future internet

MDPI

Article

OLP—A RESTful Open Low-Code Platform

Mauro A. A. da Cruz [1,2], Heitor T. L. de Paula [1], Bruno P. G. Caputo [1], Samuel B. Mafra [1], Pascal Lorenz [2] and Joel J. P. C. Rodrigues [3,4,*]

[1] National Institute of Telecommunications (Inatel), Santa Rita do Sapucaí–MG 37540-000, Brazil; maurocruz-ter@gmail.com (M.A.A.d.C.); heitortoledo@gec.inatel.br (H.T.L.d.P.); brunocaputo@gec.inatel.br (B.P.G.C.); samuelbmafra@inatel.br (S.B.M.)
[2] Network and Telecommunication Research Group, University of Haute Alsace, 34 Rue du Grillenbreit, 68008 Colmar, France; lorenz@ieee.org
[3] Post-Graduation Program in Electrical Engineering, Federal University of Piauí (UFPI), Teresina-PI 64049-550, Brazil
[4] Instituto de Telecomunicações, 6201-001 Covilhã, Portugal
* Correspondence: joeljr@ieee.org

Abstract: Low-code is an emerging concept that transforms visual representations into functional software, allowing anyone to be a developer. However, building a low-code platform from scratch can be challenging concerning the scarce available literature about the topic. In this sense, this paper proposes an Open Low-Code Platform (OLP), a low-code solution that enables regular users to create applications. Furthermore, it presents low-code's functional and nonfunctional requirements, as well as its similarities and its differences with the no-code concept. The experience obtained while developing OLP was translated into a pipeline that details how code was transformed from the visual representations into a fully fledged application. The paper demonstrates the solution's viability and is especially useful for building a low-code platform from scratch or improving an existing one.

Keywords: low-code; compilers; programming languages; transpiler; Open Low-Code Platform

Citation: da Cruz, M.A.A.; de Paula, H.T.L.; Caputo, B.P.G.; Mafra, S.B.; Lorenz, P.; Rodrigues, J.J.P.C. OLP—A RESTful Low-code Platform. *Future Internet* **2021**, *13*, 249. https://doi.org/10.3390/fi13100249

Academic Editor: Stefano Rinaldi

Received: 2 August 2021
Accepted: 22 September 2021
Published: 25 September 2021

1. Introduction

The ubiquitous presence and growing popularization of applications driven by the ease of access to computers and smartphones also increases software development demands for mobile and desktop applications. Corporations are looking for ways to make software development faster, more accessible, and more affordable to meet these demands. When a new application (for desktop or mobile) revolutionizes, creates, or disrupts a particular market, it is not uncommon for similar applications to be introduced as competitors. The issue is that these competitors are generally introduced years after the market disruption, and a monopoly is almost established. One example is the ride-sharing giant Uber, founded in 2009, while Lyft, its biggest competitor, was founded in 2012.

The delay in introducing competitors, especially in previously inexistent markets such as ride-sharing, is mainly due to recognizing the potential market, allocating funds for the project, and development time. Big corporations generally dedicate entire teams to monitoring these market changes. They can also raise capital with little effort to acquire a disruptive company or build a similar competitor. However, these big corporations prefer to acquire these disruptive companies and only create a competitor if the acquisition does not succeed. They prefer purchasing disruptive companies because building a new application from scratch demands time and great effort.

Building an application from scratch is challenging because an intuitive and responsive graphical user interface (GUI) is necessary for widespread adoption. This is the reason most ride-sharing apps have a similar interface. Another difficulty is that

algorithms should enforce business rules and ensure user satisfaction. For example, when a ride's waiting time exceeds a certain amount of time, users get impatient and can move to a competitor ride-sharing App. Building software from scratch is also tricky because a software development team must be assembled, meaning that programming languages to use must also be decided by the team. Moreover, after a software team is hired, estimating the development time is also one of the most significant software engineering issues.

A solution to reduce the software development time is the usage of low-code and no-code software development platforms [1]. These platforms do not require extensive coding knowledge and can even be used by those with no coding background to generate basic applications, which can be crucial for companies to adapt, especially in future Internet applications. Although no-code and low-code are sometimes used as synonyms, they are different concepts and can even target different user profiles. Despite low-code and no-code platforms rising in popularity, to the best of the authors' knowledge, the literature discussing how to build them is nonexistent, and the advantages and disadvantages are not well documented. Thus, the main contributions of this paper can be summarized as follows:

- An in-depth review of the state of the art on low-code and no-code platforms;
- Presentation of low-code functional and nonfunctional requirements;
- Proposal of an architecture for low-code development platforms;
- Proposal, demonstration, and validation of OLP (Open Low-Code Platform), a low-code solution that enables regular users to create applications;
- Invaluable lessons learned throughout the project.

The remainder of the paper is organized as follows: Section 2 provides a discussion on low-code and no-code topic, its main benefits and issues, and how easier and faster software development was always in demand. Low-code functional and nonfunctional requirements are addressed in Section 3, while Section 4 provides details about the OLP construction. This low-code platform was created for a better understanding of the practical difficulties in developing a low-code platform. Section 5 shares relevant lessons and crucial decisions that were learned while developing the OLP. These lessons are especially useful for building a new solution from scratch or improving an existing one. Lastly, Section 6 concludes the paper and suggests future work on the topic.

2. No-Code, Low-Code, and Traditional Approaches

In the early days of programming languages, it was common to program in assembly languages similar to the target machine code instructions designed for specific computer architectures. These assembly languages are often referred to as low-level and were a nuisance because a certain machine program would only run on a machine of the same model. These difficulties led to high-level programming languages, such as FORTRAN and, later, COBOL. High-level programming languages abstract from the machine's hardware details by using a syntax closer to the human language. Initially, high-level programming was perceived with skepticism because of its bugs. However, as time passed, the programming paradigm changed, increasing software developers' productivity and making coding easier.

The trend to simplify software development has continued up to now in low-code and no-code software development, which has been increasing in popularity since 2014 [2]. Although their essence is the same (reduce coding and simplify software development), the two terms are not synonyms, and the difference between them is minimal and, sometimes, confusing [3]. Low-code and no-code platforms rely on visual application development without the complexity associated with traditional software development [4], thereby improving the software development experience [5]. The main difference between low-code and no-code is that, while low-code reduces code writing, no-code completely eliminates it.

The no/low-code idea derives from the fourth-generation language (4GL) concept, aiming to provide a higher level of abstraction compared to previous generations of programming languages [1]. They are often categorized into different types of domain-specific languages (DSLs), such as data management languages, database management, Web development, and many more [6]. MDSD (model-driven software development) is a concept similar to no/low-code in the sense that MDSD refers to automatic source code generation based on models, and both the model and the generated source code can be debugged [7]. The main difference between MDSD and low-code is that a no/low-code platform acts as a black box where a developer has no knowledge about how the source code was generated, the used frameworks, and, in some cases, even the programming languages. Low-code is also different from integrated development environments (IDEs) because IDEs such as Eclipse and VSCode simplify source code writing. In no/low-code, developers mostly drag objects and hardly come into contact with the generated source code, unless developers intend to move away from the platform; however, even then, platform vendors do not always provide the generated source code. Then, in most cases, developers can only modify applications through the platform, which acts like its own programming language.

The most significant advantage of no-code and low-code approaches comes from the fact that it is straightforward to build complex software, allowing organizations to quickly adapt to the ever-changing market. Moreover, the low and no-code learning curve is easier when compared to regular programming languages. These characteristics enable developers to quickly prototype software, translating into faster user feedback and a better customer experience. Thus, the rapid application development (RAD) methodology is well suited for this type of development, although any methodology can also be used. According to Gartner, the low-code market leaders are OutSystems, Mendix, Microsoft Power Apps, and Salesforce [8], and they are very similar regarding functionalities. The main difference between them comes from the fact that some have built-in integration with external systems, such as customer relationship management (CRM), social media, and payment gateways. According to Gartner, low-code platforms represent approximately 65% of all development activities [9].

Like most dilemmas regarding computer science, determining which solution to use depends on the scenario and the developer's strengths. Overall, no-code is considered simpler than low-code, but this simplicity comes with the cost of less freedom for personalization in the graphical user interface and advanced business rules. Personalization limitations can be especially troublesome when integrated with external systems or using lesser-known protocols that are not supported by the platform. Therefore, it is harder to develop enterprise-grade software using no-code platforms. Furthermore, due to its simplicity, no/low-code can be used even by individuals with no programming background [10], although those with a programming background are more likely to unlock its full potential.

In no/low-code development, the application development complexity is hidden by the platform [11], which is simultaneously the biggest benefit and also an issue. Platforms hide the complexity by providing a database, front-end, backend, and generated source code, which means developers can do little to nothing to maintain the developed applications if a platform is discontinued. This happens because the "source code" available from the developer consists of visual representations that are later transpiled by the platform into a source code. The lack of a conventional source code can be troublesome for the developer because the platform generates and optimizes all the source code on the basis of visual representations. Suppose that the platform wrongly optimizes a code block. In that case, the developer might not notice until the software has various simultaneous users. When the developers notice an inefficiency, they can only wait for an update from the platform vendor. Although advertised as reducing development costs, these platforms can be costly since their business model is generally based on monthly or yearly subscriptions.

Entirely relying on a third party without a source code also reduces the developer's ability to abandon a platform since it could mean rewriting the entire software from scratch. Therefore, if developers are not satisfied with a platform because of the existing bugs, lack of new features, or other convenient reasons, abandoning it is hardly an option. This characteristic can be especially troublesome for entrepreneurs with little coding experience that decide to jumpstart an innovative idea through no/low-code platforms. Even when an idea is a success, the platform licensing or hosting cost might not be viable, which means that it is unlikely that the entrepreneur will migrate the source code or even recover the database data. The current pricing negates one of no/low-code's biggest benefits, i.e., the ability for any person within an organization to build software, since only big corporations and software houses can afford it.

Allowing any individual within an organization to build software can also present various security risks. Software developers generally have nearly unlimited access to the database, meaning that every person can potentially access confidential data (intentionally or not). This issue can be nullified through roles that limit the developer access to specific projects or even databases. Another issue is that software always has vulnerabilities, and most of them are caused by the software developer, especially those with less experience and knowledge. For example, an application developed by an inexperienced developer can gain internal popularity (i.e., it is widely adopted within the organization of the user that created it), and its developer might not even be encrypting passwords. Since most users tend to repeat passwords, this simple oversight could have severe and dangerous consequences.

In the early days of low-code, the personalization level compared to traditional software development was very lackluster. Today, some platforms such as OutSystems allow developers a high degree of personalization [12]. Overall, the pricing of no/low-code platforms is a significant barrier to its popularization. Therefore, each corporation should evaluate the benefits and drawbacks of using this type of solution for software development. It is hard to imagine that no-code can reach a degree of personalization comparable to traditional programming, but this was also an issue for low-code in the past. The next evolution of the no/low-code concept will likely be software developed by artificial intelligence (AI) solutions where users verbally explain what is desired.

As mentioned above, no/low-code platforms are very similar, and their characteristics are summarized in Table 1.

Table 1. No/low-code main characteristics.

No-Code Characteristics	Low-Code Characteristics
Removes code-writing	Reduces code-writing
Low freedom for personalization	High freedom for personalization
Easier learning curve when compared to traditional programming	
Visual application development	
Code optimization	
Any person within an organization can build software	
Hosting and/or licensing costs	
Generally hides the true source code from the user	
Less knowledgeable or inexperienced developers are likely to create vulnerabilities due to the ease of creating applications	

Since this work focuses on low-code, only concepts associated with low-code are addressed, although most of the low-code concepts can also be applied to no-code.

3. Low-Code Platform Requirements and Architecture

Any software project is built on the basis of requirements, which can be functional or nonfunctional. Nonfunctional requirements focus on the features that applications should

provide to ensure quality of service (QoS), and they are presented in Section 3.1. Functional requirements describe which application features should be deployed, and they are presented in Section 3.2. The proposed architecture is showcased in Section 3.3.

3.1. Nonfunctional Requirements

The proposed solution addresses the following nonfunctional requirements:

(1) *Ease of use*: Since the primary goal of this type of software is to simplify software development, platforms should be intuitive to handle. In practice, intuitive means that users should be able to develop applications with minimal training and effort. This low complexity is achieved by replacing traditional source code with visual representations that allow users to manipulate system elements graphically instead of textually. This crucial change is associated with a concept known as visual programming, allowing developers to focus on software functionalities, and it is also good for educational purposes [13].

(2) *Flexibility*: Since this software enables users to build other software easily, the platform must be intuitive. Nevertheless, it should support advanced use-cases and this is only possible if the platform provides a degree of flexibility to users.

(3) *Extensibility*: The platform should provide tools or documentation that allow users to refine or extend the provided functionalities through plugins or other contributions. This feature is essential because, no matter how flexible the solution is, it is unlikely to attend to all use-cases.

(4) *Interoperability*: This software should be able to integrate with external systems [14]. Today, integration with other solutions is generally accomplished through representational state transfer (REST) application programming interfaces (APIs).

(5) *Security*: It is common for programmers to have almost unlimited access inside an organization and with access to privileged information (in smaller organizations, this is even more common). Therefore, low-code platforms should provide security mechanisms that can limit access to the source code and allow accountability. In addition, similar security mechanisms should be easily applied to the developed software, allowing profile access and methods to encrypt fields within a database securely.

(6) *Privacy*: The concept of privacy is tightly linked to security because it focuses on methods and purposes of storing, analyzing, and sharing data by a service provider. For example, a recent privacy controversy was sparked by WhatsApp, which started sharing user data with its parent company Facebook. Users were fearful that their private conversations would be shared and used for targeted advertisement within Facebook, Instagram, or other third parties. Facebook later reiterated that not even WhatsApp could access private conversations due to end-to-end encryption, which meant that such a scenario would not be possible. Knowing which data are shared and how data are stored could be especially useful in a data breach, because various data could be compromised directly or indirectly. The General Data Protection Regulation (GDPR) from Europe [15] and the California Consumer Privacy Act (CCPA) from California, USA [16] are examples of privacy protection legislation.

(7) *Scalability*: The platform should provide tools or be compatible with tools that allow software to scale vertically and horizontally, adapting to the ever-increasing demands. Vertical scaling consists of running a solution in hardware with more resources (memory, HDD, CPU cores, and processing power) [17]. Horizontal scaling is characterized by distributing the workload through various hardware [18]; instead of having a single server with powerful capabilities, the workload is distributed among several servers.

(8) *Maintainability*: Low-code platforms are constantly updated by their developers, which means that a platform should be constructed in a way that is easy to maintain.

Maintainability is an attribute that describes the simplicity of modifying, fixing bugs, or improving the performance in a software solution.

(9) *Backwards compatibility*: Low-code platforms continuously iterate and improve themselves, especially regarding usability, while trying to not being so strict that too much flexibility is lost. When visual code becomes incompatible with its previous implementation, the platform should automatically upgrade it to end-users. When done correctly, even visual code from version 2 of a platform that is incompatible with version 6 can be upgraded to version 3, which can be upgraded to version 4, and so on.

3.2. Functional Requirements

The functional requirements are the following:

(1) *Data management and event management*: A software should allow "create, read, update, and delete" (CRUD) operations performed using a database or an external API. When manipulating data directly from a database, users should be able to perform advanced database query operations. These operations will generally require users to write Structured Query Language (SQL) or even NoSQL statements. Furthermore, modern systems usually execute or respond to various events automatically, according to the received data, and the platform should also provide such as a feature.

(2) *Code transpiling*: Since the source code consists of visual representations instead of the traditional text, visual representations must be translated into a textual source code. Then, it can be written in any programming language chosen by the platform developers and should be compiled or interpreted depending on the chosen programming language.

(3) *Correctness*: The generated textual source code should not contain errors or bugs, since most platforms do not offer users the source code. Thus, users will not be able to edit code that is generated by the platform. This is valid for all the coding aspects, including a REST API or a script to generate a database.

(4) *Code optimization*: The generated code should be optimized because, otherwise, the software that it produces might be difficult to scale. Moreover, the generated source code should be easily readable by the platform developers because it is likely they will need to review it in the future.

(5) *Code verification*: All the generated code should be verifiable through automated tests. This is valid for the platform itself (when it is being built or modified by developers), s well as for end-users. The developers need to use or build code verification tools because modifications to platform code can change internal functionalities and negatively impact functionalities for the application end-users. The platform should also provide similar tools for the end-users so they can verify their functionalities, detecting bugs of their own doing, as well as bugs in the sequence of platform updates. Furthermore, automated testing is a good practice in software development that saves time in the long run and ensures product quality [19].

(6) *Code compiling/interpreting*: After the transpiled source code is verified as being correct and optimized, it should be compiled or interpreted depending on the programming language. This is a necessary step in most programming languages because, otherwise, running the application will not be possible.

(7) *Application deployment*: Deployment is a process that makes a software available to its end-users and can vary depending on the targeted end-user. In a mobile application, a deployment could mean publishing the software on an App store or running the application on a smartphone. In Web applications, it could mean running on a local computer or on a cloud server.

(8) *Produce adequate messages*: A big part of programming is the textual output produced by compilers, which is generally categorized by (*i*) information, (*ii*) warnings, and (*iii*) errors. These textual outputs are essential because they give the programmer

important feedback regarding their code. Generally, errors mean that the code will not compile, and warnings imply that some aspects could be optimized or might not work as expected. An example of a warning or error (depends on the compiler) could be the following: ***variable bar in function foo (Float bar) expects a Float but receives an Integer when called***. The previous example is not a suitable warning because there is no reference to the file where the error occurred or the line it refers to. Since low-code movement goals to facilitate programming, the location where the errors or warnings occur should be easy to identify, and the user should be able to easily interpret them.

(9) *Debugging functionalities*: Even the best software presents bugs, and debuggers enable software developers to identify and remove the source of such bugs. Furthermore, debuggers allow software to run in a completely controlled environment. In practice, this means that each line of code can only be executed after programmer presses a button. In addition, the debugging process enables developers to identify and monitoring changes in various resources such as variables and should be avoided in production environments.

(10) *Visual-code export/import*: The visual code that developers use should be easily exported (saved), enabling users to import it (load) without losing functionalities.

(11) *Textual code writing*: Low-code platforms should offer ways to manually write source code because it is one of its core functionalities (minimize code-writing, but not eliminate it). However, allowing developers to write traditional source code, the platform should also have a well-defined syntax and semantic.

(12) *Versioning and collaborative development*: Most software developers verify the impacts of their code changes immediately and, with low-code, this aspect will be the same. In visual programming, it could be easy to modify or remove visual code blocks and lose functionality, which could worsen if the erased block contains textual code. Therefore, these platforms should provide version control so that developers can go back to a previous version. Furthermore, version control allows multiple developers to simultaneously work on the same project without breaking each other's code.

(13) *Visual data modeling tools*: Users should visually configure and assert constraints when creating entities, their fields, dependencies, and even relationships with other entities.

(14) *Visual programming tools*: Since low-code programming minimizes code-writing, it should rely on visual representations that allow users to control program elements graphically. Among other aspects, these tools should offer users the ability to represent the most basic aspects of any software: (i) sequences, (ii) selections, and (iii) loops [20]. It consists of a sequence of actions completed in a specified order; in traditional programming, it executes each line in sequential order (e.g., line 5 only executes after lines 1, 2, 3, and 4). Selections formulate questions to decide which subsequent lines to execute. Loops are similar to selections because they continue to formulate questions and execute subsequent code until a specific condition is reached.

3.3. Architecture

According to the presented requirements, the architecture of the low-code platform considers the following six main components: (1) visual application modeler, (2) encoder, (3) decoder, (4) source code generator, (5) compiler, and (6) deployer. These elements are presented in Figure 1 and are described as follows:

(1) *Visual application modeler*: The front-end of the low-code platform will interact with the developers and simplify software development. The visual application modeler is an enhanced integrated development environment (IDE) that implements most of the functionalities that developers interact with. These include producing code (graphical or textual), debugging, modeling data, code verification, testing,

versioning, event management, and many more. The IDE should be simple to use [21], provide a preview of the developed software, and be located at the client side.

(2) *Encoder*: The "symbols and codes" used in the visual application modeler must be imported and exported without losing functionalities. To achieve such a goal, the "code" produced by the modeler must be expressed in a self-contained representation easily interpreted by an algorithm. This entity encodes the visual representations into a flexible format such as JavaScript Object Notation (JSON) or Extensible Markup Language (XML) that can be transmitted over the Internet. The encoder is located on the client side.

(3) *Decoder*: This entity interprets the encodings of the visual representations and is located on the server side.

(4) *Source code generator*: After the data are represented in a human-readable way, they can be transpiled by the source code generator. Transpiling consists of converting the source code from one language to another. This entity should include scripts to generate and interact with a database, and it is also responsible for code correctness, optimization, and verification.

(5) *Compiler*: This is the entity responsible for compiling the source code, and it also acts as the final code correctness test.

(6) *Deployer*: This is the entity responsible for deploying the software into a target platform that will interact with end-users.

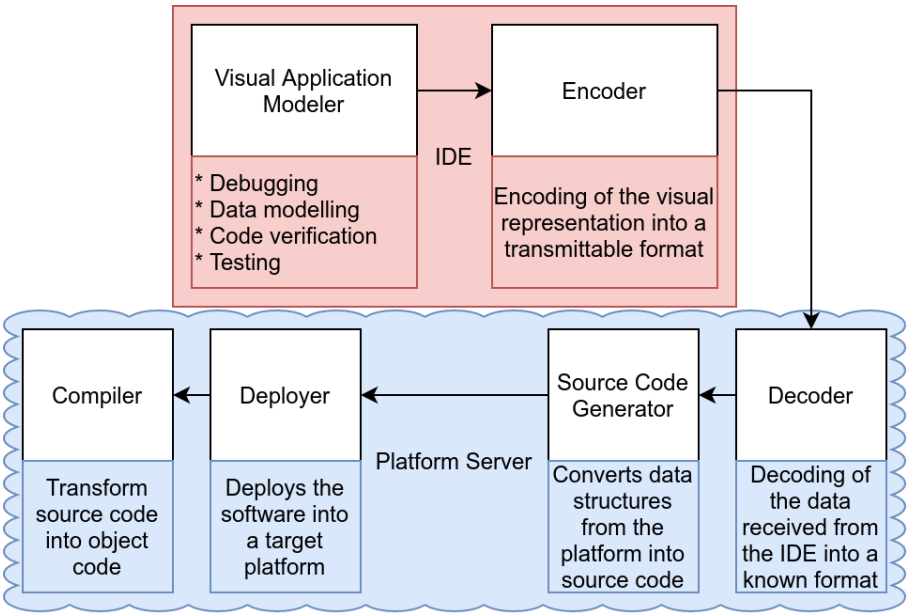

Figure 1. The general architecture of low-code platforms.

4. OLP Development and Demonstration

Developing a low-code platform from scratch is challenging because of the scarce available literature, and architectures are not always easy to interpret in practical scenarios. To facilitate the development of future low-code projects, the Open Low-Code Platform (OLP) was created. The scope was limited to Web application development, and data were persisted and consulted through REST APIs. To develop OLP, the authors drew inspiration from developing a new programming language, and the first task was to develop a syntax and, consequentially, the operator precedence. Since graphical visualizations and written source code must have syntax, it was first represented in EBNF (extended Backus–Naur form). Formalizing a syntax is helpful because the platform developers can consult it at any time, enabling productivity and cooperation. EBNF is a metasyntax used to describe other syntaxes. A logical or arithmetical expression representation

in EBNF is presented in Code 1. The other representations used in OLP can be found on the GitHub repository along with its source code [22].

Code 1. EBNF representation of an expression.

```
expression
   =   primary
   ,   [   binary-op   ,   expression   ]
   ;
```

After establishing the syntax, the developers should define a semantic that will enable the platform to differentiate between coherent and noncoherent statements. An example of an incoherent statement could be dividing a *String* by a number. Lastly, a pipeline that details how the code will be transformed from the visual representations to a written source code was established, and it is presented in Figure 2. Such a pipeline was based on the architecture presented in Figure 1 and considers two parts: IDE and code generator.

The IDE supports most of the functions that end-users will interact with and simplifies software development. These functions include producing code (graphical or textual), debugging, code verification, testing, versioning, and event management. The IDE also provides a preview of the software being developed by end-users. In OLP, the IDE is browser-based and was built using AngularJS. A browser-based IDE was chosen because Web browsers allow the HyperText Markup Language (HTML) on the screen to be inspected; OLP takes advantage of this feature and extracts it, which means that HTML is obtained directly from the browser. In addition, the code generator will translate the visual representations into source code in another programming language. When building the code generator, control flow structures frequently found in imperative languages were used because this enabled structures with a higher level of abstraction to be defined. In OLP, the code generator was built using Haskell.

In Figure 2, the upper division of the pipeline describes the processes that occur at the IDE and the bottom half describes the processes that occur on the code generator. The left side of the pipeline describes the process of analyzing and converting a low-code program into a JavaScript file, and it is called the scripting pipeline. The right inside is called the user interface (UI) pipeline and describes the process of converting the visual representations built on the IDE into CSS (Cascading Style Sheets) and HTML files. The scripting and UI pipeline occur simultaneously and are described as follows:

1: The user exports the project and transforms the visual representations into a format that can be easily encoded and decoded for future usage.
2: Programming languages generally represent data structures in compilers through a tree data structure because it resembles the production rules for syntax. In the case of the platform, an abstract syntax tree (AST) was chosen to make this representation. On the scripting side (left side) of the pipeline, the AST may contain *strings* because, even though it is a scripting language, a user is free to type any expression in text boxes. These expressions typed by the users must be parsed before being converted into a pure AST. The other side of the tree is already in the correct format as the visual representations can be converted directly into objects. In the UI (right side) of the pipeline, the tree is already assembled in the correct format because the user cannot type custom HTML.
3: The AST is converted to JSON because the AST needs to be represented in a format that humans can understand, enabling faster debugs. Then, an HTTP request is made to the compiler, where the encoded JSON is sent to the compiler.
4: The compiler receives the JSON containing the encoded AST, as well as other project data (name, the user who created it, settings, and other files).
5: On the logic side (left side of the pipeline), the JSON is decoded normally, except in the parts that were typed by the user; these are not verified by the IDE and need to

be analyzed by the semantic analyzer. The trees are already set up on the UI side (right side of the pipeline); hence, they are converted directly to ASTs.

6: On the logic side (left side of the pipeline), the semantic analyzer places data type information in the tree. It also looks at some other inconsistencies and generates warnings. This step is exclusive to logic.

7: On the logic side (left side of the pipeline), the transpiler converts the AST from step 6 into a JavaScript AST. Aspects such as data types that are supported by the platform but do not explicitly exist in Javascript are converted in this step.

8: The pretty-printer transforms ASTs into a styled and formatted source code that is easier to read. The source code is formatted and styled for the sole purpose of faster debugs, and it is the same on both sides of the pipeline.

9: The linker references the CSS and javascript files in the HTML and compresses them into a Zip file.

Most of the elements from the architecture presented in Figure 1 can be found on the pipeline. The visual application modeler represents step 1, the encoder occurs on step 2, and the decoder occurs on step 3. The source code generator was deployed in steps 4–8. The compiler from the architecture presented in Figure 1 was not included because Javascript and HTML source code does not need to be compiled (it is interpreted by the browser). The deployer will be implemented in future releases.

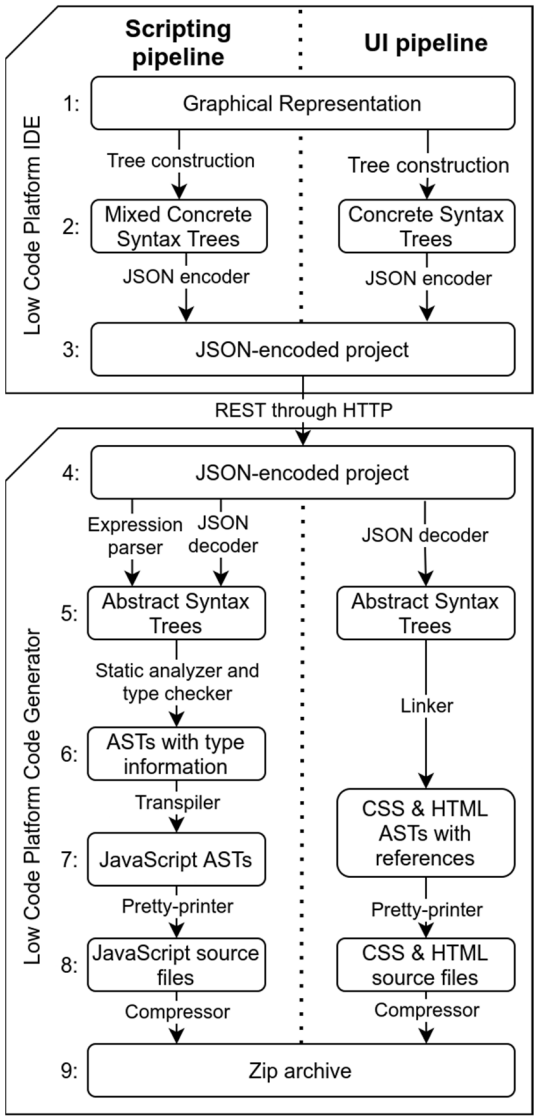

Figure 2. Pipeline detailing how the code is transformed from the visual representations into a fully fledged application.

Demonstration

OLP is a low-code platform built to better understand the practical difficulties of developing a platform from scratch. Its scope was limited to Web application development, with data persistence and consultation through REST APIs. Unlike existing platforms, the code generated is available to the users at any time, which means that users can resume development if the platform stops being updated. Although the proposed solution was built as a proof of concept, it is qualified and ready for use. The platform is also open-source software with a permissive license, allowing any user to contribute to its development or maintain its own fork. The code generated by the platform means that any user can continue maintaining his projects, even if the project stops being supported. The solution and its source code can be downloaded from the GitHub repository [22]. Figure 3 shows the IDE for an application that pulls the current temperature and displays it on a canvas. Figure 4 shows the logic part of such an application. Figure 5 shows the main function of the source code generated by the platform.

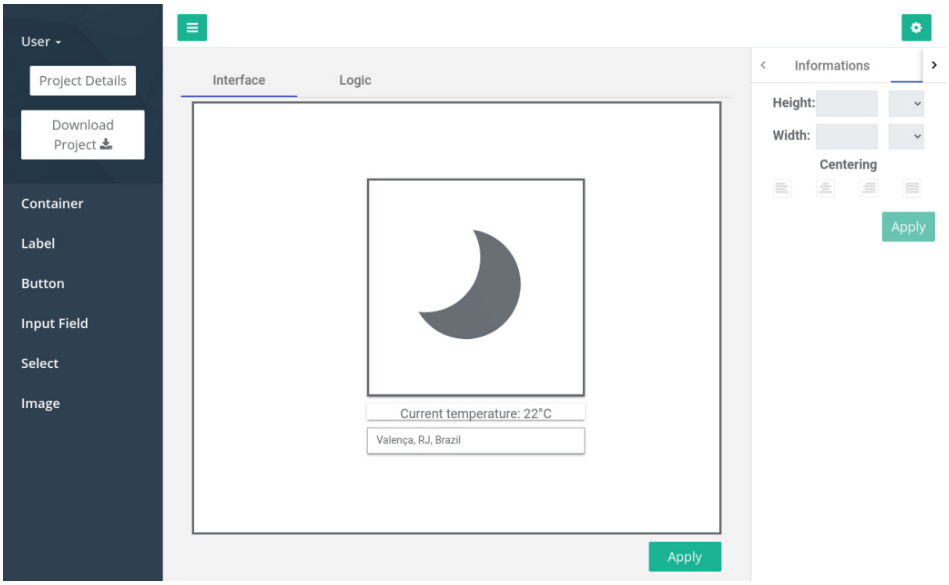

Figure 3. Interface creation screen for an application that pulls the temperature from a Website and displays its value on a canvas.

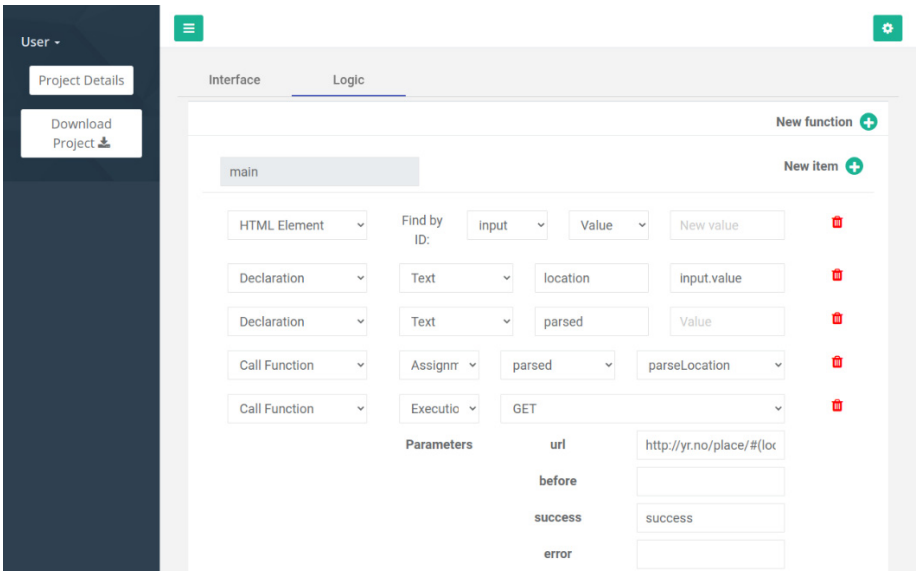

Figure 4. Logic screen for an application that pulls the temperature from a Website and displays its value on a canvas.

```
"use strict";

function main() {
    let location = input.value;
    let parsed;
    parsed = parseLocation(location);
    GET(`http://yr.no/place/${parsed}`, () => {}, success, (_0, _1, _2) => {}, () => {});
}
```

Figure 5. The main function of the source code generated by the platform from the steps presented in Figures 3 and 4.

5. Learned Lessons

Researchers and software developers interested in evaluating or developing low-code platforms can benefit from the findings of this research study and are justified by the fact that a low-code platform is software used to generate other software. The learned lessons are the following:

The project scope should be defined as soon as possible: In various projects, scope definition is one of the most crucial aspects because software is continuously optimized and new functionalities are added. A scope limitation is especially important in these projects because the functionalities are limitless, since it is a software that easily allows users to develop other applications. It considers that all software is unfinished; however, with low-code, the options are truly unlimited, and, without a clearly defined scope, it is unlikely that significant progress will be made in development. For example, to build OLP, the authors limited the scope toward Web applications that run on a Web server, and data were persisted and consulted through REST APIs. This scope limitation was made because the goal was to build a functional solution that could be realistically developed within a year.

The architecture should be flexible and support future modifications: A good architecture reflects its software requirements. With low-code, it is crucial that the architecture is flexible and can be applied to various use-cases. To verify the scalability and flexibility, the developers can discuss which elements of the architecture can be combined or removed. This aspect related to the architecture is especially valuable in building a low-code platform because, since it is a software used to build other software, scaling will be an issue that will arise soon if the architecture is not flexible.

Developers should use tools they are familiar with: Deciding which tools to use is challenging, especially while developing a low-code platform. The front-end, which will

interact with end-users, might be built in a different programming language from the backend that will generate the source code. The suggestion is for developers to use tools they are familiar with because building a low-code platform is a challenging task that will demand much effort. Then, adding the learning curve of a new programming language will only make the project more difficult.

The generated source code should be human-readable: The source code generated from the visual representations must be expressed in a flexible and self-contained manner that is human-readable. In some cases, the platform should even produce automatic commentaries because, otherwise, debugging the platform will be very difficult.

Operator precedence and blank spaces can be the source of various bugs: Operator precedence determines how expressions are evaluated. Therefore, it should be well defined and tested.

Although low-code is mostly visual, textual representations are still used in the code generator and in expressions typed by end-users. The code generator uses a textual version of the visual language to implement functionalities, and the user can type expressions in text boxes if they wish. In both situations, operators can be used, and their precedence must be considered. In early development, a mistake while coding the parsing of "XOR" and "AND" ("AND" was attributed a higher precedence) could cause bugs later. Today, most programming languages have the same operator precedence, but developers should verify whether an operator precedence is well defined to minimize bugs on the generated code. Furthermore, end-users can indent their written code in various ways and the platform should support the most common blank spaces such as "tab" and "space".

Platforms should support production and development builds: A production build is an optimized version of the end-user application. The development build is for development and application debug. For example, the most common optimization in Web applications is removing all the indentation in HTML, javascript, and CSS files because removing the indentation improves Web page load times. Then, in the production environment, the generated HTML and CSS should not be human-readable. Furthermore, removing unused libraries and directing all console logs to a log file can significantly increase performance. This is crucial because it is common for developers to include console logs or debug entry points to debug the software. Sometimes, these minor aspects slip into production and a production build could eliminate such things. Since one of the goals of low-code platforms is to optimize the generated code, a production build option for the generated code is a necessity.

Write many test programs to test the platform functionalities: Developing tools to debug and automate testing of the platform functionalities should be a top priority because it is easy to break code when the application also generates source code. Without such tools, several bugs will be introduced with each newly added functionality.

6. Conclusions and Future Works

This work described a low-code concept which consists of transforming visual representations that are easy to understand into fully fledged applications. The functional and nonfunctional requirements of low-code, the similarities with the no-code concept, and how both concepts are different from traditional programming were also explored. The paper also recognized the difficulty of developing a low-code platform due to the lack of available literature and proposed the Open Low-Code Platform (OLP), a new low-code platform. OLP is a low-code platform built to understand the practical difficulties of creating a low-code platform from scratch. The details of how the visual representations can be transformed into an application are also presented and explained through a pipeline.

Low-code platforms allow full applications to be developed extremely fast, which is crucial in today's world, as well as in the future, with time to market becoming increasingly important. Low-code projects can always improve, add new functionalities, or integrate new tools. Regarding OLP, future works can improve usability and support automatic deployment after generating the project artifacts. The project can also provide native

data storage instead of relying on external REST APIs for such purposes and allow users to write HTML code. Furthermore, support for other mediums, such as mobile applications, could vastly improve the solutions' utility. Perhaps the biggest issue with the low-code approach comes from the fact that, since source code is generated by the platform, when the code contains vulnerabilities, every application that uses the platform inherits it.

Author Contributions: M.A.A.d.C. collected and performed a deep analysis and reviewed the related literature on the topic, wrote the first draft of the document, performed the comparison study and identified some open research issues. H.T.L.d.P. and B.P.G.C. developed the platform J.J.P.C.R. supervised the study, consolidated the open issues, reviewed the structure and the text carefully. P.L. and S.B.M. tested the low-code platform, reviewed the text carefully, and analyzed the issues identified. All the authors contributed equally to the scope definition, motivation, and focus of the paper. All authors have read and agreed to the published version of the manuscript.

Funding: This work is partially funded by the *Fundo de Apoio ao Desenvolvimento das Comunicações*, presidential decree no 264/10, November 26, 2010, Republic of Angola; by RNP, with resources from MCTIC, Grant No. No 01245.010604/2020-14, under the Brazil 6G project of the Radiocommunication Reference Center (*Centro de Referência em Radiocomunicações*-CRR) of the National Institute of Telecommunications (*Instituto Nacional de Telecomunicações-Inatel*), Brazil; by FCT/MCTES through national funds and when applicable co-funded EU funds under the Project UIDB/50008/2020; and by the Brazilian National Council for Scientific and Technological Development-CNPq, via Grants No. 431726/2018−3 and 313036/2020-9.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Waszkowski, R. Low-Code Platform for Automating Business Processes in Manufacturing. *IFAC-PapersOnLine* **2019**, *52*, 376–381, doi:10.1016/j.ifacol.2019.10.060.
2. Sanchis, R.; García-Perales, Ó.; Fraile, F.; Poler, R. Low-Code as Enabler of Digital Transformation in Manufacturing Industry. *Appl. Sci.* **2019**, *10*, 12, doi:10.3390/app10010012.
3. Villegas-Ch., W.; García-Ortiz, J.; Sánchez-Viteri, S. Identification of the Factors That Influence University Learning with Low-Code/No-Code Artificial Intelligence Techniques. *Electronics* **2021**, *10*, 1192, doi:10.3390/electronics10101192.
4. Sáez-López, J.M.; del Olmo-Muñoz, J.; González-Calero, J.A.; Cózar-Gutiérrez, R. Exploring the Effect of Training in Visual Block Programming for Preservice Teachers. *MTI* **2020**, *4*, 65, doi:10.3390/mti4030065.
5. Henriques, H.; Lourenço, H.; Amaral, V.; Goulão, M. Improving the Developer Experience with a Low-Code Process Modelling Language. In Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, Copenhagen, Denmark, 14–19 October 2018; doi:10.1145/3239372.3239387.
6. Zaytsev, Vadim, Open challenges in incremental coverage of legacy software languages. In Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience, Vancouver, BC, Canada, 23–27 October 2017; doi:10.1145/3167105.
7. Völter, M.; Stahl, T.; Bettin, J.; Haase, A.; Helsen, S. MDSD—Basic idea and terminology. In *Model-Driven Software Development*, 1st ed.; Tickner, S., Eds.; John Wiley & Sons: Hoboken, NJ, USA, 2006.
8. Gartner Magic Quadrant for Enterprise Low-Code Application Platforms. Available online: https://www.gartner.com/en/documents/4005939 (accessed on 23 August 2021).
9. Woo, M. The Rise of No/Low Code Software Development—No Experience Needed? *Engineering* **2020**, *6*, 960–961, doi:10.1016/j.eng.2020.07.007.
10. Ihirwe, F.; Di Ruscio, D.; Mazzini, S.; Pierini, P.; Pierantonio, A. Low-Code Engineering for Internet of Things. In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, Virtual Event, Canada, 18–23 October 2020; doi:10.1145/3417990.3420208.
11. Bucchiarone, A.; Ciccozzi, F.; Lambers, L.; Pierantonio, A.; Tichy, M.; Tisi, M.; Wortmann, A.; Zaytsev, V. What Is the Future of Modeling? *IEEE Softw.* **2021**, *38*, 119–127, doi:10.1109/ms.2020.3041522.
12. Martins, R.; Caldeira, F.; Sa, F.; Abbasi, M.; Martins, P. An Overview on How to Develop a Low-Code Application Using OutSystems. In Proceedings of the 2020 International Conference on Smart Technologies in Computing, Electrical and Electronics (ICSTCEE), Bengaluru, India, 9–10 October 2020; doi:10.1109/icstcee49637.2020.9277404.
13. Broll, B.; Lédeczi, A.; Volgyesi, P.; Sallai, J.; Maroti, M.; Carrillo, A.; Weeden-Wright, S.L.; Vanags, C.; Swartz, J.D.; Lu, M. A Visual Programming Environment for Learning Distributed Programming. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, Seattle, WA, USA, 8–11 March 2017; doi:10.1145/3017680.3017741.

14. Sahay, A.; Indamutsa, A.; Di Ruscio, D.; Pierantonio, A. Supporting the Understanding and Comparison of Low-Code Development Platforms. In Proceedings of the 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Portoroz, Slovenia, 26–28 August 2020; doi:10.1109/seaa51224.2020.00036.
15. Barati, M.; Rana, O.; Petri, I.; Theodorakopoulos, G. GDPR Compliance Verification in Internet of Things. *IEEE Access* **2020**, *8*, 119697–119709, doi:10.1109/access.2020.3005509.
16. Stallings, W. Handling of Personal Information and Deidentified, Aggregated, and Pseudonymized Information Under the California Consumer Privacy Act. *IEEE Secur. Priv.* **2020**, *18*, 61–64, doi:10.1109/msec.2019.2953324.
17. Arteaga, C.H.T.; Anacona, F.B.; Ortega, K.T.T.; Rendon, O.M.C. A Scaling Mechanism for an Evolved Packet Core Based on Network Functions Virtualization. *IEEE Trans. Netw. Serv. Manag*. **2020**, *17*, 779–792, doi:10.1109/tnsm.2019.2961988.
18. Gouareb, R.; Friderikos, V.; Aghvami, A.H. Placement and Routing of VNFs for Horizontal Scaling. In Proceedings of the 2019 26th International Conference on Telecommunications (ICT), Hanoi, Vietnam, 8–10 April 2019; doi:10.1109/ict.2019.8798780.
19. Khorram, F.; Mottu, J.-M.; Sunyé, G. Challenges & Opportunities in Low-Code Testing. In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, Virtual Event, Canada, 18–23 October 2020; doi:10.1145/3417990.3420204.
20. Mouradian, C.; Kianpisheh, S.; Glitho, R.H. Application Component Placement in NFV-Based Hybrid Cloud/Fog Systems. In Proceedings of the 2018 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN), Washington, DC, USA, 25–27 June 2018; doi:10.1109/lanman.2018.8475055.
21. Manso, A.; Marques, C.G.; Santos, P.; Lopes, L.; Guedes, R. Algorithmi IDE—Integrated Learning Environment for the Teaching and Learning of Algorithmics. In Proceedings of the 2019 International Symposium on Computers in Education (SIIE), Tomar, Portugal, 21–23 November 2019; doi:10.1109/siie48397.2019.8970123.
22. GitHub—Heitor-Lassarote/Iolp: Inatel Open Low-Code Platform. Available online: https://github.com/heitor-lassarote/iolp (accessed on 29 July 2021).