Turning Low-Code Development Platforms into True No-Code with LLMs

Nathan Hagel, Nicolas Hili, Didier Schwab

This is a preprint version. See ACM for the final official version.
To appear in the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.
http://dx.doi.org/10.1145/3652620.3688334

Turning Low-Code Development Platforms into True No-Code with LLMs

Nathan Hagel
Univ. Grenoble Alpes, CNRS
Grenoble INP, LIG
38000 Grenoble, France
Karlsruhe Institute of Technology
Karlsruhe, Germany
nathan@hagel.dev

Nicolas Hili
Univ. Grenoble Alpes, CNRS
Grenoble INP, LIG
38000 Grenoble, France
nicolas.hili@univ-grenoble-alpes.fr

Didier Schwab
Univ. Grenoble Alpes, CNRS
Grenoble INP, LIG
38000 Grenoble, France
didier.schwab@univ-grenoble-alpes.fr

Abstract

The relevance of low-code / no-code development has grown substantially in research and practice over the years to allow non-technical users to create applications and, therefore, democratise software development. One problem in this domain still persists: many platforms remain *low-code* as the underlying modeling layer still requires professionals to write/design a model, often using Domain Specific Languages (DSLs). With the rise of generative AI and Large Language Models (LLMs) and their capabilities, new possibilities emerge on how Low Code Development Platforms (LCDPs) can be improved.

This paper shows how the capabilities of LLMs can be leveraged to turn DSL-based low-code platforms into true no-code. We analyzed how textual modeling can be replaced by generating the required model using LLMs. We conducted a user experiment to compare textual modeling with the use of LLMs for that task. Our results show that task completion time could be significantly improved, and the majority of users prefer using the LLM-aided modeling. Based on these findings, we discuss the integration of these techniques into an existing low-code platform to transform it into true no-code.

CCS Concepts

• **Software and its engineering → Software development techniques**.

Keywords

LLM, AI, low-code development platform, meta-model, model-driven engineering, DSL

1 Introduction

With information's increased value, developing software tools to manipulate, store, and analyze it has become a challenging task. The ability to quickly make decisions based on the collection of heterogeneous, possibly large amounts of data originating from different sources in real-time requires complex representations and advanced analysis tools that are now more and more integrated into large Information Systems (ISes), making their development even more complex. In addition, due to the market pressure, companies must reduce production cycle times. This has a negative impact on developers, who have to constantly innovate to develop software applications faster while maintaining robustness.

This paper intends to address this challenge by combining two paradigms, namely Low Code Development Platforms (LCDPs) and Large Language Models (LLMs). On the one hand, LCDPs promote the development of fully functional and customizable applications, using visual abstractions and graphical user interfaces while requiring a small amount of code [3]. Leveraging LCDPs for developing software systems is appealing to end-users with no programming skills who want to have a key role in the process of creating their systems. However, transitioning from "low-code" to "no-code" is still a research challenge as code is still required to i) formally describe the data entities to manipulate and their relationships; ii) describe how data is aggregated to be presented onto the screen; and iii) describe how an IS can be generated from a conceptual model. Several attempts (e.g., [13]) have been made using custom Domain Specific Languages (DSLs) while others rely on the UML and UML profiles [5]. In either case, knowledge of data modeling is still a requirement for users of LCDPs.

On the other hand, the recent advancements in research and development of LLMs open new perspectives in terms of software production from textual specifications. Yet, as pointed out several times in the literature [25, 37], LLMs often produce suboptimal results when used to generate code, forcing developers to spend time reviewing the code generated by the LLMs, thus reducing the possible benefits of using them. One reason that can explain the lack of performance from LLMs is the high expressiveness that general-purpose programming languages such as Java can offer. This limitation is, however, counterbalanced in the context of low-code development as the expressiveness a LCDP can offer is often curbed by the number of available commercially off-the-shelf functionalities (e.g., widgets in the context of Web Information Systems (WISes)), along with their customization capabilities the LCDP's User Interface (UI) can offer.

Research questions and contributions: The main research question addressed in this paper is:

RQ1. *How can existing LCDPs benefit from generative AI and LLMs?*

To answer RQ1, this paper presents an approach to how LLMs can be integrated into model-based LCDPs. The approach's goal is to consume users' specifications in natural language and produce models conforming to textual notation. These models can then be further processed in the respective LCDP to generate, e.g., the application from it. The approach is independent of any of the chosen Model-Driven Engineering (MDE) technologies (EMF/Xtext

or others) used in the running example and the case study, see Section 4.

In order to evaluate our approach, we present in this paper the results of a user experiment where we compared the perceived usability and task completion time of manually creating textual models by hand with generating them from an LLM based on the user's specifications in natural language. We used for that the OpenAI API [23] with model version GPT-4o. The results of the user experiment show that our approach could significantly improve the task completion time in the chosen case study while not impacting usability. The results, logs, and implementation of our approach can be found in the supplementary material [11].

This main research question led to the formalization of two subsidiary questions:

RQ2. *Can LLMs completely remove the necessity for coding in existing DSL-based LCDPs?*

To address RQ2, we conducted an explorative study to verify that, although the task of generating code conforming to a DSL using an LLM that was not trained for it is not trivial, the correct definition of the prompt used to interact with the LLM significantly impacts the capabilities of the LLM to produce models that are syntactically valid. During the user experiment, we observed that using our defined prompt template, all generated models were syntactically valid. However, being syntactically valid does not mean that they match the user's specification in natural language. In this paper, we discuss how our approach contributes to the semantic validation of the generated models.

RQ3. *What are the required steps to transform model-based LCDPs into true no-code by using the capabilities of LLMs?*

Finally, we discuss in this paper the opportunities and challenges in integrating generative AI in MDE processes and its benefits for LCDP end users and practitioners (RQ3). In particular, we discuss how the integration can not only benefit the user of such platforms to describe applications in a no-code approach but also how developers of such platforms can take advantage of LLMs to facilitate the customization capabilities.

Paper structure: the remainder of this paper is structured as follows: Section 2 provides background; Section 3 presents the formulated approach; Section 4 details the user experiment; Section 5 gives some pointers to turn low-code into no-code using LLMs; Section 6 discusses the current status of the implementation and future work; Section 7 presents related work; Section 8 concludes.

2 Foundations

This section gives an overview of the relevant foundations and background required for the present work.

2.1 Low-/No-Code Development

LCDPs allow *citizen developers* (developers with little or no software engineering background [22]) to rapidly deliver, set up, and deploy applications, reducing the amount of hand-written code to a minimum [29]. Therefore they are appealing to a larger group of stakeholders, as less or no software engineering knowledge is required to develop applications that meet the stakeholders specific needs.

One relevant aspect of LCDP is the rapid creation, deployment, and release of new versions of the created application. As web-based systems are often easier and cheaper to deploy than desktop ones [31], they are more aligned with low-code development [28].

LCDPs facilitate development through, e.g., drag'n'drop-oriented interactions or the composition of pre-built components or widgets. Ideally, the development process and editors are built into a single application and UI, therefore further guiding the user through development and sometimes even deployment. Further specification and customization can sometimes be added through textual specification, or the whole composition/modeling process is based on textual modeling. Business logic is usually described by workflows or processes using Business Process Model (BPM) and Notation (BPMN 2.0) [21] or equivalent languages. One important benefit of low-code tools lies in their high potential in terms of customization of the UIs to fit end-users specific needs [29]. However, implementing low-code customization capabilities in an end-user UI is not trivial. MDE has the potential to facilitate that task.

2.2 Model-driven Engineering (MDE)

MDE is nowadays an established discipline for developing complex software systems [19] by reducing platform complexity through adding a layer of abstraction on top of the programming languages [15, 30]. In MDE conceptual *models* are primary artifacts in software development processes, and techniques such as model transformation are used to progressively refine abstract models into working software solutions [15]. These system models can be represented by a variety of languages, such as UML or even custom DSLs, that are defined to model a specific problem.

Therefore, a *meta-model* (aka. *abstract syntax*) is the key ingredient to formalize models. A meta-model is a set of *meta-classes* which represent the concepts that define the domain and *relations* that specify how the concepts can be bound together in a model. Furthermore, a set of well-formed rules restrict the way concepts can be assembled to form a valid model.

Different meta-modeling frameworks exist to specify meta-models, e.g., Eclipse Modeling Framework (EMF) [34] and MetaEdit+ [14]. Dedicated to web-based technologies, new frameworks have emerged, e.g., EMF-REST [8], Ecore.js[1], the JavaScript Modelling Framework (JSMF) [33], and FlexiMeta [12]. One or many concrete syntaxes can then be specified for the same meta-model based on the abstract syntax defined using, e.g., the frameworks above. These can be graphical or textual. Using Xtext [9] (a grammar-based approach), for instance, the meta-model and a concrete textual syntax can be defined at the same time [26]. The combination of a meta-model with one or more concrete syntaxes is often also named a DSL and part of LCDPs.

Running Example DSL. The running example used in the user study in Section 4 uses a DSL to define web forms [13]. The meta-model in a graphical notation gives an overview of the available concepts, see Figure 1. It allows the definition of web forms in a row-based layout. Several widgets can be defined, e.g., a text or a

[1] A JavaScript implementation of the Eclipse Meta-Object Facility (MOF).

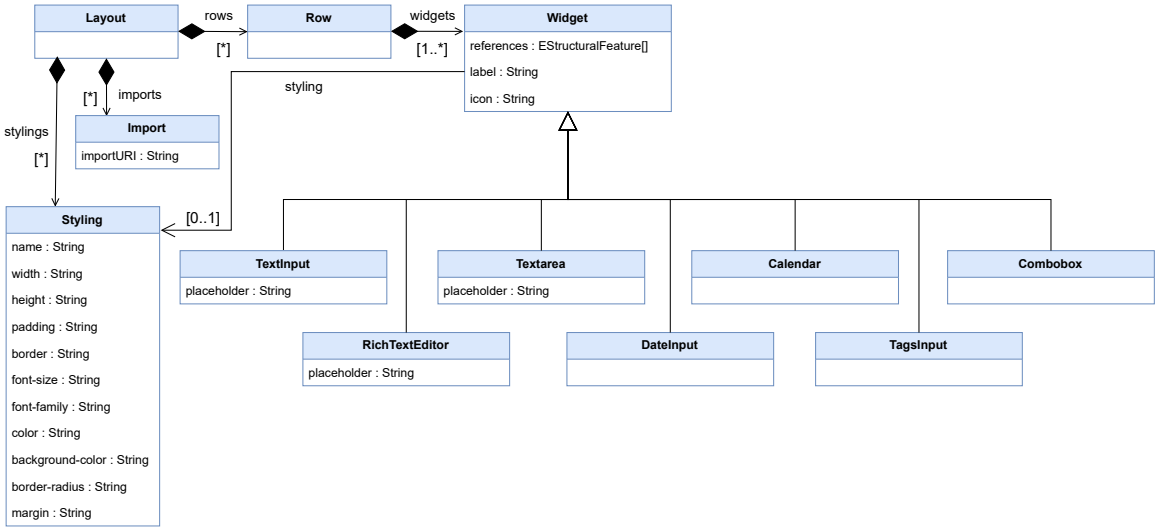

Figure 1: Running example: overview of the layout meta-model (extended version of [13]).

calendar input. This DSL can be used for both textual and graphical modeling. However, for the running example, the meta-model is defined using Xtext. This means that a textual concrete syntax for creating models is already defined. An example model using this syntax is provided in Listing 1.

Besides defining the structure of the form and the type of widgets, the DSL allows us to define to which data model attributes the widget is linked using the *references* attribute. This means that another input for the running example is a data model, which can also be defined in a textual syntax and, therefore, generated by an LLM if required. The experiment used two data models, one for a conference event and one for a hair salon. Therefore, the widget entity features the attribute *references* of type EStructuralFeature in case of a definition of the data model using Ecore. Lastly, a widget has an icon and a label to customize and adjust the form. From the layout created from Listing 1 we used MDE techniques (see Section 3) to generate the preview in Figure 2. Only the last two rows (4 widgets) are contained in the example DSL code in Listing 1 for space reasons. One of the contributions of this paper is to replace the textual creation of this model with LLMs.

2.3 Large Language Models (LLMs)

LLMs can be used to generate text based on an input provided by the user. This means, that given a sequence of input tokens, LLMs can estimate the probability of the next output token. They can be customized for specific tasks using fine-tuning, which updates the weight of parameters and, therefore, influences the token generation [27]. These capabilities, especially with the upcoming general large language models like ChatGPT [24], created a variety of use cases, from generating programming code or texts to proofreading or error detection and many more [36]. To extend these capabilities, several methods exist to adjust or improve the generated output. As mentioned above, fine-tuning is one. However, it comes with a significant drawback as for effective fine-tuning big datasets are required. Creating them can be quite costly [20, 32].

```
// missing row 1 (2 widgets) for space reasons
row {
  text input {
    references Event.location
    label "Location"
    icon "icon-location-pin"
    placeholder "Enter the location"
  }
  combobox {
    references Event.type
    label "Type of Event"
    icon "icon-options"
    placeholder "Select the event type"
  }
}

row {
  calendar {
    references Event.begin Event.end
    icon "icon-calendar"
    label "Event Dates"
    placeholder "Choose the event dates"
  }
  textarea {
    references Event.cfp
    label "Call for Papers"
    icon "icon-doc"
    placeholder "Enter the call for papers"
  }
}
```

Listing 1: Example of a model conforming to the layout meta-model given in Figure 1 in an arbitrary Xtext concrete syntax.

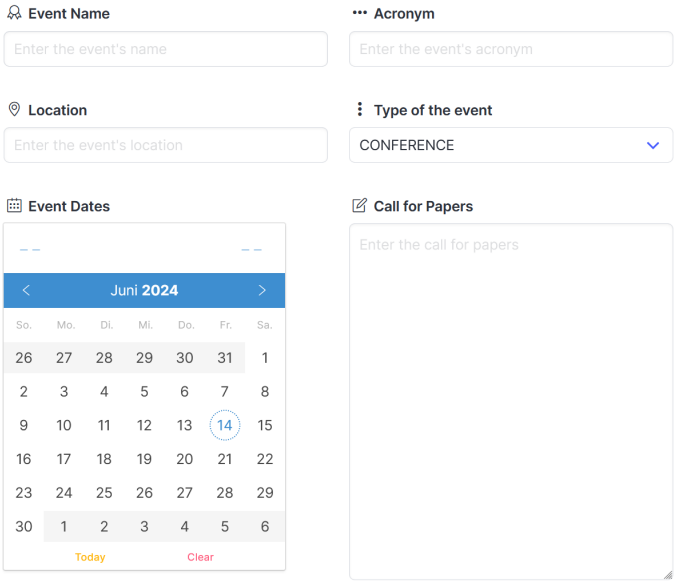

Figure 2: Preview of the layout model given in Listing 1 (the first row is hidden in Listing 1 for space reasons).

An alternative method is prompt engineering. A prompt is the information provided to the LLM, which can, in particular, contain instructions or a framing for a specific context. When generating code, for instance, these instructions could define a code style, etc. [36]. Prompt engineering is the technique of defining/creating prompts for the required task. Using prompt engineering, competitive results compared to fine-tuning can be achieved with a usually lower effort [32]. Prompt engineering can reach from simply adding small instructions to generic prompts to providing sophisticated templates that structure the requests and add additional information or instructions to improve results [36].

3 Approach

Integrating LLMs into LCDPs changes the way how the user interacts with the platform. This also depends on how the LCDP is structured and how the development process already looks. We assume in this section that the production of an application, e.g., a WIS, in an LCDP is similar to the way an application is produced following conventional MDE processes: i) the specifications of the user are captured; ii) models of the desired application are created; and iii) an implementation is produced, either through code generation or model interpretation.

In this section, we propose an approach (see Figure 3) where the modeling step is replaced with generating the required models using LLMs. The input of the approach is the user's specifications of the desired application in natural language (step 1). Then, the chosen LLM is queried (step 2), which generates the models (step 3). Finally, the models are processed (step 4), and the desired application is produced (step 5).

The proposed approach is platform-independent: it does not require a specific LLM to be used since it does not require a specific training phase; it does not depend on a specific modeling framework since the heart of the approach only relies on models conforming to dedicated textual syntaxes that any kind of parsers could process. Finally, it does not restrict the way an application is produced from the models, being, e.g., code generation or model interpretation. In our experiments, we used the OpenAI API [23] with model versions GPT-3.5 Turbo, GPT-4 Turbo, and GPT-4o for the user study as a third party LLM, Xtext for the definition of the textual notations that were used, and the LCDP presented in [13] to generate the resulting WIS.

The capabilities of the approach to generate correct models, i.e., from both a syntactical and a semantic point of view, depends on two main factors: first, the user specification alone is not enough for the LLM to generate syntactically verified models for DSLs it was not trained for. To improve this step, we enriched the prompt containing not only the user's specifications but also all relevant information an LLM requires to generate valid models. Second, due to the non-deterministic nature of an LLM to produce code from a natural language specification, some validations must be done to ensure that the resulting models fulfill the user's expectations. To do so, we implemented a feedback loop (step 6) where the user has the possibility to inspect the results (the generated model, the application or both) and make changes.

3.1 Prompt Engineering

When using new, small, or proprietary LCDPs, it is quite likely that no data, examples, or information at all was included in the training data sets these models were trained on. This limits the applicability of LLMs for LCDPs. One way to inform the LLM about the LCDP's internals and resources could be fine-tuning by creating a big training data set and adjusting the model's context. However, this approach may be too expensive [20, 32]. As mentioned earlier, prompt engineering overcomes these drawbacks, as it does not require large training data sets and is less compute-intensive than fine-tuning. This technique is particularly suitable in the context of DSLs, which are reasonably small – or at least smaller than their general-programming counterparts –, dedicated languages for specific domains.

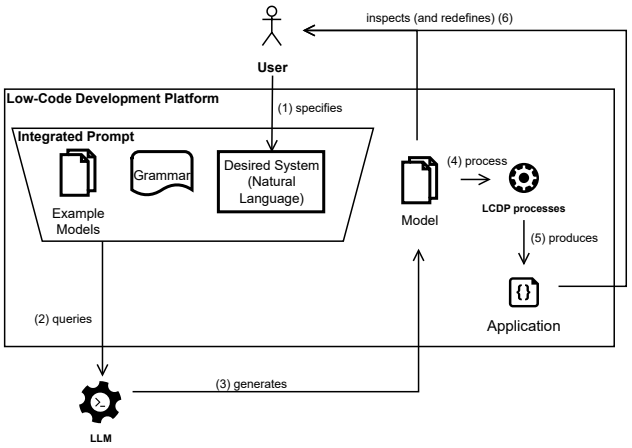

Figure 3: Overview of the proposed model-based approach integrating LLMs for developping no-code applications.

```
Using the following information:
[Additional Information]*
And this Xtext grammar as the grammar for the DSL:
[Xtext Grammar]
And the following additional information:
[Additional Information]*
[I give you the following example: [example]]+
Please create a new example that conforms to the
text below:
[User specification in natural language]
The content, naming of variables, etc., can be
freely chosen.
No additional text, explanation,etc.
Do not add markdown tags or anything.
```

Listing 2: Prompt template with placeholders that can be used to generate textual models.

The structure of the prompt has an important impact on the LLM that may (positively or negatively) impact the outcomes of the LLM. It has to be meticulously designed and tested to assess its efficiency, hence the notion of *prompt engineering*. This section discusses the design of the prompt template we used in the context of our approach.

Listing 2 details the template we defined for our approach. It consists of predefined instructions (in black) and placeholders for information and resources (in blue). An arbitrary amount of additional information, such as resources that should be referenced, other models (like the data model used in the running example and the experiment in Section 4), etc., can be inserted at two locations if required. As the input of an LLM is a text string, it is important that all information contained in the prompt can be serialized to text. In MDE, a textual representation of models (and metamodels) is the norm. This representation can be a custom textual syntax or for graphical DSLs a serialization into common formats, e.g., XMI, JSON [13, 4, 16].

The template proposed in Listing 2 assumes that the DSL's syntax is defined with Xtext [9] and that the examples given are encoded according to this grammar. However, we found out during our tests that replacing the Xtext grammar with, e.g., Extended Backus-Naur Form (EBNF), and providing additional information like the data model required for our running example in different representations (serialized Ecore models - XMI or textual representation of Fleximeta models) also provided comparable results. Therefore, the template proposed in this paper is not strictly dependent on the technologies we used, and even though we did not further investigate our findings, we are confident that alternative prompt templates would also work to generate the desired models. Comparing different templates is outside the scope of this paper, and evaluating their impacts on the generated models is future work.

During our experiments, we observed that only providing the (Xtext) grammar to the LLM is not enough to produce syntactically valid models. We found out that providing at least one example model conforming to the DSL's grammar considerably increased the quality of the produced models, and we recommend providing enough examples to cover the different parts of the grammar. It is worth noting that one limit of LLMs is the context size, which restricts the amount of data we can provide the LLM with. But, over the past years, the context size of mainstream LLMs such as ChatGPT exponentially grew, making the context size no longer a practical limit. However, mainstream LLMs and APIs, such as OpenAI API [23], rely on a financial model where the user is billed according to the consumed number of (both input and output) tokens. In the future, we plan to explore other LLMs, including open-source LLMs such as Mistral [18] or Llama 3 [17].

3.2 Incremental Process

Providing a good prompt template is enough for the LLM to produce syntactically valid models with respect to the DSL definition, but it does not provide any guarantees that the produced models are semantically correct with respect to the user's specifications. This requires a validation from the user. To do so, we implemented in our approach a feedback loop (see Fig. 3) where the user has the possibility to inspect the results and apply changes. In the tool described in Section 4, the user's feedback can be integrated in two different ways: either by making manual changes directly to the generated models or by refining his/her demand in the prompt. Obviously, the former is dedicated to MDE practitioners, while the latter better fits the expectations of end users with no modeling background. This requires the LCDP to produce a preview of the application produced from the models.

This has an impact on the prompt discussed in the previous subsection: not only the current prompt should be sent to the LLM, but also information about the history of the conversation, similar to the mechanisms applied when using tools like ChatGPT. Therefore, the template presented in Listing 2 presents only the beginning of the conversation and is further completed with the user's messages refining the original request and responses from the LLM interleaved with each other.

It is worth noting that this impacts the context size, as it will continuously grow with the number of user-requested changes. But, as mentioned earlier, recent technological advances in mainstream LLMs reduce the importance of this cap, and as soon as the context window's limit is reached, the provided history could be reduced (e.g., by deleting the oldest messages).

4 Experiment Design

To evaluate the effectiveness and usability of integrating LLMs into existing LCDPs (**RQ1**), a user study was conducted. The user study's focus was the comparison of textual modeling with the creation of the desired model and, therefore, application using LLMs and natural language specification.

Implementation. For this comparison, we chose a simple low-code approach that allows the definition of web forms that are connected to a data model. We integrated a chat interface into the UI, see Figure 4 that is similar to ChatGPT's user experience. The chat interface can connect to different LLM-versions like GPT-3.5-turbo or GPT-4o. For the experiment, we used only OpenAI's models. However, the approach, in general, is not limited to OpenAI's APIs. To improve the generated results, an engineered prompt as described in Subsection 3.1 was used. This means that the user input was inserted at the correct location in the defined prompt before being sent to the API. The response from the LLM was then directly inserted into the low-code editor, see Figure 5 on the left,

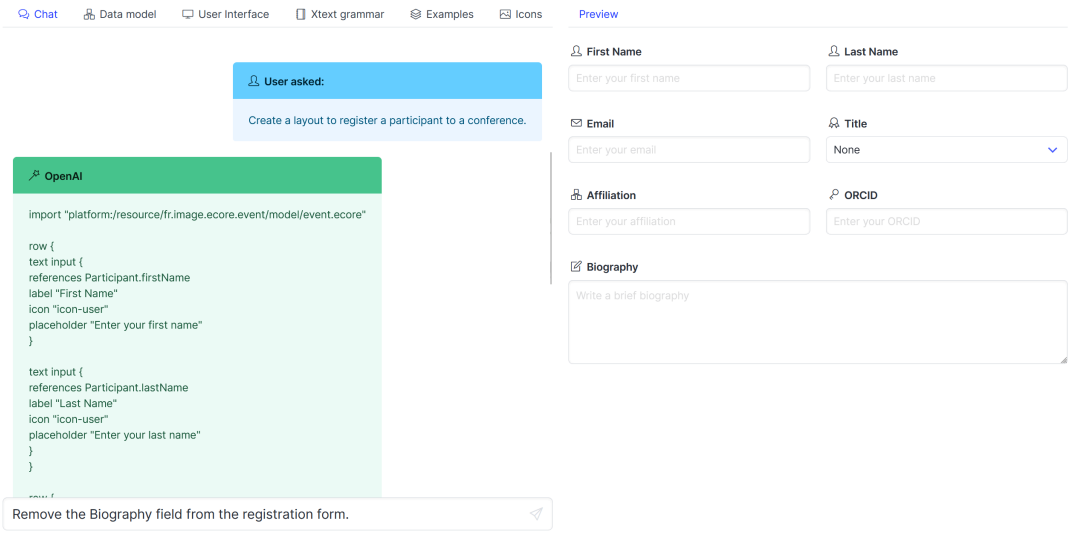

Figure 4: User Interface of the user experiment implementation (Condition 1) - LLM Chat Interface on the left, preview of the generated form on the right

which resulted in the generation of the desired web form in our implementation. Additionally, the response was displayed in the green chat box of the chat interface; see Figure 4 on the left. The web form is only generated if the generated DSL code is syntax error-free. If errors exist, they are displayed to the user in a similar way as in an IDE. The error detection was based on the grammar of the used Layout DSL in this experiment. The implementation preserved the history of the session, which means that users were able to reference previous generations or the current state of the model for error correction or improvements, see Figure 4.
The creation of web forms using the low-code approach and the chat interface was compared with the manual creation of the DSL code / the models. To create the web forms manually, a code editor was included in the UI, see Figure 5 on the left. Users could create textual models, which immediately generated the preview on the right as soon as the model was syntactically correct.

Participants. We conducted the experiment with 18 participants (9 male, 9 female), aged 18-44 (M=26,78, SD=7,69). All participants were part of computer science research groups in Grenoble Computer Science Laboratory (LIG). A minority (3 out of 18) had experience in modeling and DSLs. The chosen low-code approach / DSL was unknown to every participant. Most of the participants (66.6%) stated that they use LLMs like ChatGPT *Very Frequently* or *Frequently*. The experiment was conducted as a supervised within-subjects design. Each participant was exposed to both interactions and had to create forms using the chat interface and the manual textual modeling. The participants were randomly split into two groups. One group started with the chat interaction, whereas the other started with the manual modeling.

Experiment Setup. The experiment was conducted in a lab under supervision. For every participant, the same technical setup (computer, monitor, etc.) was used. The main interaction happened through the keyboard and mouse. As the participants were international and, therefore, preferred different keyboard layouts, we provided a suitable keyboard and settings as requested by the participant. Every participant used both conditions (chat and manual creation). At the beginning of the experiment we provided a general introduction to the experiment, the used DSL / low-code platform, the user interface etc. The participants had to create web forms for two data models, which were also provided in graphical (Ecore) and textual (Fleximeta) notation. The data models were also explained to the participants. The grammar (Xtext) of the DSL was also provided. Written scenarios described the web forms that had to be created, containing all the information required to create them and how they had to look. The written scenarios were provided to the participants one by one for each task. As an additional resource, the participants received documentation of the DSL and examples. Furthermore, an overview of the possible widgets and how they could be configured was provided. If the participant had questions about the mode, the experiment, or the resources provided, these questions were answered. For each interaction, the participant received another introduction about how the user interface works and what is expected. In total, each participant had to create 6 web forms, 3 for each data model and interaction. One task for each of the two conditions was to get familiar with the low-code approach and how the tooling works. Therefore, the data for the results is based only on two tasks for each condition. Each scenario had the same size and expected the same amount and variety of widgets. After each condition (1 + 2 tasks), the participant had to answer a questionnaire, including the System Usability Scale (SUS) questions, to assess the perceived usability. In total, the duration of each experiment was 70-90 minutes.

Task. The task was to create web forms according to a specification defined in a scenario. In total, 6 scenarios were defined,

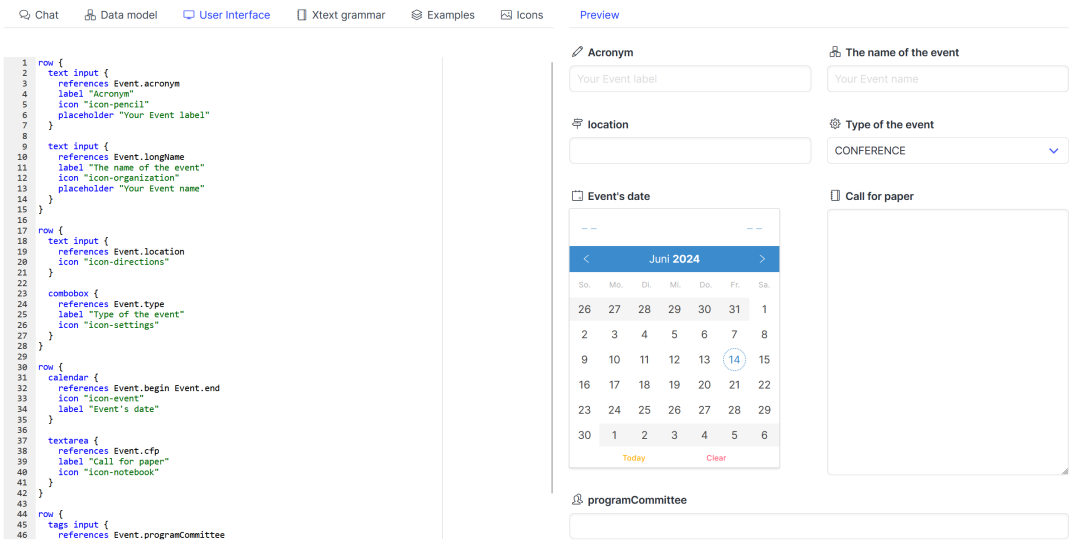

Figure 5: User Interface of the user experiment implementation (Condition 2) - Low-Code Editor for the Layout DSL on the left, preview of the generated form on the right.

therefore 6 tasks, and had to be completed by each participant. The scenarios described the data model attributes the web forms had to ask for. The different scenarios used two different data models. One is for a hair salon appointment/management tool, and one is for a conference event management tool. Furthermore, the scenarios contained a broad specification on the desired layout itself (e.g., use only one widget per row to create a mobile-first form). The participants were asked to read the scenario and ask questions if they had any. Then, the participants had to create the form using the chat interaction or manual modeling. The layouts were not strictly defined. A task/scenario was successfully completed as soon as all specified widgets were correctly included in the form. Furthermore, the labels and placeholders had to be defined appropriately for the semantics of the widget's data attribute. Also, the icon had to be set appropriately. A generated form from scenario 1 is visible in Figure 2. It is used to create a new conference event in a conference event management tool. An example of a prompt that generated this form/the respective DSL code (see Listing 1) is visible in Listing 3.

Metrics. During the experiment, we measured the following metrics:

```
Create a form to create a new conference event.
The form should ask for the event's name, its
acronym, the location and the type of the event.
The form must only allow possible choices for the
type of the event.
Lastly the form should ask for the events dates
and the call for paper.
Use two widgets per row.
```

Listing 3: Example prompt that generates a web form for scenario/task 1

- *(M1) Task completion time:* Measured from the first keystroke in the tool until the generated or created form fulfilled the task's requirements.
- *(M2) Personal preference:* Evaluated through a question in the questionnaire filled out by the participant after finishing all tasks.
- *(M3) Usability:* Evaluated through an SUS [1] score questionnaire that was filled out after each condition.

The decision why we started the timer of *M1* at the first keystroke in the tool and not the moment when the user received the task was made because each user had to read through the material, and some had questions, which we did not want to measure in this experiment. Rather, we decided to measure the time from when the user understood the task until it was completed. We informed the participants that they should start with the task as soon as they understood the task and had no questions.

Hypotheses. We defined the following hypotheses prior to our experiment:

- *(H1)* We expect that using the chat interaction to generate the models is significantly faster than creating the models manually.
- *(H2)* We expect that the majority of the participants will prefer the chat interaction to the manual creation of models.
- *(H3)* We expect that the usability of interactions is considered good while the chat interaction will receive a better usability rating.

Ethical Considerations. Before the experiment, participants were informed about the scope and purpose and potential ethical considerations. The risk was evaluated to not exceed the risk of living and using computers. Every participant explicitly agreed to participate in the study and allowed the results and artifacts to be published

for research purposes. All data, logs, etc., are fully anonymized, ensuring confidentiality.

Results. This paragraph describes the results of the conducted user experiment:
Task completion time (M1): The results of the task completion time for each scenario are depicted in the box plot in Figure 6. Scenarios 1, 2, and 5 in yellow, orange, and red, respectively, were the scenarios for the creation of models using the chat interaction. Scenarios 4, 6, and 3 in green, cyan, and blue, respectively, were the scenarios for manually creating the models. The scenarios were also used in that order, meaning that depending on the condition with which the participant began, we started with scenarios 1 or 4 and continued with 2 and 5, respectively, 6 and 3. This results in scenarios 1 and 4 being the test or learning tasks. The task completion times for these two scenarios were also included in Figure 6 but are not included in the further interpretation and evaluation of the results. We used a paired t-test to evaluate the impact of different systems on the task completion time. Results show a statistically significant difference between the two processes.

$$t(17) = -8.55, p < 0.001$$

The mean and the standard deviation of the task completion time for the chat interaction and manual creation are the following (the values origin from the combination of the two scenarios per process):

- Chat: $\overline{X} = 187.5$, $\sigma_{\text{chat}} = 44.35$
- Manual Creation: $\overline{X} = 421.56$, $\sigma_{\text{manualCreation}} = 32.49$

The results indicate that using the chat interaction is significantly faster than creating the models by hand.

Personal Preference (M2) and Perceived Usability (M3): We used a paired t-test to evaluate the impact of the two different processes on the perceived usability using an SUS score questionnaire. Results show no statistically significant difference between the two processes.

$$t(17) = 1.0095, p = 0.3268$$

The mean and standard deviation of the SUS score for chat interaction and manual creation are the following:

- Chat: $\overline{X} = 80.0$, $\sigma_{\text{chat}} = 10.74$
- Manual Creation: $\overline{X} = 75.83$, $\sigma_{\text{manualCreation}} = 9.47$

We can see that the average SUS score of both processes is between good and excellent using the categorization of Bangor et al. [1] with the chat interaction having a slightly higher mean. The standard deviation is almost the same. In literature, an average SUS score of 70 is considered average and a passing criterion for user interactions [1]. Both processes scored above that.
The questionnaire also assessed the personal preference of the participants, as every participant was exposed to both conditions. 15 out of 18 participants preferred the chat interaction.

Discussion. In the user study, the chat interaction to generate the models performed significantly better in task completion time compared to the manual modeling. The perceived usability, though having a higher mean, is not significantly better. However, participants reported that they explicitly liked the direct feedback

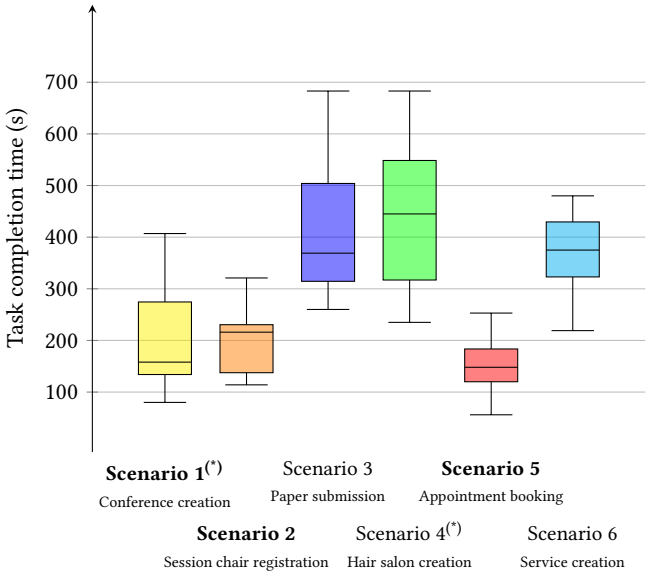

Figure 6: Box Plot of the task completion time of scenario 1 to 6. Results for test scenarios (*) are not accounted in the paired t-test evaluation. Scenarios in bold corresponds to tasks involving the LLM.

provided by the implemented UI when creating the models manually. Both ways of creating the desired models scored between *Good* and *Excellent* on the SUS scale/mapping to adjectives [1]. Therefore, our results are consistent with *(H1)* where we suspected that generating the models based on natural language is faster than manual textual modeling. As 15 out of 18 participants preferred the chat interaction, our results are also consistent with *(H2)*. *(H3)* has to be partly rejected. Both ways of modeling received a rating of at least good, though the chat interaction did not receive a significantly better usability score.
In conclusion, our results show that using LLMs with a suitable enriched and structured prompt as defined in Subsection 3.1 seems to be a good way to improve the efficiency of the modeling step and, therefore, the whole low-/no-code development process. Furthermore, a well-working integration does not reduce the perceived usability but rather adds a new way of using an existing LCDP. This answers **RQ1**. However, we expect that the benefits that LLMs can have on LCDPs are not limited to that.

5 Transformation from low-code to true no-code

Based on the conducted integration of ChatGPT into the layout web-form LCDP as well as an ongoing integration into an extension of that [13], the following section describes a set of prerequisites, steps, and challenges that we identified throughout our analysis and the conducted user study. These learnings and steps combined with the results of the experiment answer RQ3.

Prerequisites. For most LCDPs, we assume that available state-of-the art LLMs will not be able to generate correct models out-of-the-box. As described in subsection 3.1, prompt engineering can be

used to enable the LLM to generate correct models. However, we experienced that a set of resources seems to be crucial to *program* the LLM appropriately for that task [36]. We identified that (i) the language's grammar and (ii) examples that cover most or ideally all features of the modeling language should be included. This means that these resources must be prepared and defined. For LCDPs which do not use a modeling language with a graphical syntax, this can mean that defining it could be helpful. In many cases, this can be easily achieved, for instance, using Xtext when using modeling languages from the EMF ecosystem [26]. In other cases, this step could require more integration work, and Xtext could still be helpful in defining the language's grammar.

Steps. For properly using LLMs in LCDPs we think that it is crucial to directly integrate them into the existing tools, similar to Github Copilot [10] for IDEs. One indicator for this conclusion is the results of the user experiment, see section 4, where the SUS score was quite similar for using textual modeling and generating the models using the chat interface. Without integration, this would most likely mean additional steps for the user, which we expect will reduce the perceived usability and efficiency. For a successful integration, we therefore propose the following steps: (1) Define the LLM that should be used; (2) Create or define the required resources and conduct tests until the generated results match the requirements. This step could require several iterations. Then, (3) integrate them into the prompt. Find and implement an appropriate user interface within the existing LCDP. One example can be seen in Figure 4. This highly depends on the possibilities of the platform itself. (4) Directly integrate the generated models into the UI by adding a preview as in Figure 4 or a graphical or textual representation of the generated model. (5) Use a stateful chat interface that is session-based. This ensures that the user can reference the previous generation and interact with and redefine it.

Challenges. We experienced challenges such as ensuring a high rate of correct generations, which is essential for a well-working user experience. This also depends on the complexity of the used modeling language. Using a prompt template as defined in Subsection 3.1 or a modified version can help. However, examples should be wisely chosen so as not to unnecessarily increase costs when using billed APIs.

6 Discussion

Combining LLMs with MDE presents two benefits in the context of low-code/no-code development. The first benefit targets end users who can leverage generative AI to model their applications and customize the interfaces to represent the data on the screen according to their preferences. By properly integrating LLMs, this entire process can be performed without modeling or coding experience by only using natural language. This can open LCDPs to more users. Furthermore, in the experiments we conducted, we observed that models can be successfully generated using LLMs like ChatGPT and the prompt template defined in Subsection 3.1. Implementing this approach, the results of the conducted user study show that the efficiency of the chosen LCDP could be significantly increased without impacting usability. Furthermore, 15 out of 18 participants preferred defining the system in natural language, which also highlights the need to integrate these mechanisms into LCDPs. These improvements were possible without fine-tuning a model but only by applying prompt engineering and the integration into the UI. Removing the need for fine-tuning reduces not only the costs but also the additional required environmental resources. However, the evaluation of the approach lacks the comparison if LLMs could also be used to reduce the customization complexity of LCDPs, which could be an extended application. In [13], we presented a platform where the graphical user interfaces can be easily modified by means of drag'n'drop and by changing one widget by another. The experiment should be extended to compare the two different modalities (standard mouse and click capability vs. the use of LLMs).

The previous observation leads to a second, less obvious benefit LCDP practitioners can take advantage of by extending the customization capabilities of LCDPs. The customization capabilities are often constrained by the set of, e.g., customizable widgets in the domain of WISes an LCDP has to offer, along with the user interfaces to customize the appearance and behavior of those widgets [2]. As expectations about LCDPs grow, widgets become increasingly complex in size and customization capabilities, and LCDP developers must make a trade-off between customization and the complexity of the platform. A commonly accepted compromise is to implement coarse-grained customization capabilities available through no-code user interfaces and fine-grained ones only through extending the created code or additional textual settings. Hence, improving the usability of such interfaces for end-users LLM could play a relevant role in facilitating the fine-grained customization of applications when no alternative UI capabilities are available or too much work to implement. Although we can't answer **RQ**2 for all LCDPs, we can for confidently say that it is indeed possible to eliminate coding entirely from DSL-based LCDPs by properly integrating the capabilities of LLMs. However, the limitations of this approach will likely co-develop with the limitations of state-of-the-art LLMs.

7 Related Work

This section reviews related work on the topic focussing on the application of LLMs for DSLs and LCDPs. Busch et al. [6] present a Low-/No-Code approach to creating applications using first, a graphical model created by the user and second, a prompt frame. This prompt frame is then filled with a natural language description of the user to extend the existing model with e.g., semantics. The goal of the prompt is to generate code, which is added to the code generated from the graphically defined model. Together, these two pieces create the final application. The approach was successfully implemented in a running example to create a point-and-click adventure game. Grammar Prompting by Wang et al. focuses on the generation of grammar-based DSLs in general (not only DSLs in the context of LCDPs) [35]. However, their approach could be applied and useful when integrating LLMs into existing LCDPs as it describes a way to create prompts that successfully generate DSLs by only providing parts of the grammar. The full grammar is only provided for small DSLs. This approach, similar to subsection 3.1, also requires examples in the prompt, which are in contrast to this proposal, labeled. They contain a query, a minimal provided grammar, and the result. Their prompt template requests the LLM to first

generate a minimal grammar before generating the actual DSL. Besides these grammar and textual DSL based approaches, other work focuses on the interpretation and translation of hand-drawn models into more formal representations like UML models [7]. These sketches can then be used as input for an LLM to generate UML models, which then form the input for LCDPs or a modelling pipeline. Integrating this approach into LCDPs could just as well improve the usability for MDE and low-code approaches.

8 Conclusion

In this paper, we presented an approach to transform DSL based LCDPs into true no-code. The transformation is achieved by allowing users to define the desired application in natural language, from which first the models of the LCDP are generated, and the LCDP then creates the desired application. The approach is validated through a user experiment with 18 participants, where results show, that the task completion time could be significantly reduced compared to manually using the LCDP without impacting the perceived usability. We also presented a prompt template to generate textual models/DSLs which can help when integrating LLMs into existing LCDPs. In conclusion, LLMs can help to improve existing LCDPs. They can be used to either replace or aid the modeling step where the application is defined and improve the development time.

Acknowledgments

This work was supported by MIAI@Grenoble Alpes, (ANR-19-P3IA-0003) and the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF).

References

[1] Aaron Bangor, Philip Kortum, and James Miller. 2009. Determining what individual SUS scores mean: Adding an adjective rating scale. *Journal of Usability Studies*, 4, 3, 114–123.

[2] Tina Beranic, Patrik Rek, and Marjan Hericko. 2020. Adoption and usability of low-code/no-code development tools. In *Central European Conference on Information and Intelligent Systems*. Faculty of Organization and Informatics Varazdin, 97–103.

[3] Alexander C Bock and Ulrich Frank. 2021. In search of the essence of low-code: an exploratory study of seven development platforms. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 57–66.

[4] Alexandre Bragança, Isabel Azevedo, Nuno Bettencourt, Carlos Morais, Diogo Teixeira, and David Caetano. 2021. Towards supporting SPL engineering in low-code platforms using a DSL approach. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 16–28.

[5] Alan Bubalo and Nikola Tankovic. 2023. Modeling Low-Code Databases With Executable UML. *Human Systems Engineering and Design (IHSED 2023): Future Trends and Applications*, 112, 112.

[6] Daniel Busch, Gerrit Nolte, Alexander Bainczyk, and Bernhard Steffen. 2023. ChatGPT in the loop: a natural language extension for domain-specific modeling languages. In *International Conference on Bridging the Gap between AI and Reality*. Springer, 375–390.

[7] Aaron Conrardy and Jordi Cabot. 2024. From image to uml: first results of image based uml diagram generation using llms. *arXiv preprint arXiv:2404.11376*.

[8] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, Abel Gómez, Massimo Tisi, and Jordi Cabot. 2015. Emf-rest: generation of restful apis from models. *arXiv preprint arXiv:1504.03498*.

[9] Sven Efftinge and Markus Völter. 2006. OAW xText: a framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit* number 118. Vol. 32.

[10] GitHub. 2024. GitHub Copilot · Your AI pair programmer. https://github.com/features/copilot. Accessed: 2024-07-02. (2024).

[11] Nathan Hagel, Nicolas Hili, and Didier Schwab. Supplementary Material for "Turning Low-Code Development Platforms into True No-Code with LLMs". Zenodo, (July 2024). DOI: 10.5281/zenodo.12733193.

[12] Nicolas Hili. 2016. A Metamodeling Framework for Promoting Flexibility and Creativity Over Strict Model Conformance. In *Flexible Model Driven Engineering Workshop* (Flexible Model Driven Engineering). Vol. 1694. Davide Di Ruscio and Juan de Lara and Alfonso Pierantonio. CEUR-WS, Saint-Malo, France, (Oct. 2016), 2–11. https://hal.archives-ouvertes.fr/hal-01464800.

[13] Nicolas Hili and Raquel Araújo de Oliveira. 2022. A light-weight low-code platform for back-end automation. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 837–846.

[14] Panos Constantopoulos, John Mylopoulos, and Yannis Vassiliou, (Eds.) 1996. *Advanced Information Systems Engineering: 8th International Conference, CAiSE'96 Heraklion, Crete, Greece, May 20–24, 1996 Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg. Chap. MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment, 1–21. ISBN: 978-3-540-68451-0. DOI: 10.1007/3-540-61292-0_1.

[15] Stuart Kent. 2002. Model driven engineering. In *Integrated Formal Methods*. Michael Butler, Luigia Petre, and Kaisa Sere, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, 286–298. ISBN: 978-3-540-47884-3.

[16] Faezeh Khorram, Jean-Marie Mottu, and Gerson Sunyé. 2020. Challenges & opportunities in low-code testing. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 1–10.

[17] Meta. 2024. Llama 3. https://llama.meta.com/llama3. Accessed: 2024-07-05. (2024).

[18] Mistral. 2024. Mistral AI. https://docs.mistral.ai/. Accessed: 2024-07-05. (2024).

[19] Gunter Mussbacher et al. 2014. The relevance of model-driven engineering thirty years from now. In *Model-Driven Engineering Languages and Systems*. Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, (Eds.) Springer International Publishing, Cham, 183–200. ISBN: 978-3-319-11653-2.

[20] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Nick Barnes, and Ajmal Mian. 2023. A comprehensive overview of large language models. *arXiv preprint arXiv:2307.06435*.

[21] Object Management Group. 2011. Business Process Model and Notation - (BPMN 2.0). (2011). http://www.omg.org/spec/BPMN/2.0.

[22] Marten Oltrogge, Erik Derr, Christian Stransky, Yasemin Acar, Sascha Fahl, Christian Rossow, Giancarlo Pellegrino, Sven Bugiel, and Michael Backes. 2018. The rise of the citizen developer: assessing the security impact of online app generators. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 634–647.

[23] OpenAI. 2024. ChatGPT. https://platform.openai.com/docs/overview. Accessed: 2024-07-02. (2024).

[24] OpenAI. 2024. ChatGPT. https://chatgpt.com/. Accessed: 2024-06-12. (2024).

[25] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2023. LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation. *arXiv preprint arXiv:2308.02828*.

[26] Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. 2014. A tutorial on metamodelling for grammar researchers. *Science of Computer Programming*, 96, 396–416. DOI: https://doi.org/10.1016/j.scico.2014.05.007.

[27] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training.

[28] Clay Richardson and John R Rymer. 2016. Vendor landscape: the fractured, fertile terrain of low-code application platforms. *FORRESTER, Janeiro*.

[29] Clay Richardson, John R Rymer, Christopher Mines, Alex Cullen, and Dominique Whittaker. 2014. New development platforms emerge for customer-facing applications. *Forrester: Cambridge, MA, USA*, 15.

[30] Bran Selic. 2003. The pragmatics of model-driven development. *IEEE software*, 20, 5, 19–25.

[31] Tao Shi, Hui Ma, Gang Chen, and Sven Hartmann. 2021. Cost-effective web application replication and deployment in multi-cloud environment. *IEEE Transactions on Parallel and Distributed Systems*, 33, 8, 1982–1995.

[32] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. 2023. Prompt engineering or fine tuning: an empirical assessment of large language models in automated software engineering tasks. *arXiv preprint arXiv:2310.10508*.

[33] Jean-Sébastien Sottet and Nicolas Biri. 2016. JSMF: a Javascript Flexible Modelling Framework. In *Proceedings of the 2nd Workshop on Flexible Model Driven Engineering*. Vol. 1694. CEUR Workshops Modeling, 42–51.

[34] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF Eclipse Modeling Framework. The Eclipse Series*. Addison Wesley.

[35] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A Saurous, and Yoon Kim. 2024. Grammar prompting for domain-specific language generation with large language models. *Advances in Neural Information Processing Systems*, 36.

[36] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatGPT. *arXiv preprint arXiv:2302.11382*.

[37] Michael Wollowski. 2023. Using ChatGPT to produce code for a typical college-level assignment. *AI Magazine*, 44, 1, 129–130.