Towards LLM-powered consistency in model-based low-code platforms

1st Nathan Hagel
KASTEL
Karlsruhe Institute of Technology
Karlsruhe, Germany
nathan.hagel@kit.edu

2nd Nicolas Hili
Univ. Grenoble Alpes, CNRS
Grenoble INP, LIG
38000 Grenoble, France
nicolas.hili@univ-grenoble-alpes.fr

3rd Alexander Bartel
Dep. Information Management
Neu-Ulm University of Applied Sciences
Neu-Ulm, Germany
alexander.bartel@hnu.de

4th Anne Koziolek
KASTEL
Karlsruhe Institute of Technology
Karlsruhe, Germany
anne.koziolek@kit.edu

***Abstract*—Low-code platforms often use various models that define the application built by citizen developers. With the increasing size and complexity of the applications built using low-code platforms, the number of required models and the dependencies between them expand. However, with increased complexity, keeping these models consistent during the development or evolution of the application is crucial and often a non-trivial task for citizen developers.**

In this paper, we present four approaches to how LLMs can be used and integrated on the architecture level into low-code platforms to (i) create consistent models automatically, (ii) keep models consistent based on different types of dependencies, and (iii) support users in maintaining consistent models during the development. We implemented the approaches in a prototype and evaluated them in an exploratory study. The results show that state-of-the-art LLMs are capable of preserving consistency for low-code models as well as generating correct and consistent models in various scenarios.

***Index Terms*—LLM, AI, low-code development platform, metamodel, model-driven engineering, DSL, consistency**

I. Introduction

To shorten the development time of software applications, low-code platforms can enable citizen developers (developers with little or no software engineering background [1]), such as domain experts, to build functional applications effectively. These platforms typically offer graphical or textual interfaces to specify a system's behavior, user interface, or API, reducing the need for manual coding.

As low-code platforms target citizen developers with no programming skills, models are often used to abstract away the complexity of writing code manually. Depending on the concrete low-code platform, these models are explicitly defined, and Model-driven Engineering (MDE) techniques are integrated into the low-code platform [2], treating these models as a primary artifact. An alternative is implicitly integrating these models into the workflow of creating an application [3]. In any case, the code is generated from a model-like artifact or a set of artifacts.

Using a single model may not be sufficient for building complex applications. Dividing a model into multiple components can better manage large, more sophisticated applications. Therefore, citizen developers must create multiple models, such as those for the system's user interface, data model, and application behavior. Creating and maintaining consistency across numerous models can become a complex task, contradicting the idea that low-code platforms offer a straightforward development experience. AI-assisted low-code platforms can help citizen developers to create applications faster [4]. However, keeping the models consistent is still a challenging task. In this paper, we present four approaches for integrating Large Language Models (LLMs) into Low Code Development Platforms (LCDPs). These approaches aim to ease consistency preservation across models or to automate the process of keeping models consistent. By leveraging LLMs, the development effort required for implementing consistency mechanisms could be significantly reduced while enhancing the tools available for citizen developers and facilitating more efficient application creation.

We continue with background in Section II. Section III discusses related work. In Section IV, we describe the approaches on how LLMs can be implemented on low-code platforms. We evaluate the approaches in Section V and discuss the findings in Section VI. Section VII provides an outlook, and Section VIII concludes.

II. Foundations

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

A. Low-Code Development

LCDPs allow so-called *citizen developers* like domain experts to create, set up, and deploy software applications. They facilitate the development of applications through simple

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1608 – 501798263

graphical, e.g., drag'n'drop-oriented or textual interactions or the composition of prebuilt components and widgets. Users are often guided through structured development processes or receive contextual hints. Using these simple interactions during the development can reduce the amount of hand-written code to a minimum [5]. Therefore, they appeal to a larger group of stakeholders, as less or no software engineering knowledge is required to develop applications that meet the stakeholders' specific needs. However, as no programming skills can be expected, the LCDP must provide straightforward and easy-to-use mechanisms to ensure a smooth and well-working development experience. This also means that mechanisms to enhance the usability of the development platform should be given high priority.

By defining the desired application in the LCDP, users are either explicitly or implicitly creating one or several models. These models are then used by the LCDP to generate the running application using techniques related to MDE.

Business logic can be described by workflows or processes using Business Process Model (BPM) and Notation (BPMN 2.0) [6] or equivalent languages, while a data model used could be defined with a notation similiar to a class diagram. In the end, the artifacts often are related, like a database attribute that is used in a use case of the later application. Therefore, they must be kept consistent during development and evolution of an application. Working with several artifacts and models while creating an application in an LCDP is a challenge for users and developers of the LCDP as they have to keep the process as simple as possible without reducing the configuration space.

B. Model-driven Engineering

MDE is an established technique to develop software systems of various size and complexity [7]. Using MDE-techniques, an additional layer of abstraction is added on top of the programming languages [8], [9], which is also required in LCDPs. However, creating and defining good modeling languages that fulfill the LCDP's requirements as well as managing their complexity is challenging. MDE-techniques such as meta-modelling or modularity at the model level can help here as the MDE community faced similar problems.

In MDE, models are treated as primary artifacts in the development process and using model transformations, they are used for the generation of software applications [8]. The models themselves do not necessarily have to adhere to any known modelling language like UML. They can also be custom-defined graphical or textual Domain Specific Languages (DSLs). Using custom DSLs usually means that a *meta-model* is defined (e.g., using Eclipse Modeling Framework (EMF) [10] or FlexiMeta [11]) to formalize the DSL. This meta-model contains a set of *meta-classes*, representing the concepts of the domain and their *relations* to other model elements. Additionally, well-formed rules and even relations to other meta-models / meta-classes can be added to a meta-model's specification, establishing the criteria for valid models. These relations to other meta-models can be a part of the consistency rules between a set of models.

C. Large Language Models

Even though MDE techniques can facilitate the creation of models for LCDPs, creating models by hand remains difficult for citizen developers. Therefore LLMs can be used to simplify the creation of models [4].

Large Language Models (LLMs) are designed to generate coherent text based on input sequences provided by users. These models operate by estimating the probability of the next token in a sequence, given a set of preceding tokens. LLMs can be customized for specific tasks through fine-tuning, which involves updating their parameter weights to influence token generation [12]. This capability, along with the advent of general-purpose LLMs like ChatGPT [13], has enabled a wide range of applications, such as generating programming code, drafting text, proofreading, and detecting errors, among others [14].

To further extend the utility of LLMs, several methods can be employed to refine or enhance their outputs. Fine-tuning is one such method. However, it comes with significant challenges, primarily the need for large and often expensive-to-create datasets to achieve effective fine-tuning [15], [16].

An alternative to fine-tuning is prompt engineering. Prompts consist of the input information provided to an LLM, often including instructions or framing for a specific task or context. For instance, when generating code, prompts might specify the desired coding style or structure [14]. Prompt engineering involves crafting these prompts to achieve the desired output. It can range from simple modifications to generic prompts to creating sophisticated templates that structure requests and incorporate additional information or instructions for improved results [14]. Besides additional context information, one-shot or few-shot prompting can be used to improve the generation [17]. This means, that the LLM is provided with one or a few examples of the task in the prompt before requesting the desired generation. Compared to fine-tuning, prompt engineering is often less resource-intensive and can deliver competitive outcomes [16].

III. Related Work

Keeping consistency across models in modeling tools (facing similar problems as model-based LCDPs) is a challenging task. A large portion of modeling tools incorporating several models with interdependencies do not provide sufficient automatic consistency checks or support [18]. Therefore, finding good and applicable ways of checking and preserving consistency is still an open question [18]. One way is to explicitly define consistency rules between models as in [19]. These rules can then be extended by model transformations that automatically keep the models consistent as in the Vitruvius approach by [20]. Explicit, transformation-supported approaches have the advantage of lower computational overhead compared to LLM-based solutions. Though, modeling consistency dependencies explicitly is time consuming, especially when combined with the creation of transformations to preserve consistency [18]–[20].

Work from Wang et al. [21], Netz et al. [22] and Hagel et al. [4] investigated how models can be generated consistent to a defined meta-model or grammar specification using LLMs. Their results show, that given a modeling language specification and examples, it was possible to generate textual models consistent with the formal specification. However, none of these approaches, though sometimes also generating models with implicit or explicit external dependencies, focuses on consistency preservation across models. In the context of low-code platforms, the question of how such mechanisms can be integrated and which work best in different scenarios is still an open research question.

IV. Approach

Larger and more complex LCDPs often require a set of models (e.g., structural data model, model for the UI or for the applications behavior) to fully specify the application. To keep the models consistent, rules can be specified in different ways to either statically check consistency, or when using LLMs provide information how consistency must be ensured. We identified three methods to integrate consistency rules in LCDPs:

1) Formally specifying the consistency rules on meta-model level or by using constraints.
2) Adding consistency rules using natural language in predefined prompt templates.
3) Requiring the user of the LCDP to provide the desired consistency in natural language.
4) Relying on the implicit context of the LLM and its reasoning to connect the models without further specification.

These methods can be integrated to use LLMs in LCDPs for consistency-keeping when implementing the proposed approaches below. In the following, we propose four approaches on how LLMs can be integrated into LCDPs to ease consistency preservation during the use of the LCDP. The approaches explicitly do not cover how syntactically correct models can be generated, as this is covered in other work (e.g. [4], [22]).

A. Generating consistent models based on natural language

Creating correct models fulfilling the stakeholders' requirements can be a challenging task, even for experienced software engineers or modelers [23]. Working with several models interacting with each other, like a data model and a model for a user interface like a form or a dashboard, makes this task even more complicated. However, specifying the application's problem or goal in natural language can be easier for inexperienced users. Therefore, one solution is to directly generate consistent models in an iterative approach (see Figure 1 left side - A) based on the provided specification in natural language (step 1). This approach requires integrating a prompt into the LCDP, where the developer describes the application in natural language. This can range from quite concrete instructions, like the demand for a database table for customers, to more broad descriptions depending on the domain or type of applications the LCDP covers. Using prompt engineering techniques, like few-shot prompting, the LLM receives the specification along with examples of correctly generated models or model parts to improve the quality and consistency of the output. Providing additional context information through a structured prompt can be a way to specify dependencies between models. The LLM then generates the models or only a subset of them while ensuring consistency, based on the provided user specification (step 2). For example, if a user prompts the LLM "I want to store customers with their name and email", the user prompt can be enriched with structured instructions such as: "First, create the data model based on the specification. Then keep the user interface model consistent by adding or removing widgets and their references to the datamodel.". Previous work [4] showed that examples are helpful to improve the generated models, which therefore should be integrated into the prompt. This enriched prompt helps the LLM generate the models while maintaining their interdependencies. After model generation, the LCDP either generates the application or allows the user to inspect and modify the models further (step 3). The user can adjust the prompt or request changes, such as renaming a field or adding a new one, with the LLM ensuring consistency throughout the iterations. In each iteration, the LLM ensures consistency between each of the models.

B. Detecting inconsistencies after manual model changes

Another approach in using LLMs to ensure model consistency without the need to implement and specify each consistency rule during development of the LCDP is to use an LLM agent that repeatedly checks the models created by the user for inconsistencies. This approach is depicted in Figure 1 in the middle (B). While modeling an application in the LCDP (step 1), the LLM agent observes each model (step 2). Again, it is aware of the dependencies between the models, e.g., that a field in a defined model of a web form should reference an existing attribute in a data model. These dependencies can be for instance integrated into the prompt the LLM-agent is using to repeatedly query the LLM. This means, that these dependencies do not need to be defined formally using constraint languages like Object Constraint Language (OCL) or other formal specifications to statically check the models' correctness. They can even be defined in natural language. When detecting an inconsistency (step 3), the agent informs the user (step 4.1) about the inconsistency and directly proposes a change to one or several models to preserve consistency. The user can then decide if the change is accepted or ignored while keeping complete control over the models.

C. Detecting inconsistencies and preserving the consistency automatically

This approach is similar to the previous presented integration (see Figure 1 in the middle - C). While changing one or several models (step 1), the LLM agent observes these changes (step 2) and repeatedly tries to find inconsistencies (step 3). In contrast to earlier (step 4.1), the agent directly regenerates the models (step 4.2) that are inconsistent with the recent change

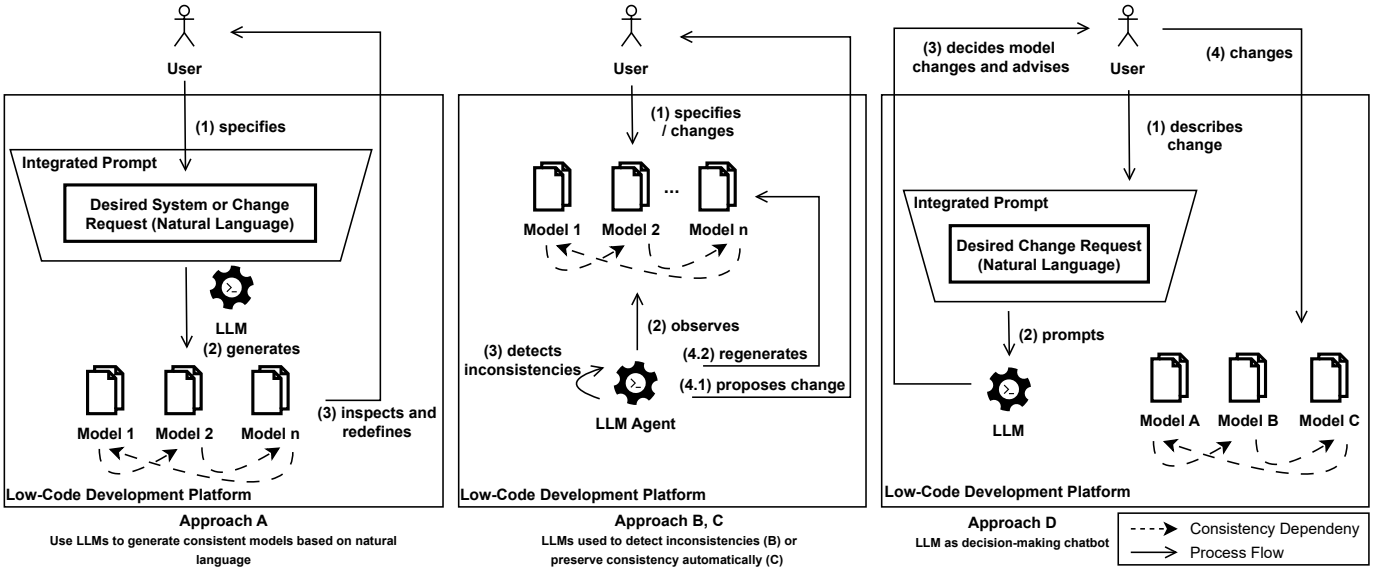

Fig. 1. Approaches A-D to integrate LLMs into low-code development platforms.

or changes. The LCDP can decide if the user is informed about these changes depending on the chosen user interface and development process.

D. LLM as a Decision-Making Chatbot for Consistency Preservation

Whereas the proposed approaches so far focus on either proactive generation or reactive correction of inconsitencies, another approach would be to put an LLM in the role of an active decision-maker. In this approach (see Figure 1 right side - D), the LLM acts as a decision-making chatbot that interacts with the user to ensure consistency preservation. The user first describes the desired modification to the chatbot (step 1). The LLM chatbot evaluates the user prompt (step 2) and decides which model(s) should be modified (step 3). For example, if there are three models A, B, and C, and model C depends on A and B, the chatbot determines whether changes should be made to model A, B, or C based on the user input.

Once the decision is made (e.g., to change model C), the chatbot explicitly describes the required modifications to model C which maintain consistency with models A and B. Depending on the concrete implementation, the user either accepts the proposed changes or performs them manually (step 4). This approach leverages the LLM's ability to understand dependencies and context and provides a more interactive as well as guided user experience, incorporating a multi-step decision-making process.

V. Experiment

As the running example we used a small LCDP prototype incorporating a domain model and a set of user interface models for web forms [2], [4]. For the tests, a textual concrete syntax for both modeling languages was chosen. A usual workflow when creating applications with this LCDP is the definition of a domain (like a hair salon business) with concepts and their attributes and relations. Then the user can define one or several user interfaces like web forms (e.g., to book an appointment). When generating models from natural language, an integrated chat interface (see Figure 2) is used. As described in Section IV, an ideal integration of each approach requires different development processes within the platform. Therefore, when testing the approaches, we targeted the question, if we can find working prompts for each of these approaches. We tested two scenarios using GPT-4o as the LLM and the hair salon example from [4] as the domain. First, a new widget for the birthday of the customer is added to an existing web form model, where the attribute already exists in the domain model. Therefore, depending on the approach, the model must be correctly extended (approaches A and C) or the correct hints or decision must be generated by the LLM (approaches B and D). In the end, both models or dictated model changes had to be consistent. For each approach using different prompts, the LLM was able to generate a correct extension of the model or description of the required change. The second scenario required that the birthday should be removed from the data model and the web form model had to be co-evolved or a description had to be generated which model had to be changed in which way. Here too, the generated results were generally correct, both when the model changes were directly generated for approaches A and C and when a description or change decision was required for approaches B and D. An exemplary prompt used for approach D is pictured in Listing 1. This version omits the use of formal consistency rules (type 1) and instead relies on the LLM's reasoning capabilities (type 4) to co-evolve the models. The user was not required in this case to add any consistency rules.

Though, being small and simple examples for consistency dependencies, the experiment shows that GPT-4o is capable of preserving consistency using the described approaches. Further

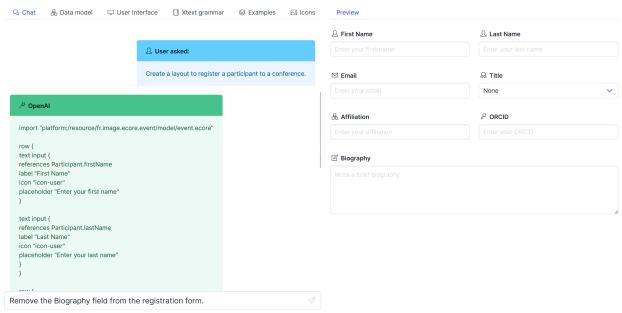

Fig. 2. User Interface of the LCDP [4] - LLM chat interface on the left, preview of the generated model on the right

```
Provided the following domain model:
[Domain Model in textual syntax]
And the following models of forms that define
the User Interface of the system:
[Layout Model]*
And the following desired change of the models
by the user:
"We do not longer store the birthday of our
customers."
Provide an explanation, how all the models
must be changed to ensure consistency between
every model, especially ensuring consistency
between the domain model and the UI.
```

Listing 1. Example prompt template used for approach D providing an explanation of the necessary steps to evolve the models consistently based on the desired user change.

experiments are still necessary to verify whether this also works in other scenarios as well as with other LLMs and to determine which approach provides the best user experience.

VI. Discussion

In the following, we discuss the approaches and their limitations and compare them with alternatives in preserving consistency in LCDPs.

As LLMs are capable of generating syntactically correct models in LCDPs [4], [21], [22], and our initial experiments show that this is most likely also the case when generating a set of models with dependencies between each other, using LLMs for consistency preservation in LCDPs seems to be a valid option for platform developers. However, the approaches have a different impact on the platform and how the platforms are used and also require some technical aspects that should be integrated on the architecture level of a LCDP.

Previous work [4] showed that adding LLMs in a chat-like interface similar to known providers such as ChatGPT [13] can increase the perceived usability and efficiency of LCDPs. This can be said not only for directly generating models or other artifacts but also for helpful chatbots that answer questions. Extending this work by generating all models and keeping them consistent in the development process should have a similar effect. This means that approach A - generating consistent models or generating the desired change described in natural language (see Figure 1 left side - A) - should be a valid choice for LCDP developers. However, an approach like this can change the use or the process how applications are created in the LCDP. One concern would be that the user could lose the overview over the system or the artifacts that must be created by only using the chat interface and mixing up e.g., the behavior of the system with a datamodel. This could even harm the resulting application or usability of the platform. In any case, this approach means that the user would need a chat interface integrated to the user interface of the LCDP with a way to inspect the generated results, either by inspecting the model or, if applicable, directly the generated application.

An alternative integration of LLMs to improve or add consistency preservation to an LCDP is to add an LLM agent observing the models created by the user, see approaches B and C. Integrating approach B - observing model changes and proposing changes or only hints which models must most likely be changed - does not change existing development processes or the user interface of the LCDP. It is not necessary to integrate, e.g., a chat interface to the system. However, the LLM agent would constantly observe the model changes and calculate in the background propositions to preserve the consistency of the models. This means that the computational overhead and therefore costs introduced to the LCDP is probably higher than with approach A. On the other side, the changes to the user interface and possibly already known development processes within the LCDP do not need be changed that dramatically.

Approach C - automatically regenerating models when a consistency violation is detected - is a stronger form of approach B, as the user would not have the choice how the violation would be resolved. Especially if the violation could be resolved in several ways, this decision would be made by the LLM, which does not necessarily have to but could result in unwanted changes or worse results. Also, as approach B, an integration of an LLM like this would also require more computational costs than approach A, while allowing to keep already known interactions with the platform intact.

Integrating approach D - using an LLM as a decision-making chatbot for consistency preservation - into an LCDP could enhance user interaction and satisfaction by providing a more guided and interactive experience. However, one concern is the increased complexity in the user interface, as the chatbot needs to interact with the user frequently, which might overwhelm some users. Additionally, the decision-making process of the LLM might not always align with the user's intentions. This can lead to frustration, escpecially if the suggested changes are not what the user expected.

All presented approaches use LLMs to preserve the consistency between the models. However, with formally well-defined meta-models or other technical setups, the consistency, at least parts of it, could also be checked statically without relying on LLMs. One approach is, for instance, the Vitruvius approach by Klare et al. [20]. Here, several models are also kept consistent with strictly defined model transformations, which propagate a change in one model directly to all other models, keeping the models always consistent. Though there is

probably less computational overhead during use, approaches like this or defining consistency together with required changes programmatically require a lot of knowledge about the domain when defining the meta-models. Furthermore, they usually require well-defined meta-models, while LLMs can often deal with incomplete specification, filling the gaps with provided or implicit context which has the potential to reduce the required effort when integrating consistency-keeping mechanisms into LCDPs. On the other hand, we assume that for large models with a vast number of model elements, the proposed approaches may not be applicable due to the limited context size of LLMs.

In the end, we think that, for many LCDPs, integrating LLMs could have a significant positive impact on the usability and efficiency of the platform while requiring a limited development effort from the platform developers. Especially for smaller and medium sized platforms or where the models have a limited size, these approaches could be more applicable than static solutions like the Vitruvius approach.

VII. Future Work

As the proposed approaches are still early work, further experiments must be conducted to validate them and gain further insights. A user study is planned to compare the different approaches, with a focus on usability, efficiency in completing the modeling tasks, and exertion compared to manual modeling. Additionally, to analyze the limitations of state-of-the-art LLMs to preserve consistency, another experiment with different LCDPs and with larger scaled models is planned. The results will be compared with a statical approach like Vitruvius [20] to preserve consistency. Such work would contribute to the research question of which approach is best and when LLM-based consistency preserving mechanisms are applicable.

VIII. Conclusion

In this paper, we presented four approaches to implement consistency preservation using LLMs featuring either automated consistency keeping of models in LCDPs or hints/instructions how models have to be consistently changed. The approaches do not necessarily require an explicit specification of consistency rules between models. We tested the approaches in a small experiment with promising results. Further validation of the approaches through user experiments with varying scenarios and levels of LLM involvement is subject of future work.

References

[1] M. Oltrogge, E. Derr, C. Stransky, *et al.*, "The rise of the citizen developer: Assessing the security impact of online app generators," in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 634–647.

[2] N. Hili and R. A. de Oliveira, "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*, 2022, pp. 837–846.

[3] Airtable, *Airtable*, https://www.airtable.com/, Accessed: 2025-01-04, 2024.

[4] N. Hagel, N. Hili, and D. Schwab, "Turning low-code development platforms into true no-code with LLMs," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS Companion '24, Linz, Austria: Association for Computing Machinery, 2024, pp. 876–885, ISBN: 9798400706226. DOI: 10.1145/3652620.3688334. [Online]. Available: https://doi.org/10.1145/3652620.3688334.

[5] C. Richardson, J. R. Rymer, C. Mines, A. Cullen, and D. Whittaker, "New development platforms emerge for customer-facing applications," *Forrester: Cambridge, MA, USA*, vol. 15, 2014.

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

[7] G. Mussbacher, D. Amyot, R. Breu, *et al.*, "The relevance of model-driven engineering thirty years from now," in *Model-Driven Engineering Languages and Systems*, J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, Eds., Cham: Springer International Publishing, 2014, pp. 183–200, ISBN: 978-3-319-11653-2.

[8] S. Kent, "Model driven engineering," in *Integrated Formal Methods*, M. Butler, L. Petre, and K. Sere, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 286–298, ISBN: 978-3-540-47884-3.

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

[10] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF Eclipse Modeling Framework* (The Eclipse Series). Addison Wesley, 2009.

[11] N. Hili, "A Metamodeling Framework for Promoting Flexibility and Creativity Over Strict Model Conformance," in *Flexible Model Driven Engineering Workshop*, ser. Flexible Model Driven Engineering, Davide Di Ruscio and Juan de Lara and Alfonso Pierantonio, vol. 1694, Saint-Malo, France: CEUR-WS, Oct. 2016, pp. 2–11. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01464800.

[12] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, *et al.*, "Improving language understanding by generative pre-training," 2018.

[13] OpenAI, *ChatGPT*, https://chatgpt.com/, Accessed: 2024-06-12, 2024.

[14] J. White, Q. Fu, S. Hays, *et al.*, "A prompt pattern catalog to enhance prompt engineering with ChatGPT," *arXiv preprint arXiv:2302.11382*, 2023.

[15] H. Naveed, A. U. Khan, S. Qiu, *et al.*, "A comprehensive overview of large language models," *arXiv preprint arXiv:2307.06435*, 2023.

[16] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati, "Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks," *arXiv preprint arXiv:2310.10508*, 2023.

[17] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[18] W. Torres, M. G. Van den Brand, and A. Serebrenik, "A systematic literature review of cross-domain model consistency checking by model management tools," *Software and Systems Modeling*, vol. 20, pp. 897–916, 2021.

[19] A. Qamar, C. J. Paredis, J. Wikander, and C. During, "Dependency modeling and model management in mechatronic design," *Journal of Computing and Information Science in Engineering*, vol. 12, no. 4, p. 041 009, 2012.

[20] H. Klare, M. E. Kramer, M. Langhammer, D. Werle, E. Burger, and R. Reussner, "Enabling consistency in view-based system development—the Vitruvius approach," *Journal of Systems and Software*, vol. 171, p. 110 815, 2021.

[21] B. Wang, Z. Wang, X. Wang, Y. Cao, R. A Saurous, and Y. Kim, "Grammar prompting for domain-specific language generation with large language models," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[22] L. Netz, J. Reimer, and B. Rumpe, "Using grammar masking to ensure syntactic validity in LLM-based modeling tasks," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS Companion '24, Linz, Austria: Association for Computing Machinery, 2024, pp. 115–122, ISBN: 9798400706226. DOI: 10.1145/3652620.3687805. [Online]. Available: https://doi.org/10.1145/3652620.3687805.

[23] M. Ozkaya and F. Erata, "Understanding practitioners' challenges on software modeling: A survey," *Journal of Computer Languages*, vol. 58, p. 100 963, 2020.