The low-code handbook

Learn how to unlock faster and better software development with low-code solutions

Jordi Cabot

ISBN: 978-99987-785-0-4

DEDICATION

To all the colleagues with whom I have had endless exciting discussions on how to make better software faster. Special thanks to the reviewers that helped so much to improve this book and to the members of my BESSER team. Together, we are trying to make the low-code revolution happen for all.

PRAGMATIC ANSWERS TO ALL YOUR LOW-CODE QUESTIONS

1. BASIC LOW-CODE QUESTIONS AND ANSWERS

First things first. Let's start by clarifying what low-code is (and what it is not), where it comes from and who is it for.

What is NOT low-code?

I think it's easier to begin by listing all things low-code is NOT:

- Low-code is **not a specific language**. You can use any specification/design language you like in a low-code approach.
- Low-code is **neither a graphical nor a textual notation.** It's not a particular set of symbols or shapes to be used when describing your systems.
- Low-code is **not a development method**. Low-code does not prescribe any set of concrete steps to follow, nor specific guidelines to drive your development process.
- Low-code is **not a tool**, many tools can support your low-code approach as long as they come with a certain number of features that we'll see later.

And, definitely, low-code is **no silver bullet**. As explained in the famous paper[1], "*there is no single development, in either technology*

[1] Brooks, Frederick P. (1986). "No Silver Bullet—Essence and Accident in Software Engineering". Proc of the IFIP Tenth World Computing Conference: 1069–1076.

or management technique, which by itself promises even one order of magnitude improvement within a decade in productivity, in reliability and simplicity".

Low-code can help with all the above, but if you are leading a dysfunctional software team, adopting a low-code approach alone will not give your team superpowers not fix their poor dynamics and performance.

So, what is low-code then?

The most well-known definition of this term, coming from the Forrester Report from 2014[2], states that "*low-code accelerates app delivery by dramatically reducing the amount of hand-coding required"*. Note the two key elements in this definition:

- The increase in productivity
- Writing less code as a strategy to obtain such increase.

But, how can we produce more software while writing less code? Well, by automating the generation of that software. From where? From higher-level specifications. Instead of writing all code by hand, we model, we design, we specify, the software system, and we use those specifications to generate (most of) the actual software.

Honestly, not that different from the way other engineering disciplines work. Nobody starts building a bridge or a house by putting one brick on top of the other. We first draw the blueprints, examine them and, only when we are sure they are correct, we launch the construction process. Low-code propose to do the same with the software, just with the important difference that software is a digital asset, and therefore we can automate the derivation of the asset itself from the specification[3].

And if you think about it, this type of evolution of moving up the abstraction ladder has been, in fact, a constant in the history of software development, e.g. compare modern languages with assembler or even C. This was nicely put by Grady Booch.

[2] New Development Platforms Emerge for Customer-facing Applications. Forrester: Cambridge, MA, USA 15 (2014)

[3] With 3D printing we do the same for basic structures, automatically built from their blueprints.

The entire history of software engineering is about raising the level of abstraction – Grady Booch

I'd like to highlight that the low-code definition mentions reducing (not eliminating!) the amount of hand-code required. So, in a low-code approach, we still acknowledge there are software parts that you may want to write by hand because they are either too complex to model or need such a degree of optimization that it's better to directly code that part. This is in contrast with no-code approaches, see Chapter 9.

Note also how the definition doesn't say how to put in practice this low-code approach. As we just discussed, low-code does not prescribe a specific language or methodology, low-code is more a philosophy or development style that you decide how "instantiate"[4] based on your specific requirements and context.

A little bit of history

While the low-code term may be new, the truth is that it is just the latest incarnation of a long tradition of model-based approaches (known under a variety of names such as model-driven engineering, model-driven architecture, model-driven development, …[5]) which in turn can be traced back to the older tradition of CASE (Computer-Aided Software Engineering) tools. In fact, already in 1991, in the 1st edition of the well-known CAiSE conference, we could find papers stating concepts like: "*Given the final model, the complete computerized information system can be automatically generated*" or "*we arrive at a specification from which executable code can be automatically generated*", which sound very similar to the promises of today's low-code tools as they have the same goal, automate software development from an initial set of high-level models.

[4] The choice of a specific low-code tool will in fact already make plenty of these decisions for you. Keep this in mind when choosing the right tool, a topic I will cover in Chapter 6.

[5] Model-driven engineering and related concepts have been covered in detail in my previous book: Model-Driven Software Engineering in Practice co-authored with Marco Brambilla and Manuel Wimmer. See https://mdse-book.com/ for full details.

I believe that low-code can be defined as a pragmatic version of these previous approaches where instead of using many models (and many transformations between them) we go for a smaller and fixed set of models (enough for the reduced target of application types low-code aims for) together with a simple one-step transformation from those models to the final code.

We trade expressiveness for simplicity, which, in turn, helps us to broaden the user base of low-code compared with the more tech-savvy requirements of traditional modeling approaches.

All this combined with good tool support that may even seamlessly deploy the models (and/or the code generated from them) in a cloud-based infrastructure. Another factor that strongly contributes to broaden the user base of low-code.

And let’s not disregard the marketing of the term. All previous terminology focused on the word model, an overloaded term, even more now in these AI-times. And one that alienates programmers as it suggests coding is not necessary and that their job as programmers is going to become redundant. Instead, calling the approach “low-code”, keeps them in a familiar territory. Programmers will still program, it's just that they will not program the full application but parts of it, while other parts will be generated for them.

In my opinion, all this explains the explosion of popularity of the approach, see Figure 1-1, reaching communities that modeling paradigms were never able to breach into.

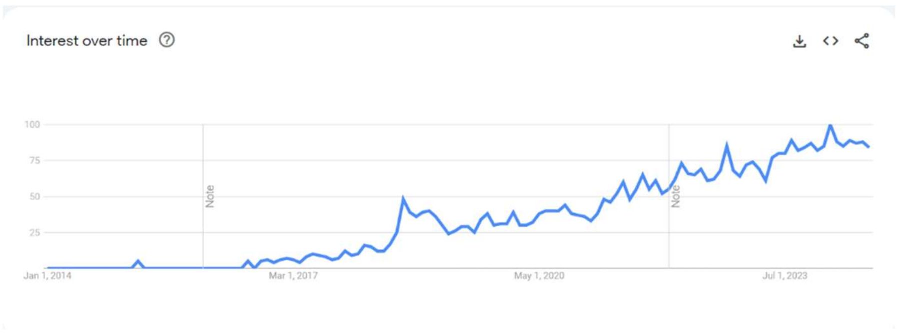

Figure 1-1 Google trends analysis on the popularity of the low-code term

Is low-code for me? Can I trust low-code has a bright future?

Theoretically speaking, low-code approaches can be adapted to build any type of application. Realistically, low-code tools specialize in the development of data-intensive applications with a relational database in the backend and a lot of CRUD (Create-Read-Update-Delete) web-based forms in the frontend with limited business logic and automations. With as many exceptions as you like, this is today the bread-and-butter of low-code.

Therefore, low-code is for you if you need to build this type of application on a repetitive basis[6]. Instead, if you are building a critical software infrastructure, that you'll need to get certified before running it in a production environment, then low-code is not for you yet. It could still help, e.g. by generating tests for the code using model-based testing techniques, but it can't be the core of your solution[7].

For any in-the-middle scenario, I'd recommend giving it a try and decide for yourself. Even if you end up deciding against it, I think the lessons learned will be useful, as there is a strong likelihood you will adopt a low-code approach in a future project anyway. Already today, the low-code market is estimated to be around $13-25B (depending on the low-code scope used in the calculations) with an expected compound annual growth rate of 20% annually. Moreover, as low-code tools evolve and improve, there will be more and more domains where applying low-code tools will make sense, so your early experiments will pay off. Either in future projects at your company or if you move to a different company or sector.

Keep in mind as well that the Big Five in Tech[8] (and others such as Oracle or SAP) are all embracing low-code and developing their own offerings for this market. If you think about it, it makes perfect sense

[6] If you're sure you're going to build just one, then, just coding is also acceptable, as there is an initial adoption cost linked to the adoption of a low-code tool, or any new technology for that matter, see the discussion in Chapter 7.
[7] You could certify the code generator to use low-code in critical environments, but this goes beyond the concepts covered in the book.
[8] See the list of companies in the Big Five or Magnificent Seven group https://en.wikipedia.org/wiki/Big_Tech

that these companies invest in low-code tools. For them, it's a way to reach a broader audience of potential users that could become consumers of their platforms, services and infrastructure. They are not stupid so the low-code tools they develop are especially good when it comes to connect the app-to-be to their own specific commercial offerings. Basically, their line of thought is: the more people I help to build software, the more people will pay me to run that software as I'm the one providing the best infrastructure for it. This way of delegating the gains to the deployment and run-time phases helps them to offer a pricing model for the low-code phase much more reasonable than that of other low-code solutions.

And while low-code still shows some limitations and open challenges, as I will discuss in this book, it is getting more and more attention, also from the research community[9]. This makes me confident we will see a steady improvement in the features and capabilities of the low-code tools in the mid and long-term future.

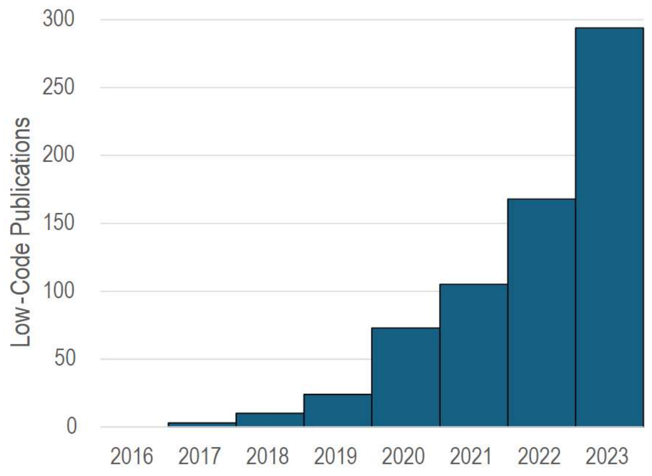

Figure 1-2 Increase in the number of research publications targeting low-code challenges in the last years

[9] A Metascience Study of the Adoption of Low-Code terminology in Modeling Publications https://modeling-languages.com/a-metascience-study-of-the-adoption-of-low-code-terminology-in-modeling-publications/

How to read this book?

After this first introduction to the low-code concept, allow me to give you some advice on what chapters to focus on next depending on your main interests and profile:

- Low-code Manager: if you just want to get an overview of the low-code domain, benefits, limitations… but you're not the one that will be using the low-code tools, read chapters 2,5,6,7, 9 and 10.
- Low-code user: if you'll be the one building software applications with low-code tools, all chapters are relevant except for chapter 8.
- Low-code tool developer: if you'll be in charge of creating or adapting existing low-code tools to better fit the needs of your team, I'm sure you'll enjoy the full book, don't skip a single chapter!

2. HOW TO CREATE MY FIRST LOW-CODE APPLICATION?

Creating a low-code application is as simple as modeling the app, choosing the generator for your target platform, (optionally) refactoring and optimizing the code, and running the (improved) code. That's it.

And with some tool vendors, the process is even more straightforward. You just model the app, and from there the low-code platform generates internally the code and automatically deploys it on its own cloud-based infrastructure. So, once the models are completed, you just click on the "Run" button, and you'll get a URL pointing to where your new app has been deployed. Keep in mind that, as usual, the simpler the process, the less control you have over it.

Nevertheless, one way or the other, the internal architecture of all low-code platforms is similar. For instance, Figure 2-1 shows the architecture of the BESSER platform. BESSER is a Python-based open-source low-code platform that I'll be using as an example platform in different occasions across the book. As depicted in the Figure, BESSER includes a number of components to specify your application using various input formats and another group of components to generate and run it on your desired target platform.

Let's take a closer look at these two key low-code phases before we go deeper into each step over the next chapters. This will help us to better understand some of the design decisions we'll discuss later.

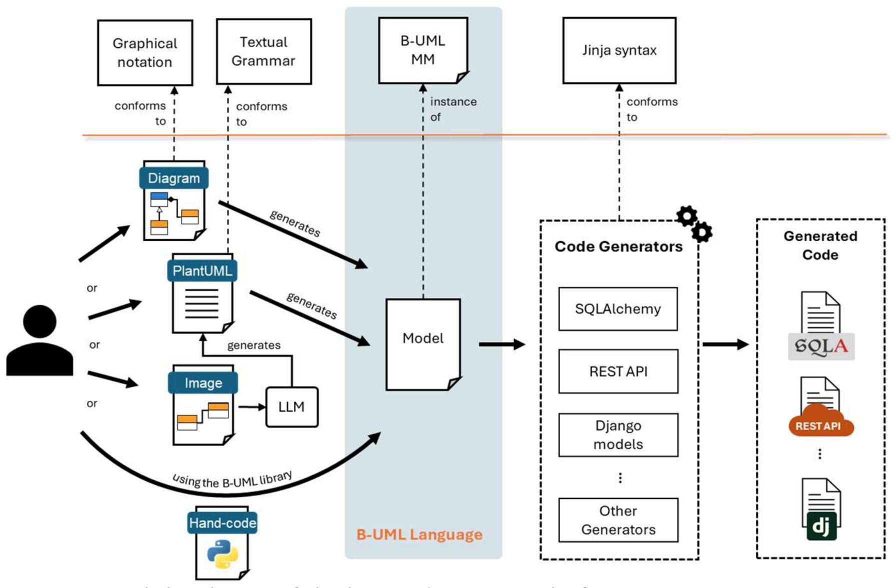

Figure 2-1 High-level view of the low-code BESSER platform

Phase 1 – Modeling your software application

First, in the modeling stage, the low-code expert specifies the software models using the modeling language provided by the tool (called *B-UML* in the *BESSER* example platform, Figure 2-1). These models are specified using the notation/s provided by the language. There could be more than one option for the same language, e.g. a graphical notation and a textual notation. For instance, in BESSER, you can use a graphical modeling environment, a textual syntax or even an AI-based approach able to transform handwritten drawings into models.

Either way, this input is internally stored as a set of objects conforming to the *metamodel* of the language, the B-UML metamodel in our example (*B-UML MM* in the Figure). Like the role of a grammar for programming languages, metamodels express what modeling primitives can be used and how they can be combined to create syntactically correct models. Each low-code tool comes with its own set of (sub)languages together with their corresponding notations and metamodels). We'll cover this part in Chapter 3.

I can't help but highlight the key role of modeling in a low-code development process. The less the code the more the (explicit) modeling!

Modeling a software system is not an option. The only decision is whether to make those models explicit – Antoni Olivé

As Antoni Olivé puts it, in any project, you always start by modeling the system, but in a programming-centric development project, you have the option not to make the models explicit, you could just model "in your head", and proceed to directly write down the code based on your mental designs. Instead, in a low-code approach, this is not an option, you need to make all models explicit as they are the required input of the whole process.

Phase 2 – Generating your software application

Once the models are created, we move to the code-generation phase, where the low-code expert will choose the right code generator, among those provided by the tool, depending on the desired output stack. In short, a code generator is a template (written in *Jinja* in the example) that transforms an input model into a textual output. Basically, the template engine will traverse the input model, identify the elements that need to be transformed and apply to each of them the relevant subset of the generation template. For instance, in a classical UML class diagram to SQL transformation, the template would transform each class into a table, each attribute into a column, each association into a foreign key or a new table, depending on the type of association, and so on.

If the produced source code is available[10], you can now modify it to complete the app (if any aspect of the app functionality had not been modeled and therefore not generated) or improve the resulting code. The former is easier than the latter, as the latter involves modifying the generated code, and there is a risk that your changes will be overwritten if you need to regenerate the software from the models later on, e.g. as part of an evolution process. Typically, tools will aim

[10] Remember that this code-generation approach is the simplest and most transparent one, as you can see and play with the generated code. But it's not the only alternative, as I discuss in Chapter 4. And even if the low-code tool follows a code-generation strategy it may not provide you access to the code itself.

to avoid this issue by letting you annotate your modifications[11] (using, for instance, some kind of "protected areas" tags), but it's always something to keep in mind.

A word of advice:

It's better to complete 100% of 8 things than of 80% of 10 things - Dave Kellogg

This quote is fully applicable to the code-generation process. In general, try to fully specify parts of the app (so that these parts can be 100% generated) instead of partially modeling everything, as this will increase the need to co-evolve the models and the code together every time the app needs to be updated, increasing the risks I just mentioned.

For instance, it is better to be able to generate a full database implementation for the modeled app even if there is zero code generated to control the business logic of the app (as maybe we decided to skip any type of behavioral modeling) than generating a little bit of both. In the former scenario, we only need to modify either the model (if we update the database) or the code (if we evolve the business logic). In the latter, we are forced to manually complete and refine all artifacts, both at the code and model level, every time we change anything on the app.

The next chapters aim to go deeper into all these aspects by answering questions such as: How many models do I need? How much detail in each model? What is the best code generation strategy? etc.

[11] Some tools will ask you to write your code excerpts as part of the model definition in a kind of "black-boxes" that will just be added during the code generation at the points where you indicated