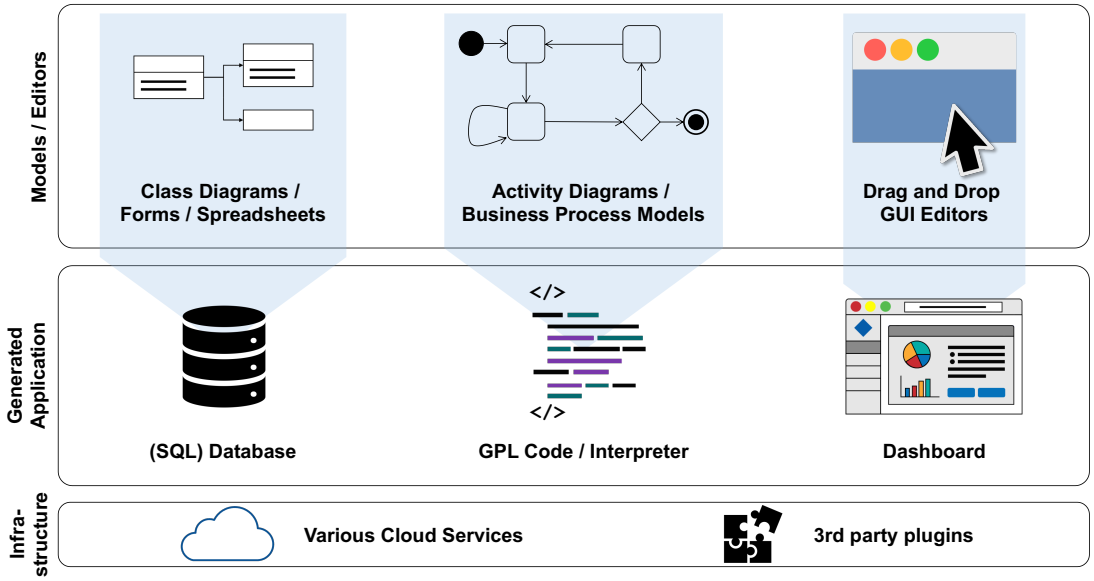


Figure 2. Common architecture of low-code tools.

particularly favorable. Also, there are open challenges LCTs have to tackle in the future.

A. Observations

Web for the win? Apparently, web-based platforms are clearly on the rise. This is likely due to the greater approachability and out-of-the-box usability, even for novice users. The necessity of a local setup represents an entry hurdle for domain experts without software engineering experience.

Furthermore, the embedding of external services turned out to be beneficial for LCTs. Functionalities such as Google services or notifications are already established technological standards that can be easily embedded in applications.

Where is the domain-specificity? As domain experts are usually not software experts, low-code platforms should leverage domain-specific modeling techniques and not constitute mere abstractions of a GPLs. Terminology from the domain picks up users in their field of knowledge, supporting them in their modeling activity. However, current LCTs mostly do not incorporate domain knowledge, which makes it questionable whether they are currently not rather an aid for skilled software developers (who have an intrinsic understanding of the underlying programming concepts) to work more efficiently.

Integrated hosting—Blessing or curse? Several LCTs provide integrated deployment and hosting. This is both an advantage and a disadvantage for users. On the one hand, direct deployment and hosting offer a convenient solution to build and use the target system immediately and without coping with any technological concerns. On the other hand, this often results in a vendor-lock scenario, as the platform providers profit from direct hosting, which either prevents or at least impedes in-house or third-party hosting.

How should the models look like? Most LCTs support modeling using graphical editors to describe structure and behavior of the target application. While graphical models yield various benefits, including improving communication quality when discussing the models and improving the recall of design details, textual models can be organized differently more easy to include additional information, such as design rationale and can foster understanding of design decisions better [9]. This suggests that for optimal use, LCTs should provide boths kinds of views on their models.

Where are the the real-world examples? LCTs aspire to be used by software developers without formal software engineering education. Yet, most examples we have encountered were small-scale / toy examples only that leave the intended LCT users behind once the domain problems to be solved become more challenging.

Where are the language workbenches? The analyzed LCTs are not implemented using common language workbenches. This suggests that this information either is hidden on purpose or that the modeling languages of the LCTs and model-processing tooling are crafted without using a language workbench. If the latter, we can expect LCTs having to deal with many challenges solved by language workbenches (such as language composition [10], generation [11], and evolution of model processors, ...) manually.

LCT and MDE—"Two sides of the same coin?" [7] This finding also aligns with [7], stating that MDE and low-code should complement each other. Therefore, LCTs should be built on top of already established MDE techniques to harness their advantages. This also incorporates functionalities usually provided by default in MDE applications. An example is the possibility to regenerate multiple times, while many low-code platforms currently only support a one-shot generation. The option of retriggering the generation process is essential for the evolutionary development of a system, in particular, if additional handwritten code snippets should be integrated. It

is also necessary to integrate generated and handwritten code appropriately while keeping their sources separate (following the notion of separation of concerns [12]). For this purpose, LCTs could benefit vastly from known patterns from software language engineering, such as the generation gap [13] or one of its extensions [14].

B. Challenges

Computational Thinking. LCTs ultimately aim to enable domain experts to contribute to or create software solutions themselves. Obviously, these experts come from a variety of domains and have internalized various different methods, concepts, and paradigms to solve challenges in their domain that may not be directly translatable to software development. Instead, using an LCT properly often still demands understanding certain concepts from software engineering, such as data structures, encapsulation, algorithmic thinking, and similar. Hence, an essential challenge for each LCT is the onboarding of domain experts into the computational thinking required to provide useful low-code software solutions.

Brownfield Low-Code. Software projects rarely start completely from scratch. Likewise, when low-code development is employed in a context in which large bodies of code reifying domain knowledge and business expertise already exist, the integration of low-code with the code base is necessary. With many LCTs being walled gardens that prevent the users from accessing the (generated) code, this entails that existing code must become low-code, i.e., models, as well. This demands means to extract (LCDP-specific) models from existing code bases automatically. Otherwise, the manual extraction of LCT models from code would bind software engineering resources that the users of LCTs obviously cannot employ.

LCT Migration. Even a walled garden might wither and when the low-code models therein reify important business value to their owners, there must be means to migrate models from one LCT to another. Aside from contrasting the business model of many LCTs, this also entails the semantically correct and, ideally automated, translation of models from the language(s) of one LCT into the languages of another LCT.

Tailoring LCTs. Most LCTs aim for supporting as many use cases as possible by using rather generic approaches, such as providing generic modeling languages for data flows, data structures, constraints, and views to create some kind of application. However, in many domains there are very specific standards, concepts, and terminologies that the respective domain experts are familiar with and expect to be able to use [15]. This demands means to easily tailor the modeling languages of LCTs but also the LCTs themselves to a specific domain.

VI. Related Work

In the literature, there are several analyses referring to low-code platforms. While we have already included and discussed related work inline, in the following, we go into more detail about other relevant studies in this domain and compare those with our findings. Primarily, these studies are based on a purely conceptual investigation. This often omits aspects of actual usability and technology readiness, which, however, are important features, especially for low-code platforms to support citizen developers out-of-the-box [7].

Gartner analyzed numerous low-code platform in one of their magic quadrants [16], evaluating tools according to their *ability to execute* and their *completeness of vision*. In this analysis, Outsystems, Mendix and Microsoft Powerapps were rated best. While Gartner also includes features of the product in their evaluation, their analysis focuses clearly on business aspects. Our analysis focuses more clearly on technical aspects and excludes aspects such as pricing, customer experience or the business model. Gartner advises software engineering leaders to develop their low-code tools based on technology fit, commercial considerations, and developer talent and skills. Our analysis provides an extended insight to evaluate the technology fit.

Similar to our study, [17] and [18] analyze the features of a set of low-code tools. Compared to these studies, our study offers a lower number of features but a larger number of tools, giving a broader market overview. Moreover, [17] also includes an overview of the main components of low-code platforms, similar to the common architecture identified in our analysis. A survey of low-code tools with a special focus on blockchain application can be found in [19].

VII. Conclusion

We have investigated the features of 21 low-code tools and found that the majority of these tools leverages largely domain-independent modeling techniques using web-based, graphical editors to create web applications over SQL databases. To this end, many LCTs also support integrating handwritten code as well as third-party plugins. The main discriminating decisions in choosing a LCT seem to be whether (a) a specific database technology, (b) live collaboration, or (c) desktop applications are required. To further support LCT users in choosing the best-suitable LCT, the supported degrees of automation by the LCT as well as the supported features in the target applications should be compared as well. Overall, the lack of truly domain-specific abstractions in the investigated LCTs suggests that these aim less at being used by domain experts–unless the domain is web applications–and more by citizen developers with an interest in software development. Despite the rather simple modeling techniques encountered in the LCTs, we found little (documented) reuse of established language workbenches is as astonishing as well.

Acknowledgements

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy—EXC 2023 Internet of Production—390621612. Website: https://www.iop.rwth-aachen.de

References

[1] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," *Future of Software Engineering (FOSE '07)*, pp. 37–54, May 2007.

[2] B. Selic, "The pragmatics of model-driven development," *IEEE software*, vol. 20, no. 5, pp. 19–25, 2003.

[3] A. C. Bock and U. Frank, "Low-code platform," *Business & Information Systems Engineering*, vol. 63, pp. 733–740, 2021.

[4] N. Prinz, C. Rentrop, and M. Huber, "Low-code development platforms-a literature review." in *AMCIS*, 2021.

[5] M. Tisi, J.-M. Mottu, D. S. Kolovos, J. De Lara, E. M. Guerra, D. Di 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: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019)*, 2019.

[6] A. Bucchiarone, F. Ciccozzi, L. Lambers, A. Pierantonio, M. Tichy, M. Tisi, A. Wortmann, and V. Zaytsev, "What is the future of modeling?" *IEEE software*, vol. 38, no. 2, pp. 119–127, 2021.

[7] D. Di Ruscio, D. Kolovos, J. de Lara, A. Pierantonio, M. Tisi, and M. Wimmer, "Low-code development and model-driven engineering: Two sides of the same coin?" *Software and Systems Modeling*, vol. 21, no. 2, pp. 437–446, 2022.

[8] H. Mehmanesh, S. Gandenberger, A. Weiss, T. Kneist, and S. Lorenz, "A12 low code für individuelle enterprise software," [Online]. Available: https://www.mgm-tp.com/documents/mgm-A12_Whitepaper_Low-Code-Enterprise-Software_2022_DE.pdf, Last accessed: 03.06.2023, mgm technology partners GmbH, Tech. Rep., 2022.

[9] R. Jolak, M. Savary-Leblanc, M. Dalibor, A. Wortmann, R. Hebig, J. Vincur, I. Polasek, X. Le Pallec, S. Gérard, and M. R. Chaudron, "Software engineering whispers: The effect of textual vs. graphical software design descriptions on software design communication," *Empirical software engineering*, vol. 25, pp. 4427–4471, 2020.

[10] A. Haber, M. Look, P. Mir Seyed Nazari, A. Navarro Perez, B. Rumpe, S. Völkel, and A. Wortmann, "Composition of Heterogeneous Modeling Languages," in *Model-Driven Engineering and Software Development*, ser. Communications in Computer and Information Science, vol. 580. Springer, 2015, pp. 45–66.

[11] B. Rumpe, *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[12] W. L. Hürsch and C. V. Lopes, "Separation of Concerns," 1995.

[13] J. Vlissides, *Pattern Hatching: Design Patterns Applied*. Addison-Wesley Longman Ltd., 1998.

[14] F. Drux, N. Jansen, and B. Rumpe, "A Catalog of Design Patterns for Compositional Language Engineering," *Journal of Object Technology (JOT)*, vol. 21, no. 4, pp. 4:1–13, October 2022.

[15] A. Wortmann, O. Barais, B. Combemale, and M. Wimmer, "Modeling languages in industry 4.0: an extended systematic mapping study," *Software and Systems Modeling*, vol. 19, pp. 67–94, 2020.

[16] P. Vincent, K. Iijima, A. Leow, M. West, and O. Matvitskyy, "Magic Quadrant for Enterprise Low-Code Application Platforms," Gartner, Tech. Rep., 2022.

[17] 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.

[18] A. C. Bock and U. Frank, "In search of the essence of low-code: An exploratory study of seven development platforms," in *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2021, pp. 57–66.

[19] S. Curty, F. Härer, and H.-G. Fill, "Design of blockchain-based applications using model-driven engineering and low-code/no-code platforms: a structured literature review," *Software and Systems Modeling*, 2023. [Online]. Available: https://doi.org/10.1007/s10270-023-01109-1