



**Low-code has the potential to empower more people to automate tasks by creating computer programs.**

BY MARTIN HIRZEL

# Low-Code Programming Models

LOW-CODE IS THE subject of much current enthusiasm stirred by market research companies and confirmed by vendors rushing to embrace the label.<sup>7,31</sup> But what low-code programming means is somewhat cryptic, let alone how it works. Moreover, scientific literature rarely uses the term. We can decode the term by breaking it into its components. Programming means developing computer programs, which comprise instructions for a computer to execute. Traditionally, programming means writing code in a textual programming language, such as C, Java, or Python. In contrast, low-code programming minimizes the use of a textual programming language. Instead, it aims to use alternative techniques closer to how users naturally think about their task.

Users of low-code range from professional developers to so-called citizen developers. A citizen developer is an amateur programmer with little professional programming education. Citizen

developers, having chosen a career different from programming, tend to have more domain expertise. Low-code enables domain experts to become citizen developers. At the same time, low-code platforms should also strive to make pro-developers (professionals with an education or career in software development) more productive.

Whether used by a citizen developer or a pro-developer, low-code programming aims to save the time and tedium of performing a task by hand.<sup>35</sup> Further motivation for individuals comes from the joy of creating something useful, thinking about tasks in a computational way, and acquiring programming skills that can advance their career. Businesses may have their own motivation for adopting low-code platforms, which can alleviate the shortage of pro-developers, reduce mistakes of tedious manual tasks, and multiply the time savings from one individual's low-code program to their colleagues.<sup>31</sup> Another factor driving low-code is the rise of cloud-based software as a service, providing both more interfaces to automate and a platform on which to deploy automations.

A few concepts are closely related to low-code programming. No-code programming is more purist, with zero handwritten code in a textual programming language. End-user programming (EUP) puts the emphasis on who is doing the programming (the end-user as citizen developer) rather than on how they are not doing their programming (not with textual code).<sup>6</sup> This term is common in the academic litera-

## » key insights

- Low-code minimizes the use of textual programming languages and instead uses alternatives such as visual or natural languages.
- Progress in AI fuels progress in low-code in proportion to the ambiguity of the low-code technique.
- Domain-specific languages and the model-view-controller pattern constitutes a backbone and unifying principle across low-code techniques.



ture and overlaps with low-code, but does not preclude the use of a textual programming language. Another gap between EUP and low-code is the latter aims to serve not just end users but also pro-developers.<sup>7,31</sup>

Bock and Frank<sup>7</sup> and Sahay et al.<sup>31</sup> recently compared commercial low-code platforms, and Barricelli et al. recently mapped the EUP literature.<sup>6</sup> In contrast, this article bridges the gap between low-code and the academic literature and adds missing details and perspective. Low-code encompasses more specialized techniques, such as visual programming languages (VPLs), programming by demonstration (PBD), programming by example (PBE), robotic process automation (RPA), programming by natural language (PBNL), and others. Surveys on these techniques are more specific and often dated.<sup>4,8,20,35</sup> In

contrast, this article reviews recent literature across all these techniques.

Given that low-code offers citizen developers a model to create computer programs, this article explores low-code from the perspective of programming models. A programming model is a set of abstractions that supports developing computer programs. Programming models can be low-code or not, and they can be domain-specific or general-purpose. Some programming models are languages; for example, Java is a general-purpose language and SQL is domain-specific, and neither is low-code. Scratch is a low-code programming model for kids that is media-centric,<sup>29</sup> making it domain-specific. The programming-model perspective helps highlight common techniques for writing, reading, and executing programs, and it helps relate low-code to

research into program synthesis and domain-specific languages.

This article includes a deep-dive into three prominent low-code techniques: visual programming, programming by demonstration, and programming by natural language. The deep-dive focuses on fundamental building blocks and a unifying framework common to all three. The citations in this article cover both seminal work and recent advances in low-code programming models, for instance, based on artificial intelligence. Moreover, this article aims to cut through the buzz surrounding low-code so as to expose the technical foundations underneath. Hopefully, doing so will foster better development of the field through awareness of existing (albeit scattered) research, and will ultimately lead to even more empowered citizen developers.

## Problem Statement

If low-code is the solution, then what is the problem? Given the term low-code, it might seem the answer is obviously code. Unfortunately, that answer is superficial and nonconstructive. Defining a thing solely by what it is not, as the term low-code appears to do, causes confusion. Consider two other recent similarly named trends: NoSQL and serverless. At the surface, one might think NoSQL was mostly about rejecting SQL, but in fact, it was more about flexible data and consistency models than about the query language. Similarly, serverless computing was not about eliminating compute servers, but about hiding them behind better abstractions. Defining a new trend by rejecting an old one grabs attention at the expense of being misleading. Just like serverless still needs servers, low-code (and even no-code!) still needs code.

The three terms—low-code, NoSQL, and serverless—have one thing in common: a desire to avoid specific baggage while preserving core value. In NoSQL, the core value is durable and consistent storage. In serverless, it is portable and elastic compute. What then is the core value that low-code aims to preserve? This article argues it is computer programming. Programming is to low-code what computing is to serverless. Low-code is about creating instructions for a computer to execute or interpret. These instructions form a computer program, typically in a domain-specific language (DSL). For instance, low-code is often based on search-based program synthesis, and synthesis usually targets a DSL carefully crafted for the purpose.<sup>2</sup> The program may not be exposed to the user, but it is there.

One way to better understand the problem statement behind low-code is to look at who it is for. The top portion of Figure 1 shows the spectrum of low-code users. They range from citizen developers at one end to pro-developers at the other, with intermediate stages here dubbed semi developers. In this simplified view, users at the citizen developer end of the spectrum tend to have the most domain knowledge and users at the pro-developer end have the most programming expertise. Low-code can enable citizen developers to self-serve their programming needs instead of depending on pro-developers. At the



**Just like  
serverless still  
needs servers,  
low-code  
(and even  
no-code!) still  
needs code.**



same time, low-code can make pro-developers more productive, for example, in a new domain. Finally, low-code can break barriers between developers across the spectrum and help them collaborate on common ground.

The middle portion of Figure 1 shows three representative low-code techniques. Programming expertise induces a Venn diagram over the users, with the smallest subset being able to use the largest range of programming techniques. An edge between a set of users and a low-code technique indicates the users write or read a program with that technique. Specifically, all users can use programming by demonstration and programming by natural language (edges to the outermost set of users encompassing citizen-, semi-, and pro-developers). Only semi-developers and pro-developers can readily use visual programming, though citizen developers may be easily trained to do so, as evidenced by Scratch.<sup>29</sup> And only pro-developers are likely to directly use a DSL. Therefore, while low-code typically targets a DSL, that DSL may not be exposed, or if it is, may only be exposed to pro-developers. That is especially true in the common case of a DSL that is embedded<sup>17</sup> in a general-purpose textual programming language such as Python.

If the core value of low-code is to create computer programs, what exactly is it about created programs that is deemed valuable? One way to shed more light on this question is to look at a seemingly opposing trend, namely the as-code movement. The as-code movement started with infrastructure as code, which automates standing up compute resources and the services running on them from a source code repository and a backup.<sup>18</sup> Treating this process as code can speed it up, reduce mistakes, and facilitate testing. Another instance of as-code is security as code, where security policies, templates, and configuration files all live in a source code repository.<sup>25</sup> Treating them as code lets them be versioned, inspected by humans, and checked by machines. To summarize, the as-code movement sees value in programs that are repeatable, tested, versioned, human-readable, and machine-checkable. These are also desirable properties for low-code programs.

When citizen developers use low-

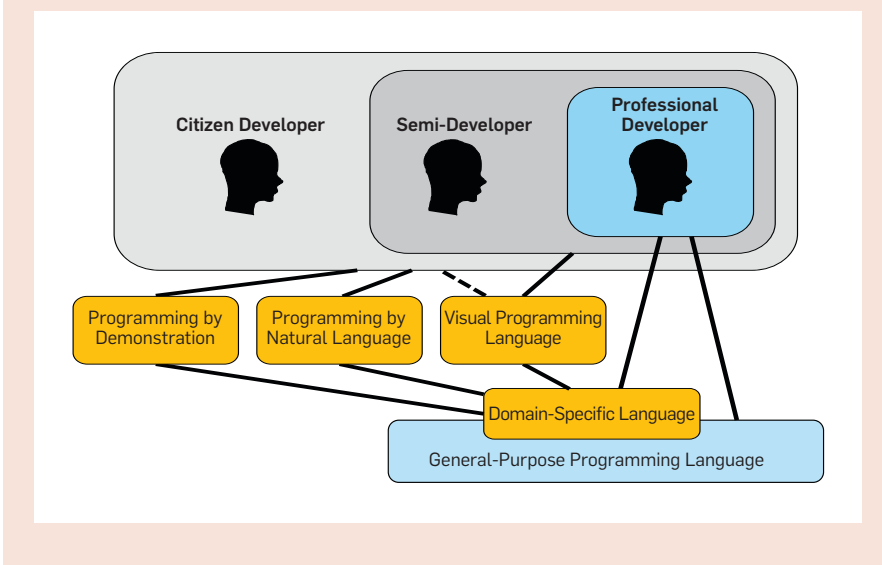
code, it is typically to create a program for a task they would otherwise do by hand. So what tasks is low-code good for? Generally speaking, low-code helps if it shaves off more time from a task than the time spent doing the low-code programming. This is true for tasks that are repetitive or time-consuming. Of course, the equation shifts when the program can be used not just by the developer who created it, but also by others, shaving time off their tasks as well. In the extreme, pro-developers create programs used by millions. Low-code is most appropriate when it saves time, but not enough time to make professional coding economically feasible. Low-code is suitable for tasks that are rule-based and low on exceptions. And besides the time savings, it can be even more beneficial when the tedium of doing the task by hand causes errors.

### Techniques

Here, we take a deep-dive into three representative techniques for low-code programming: VPLs, PBD, and PBNL. These three are a good set for the following reasons. Sahay et al.'s paper declares low-code as synonymous with just one technique, VPLs,<sup>31</sup> but that perspective seems too narrow. Barricelli et al. list 14 different techniques for EUP,<sup>6</sup> but they are not clearly separated, and reviewing them all in detail would get too long-winded. In the past, the dominant low-code technique has been spreadsheets.<sup>9</sup> The three chosen techniques instead align with present and future trends: VPLs are central to current commercial low-code platforms;<sup>31</sup> PBD is the backbone of RPA, which often uses record-and-replay;<sup>35</sup> and PBNL is poised to grow thanks to advances in deep learning-based large language models.<sup>11,33,39</sup>

Furthermore, the three techniques are well-suited for citizen developers by drawing upon universal skills: VPLs draw upon seeing, PBD draws upon the ability to use a computer application, and PBNL draws upon speaking. In fact, low-code can offer an alternative modality when some other approach is impeded, such as using speech interfaces when a user's hands or eyes are unavailable. Finally, VPLs, PBD, and PBNL are sufficient to span a set of building blocks that can also be ar-

**Figure 1. Low-code users and techniques.**



ranged differently for use with other low-code techniques, such as spreadsheets, rules, wizards, or templates. Not all building blocks appear in all techniques, but the following blocks recur enough to warrant brief up-front definitions:

- code canvas: renders code, for example, visually as a flow graph;
- palette: offers components for drag-and-drop selection;
- text box: holds natural-language text used for code search, description, or generation;
- player: has buttons for capture, replay, pause, or step;
- stage: shows the effect of code execution; and,
- configuration pane: lets the user customize components, for example, via graphical controls such as checkboxes or sliders, or textually by typing small formulas.

Low-code techniques support not just writing programs, but also reading and executing them. A low-code system can execute the program immediately after it is written or save it for later, and the user may choose to execute the program multiple times, for example, after input data changes. Low-code techniques differ in which of the listed building blocks are engaged to read, write, or execute programs. Whereas Figure 1 blurred the read/write/execute distinction by using undirected edges, the rest of this section explicates the distinction by using directed edges and

colors (orange for read, dark blue for write, and purple for execute).

**Visual programming languages.** *The user drags visual components from a palette to a canvas, connects them, and configures them.*

*Description.* Visual programming languages let users write programs by directly manipulating their visual representation. There is a plethora of possible visual representations,<sup>8</sup> often inspired by domain notation, such as electrical circuit diagrams. Two prominent domain-independent visual representations are boxes-and-arrows (for example, BPMN<sup>27</sup>) or interlocking puzzle pieces (for example, Scratch<sup>29</sup>). Here, boxes or puzzle pieces represent instructions in the program, and arrows between boxes or the interlock of pieces represent how data and control flows between instructions.

Despite the diversity in visual languages, their programming environments tend to comprise similar building blocks, as depicted in Figure 2. The central building block is the code canvas, where the user can both read (orange arrow from canvas to eye) and write (dark blue arrow from hand to canvas) the program. Writing the program also involves dragging components from the palette to the canvas and possibly configuring them in a separate configuration pane. The programming environment also often includes a stage, which visually shows a program execution, ideally live.<sup>34</sup> For

example, in Scratch, the stage shows sprites in a virtual world. Besides making the environment more engaging, the stage is also crucial for program understanding and debugging. To facilitate this, the stage is usually tightly connected to the canvas, helping the user navigate back and forth.

*Strengths, weaknesses, and mitigations.* One strength of VPLs is they tend to be easy to read, especially when re-using notation already familiar to the

domain expert,<sup>8</sup> or in visual builders for graphical user interfaces.<sup>24</sup> Another strength is that, in contrast to PBD or PBNL, VPLs are usually unambiguous, thus increasing programmer control and reducing mistakes. Finally, compared to textual programming languages, visual languages can rule out syntax errors<sup>37</sup> and even simple type errors<sup>29</sup> by construction.

In the context of low-code programming, the main weakness of visual

programming languages is they are not always self-explanatory; that is why Figure 1 connects them to semi-developers. The mitigation for this need-to-learn is user education, and for some VPLs, education is a primary purpose.<sup>29</sup> The visual notation can take up a lot of screen real estate; the mitigation is to elide detail, for example, by requiring a configuration pane or via modular language constructs.<sup>3,26</sup> Even the palette can get too full, hindering discoverability, which can be mitigated by search facilities. A drawback of visual languages compared to textual languages is they tend to be co-dependent on their visual programming environment, hindering the use of basic tools such as diffing or search, or of third-party tools such as linters or code generators. This can be mitigated by backing the visual language with a textual domain-specific language.<sup>37</sup>

*Literature.* Some seminal VPLs include BPMN-on-BPEL for modeling and executing business processes<sup>27</sup> and the Scratch language for teaching kids programming.<sup>29</sup> Boshernitsan and Downes chronicle early VPLs and categorize them into purely visual vs. hybrid (mixed with text), and complete (sufficient procedural abstraction and data abstraction to be self-hosting) or not.<sup>8</sup> Today, VPLs are central to commercial low-code platforms such as Appian, Mendix, and OutSystems.<sup>31</sup>

Other papers address VPL implementation approaches, such as meta-tools (tool used to implement other tools) and the model-view-controller (MVC) pattern, which lets users manipulate the same model through multiple synchronized views. VisPro is a meta-tool for creating visual programming environments.<sup>40</sup> VisPro advocates for a coordinated set of visual and textual languages, using MVC to expose the same program (model) via multiple languages (views). More recently, Blockly is a meta-tool for creating VPLs with interlocking puzzle pieces<sup>28</sup> such as those in Scratch. Some VPLs target pro-developers and are embedded in professional programming environments or languages. Projectional editing, such as in MPS,<sup>37</sup> doubles down on the MVC paradigm, where even the textual language is projected into a view precluding syntax errors. More recent work has demonstrated VPLs as li-

Figure 2. Visual programming languages.

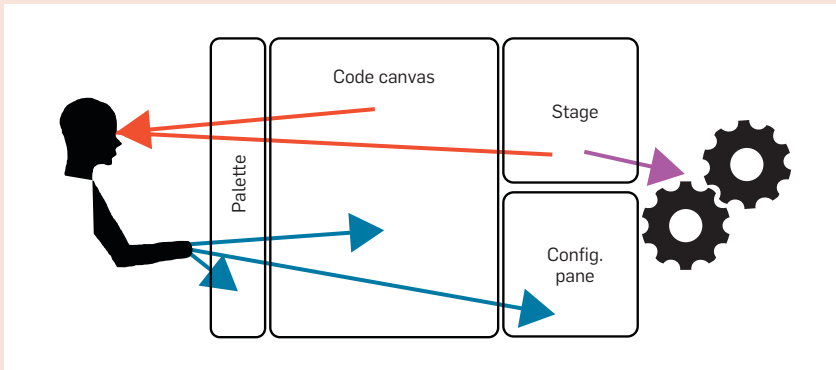


Figure 3. Programming by demonstration.

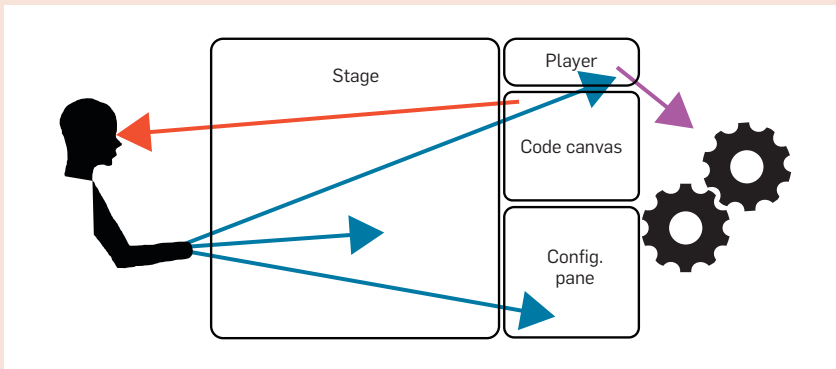
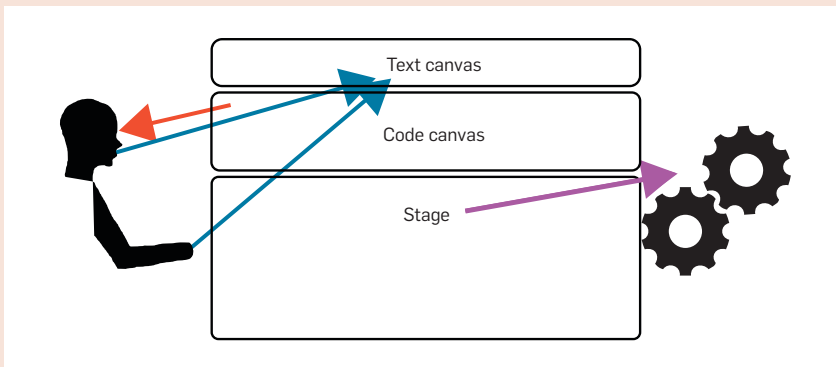


Figure 4. Programming by natural language.





braries extending textual languages. A livelit is a user-defined VPL widget that can be used in place of a textual literal,<sup>26</sup> and Andersen et al. let users implement VPL widgets for literals, patterns, and templates.<sup>3</sup>


#### **Programming by demonstration.**

*The user demonstrates the behavior on a canvas, with some configuration during or after recording.*


*Description.* In PBD, the user demonstrates how to perform a task by hand via the mouse and keyboard, and the PBD system records a program that can perform the same task automatically. As shown in Figure 3, the demonstration happens on a stage, which may be a specific application like a spreadsheet, or a Web browser visiting a variety of sites and apps, or even a general computer desktop or smartphone screen. Ideally, the recorded program abstracts from perceptions to a symbolic representation, for instance, by mapping pixel coordinates to a user-interface widget, or several keystrokes to a text string. Besides the stage, most PBD systems have a player with buttons to record and replay, plus often additional buttons such as pause or step (reminiscent of interactive debuggers).

The program is most useful if executing it does not yield the same behavior as the initial demonstration, but rather, generalizes to different data. For example, a program for ordering a taxi to any new location is more general and more useful than a program for ordering a taxi to only a single hard-coded location. Generalizing typically requires identifying variables or parameters, and may even entail adding conditionals, loops, or function calls. Unfortunately, a single demonstration is an inherently ambiguous specification for such a more general program. Therefore, PBD systems often provide a configuration pane that allows users to disambiguate the generalization either during or after demonstration. Some PBD systems also have a code canvas that renders the recorded program for the user to read, for example, visually or in natural language.

*Strengths, weaknesses, and mitigations.* The main strength of programming by demonstration is that the user can work directly with the software applications they are already famil-



**To turn a demonstration into a program, it must be generalized, and automatic generalization may not capture user's intent.**



iar with from their day-to-day work.<sup>21</sup> This makes PBD well suited for citizen developers, as there is no indirection between programming and execution. Furthermore, a demonstration is more concrete than a program in a different paradigm, since it works on specific values and has a straight-line flow of control and data.

Unfortunately, being so concrete is also PBD's main weakness: to turn a demonstration into a program, it must be generalized, and automatic generalization may not capture the user's intent.<sup>14</sup> Mitigations include hand-configuration<sup>21</sup> or multishot demonstration.<sup>15</sup> PBD can be brittle with respect to the graphical user interface of the application on stage, especially when that changes; mitigations include heuristics and specialized recorders that can map perception to application-level concepts.<sup>32</sup> Generalization can also overshoot, allowing a program to plow ahead even in unforeseen circumstances.<sup>16</sup> This can be mitigated by providing guardrails, such as an attended execution mode that asks the user to confirm before certain actions. Finally, PBD can result in programs that are difficult to understand because they include spurious steps or are too fine-grained, which is of course a problem in low-code programming.<sup>10</sup> This can be mitigated by pruning and by discovering macro-steps.

*Literature.* A good example of a PBD system is CoScripter, where the stage is a Web browser and the code canvas displays the program in natural language.<sup>21</sup> The CoScripter paper describes interviews that informed its design, as well as experiences from real-world usage in a business setting. In Rousillon, the stage is also a Web browser and the canvas displays the program in a VPL, fusing sequences of several low-level steps into a single puzzle piece.<sup>10</sup> In VASTA, the stage is the display of a mobile phone, and the system uses machine learning to reverse-engineer screenshots into user interface elements.<sup>32</sup> In DIYA, the stage is a Web browser and users customize the program during recording via voice input.<sup>14</sup> PBD is used in commercial robotic process automation products (such as UIPath, Automation Anywhere, and BluePrism) that let a human demonstrate a process on the

existing software and then refer to the automatic replay engine as a robot.<sup>35</sup>

PBD is closely related to PBE, since a demonstration is an elaborate example. FlashFill is a seminal PBE system that uses example input and output columns in a spreadsheet to synthesize a program for transforming inputs to outputs.<sup>15</sup> Both PBD and PBE are based on program synthesis.<sup>2</sup> Recent work has harnessed novel machine-learning techniques for program synthesis, such as learned search strategies in DeepCoder<sup>5</sup> and learned libraries in DreamCoder.<sup>13</sup>

PBD can be profitably combined with other low-code techniques. The play-in/play-out approach is a PBD system codesigned with its own VPL based on sequence diagrams.<sup>16</sup> And SwaggerBot is a PBD system embedded in a natural-language conversational agent, enabling a form of PBNL.<sup>36</sup>


#### **Programming by natural language.**

*The user enters natural language text via keyboard or voice, and the system synthesizes a program.*


*Description.* In this low-code technique, the user enters text in natural language, either by typing on the keyboard or via speech-to-text. Figure 4 indicates these two possibilities via blue arrows from the user's hand or mouth to the text canvas. The PBNL system translates the user's text, or utterance, to a program. The system can optionally render the program on a code canvas for the user to read. This rendering might use a VPL, or it might use a controlled natural language<sup>20</sup> for a disambiguated version of the user's utterance. The system can also optionally show the effect of the program's execution on a stage. For example, if the program is a query in a spreadsheet, the spreadsheet is the stage, and the result can be shown as a new table.

*Strengths, weaknesses, and mitigations.* The main strength of PBNL is that it is not just low-code, but more generally, low on demands during programming. As shown in Figure 4, its programming environment has only three building blocks (text canvas, code canvas, and stage), all optional. That means PBNL in principle even works in circumstances where the user's hands and eyes are otherwise occupied.

PBNL makes it particularly easy for citizen developers to create programs,



**A strength of programming by natural language is its expressiveness: natural language can express virtually anything humans want to communicate.**



but unfortunately, those programs are often wrong.<sup>4</sup> Natural language is ambiguous, since humans are often vague and tend to assume common ground and omit context. On top of that, natural language processing (NLP) technologies are imperfect. The optional code canvas and stage can mitigate this weakness, by showing the user the synthesized program or its effect, thus giving them a chance to correct it. Another mitigation is to encourage users to keep their utterances short and not take advantage of the full expressiveness of natural language, since simpler programs are easier to get right.<sup>22</sup> Furthermore, some PBNL systems support hand-editing the program.

Another strength of PBNL is its expressiveness: natural language can express virtually anything humans want to communicate. In theory, PBNL restricts neither the sophistication nor the domains of programs. On the flip-side, PBNL systems often require an aligned corpus of utterances and programs to train NLP models, and obtaining such a corpus is expensive. Mitigating this is an active research topic in the machine-learning research community.<sup>33,38</sup>

*Literature.* As an interdisciplinary field of research, PBNL is best illuminated through multiple surveys. Androutsopoulos et al. surveyed natural-language interfaces to databases, a prominent form of PBNL going back to the 1960s.<sup>4</sup> A common approach is to parse a natural-language utterance into a tree and then map that tree to a database query. Kuhn surveyed controlled natural languages (CNLs), which restrict inputs to be unambiguous while preserving some natural properties.<sup>20</sup> Compared to unrestricted natural language, CNLs may make it harder for citizen developers to write programs but may make it easier to write correct programs. Allamanis et al. surveyed machine learning for code, arguing that code has a “naturalness” that makes it possible to adapt various NLP technologies to work on code.<sup>1</sup> The survey covers some code-generating models relevant to PBNL.

The most successful NLP technology applied to PBNL is semantic parsers, which are machine-learning models that translate from natural language to an abstract syntax tree (AST) of a pro-

gram. For instance, SILT learns rule-based semantic parsers that have been demonstrated for programs that coach robotic soccer teams or for programs that query geographic databases.<sup>19</sup> The Overnight paper addresses the problem of obtaining an aligned corpus for training a semantic parser via synthetic data generation and crowdsourced paraphrasing.<sup>38</sup> Pumice tackles the ambiguity of natural language by a dialogue, where the system prompts for clarification which the user can provide via natural language or demonstration.<sup>22</sup> And Shin et al. show how to coax a pretrained large language model into doing semantic parsing without requiring fine-tuning.<sup>33</sup>

Another approach to PBNL is program synthesis, which typically searches a space of possible programs.<sup>2</sup> Desai et al. describe a meta-synthesizer that, given a DSL grammar and an aligned corpus, creates a synthesizer from natural language to programs in the DSL.<sup>12</sup> PBNL is not limited to domain-specific languages for citizen developers. Yin and Neubig describe a semantic parser that uses deep learning to encode a sequence of natural-language tokens, then decodes that into a Python AST.<sup>39</sup> Codex is a pre-trained large language model for natural language first fine-tuned on unlabeled code, then fine-tuned again on an aligned corpus of utterances and programs.<sup>11</sup>

## Perspectives

While the previous discussion covered three low-code techniques in depth, here we cover cross-cutting topics beyond any single technique. The accompanying table compares the techniques discussed earlier. The Activity columns indicate how each technique supports the user in writing, reading, and executing programs. The main difference is in the Write column: users write programs mainly on the code canvas for VPLs, the stage for PBD, and a text canvas in

PBNL. On the other hand, there is little difference in the Read and Execute columns: users read programs on a code canvas (if provided), and watch them executing on the stage (if visible). That hints at an opportunity for reuse across tools for different techniques.

A core problem with low-code programming is ambiguity. While visual programming languages can be rigorous and unambiguous, there is ambiguity in how to generalize from a demonstration to a program that works in different situations, and natural languages are inherently ambiguous as well. More ambiguous techniques may only work reliably on small and simple problems. Systems for PBD and PBNL must guess at the user's intent and are likely to guess wrong when programs get complicated. This motivates offering users an option to read or even correct programs or their executions.

A core goal of low-code programming is to reduce the need to learn a programming language. Citizen developers can demonstrate a program or describe it in natural language without having been taught how to do so. Visual programming is often less self-explanatory, which is why Figure 1 associates it more with semi-developers. On the other hand, depending on the user's attitude, the need-to-learn can also be good, since it grows computational thinking skills.

### *Artificial intelligence for low-code.*

Does the ongoing rapid progress in AI fuel progress in low-code? This article argues that yes, it does, in proportion to the ambiguity of the low-code technique. Out of the three techniques in the table, AI is most prominent for PBNL, which is also the most ambiguous. PBNL can hardly avoid AI except by using a controlled natural language,<sup>20</sup> but that would make it feel more like code. Currently a rising AI approach for PBNL is to use large language models with code generation.<sup>11,33</sup> PBNL will likely

grow along with relevant advances in AI. AI is also prominent in PBD, characterized in the table as medium ambiguity. For example, DeepCoder shows the interplay between program synthesis for defining a space of possible programs and checking whether a given program is correct, and AI for guiding the search through that space.<sup>5</sup> As another example, VASTA uses speech recognition, object recognition, and optical character recognition to better understand a user's demonstration of a task.<sup>32</sup>

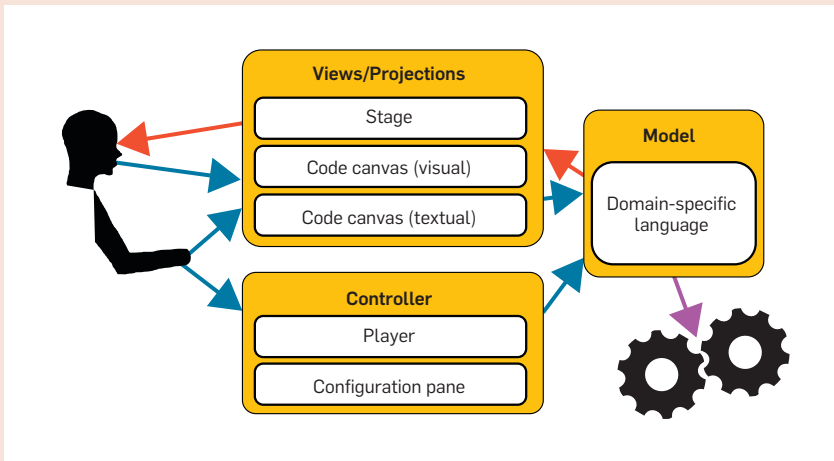
*Communicating with humans and machines.* Pro-developers use code in textual programming languages to communicate with a computer, telling it what to do. In addition, developers can also use programming languages to communicate with each other or with their own future self. A low-level programming language such as C gives developers more control of the computer, whereas a high-level language such as Python arguably makes communication among humans more effective. Similarly, low-code programs can serve both to communicate instructions to a computer and to communicate among low-code users. Being even more high-level than, say, Python, low-code can serve as a lingua franca to help citizen developers and pro-developers communicate more effectively with each other. For instance, a citizen developer might use PBD to communicate a desired behavior to a pro-developer to flesh out.<sup>16</sup> Conversely, a pro-developer might use PBNL or a VPL to communicate a proposed behavior to a domain expert for explanation or approval.<sup>21,27</sup>

*Domain-specific languages for low-code.* All three low-code techniques noted earlier are intrinsically related to DSLs: most VPLs are DSLs (for example, Scratch<sup>29</sup>), and both programming by demonstration and programming by natural language usually target DSLs (DIYA targets its co-designed Thing-Talk 2.0 DSL<sup>14</sup>). Mernik et al. list further

### Comparing low-code techniques.

Technique	Activity			Ambiguity	Need to learn
	Write	Read	Execute		
Visual programming languages	code canvas, palette, config. pane	code canvas	stage	low	medium
Programming by demonstration	stage, player, config. pane	code canvas	player, stage	medium	low
Programming by natural language	text canvas	code canvas	stage	high	low



**Figure 5. Model-view-controller for low-code.**

benefits of DSLs: they facilitate program analysis, verification, optimization, parallelization, and transformation (AVOPT).<sup>23</sup>

While reviewing the low-code literature reveals a close tie to DSLs, those DSLs are not always exposed to the user. For instance, the DSL may manifest as a proprietary file format or as an undocumented internal representation. If the DSL is exposed, users can more easily read, test, and audit programs, version them and store them in a shared repository, and manipulate them with tools for program transformation or generation. Also, an exposed DSL is less locked into a specific programming environment or its vendor. When exposed, the DSL should be designed for humans, possibly based on interviews and user studies as role-modeled by Leshed et al.<sup>21</sup> On the other hand, a DSL that is not exposed will be shaped by different factors, such as the ease of enumerating valid programs, which can be improved by breaking symmetries in the search space.<sup>13</sup>

DSLs (including DSLs for low-code) may be embedded in a general-purpose language. Compared to a stand-alone DSL, an embedded DSL is often easier to implement (for example, due to not requiring a custom parser) and easier to use (due to syntax highlighting and auto-completion tools of the host language). The approach to implementing an embedded DSL depends on the facilities of the host language. One approach is Pure Embedding, which uses higher-order functions and lazy evaluation, such as in Haskell.<sup>17</sup> Another example

is Lightweight Modular Staging, which uses operator overloading and dynamic compilation, such as in Scala.<sup>30</sup>

*Model view controller.* The current state-of-the-art VPLs and associated meta-tools are based on the MVC pattern.<sup>28,40</sup> And in PBD or PBNL, even though the user does not use a code canvas to write a program, the system may optionally provide one for reading it, again using MVC. Figure 5 illustrates MVC with a superset of the components from each low-code technique. Low-code programming tools provide one or more views of the program. Some of these views are read-only, while others are read-write views. When multiple views are present, the system keeps them in sync with a single joint model, and through that, with each other. Edits in one view are projected live to all other views. The model is a program in a DSL. Optionally, the system may even expose the textual DSL as another view, for instance, in a structure editor.<sup>37</sup> Besides the model and the view, the third part of the MVC pattern is the controller, which, for low-code, can contain a player and/or a configuration pane.

*Combining multiple low-code techniques.* When users write a program by demonstration or by natural language, the system may let them read it on a code canvas. And once a system lets users read programs on a code canvas, a logical next step is to also let them write programs there, such as, to correct mistakes from generalization or from natural language processing. This yields a combination of low-code techniques,

where users can write programs in multiple ways. Such combinations can compensate for weaknesses of techniques. For example, in Rousillon, the user first writes a program by demonstrating how to scrape data from web pages;<sup>10</sup> since one weakness of PBD is ambiguity, Rousillon lets the user read the resulting program in a scratch-like VPL.<sup>10</sup> Pumice combines PBD with PBNL: the user first writes a program via natural language; since one weakness of PBNL is ambiguity, Pumice next lets the user clarify with PBD.<sup>22</sup>

*Meta-tools and meta-circularity.* A meta-tool for low-code is a tool used to implement low-code tools. In traditional programming languages, meta-tools (such as parser generators) have long been an essential part of the tool-writer's repertoire. Similarly, meta-tools for low-code can speed up the development of low-code tools by automating well-known but tedious pieces. Thus, meta-tools make it easier to build several tools or variants, for instance, to experiment with the user experience. There are examples of meta-tools for all three low-code techniques discussed previously. Blockly<sup>28</sup> is a tool for creating VPLs that look similar to Scratch; DreamCoder<sup>13</sup> is a tool for learning a library of reusable components along with a neural search policy for PBE; and Overnight<sup>38</sup> is a tool for building semantic parsers for PBNL with synthetic training data.

A meta-circular tool for low-code is a meta-tool for low-code that is itself a low-code tool. Not all meta-tools are metacircular tools, as that requires them to be powerful enough for serious software development. Supporting all that power can compromise the tool's low-code nature: complex features can get in the way of learning easy ones. On the positive side, meta-circular tools can democratize the creation of low-code tools themselves. Furthermore, tool developers who use their own tools may empathize more with their users' needs. Examples for meta-circular low-code tools include VisPRO<sup>40</sup> and Racket<sup>3</sup> (both for VPLs).


*Low-code foundation.* In addition to meta-tools, are there other reusable modules that make it easier to build new low-code tools? The beginning of the Techniques section listed several reusable building blocks for low-code programming interfaces: code can-

vas, palette, text box, player, stage, and configuration pane. Besides making it easier to create low-code tools, such reuse can also give different tools a more uniform look-and-feel, thus reducing the need-to-learn. In the case of multiple low-code tools for the same domain, reusing the same domain-specific language makes them more interoperable. Of course, low-code tools in different domains will require different DSLs, but they may still be able to reuse some sublanguage, such as expressions or formulas with basic arithmetic and logical operators and a function library. There are also AI components that can be reused across low-code tools, such as speech recognition modules, a search-based program synthesis engine, semantic parsers, or language models.

**End-user software engineering.** Most of the discussion on low-code programming focuses on writing a program: low-code enables citizen developers to rapidly create a prototype. But what happens over time when these programs stick around, get used in new circumstances that the developer did not foresee, get modified or generalized, and proliferate? At that point, users need end-user software engineering (EUSE) for quality control, for instance, by showing test coverage, letting users add assertions, and helping them localize faults directly in their low-code programming environment.<sup>9</sup> Citizen developers often struggle with anticipating exceptional contexts for their programs; Pumice is a low-code tool that lets users extend programs with new branches when the unforeseen happens.<sup>22</sup> Another way to support EUSE is to expose the DSL, which makes it easier to adopt established software development workflows and the associated tools (such as version-controlled source code repositories, regression tests, or issue trackers) for low-code. Those tools also facilitate collaboration between citizen developers and professional software engineers.

## Conclusion

This article reviews research relevant to low-code programming models with a focus on visual programming, programming by demonstration, and programming by natural language. It maps low-code techniques to target

users and discusses common building blocks, strengths, and weaknesses. This article argues that domain-specific languages and the model-view-controller pattern constitute a common backbone and unifying principle across low-code techniques. 

## References

- Allamanis, M. et al. A survey of machine learning for big code and naturalness. *ACM Computing Surveys* 51, 4 (July 2018), 81:1–81:37; <https://doi.org/10.1145/3212695>
- Alur, R. et al. Search-based program synthesis. *Commun. ACM* 61, 11 (Nov. 2018), 84–93; <https://doi.org/10.1145/3208071>
- Andersen, L. et al. Adding interactive visual syntax to textual code. In *Proceedings of 2020 Conf. Object-Oriented Programming, Systems, Languages, and Applications*; <https://doi.org/10.1145/3428290>
- Androulopoulos, I. et al. Natural language interfaces to databases—An introduction. *Natural Language Engineering* 1, 1 (1995), 29–81; <https://doi.org/10.1017/S135132490000005X>
- Balog, M. et al. DeepCoder: Learning to write programs. In *Proceedings of 2017 Intern. Conf. Learning Representations*; <https://openreview.net/forum?id=BydLrqlx>
- Barricelli, B.R. et al. End-user development, end-user programming and end-user software engineering: A systematic mapping study. *J. Systems and Software* 149, (2019), 101–137; <https://doi.org/10.1016/j.jss.2018.11.041>
- Bock, A.C., and Frank, U. Low-code platform. *Business & Information Systems Engineering* 63, (2021), 733–740; <https://doi.org/10.1007/s12599-021-00726-8>
- Boshernitsan, M., and Downes, M. Visual Programming Languages: A Survey. *Technical Report UCB/CSD-04-1368, 2004, UC Berkeley*; <https://bit.ly/3P0SaX5>
- Burnett, M. et al. End-user software engineering. *Commun. ACM* 47, 9 (Sept. 2004), 53–58; <https://doi.org/10.1145/1015864.1015889>
- Chasins, S. et al. Rousillon: Scraping distributed hierarchical Web data. In *Proceedings of 2018 Symp. User Interface Software and Technology*. 963–975; <https://doi.org/10.1145/3242587.3242661>
- Chen, M. Evaluating large language models trained on code. (2021); <https://arxiv.org/abs/2107.03374>
- Desai, A. Program synthesis using natural language. In *Proceedings of 2016 Intern. Conf. Softw. Eng.*, 345–356; <https://doi.org/10.1145/2884781.2884786>
- Ellis, K. DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of 2021 Conf. Programming Language Design and Implementation*. 835–850; <https://doi.org/10.1145/3453483.3454080>
- Fischer, M.H. et al. DIY assistant: A multi-modal end-user programmable virtual assistant. In *Proceedings of 2021 Conf. Programming Language Design and Implementation*. 312–327; <https://doi.org/10.1145/3453483.3454046>
- Gulwani, S. Automating string processing in spreadsheets using input-output examples. In *Proceedings of 2011 Symp. Principles of Programming Languages*. 317–330; <https://doi.org/10.1145/1926385.1926423>
- Harel, D., and Marelly, R. Specifying and executing behavioral requirements: The play-in/play-out approach. *Software and Systems Modeling* 2, (2003), 82–107; <https://doi.org/10.1007/s10270-002-0015-5>
- Hudak, P. Modular domain specific languages and tools. In *Proceedings of 1998 Intern. Conf. Software Reuse*. 134–142; <https://doi.org/10.1109/ICSR.1998.685738>
- Jacob, A. Infrastructure as code. *Web Operations: Keeping the Data on Time*. J. Allspaw, and J. Robbins, (eds). O'Reilly, Chapter 5, (2010), 65–80.
- Kate, R.J. et al. Learning to transform natural to formal languages. In *Proceedings of 2005 Conf. Artificial Intelligence*. 1062–106; <http://www.aaai.org/Library/AAAI/2005/aaai05-168.php>
- Kuhn, T. A survey and classification of controlled natural languages. *Computational Linguistics* 40, 1 (2014), 121–170; <https://bit.ly/42t3JsY>
- Leshed, G. et al. CoScripter: Automating and sharing know-to knowledge in the enterprise. In *Proceedings of 2008 Conf. Human Factors in Computing Systems*, 1719–1728; <https://doi.org/10.1145/1357054.1357323>
- Li, T.J. et al. PUMICE: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. In *Proceedings of 2019 Symp. User Interface Software and Technology*. 577–589; <https://doi.org/10.1145/3332165.3347899>
- Mernik, M. et al. When and how to develop domain-specific languages. *ACM Computing Surveys* 37, 4 (2005), 316–344; <https://doi.org/10.1145/1118890.1118892>
- Myers, B. et al. Past, present, and future of user interface software tools. *Trans. Computer-Human Interaction* (Mar. 2000), 3–28; <https://doi.org/10.1145/344949.34495>
- Myrbakken, H., and Colomo-Palacios, R. DevSecOps: A multivocal literature review. *Software Process Improvement and Capability Determination* (2017), 17–29. [https://doi.org/10.1007/978-3-319-67383-7\\_2](https://doi.org/10.1007/978-3-319-67383-7_2)
- Omar, C. et al. Filling Typed Holes with Live GUIs. In *Proceedings of 2021 Conf. Programming Language Design and Implementation*. 511–525; <https://doi.org/10.1145/3453483.3454059>
- Uyang, C. From BPMN process models to BPEL Web services. In *Proceedings of Intern. Conf. Web Services*. (2006); <https://doi.org/10.1109/ICWS.2006.67>
- Pasternak, E. et al. Tips for creating a block language with Blockly. In *Proceedings of Blocks and Beyond Workshop* (2017); <https://doi.org/10.1109/BLOCKS.2017.8120404>
- Resnick, M. et al. Scratch: Programming for all. *Commun. ACM* 52, 11 (Nov. 2009), 60–67; <https://doi.org/10.1145/1592761.1592779>
- Rompf, T., and Odersky, M. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM* 55, 6 (June 2012), 121–130; <https://doi.org/10.1145/2184319.2184345>
- Sahay, A. et al. Supporting the understanding and comparison of low-code development platforms. In *Proceedings of Euromicro 2020 Conf. Software Engineering and Advanced Applications*. 71–178; <https://doi.org/10.1109/SEAA51224.2020.00036>
- Sereshkeh, A.R. et al. VASTA: A vision and language-assisted smartphone task automation system. In *Proceedings of 2021 Conf. Intelligent User Interfaces*. 22–32; <https://doi.org/10.1145/3377325.3377515>
- Shin, R. Constrained language models yield few-shot semantic parsers. In *Proceedings of 2021 Conf. Empirical Methods in Natural Language Processing*, 7, 699–7715; <https://doi.org/10.18653/v1/2021.emnlp-main.608>
- Tanimoto, S.L. A perspective on the evolution of live programming. In *Proceedings of Intern. Workshop on Live Programming*. (2013), 31–34; <https://doi.org/10.1109/LIVE.2013.6617346>
- van der Aalst, W.M. et al. Robotic process automation. *Business Spsampss Information Systems Eng.* 60, (2018), 269–272. <https://doi.org/10.1007/s12599018-0542-4>
- Vaziri, M. et al. Generating Chat Bots from Web API Specifications. In *Proceedings of Symp. New Ideas, New Paradigms, and Reflections on Programming and Software*. (2017), 44–57; <http://doi.acm.org/10.1145/3133850.3133864>
- Voelter, M., and Lissón, S. Supporting diverse notations in MPS' projectional editor. In *Proceedings of Workshop on the Globalization of Modeling Languages*. (2014), 7–16; <https://hal.inria.fr/hal-01074602/file/GEMOC2014-complete.pdf#page=13>
- Wang, Y. et al. Building a semantic parser overnight. In *Proceedings of the Annual Meeting of the Assoc. for Computational Linguistics*. (2015), 1332–1342; <https://www.aclweb.org/anthology/P15-1129.pdf>
- Yin, P., and Neubig, G. A syntactic neural model for general-purpose code generation. In *Proceedings of the Annual Meeting of the Assoc. for Computational Linguistics*. (2017), 440–450; <http://dx.doi.org/10.18653/v1/P17-1041>
- Zhang, K. et al. Design, construction, and application of a generic visual language generation environment. *IEEE Trans Softw Eng.* 27, 4 (2001), 289–307; <https://doi.org/10.1109/32.917521>

**Martin Hirzel** is a research staff member and manager at IBM Research, Yorktown Heights, NY, USA.



This work is licensed under a Creative Commons Attribution International 4.0 License.