White-box LLM-supported Low-code Engineering: A Vision and First Insights

Thomas Buchmann*
thomas.buchmann@hof-university.de
Hof University of Applied Sciences
Hof, Germany

René Peinl*
rene.peinl@hof-university.de
Hof University of Applied Sciences
Hof, Germany

Felix Schwägerl*
felix.schwaegerl@oth-regensburg.de
OTH Regensburg
Regensburg, Germany

Abstract

Low-code development (LCD) platforms promise to empower citizen developers to define core domain models and rules for business applications. However, as domain rules grow complex, LCD platforms may fail to do so effectively. Generative AI, driven by large language models (LLMs), offers source code generation from natural language but suffers from its non-deterministic black-box nature and limited explainability. Therefore, rather than having LLMs generate entire applications from single prompts, we advocate for a white-box approach allowing citizen developers to specify domain models semi-formally, attaching constraints and operations as natural language annotations. These annotations are fed incrementally into an LLM contextualized with the generated application stub. This results in deterministic and better explainable generation of static application components, while offering citizen developers an appropriate level of abstraction. We report on a case study in manufacturing execution systems, where the implementation of the approach provides first insights.

CCS Concepts

• **Software and its engineering** → **Automatic programming**; **System modeling languages**; • **Computing methodologies** → **Natural language processing**.

Keywords

Model-driven engineering, large language models, low-code, semi-formal, artificial intelligence

ACM Reference Format:
Thomas Buchmann, René Peinl, and Felix Schwägerl. 2024. White-box LLM-supported Low-code Engineering: A Vision and First Insights. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3652620.3687803

1 Introduction

Low-code development (LCD) platforms [15] come with the promise to enable non-technical *citizen developers* in specifying the core *domain model* and *domain rules* of a business application, so that the latter can be automatically generated and deployed; see Figure 1a. However, the more complicated domain rules become, the more formalized the language and interface for domain rules must be, so that citizen developers once again are exposed to complicated syntax and semantics. Like *model-driven engineering* (MDE) [18], LCD relies on well-defined modeling languages, and many studies, e.g., [9] show that it is not structural elements (such as classes, attributes, or associations), but complex constraints and operations languages that raise the entrance barrier into LCD.

Recently, a new player has emerged – generative AI, mainly supported by large language models (LLMs) [11, 19]. While the first applications of LLMs aimed at providing users with more powerful ways to retrieve information (compared to standard internet search engines), generative AI has conquered many areas, from generating content for audio, video, and even program source code (see Figure 1b) based on the developer's natural language input, called *prompt* [10, 16]. First experiences have shown, however, that of LLM-based software development implies lacks of *precision* (important requirements may be overlooked or non-existing requirements may be hallucinated), *explainability* (intermediate steps such as software design are not captured), and *determinism* (repeated generation with same prompt may cause different results). We argue that these drawbacks are due to the *black-box* nature of the LLM-based software engineering approach: The source code is derived exclusively from prompts, without giving the software engineer the possibility to determine an invariable software core.

In this paper, we envision a novel approach using LLMs in conjunction with LCD in order to get the best out of both worlds; see Figure 1c. Rather than generating an entire application from a single prompt (sequence), we let the citizen developer define an invariable domain model, which consists only of structural elements (e.g., classes, attributes, and associations). All further constraints and operations are phrased as natural-language annotations and remain attached to the domain model until the application stub has been generated. Thereafter, every annotation is passed incrementally as an individual prompt into an LLM, which has been previously contextualized with the generated application stub. In this way, the static parts of the application are generated in a precise, explainable and deterministic way, while the constraints and operations can be phrased at a level of abstraction adequate to citizen developers. The novelty of our approach is rooted on the fact that every part of the generated source code can be traced down either to a static element of the (fully explainable) domain model or a fine-grained LLM prompt (with high precision and limited scope); therefore, we classify our work as a *white-box* approach.

*All authors contributed equally to this research.

MODELS Companion '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3687803

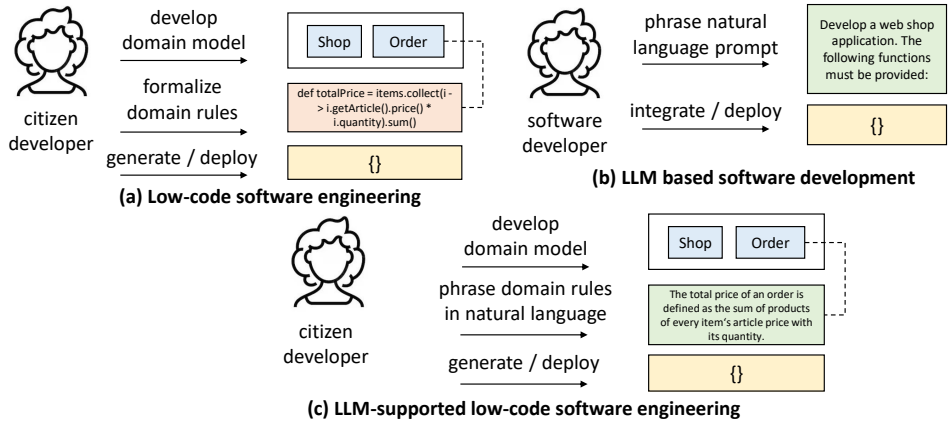

Figure 1: Envisioned white-box approach to combined low-code and LLM-based software engineering

2 Related Work

Since LLMs are currently the most advanced form of AI regarding their generalizability, their application in various text-based tasks is explored in [11, 19]. Code generation capabilities have not only been improved a lot in the past months, but this capability also contributed to general reasoning features of LLMs [10, 16]. Some limitations of LLMs like their well-known weakness in doing maths can be overcome by teaching them how to use a calculator. This is a special form of what is known as "tool-use" of LLMs [6]. These tools are usually available in form of an API with a formal syntax that is close to natural language. With such an API, an LLM can e.g. do an online search for information on a certain topic to be able to answer a question based on this context information, or trigger real-world actions like booking a hotel. An important question to address is whether models need fine-tuning , should get a low rank adaptation or it is enough to do few-shot learning by presenting them some examples to be able to generate the desired output.

One of the first to evaluate the use of LLMs in model driven development is [3]. They use ChatGPT to create ER, UML, BPMN and Heraklit diagrams with reasonable success. Vidgof et al. [17] analyze the applicability of LLMs to Business Process Management and argue that they are especially well-suited for the identification of processes based on documents. For process analysis, LLMs could be used to find filed complaints and summarize them. According to Vidgof et al., LLMs can also be used for redesign and implementation of business processes. However, the argumentation seems overly optimistic and lacks proof of how well their ideas work in practice. They state that future research needs to show "which tasks can be achieved with already existing models" and that data sets, benchmarks and new LLMs specialized on process management are required. Forell and Schüler [4] use GPT-4 to generate Petri-Net based process models in a two-step process. They found that the LLM had problems with splits and joins in the process. Nevertheless, for less complex models the approach was working. Kourani et al. [8] also use a two-step process, but use Partially Ordered Workflow Language (POWL) as an intermediate process representation that can be used to generate BPMN or Petri nets. They find GPT-4 to perform much better than Gemini. Grohs et al. [5] analyze the tasks of mining imperative process models from textual descriptions, mining declarative process models from textual descriptions, and assessing the suitability of process tasks from textual descriptions for robotic process automation. For all three tasks, GPT4 performs similarly to or better than the benchmark, i.e., specific applications for the respective task. Given process data in an event log or relational format, Kampik et al. [7] propose the use of large process models, a combination of LLMs with symbolic reasoning, to automatically identify the domain of a specific process as well as the context of the organization that runs it, to then generate insights and action recommendations, using a collection of tools for process design, analysis, execution, and prediction. The authors in [1] couple Retrieval-Augmented Generation with fine-tuning, to enrich process-specific knowledge and enable actionable conversations with human workers. A process-aware chunking approach is incorporated to enhance the pipeline. It has been evaluated in various experimental scenariosand showed promising results for use in process-aware Decision Support Systems.

3 Example

We detail the envisioned approach by means of an example scenario, a *web shop*. Since our envisioned tool chain has not been realized yet, we rely on three semi-automated steps (further detailed in subsections below):

(1) Design of an initial domain model consisting of static elements and stubs with optional human-language annotations for domain rules.
(2) Partial code generation, using the domain model as input and producing intermediate Java code as output. The generated Java code contains static elements such as classes, fields and getters/setters. For dynamic elements such as constructors or operations, we generate stubs including the human-language annotations as comments.
(3) Usage of a LLM contextualized with the existing source code in order to fill in the generated stubs one after another. For every operation, an implementation is retrieved from a generated prompt. The generated code is then copied back into the source code generated in the previous step.

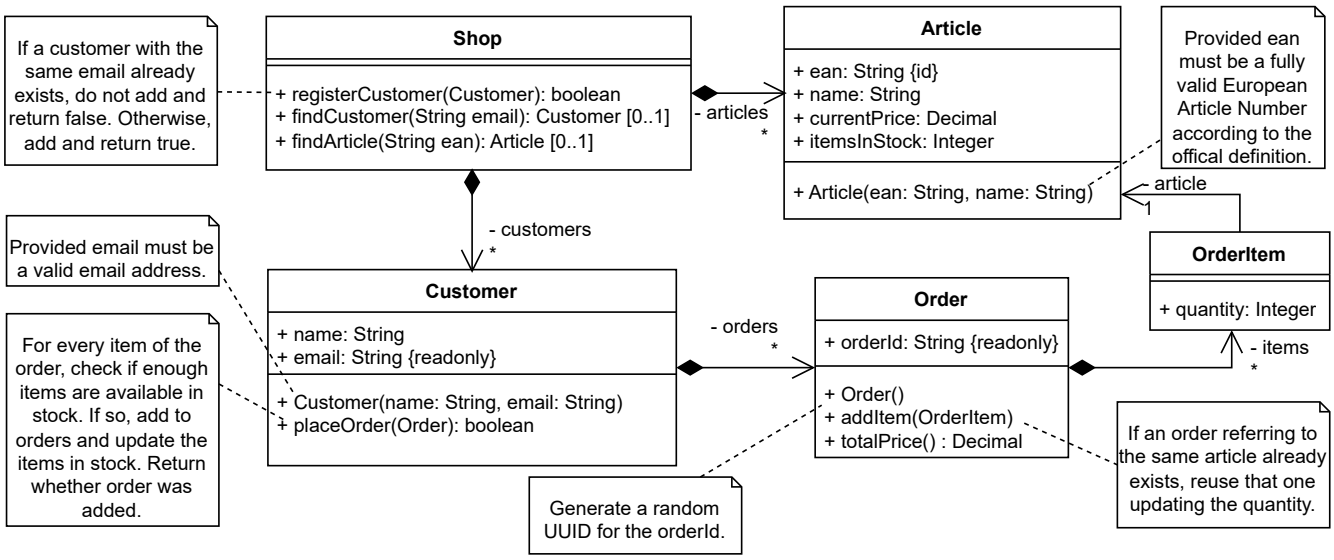

Figure 2: The domain model for the *shop* example, including natural-language annotations.

3.1 Domain Model

Figure 2 depicts the domain model as it typically results from a domain analysis. Its static part consists of five classes, five associations, eight attributes. The dynamic part is modeled by six operations and three constructors. All constructors as well as three out of the six operations carry natural-language annotations; for the remaining operations, it was assumed that the implementation can be easily inferred from the context (i.e., operation name and structural elements of parent and adjacent classes). The natural language annotations contain constraints whose implementation can be inferred either from the context (e.g., "If a customer with the same email already exists, ...") or from common knowledge (e.g., the specific constraints underlying e-mail addresses or European Article Numbers).

The here depicted domain model has been created using the MDE tool *Valkyrie* [2]. Both the model and the generated code can be found in an on-line repository.[1] For the implementation of our suggested approach, we envision a low-code modeling environment containing a language that is deliberately *semi-formal*.

3.2 Partial Code Generation

As another internal component of the LLM-supported low-code environment, we envision a specific code generator capable of creating and maintaining the intermediate source code with incremental updates. For this example, we resort to manual transformation into Java source code, using only built-in standard library classes (e.g., `ArrayList`).

The intermediate source code can be found in the on-line repository, too.[2] Figure 3 shows `Order` as representative of the five generated Java classes. Model attributes are mapped to private fields and corresponding getters or setters (depending on their visibility and read-only state). For operations, only the signatures have been generated into corresponding Java methods; the stubs are non-functional. Human-language annotations attached to operations are mapped to Java comments placed above the generated stubs.

```
public class Order {
 private String orderId;
 private List<OrderItem> items =
  new ArrayList<>();

 /*Generate a random UUID for the orderId.*/
 public Order() {
  throw new UnsupportedOperationException();
 }

 public String getOrderId() {
  return orderId;
 }

 public List<OrderItem> getItems() {
  return Collections.unmodifiableList(items);
 }

 /* If an order referring to the same article
  * already exists, reuse that one updating
  * the quantity. */
 public void addItem(OrderItem orderItem) {
  throw new UnsupportedOperationException();
 }

 public double totalPrice() {
  throw new UnsupportedOperationException();
 }
}
```

Figure 3: Java code generated for the class `Order`.

[1] https://github.com/scx-othr/mdeintelligence24-shop-example/tree/main/src/valkyrie

[2] https://github.com/scx-othr/mdeintelligence24-shop-example/tree/main/src/stage1

```
public class Order {
 // (existing fields and getters omitted)

 public Order() {
  this.orderId = UUID.randomUUID().toString();
 }

 public void addItem(OrderItem orderItem) {
  for (OrderItem item : this.items) {
   if (item.getArticle().equals(
         orderItem.getArticle())) {
          item.setQuantity(item.getQuantity() +
            orderItem.getQuantity());
          return;
         }
   }
  this.items.add(orderItem);
 }

public double totalPrice() {
 double total = 0.0;
  for (OrderItem item : this.items) {
   total += item.getArticle().getCurrentPrice()
         * item.getQuantity();
   }
  return total;
 }
}
```

Figure 4: Missing operation implementations added by LLM.

3.3 LLM-based Refinement

The final steps consists in a refining in-place transformation of the partially generated Java source code. To this end, we use the commercial code LLM GitHub Copilot[3]. The envisioned tool will be able to plug-in arbitrary LLMs via a standardized interface such as *Ollama*.[4]

In our semi-automatic experimentation, we derive a straightforward prompt for every missing method stub. For instance, for the class `Order`:

- Prompt 1: "implement constructor taking into consideration the comment above"
- Prompt 2: "implement `addItem` taking into consideration the comment above"
- Prompt 3: "implement `totalPrice`"

In the envisioned approach, these prompts will be generated from a template, so that the specifics of both the modeling environment and the used LLM can be considered.

Creating the resulting source code requires manual copy-and-paste in our experimentation, but will be automated in our envisioned approach. The so completed source code for class `Order` is shown in Figure 4. The on-line repository[5] contains detailed prompts and the LLM-generated code for all classes. It also contains a more sophisticated example: The code generated by the LLM for the constructor of `Article` contains an additional private method `isValidEan`; such additions must also be supported in the final solution.

[3]https://github.com/features/copilot
[4]https://ollama.com/
[5]https://github.com/scx-othr/mdeintelligence24-shop-example/tree/main/src/stage2

3.4 Further Aspects

During the practical experimentation of this example, seven of the nine operation bodies were provided correctly by the LLM, while we had to re-phrase two of them. This was due to an incomplete validation of the `ean` in the `Article` constructor, and a logical error in the `placeOrder` method, respectively. The approach as presented so far is capable of updating rules and operations *incrementally* by re-phrasing their attached annotations and repeating step three for the affected elements. We believe, however, that the user experience can be even improved by applying exploiting the *conversational* nature of LLMs. Concretely, the developer might provide natural-language feedback, based on which both the annotations are updated and the code generation is repeated until the specification is met.

The here presented example contains simplifications not applicable in real-world LCD environments. For instance, we focus on the data model perspective, ignoring further aspects typically part of LCD, such as *user interface design*. We expect, however, that the here envisioned approach is applicable, also to input field validation and the like.

4 Case Study and Insights

Within the BMBF project MOONRISE, the efficient introduction of the highly customizable manufacturing execution system HiCuMES in six manufacturing SMEs is studied [13]. It uses a BPMN-based workflow engine as well as a data schema editor and schema mapper to adapt the MES to the specifics of the organization as part of the customization process and connect it to other information systems, especially ERP-systems.

In this customization process, the process model is the central element representing the production processes. An LLM could aid here in multiple aspects.

(1) The process model generated by a domain expert will likely miss important technical aspects that hinder the Camunda workflow engine to execute the model. These aspects go beyond the capabilities of a linter or pure syntax validation. If the LLM is able to understand both the process as well as the technical requirements it can guide the user enhance the model in these aspects.
(2) Validation functions and conditional flows need to be stated in GroovyScript or JavaScript. This might be beyond the capabilities of a domain expert, although the main challenge usually is the correct identification of the data in object-oriented syntax (e.g. machine-occupation.production-order.planned-start)
(3) Adding form fields to the process [12] is another important customization task, in order to adapt the UI to show all required information. Although this is not extremely complicated, in our tests, process consultants still failed to do that with only a short introduction to the task. An LLM could serve as a technical guide that explains mistakes that were made and give tips for successful implementation.
(4) Finally, the form fields need to be mapped to HiCuMES' database schema, which can also be extended to incorporate company-specific data. Here, the aid could be in identifying possible matches based on semantic similarities between the

field and schema names, since the database schema can easily become confusing. Furthermore, the LLM could identify problems with datatype mismatches between form fields and database schema and suggest fixes.

In an even more visionary setting, the LLM could generate the initial BPMN processes from natural language input, maybe even in form of audio that is transcribed by automatic speech recognition systems like Whisper [14]. This should be an iterative process in a dialog style, where the visual output of the resulting BPMN would be the input for humans to dictate further extensions or refinements. Therefore, humans would only have to care about the layout.

5 Conclusion

In this visionary paper, we propose a novel white-box approach integrating generative AI, particularly large language models (LLMs), into low-code development (LCD) platforms. This approach empowers citizen developers by allowing them to define domain models using structural elements, while complex constraints and operations are expressed as natural-language annotations. This fusion of LCD and LLM technology promises precise, explainable, and deterministic generation of applications, bridging the gap between technical complexity and non-technical user proficiency.

While we still cannot exclude the possibility of instances of the typical drawbacks of LLM-based code generation, namely lack of precision, potential hallucinations and limited explainability, we argue that their impact will be considerably weaker due to the locality of the prompts in our suggested white-box approach.

References

[1] Mario Luca Bernardi, Angelo Casciani, Marta Cimitile, and Andrea Marella. [n. d.]. Conversing with business process-aware Large Language Models: the BPLLM framework. ([n. d.]). PREPRINT (Version 1) available at Research Square.

[2] Thomas Buchmann. 2012. Valkyrie: A UML-based Model-driven Environment for Model-driven Software Engineering. In *Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT 2012)*, Slimane Hammoudi, Marten van Sinderen, and José Cordeiro (Eds.). ScitePress, 147–157.

[3] Hans-Georg Fill, Peter Fettke, and Julius Köpke. 2023. Conceptual Modeling and Large Language Models: Impressions From First Experiments With ChatGPT. *Enterp. Model. Inf. Syst. Archit. Int. J. Concept. Model.* 18 (2023), 3.

[4] Martin Forell and Selina Schüler. 2024. Modeling meets Large Language Models. Modellierung 2024 Satellite Events.

[5] Michael Grohs, Luka Abb, Nourhan Elsayed, and Jana-Rebecca Rehse. 2023. Large Language Models Can Accomplish Business Process Management Tasks. In *Business Process Management Workshops - BPM 2023 International Workshops, Utrecht, The Netherlands, September 11-15, 2023, Revised Selected Papers (Lecture Notes in Business Information Processing, Vol. 492)*, Jochen De Weerdt and Luise Pufahl (Eds.). Springer, 453–465.

[6] Yaru Hao, Haoyu Song, Li Dong, Shaohan Huang, Zewen Chi, Wenhui Wang, Shuming Ma, and Furu Wei. 2022. Language Models are General-Purpose Interfaces. *CoRR* abs/2206.06336 (2022).

[7] Timotheus Kampik, Christian Warmuth, Adrian Rebmann, Ron Agam, Lukas N. P. Egger, Andreas Gerber, Johannes Hoffart, Jonas Kolk, Philipp Herzig, Gero Decker, Han van der Aa, Artem Polyvyanyy, Stefanie Rinderle-Ma, Ingo Weber, and Matthias Weidlich. 2023. Large Process Models: Business Process Management in the Age of Generative AI. *CoRR* abs/2309.00900 (2023).

[8] Humam Kourani, Alessandro Berti, Daniel Schuster, and Wil M. P. van der Aalst. 2024. Process Modeling With Large Language Models. *CoRR* abs/2403.07541 (2024).

[9] Yajing Luo, Peng Liang, Chong Wang, Mojtaba Shahin, and Jing Zhan. 2021. Characteristics and challenges of low-code development: the practitioners' perspective. In *Proceedings of the 15th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*. 1–11.

[10] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *CoRR* abs/2306.08568 (2023).

[11] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023).

[12] René Peinl and Ornella Perak. 2019. BPMN and DMN for Easy Customizing of Manufacturing Execution Systems. In *BPM 2019 International Workshops, Vienna, Austria, September 1-6, 2019, Revised Selected Papers*, Chiara Di Francescomarino, Remco M. Dijkman, and Uwe Zdun (Eds.), Vol. 362. Springer, 441–452.

[13] René Peinl, Susanne Purucker, and Sabine Vogel. 2023. Dependencies between MES features and efficient implementation. *Procedia Computer Science* 219 (2023), 897–904. CENTERIS / ProjMAN / HCist 2022.

[14] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine Mcleavey, and Ilya Sutskever. 2023. Robust Speech Recognition via Large-Scale Weak Supervision. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*. PMLR, 28492–28518.

[15] Karlis Rokis and Marite Kirikova. 2022. Challenges of low-code/no-code software development: A literature review. In *International Conference on Business Informatics Research*. Springer, 3–17.

[16] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023).

[17] Maxim Vidgof, Stefan Bachhofner, and Jan Mendling. 2023. Large Language Models for Business Process Management: Opportunities and Challenges. In *BPM 2023 Forum, Utrecht, The Netherlands, September 11-15, 2023, Proceedings (Lecture Notes in Business Information Processing, Vol. 490)*. Springer, 107–123.

[18] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. 2006. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons.

[19] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent Abilities of Large Language Models. *Trans. Mach. Learn. Res.* 2022 (2022).