

To Model, to Prompt, or to Code? The Choice Is Yours: A Multi-Paradigmatic Approach to Software Development

Thomas Buchmann¹^a, Felix Schwägerl²^b and René Peinl¹^c

¹Hof University of Applied Sciences, Hof, Germany

²OTH Regensburg, Regensburg, Germany

Keywords: Model-Driven Development, Large Language Models, Software Engineering, Code Generation.

Abstract: This paper considers three fundamental approaches to software development, namely manual coding, model-driven software engineering, and code generation by large language models. All of these approaches have their individual pros and cons, motivating the desire for an integrated approach. We present MoProCo, a technical solution to integrate the three approaches into a single tool chain, allowing the developer to split a software engineering task into modeling, prompting or coding sub-tasks. From a single input file consisting of static model structure, natural language prompts and/or source code fragments, Java source code is generated using a two-stage approach. A case study demonstrates that the MoProCo approach combines the desirable properties of the three development approaches by offering the appropriate level of abstraction, determinism, and dynamism for each specific software engineering sub-task.

1 INTRODUCTION

This paper contributes and discusses an integrated approach combining the three software development paradigms summarized in Figure 1, namely modeling, prompting and coding. Our suggested multi-paradigmatic approach is implemented by the tool *MoProCo* (= modeling, prompting and coding woven together), which is documented in this paper and published as VSCode extension.

1.1 Background and Motivation

Manual *coding* is the traditional paradigm to software development and still preferred for many scenarios, e.g., when performance or safety are crucial. Higher-level programming languages are *deterministic* by design, meaning that the same syntactical input always produces equivalent run-time semantics. Furthermore, they are suited for describing the *dynamic* behavior of software systems. On the downside, programs have to be specified on a high level of *detail*, requiring a deep understanding of programming languages, algorithms, and design patterns, making the

approach time-consuming and error-prone.

The *modeling* paradigm has gained significant attention in recent decades. Model-Driven Development (MDD) (Völter et al., 2006) focuses on creating *abstract* models of software systems, which are then transformed into concrete implementations using automated and *deterministic* generators. Relying on standards like the Uniform Modeling Language (UML) (OMG, 2017) or domain-specific languages (Fowler and Parsons, 2010), this approach emphasizes traceability, consistency and reusability. It was, however, demonstrated that MDD can be challenging to adopt in practice, particularly when it comes to modeling *dynamic* behavior of systems. Therefore, modeling is often used for the static structure as a starting point, and dynamic behavior is added, e.g., by manual coding (Steinberg et al., 2009).

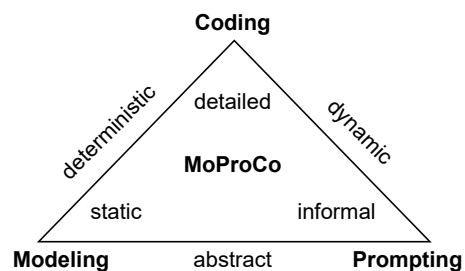





Figure 1: A triad of software development paradigms.

^a <https://orcid.org/0000-0002-5675-6339>

^b <https://orcid.org/0009-0004-9852-5418>

^c <https://orcid.org/0000-0001-8457-1801>

Third, *prompt*-based software engineering using Large Language Models (LLMs) has recently revolutionized software development, particularly in code completion, bug fixing, and program synthesis tasks (Lyu et al., 2024). Being based on statistics and heuristics, LLMs are inherently informal and lack *determinism*, making it challenging to ensure the correctness and reliability of generated code. LLMs, having been trained on vast amounts of data and being able to generalize to new scenarios, accept *informal* input, making them particularly well-suited for generating *dynamic* parts of software systems. LLMs generate code from informal natural language prompts usually phrased at a high level of *abstraction*.

1.2 Contributions

Previous work (Buchmann et al., 2024) suggests that a combination of MDD and LLMs is promising; MDD provides a deterministic structure which “tames” the inherently volatile LLMs by using them only for local prompts with limited scope. To date, integrated tool support for interweaving model-driven and LLM-generated artifacts is still missing. This is the research gap where the paper at hand steps in by making the following contributions:

1. An extension to an existing textual modeling language for UML-like class models by two types of annotations: natural language prompts and Java code snippets.
2. Fully automated transformation tooling consisting of two components, one of which is responsible for deterministically transforming models (and their contained Java code snippets) into an initial code base. The second component then uses LLMs to generate method bodies based on the natural language prompts and weaves them into the code base.
3. A case study (on-line shop) demonstrating the feasibility and effectiveness of the proposed approach.
4. A classification of related research in the intersecting fields of MDD and LLMs.

1.3 Outline

The remainder of this paper describes the design and implementation of our Toolchain (Section 2), provides a qualitative discussion (Section 3), and aligns with related work (see Section 4). Future work is outlined in the conclusion.

2 THE MoProCo TOOLCHAIN

In this section, we present the architecture and usage of the MoProCo toolchain. Figure 2 summarizes the explanations below. Source code and build instructions are available in two separate GitHub repositories⁴. We implemented the toolchain as two extensions to the popular editor VSCode⁵. The extensions’ core components are:

Editor A textual editor for a class model language (a subset of UML class diagrams). It describes the static structure of a software system to be developed. Operations may optionally carry natural language prompts and Java code snippets.

Generator A deterministic code generator that transforms the class models into two artifacts, namely a visualization file for PlantUML (see below) and an initial Java code base. The latter contains the static class structure derived from the model (e.g., considering bidirectional and composition semantics for associations). Code snippets are directly copied into the generated code. Natural language prompts attached to operations are generated into JavaDoc comments, which are in turn processed by the downstream Weaver component.

PlantUML Viewer We reuse the PlantUML extension for VSCode to graphically visualize the class model as diagram while the developer creates them using the textual syntax; see Figure 3. The PlantUML file (extension `.classdiag`) is generated by the Generator component automatically when the class model file is updated and saved.

Weaver The Weaver component is responsible for generating method implementations based on the natural language prompts attached to methods in the intermediate Java code. The Weaver uses a large language model (LLM) to generate Java code fragments based on the prompts and the existing codebase. The generated code snippets are then woven into the intermediate Java code produced by the upstream Generator component.

LLM Server For generating code snippets from JavaDoc annotations, the Weaver component interacts with a remote server hosting the LLM. Our architecture does not assume a specific LLM; rather it offers integrations based on two de-facto standard interfaces, namely *llama.cpp*⁶ (best performance if the LLM runs locally) and OpenAI-

⁴<https://github.com/tbuchmann/class-diag-langium>,
<https://github.com/tbuchmann/mdellmprocessor>

⁵<https://code.visualstudio.com/>

⁶<https://github.com/ggml-org/llama.cpp>

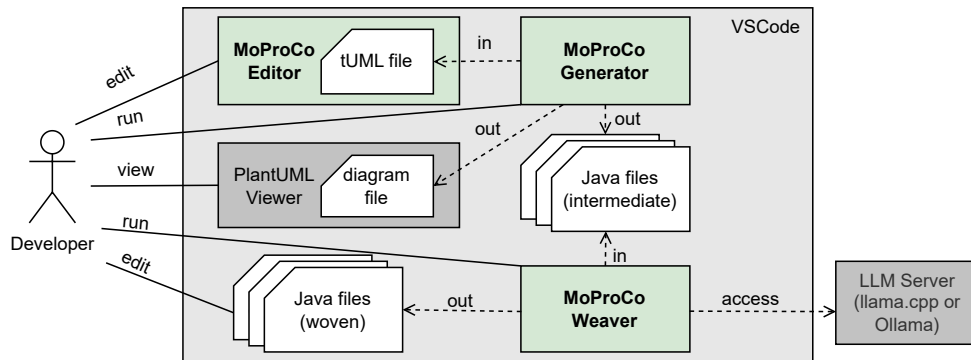


Figure 2: A context view of the architecture of the MoProCo toolchain. Gray boxes indicate existing components, while green boxes indicate components developed as part of the new tool chain.

compatible endpoints such as *Ollama*⁷ (if the LLM runs on a remote server). The component is configurable with respect to the URL of the LLM server, as well as the language model and variant.

2.1 MoProCo Editor

The first component, MoProCo Editor, is based on the textual modeling language *tUML* (Buchmann and Schwägerl, 2025), which allows to describe domain models at a level of abstraction between UML and source code. It has been developed with Langium⁸ and TypeScript and integrates with PlantUML⁹ to visualize the model as non-editable diagram.

A minimal example (an excerpt of the case study initially presented in (Buchmann et al., 2024)) of the textual syntax is shown in Listing 1. Packages may contain primitive types, classes, associations (**assoc**), and interfaces as well as nested packages (not shown in the example). Classes contain attributes with primitive types and optional cardinalities (lower .. upper; -1 is for unbounded). Operations of classes are modeled with visibility, parameters and return types. Associations have two ends, each referencing a class and carrying visibility and multiplicity. One association end may be marked as **composite**. A more detailed description of the *tUML* syntax (not considering the *prompt* and *code* paradigms) is provided in (Buchmann and Schwägerl, 2025).

In the *model* syntax, the connection to the *prompt* and *code* paradigms is achieved as follows: Operations may optionally be annotated with either a prompt (**desc**, followed by a natural language description of the expected dynamic behavior) or an im-

plementation (**impl**, followed by Java source code). These annotations are used by the downstream Generator to produce JavaDoc comments and Java code.

```
package shop {
    primitive Integer
    // more primitive types
    class Customer {
        public names : String
        public email : String [0..1]
        public placeOrder(o : Order) : Boolean
        desc "For every item of the order,
        check if enough items are in stock.
        If so, add to orders and update the
        items in stock. Return whether
        order was added"
    }
    class Order {
        public orderID : String
        public addItem(item : OrderItem)
        : Boolean desc "If an order
        referring to the same article
        already exists, reuse that one
        updating the quantity and returning
        true. Else return false."
        public totalPrice() : Decimal
        impl "return items.stream()
        .mapToDouble(i -> i.getQuantity()
        * i.getArticle().getCurrentPrice()
        ).sum();"
    }
    // more classes and associations
    assoc hasOrders {
        public customer : Customer [1..1]
        public orders : Order [0..-1]
        composite
    }
    assoc hasItems {
        public order : Order [1..1]
        public items : OrderItem [0..-1]
        composite
    }
}
```

Listing 1: A minimal example of a *tUML* file.

⁷<https://ollama.com/>

⁸<https://langium.org/>

⁹<https://plantuml.com/>

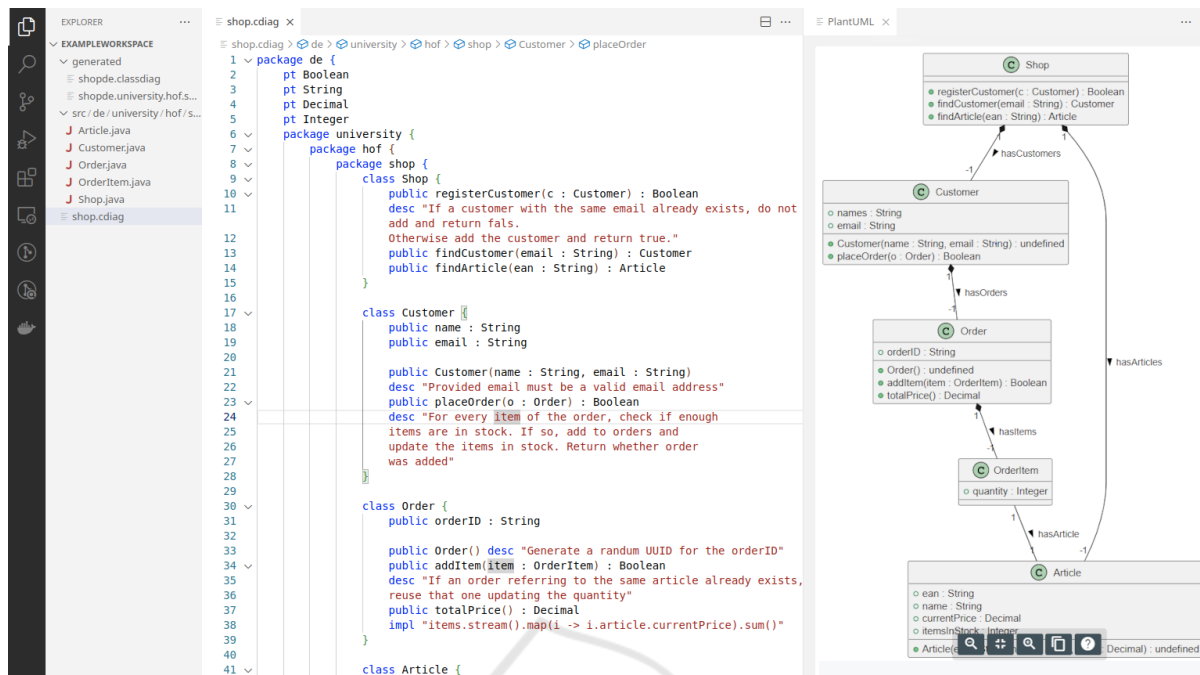


Figure 3: A screenshot showing the MoProCo editor and the PlantUML Viewer in VSCode.

2.2 MoProCo Generator

The generator receives as input a *tUML* file and produces two artifacts: a PlantUML file for visualization and the intermediate Java file. For consistency reasons, the visualization is automatically generated when the developer saves a changed class model, whereas the Java code generation must be manually triggered using the VSCode command *Generate Java Code* contributed by our extension. We here describe the Java code generation focusing on the *prompt* and *code* paradigms; general Java code generation as well as PlantUML generation is described in (Buchmann and Schwägerl, 2025).

Listing 2 shows an excerpt of the intermediate code generated for Listing 1; the full code is available in the GitHub repository referenced above.

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

    public String getOrderId() {
        return this.orderId;
    }
    public void setOrderId(String orderId) {
        this.orderId = orderId;
    }

    public Customer getCustomer() {
        return this.customer;
    }
}
```

```
    }
    public void setCustomer(Customer c) {
        if (this.customer != caption) {
            Customer o = customer;
            if (o != null) {
                this.customer = null;
                o.removeFromOrders(this);
            }
            this.customer = c;
            if (c != null)
                c.addToOrders(this);
        }
    }

    public List<OrderItem> getItems() {
        return unmodifiableList(this.items);
    }
    public void addToItems(OrderItem o) {
        if (!this.items.contains(o)) {
            this.items.add(o);
            o.setOrder(this);
        }
    }
    public void removeFromItems(OrderItem o) {
        if (this.items.contains(o)) {
            this.items.remove(o);
            o.setOrder(null);
        }
    }

    /** @prompt If an order referring to the
        same article already exists,
        reuse that one updating the
        quantity. */
    public Boolean addItem(OrderItem item) {
```

```

        // to be inserted by LLM
    }

    public Double totalPrice() {
        return items.stream().mapToDouble(i ->
            i.getQuantity() * i.getArticle()
                .getCurrentPrice()).sum();
    }
}

```

Listing 2: Generated intermediate Java code excerpt.

The generated Java code implements the class structure defined in the model, considering the semantic details. For instance, attributes are transformed into Java fields with getters and setters, associations are transformed into pairs of Java fields with consistency-preserving bidirectional accessor methods (e.g., `setCustomers` and `addToItems/removeFromItems`). Being based on templates, the code generated here is *deterministic* and traceable to the static model structure; a property that LLMs cannot guarantee due to their non-deterministic nature.

Operations are generated as Java methods based on the the following rules:

- If no annotation is provided, only a stub is generated, instructing the developer to implement the method afterwards.
- If a `desc` annotation is provided, a JavaDoc comment is generated with the content of the annotation. The comment serves as a controlled prompt for the Weaver component.
- If an `impl` annotation is provided, the content of the annotation is directly copied into the body.

2.3 MoProCo Weaver

The MoProCo Weaver extension is responsible for generating method implementations based on the natural language prompts attached to methods in the intermediate Java code as JavaDoc annotations. Triggered by a context menu action *Process Java files*, the Weaver proceeds as specified by the following algorithm:

- For every Java file containing a `@prompt` tag
 - For every occurrence of `@prompt` in the file
 - * Extract method name and memorize range of method body (embraced by `{}`) of the subsequent Java method.
 - * Extract the value of the `@prompt` tag.
 - * Send a request to the LLM on the LLM server. The request includes:
 - all Java files of current project as context,

- a system prompt instructing the LLM to generate Java code for the method based on the provided user prompt without additional explanations or markup in the response.
- The specified prompt as user prompt.
- * Extract the content of the method body from the response produced by the LLM.
- * Replace memorized range of the Java file with the extracted method body.

Listing 3 shows the generated Java code after the Weaver has processed the `@prompt` annotations.

```

public class Order {
    // existing contents

    /** @prompt If an order referring to the
        same article already exists, reuse that
        one updating the quantity and returning
        true. Else return false. */
    public Boolean addItem(OrderItem item) {
        for (OrderItem i : this.items) {
            if (i.getArticle().equals(
                item.getArticle())) {
                i.setQuantity(i.getQuantity()
                    + item.getQuantity());
                return true;
            }
        }
        this.items.add(item); return false;
    }

    public Double totalPrice() {
        return items.stream().mapToDouble(i ->
            i.getQuantity() * i.getArticle()
                .getCurrentPrice()).sum();
    }
}

```

Listing 3: An excerpt of the woven Java code.

Method `addItem` has been completed by the LLM based on the provided prompt. To provide traceability to the original model, the `@prompt` annotation is retained in the JavaDoc comment. Method `totalPrice` has not been touched by the Weaver since it does not carry a `@prompt` annotation.

3 DISCUSSION

Qualitative experiments of our tool based on the presented case study raised several questions.

3.1 Determinism and Traceability

The added determinism of our hybrid approach is a key advantage over pure (black-box) LLM-based

code generation (Buchmann et al., 2024). The Mo-ProCo Generator component ensures that the generated code is consistent with the static model and that the generated methods are traceable to their originating natural language prompts. This allows developers to reason about the generated code and to understand the relationship between model, prompts, and code.

In contrast to the black-box style, the code fragments to be generated by LLMs are much smaller, giving the developer more control over the generated code and reducing the risk of, e.g., hallucinations. The Weaver component further enhances the traceability of the generated code by preserving the original prompts in the JavaDoc comments. This allows developers to understand why a certain piece of code was generated and to verify that the generated code is consistent with the intended behavior. Nevertheless, the decision to employ LLMs only locally still does not make our presented approach fully deterministic. Varying results and hallucinations may still occur.

3.2 Quality and Consistency

The quality of the code produced by the Weaver component depends on both the capabilities of the connected LLM and the quality of the natural language prompts provided by the developer.

As soon as the LLM generates incorrect code, however, it is hard to determine if the problem is due to the prompt or due to the LLM. Our contribution does not intend to overcome this general problem, but to provide a tool chain that allows developers to experiment with different LLMs and prompts in order to engineer the best possible solution iteratively.

In our own experimentation, we used both *Ollama* and *llama.cpp* with small-size LLMs including *llama3.2* (3 billion parameters) and *qwen2.5-coder* (3 billion parameters). In all cases, the generated code was consistent with the natural language prompts.

3.3 Tooling and Integration

Unexpected results may also be due to technical limitations. For instance, our approach may not overcome the general limits of LLMs such as context size.

It is obvious that the editing support for the `impl` construct is not yet optimal as it is represented as a plain string without syntax highlighting or validation. This can be worked around in the following way (see future work): The intermediate or final code has been generated, the developer may either implement methods manually using the full-fledged Java editor or have the LLM generate fragments using the `desc` annotation, and propagate the code back into the

model. In the latter case, this also resolves the problem of non-determinism and accelerates future generation steps by reducing the number of LLM iterations.

3.4 Domain Unspecificity

The presented approach is limited to UML-like class models, which are initially domain-unspecific and technology-neutral. Speaking in terms of *Model-Driven Architecture* (Frankel, 2003), our tooling allows to create and maintain *platform-independent* (yet programming language specific, namely Java) models, but the transfer to specific technology (e.g., a specific Java framework for web development) is not supported. While this is perceived as a common limitation of MDD approaches, LLMs provide the opportunity for manual intervention to adapt the generated code to the target platform. To this end, one additional building block could be *global prompts* attached not to operations, but to entire classes or packages, to guide the LLM in generating code that is consistent with the target platform. For instance, we could imagine a prompt like “Generate a REST API offering CRUD functionality for this domain model” to guide the LLM in generating the appropriate boilerplate code. While this is a promising idea in general, we have to keep in mind that it may negatively impact the improved *determinism*.

4 RELATED WORK

Although the incredibly fast-paced progress in LLM development has slowed down, the idea of using LLMs to aid in software development is still striking and researched manifold. Current coding benchmarks show, that models are often optimized towards a few selected popular benchmarks, intended or not (Jain, Naman et al., 2024), but fail to show their generalization abilities when tested in different benchmarks. HumanEval (Chen, Mark et al., 2021) for example shows that many models are able to achieve 85% accuracy and more, including a few small models like Qwen 2.5 Coder 7B, whereas the same models achieve less than 30% accuracy in SciCode (Tian et al., 2025) and LiveCodeBench (Jain, Naman et al., 2024), two more recent and more practically relevant coding benchmarks according to ArtificialAnalysis.ai¹⁰. For small models, the drop is especially severe. Qwen Coder 2.5 7B drops from 90% HumanEval to 9% LiveCodeBench and 14% SciCode, whereas Qwen 2.5 70B achieves only 73% in Hu-

¹⁰<https://artificialanalysis.ai/models>

manEval, but 28% and 27% in the other two benchmarks. This impression is supported by findings from Bytedance (Cheng, Yao et al., 2024) that notice that QwenCoder models perform much better on HumanEval than on their self-constructed FullStack Bench, whereas nearly all other models show a linear correlation between HumanEval performance and FullStack Bench performance.

One idea to overcome this is combining the creativity and background “knowledge” of LLMs with the formal precision of MDD code generators. This can be seen as an incarnation of the ongoing debate between symbolic and subsymbolic AI (Saba, 2023) and their compromise resolution in neuro-symbolic approaches combining formal reasoning and LLMs (Dinu et al., 2024; Mirzadeh et al., 2024). It remains unclear whether we should rather “directly go from natural language requirements to code, or is there still [...] a sweet spot for using models in such highly automated processes” (Burgueño et al., 2025).

Blinn et al. propose to use type hints and appropriate code context by using the Hazel Language Server in order to aid LLMs in correctly filling holes in the code (Blinn et al., 2024). This kind of support is similar to what IDEs provide for human programmers. Blinn et al. show that this increases model performance dramatically while reducing the inferencing time significantly, especially during error rounds.

Although general purpose LLMs are highly successful, specific LLMs for coding can be more effective (Hui, B. et al., 2024). Nevertheless, using specific system prompts targeting the various tasks in aiding software developers to create a multi-agent system can further enhance the LLM’s capabilities (Burgueño et al., 2025). Typical multi-agent setups for code generation focus on role specialization and iterative feedback loops to optimize collaboration among agents like an Orchestrator, Programmer, Reviewer, Tester, and Information Retriever (He et al., 2025). On the other hand, large reasoning models following the Quiet-Star (Zelikman et al., 2024) or DeepSeeks R1 (Guo, Daya et al., 2025) pattern are taught via reinforcement learning to look at a problem from multiple perspectives themselves without the need for different system prompts. This allows both the general LLMs to be highly successful in coding benchmarks, but is also translated to specific coding LLMs (Li, Dacheng et al., 2025). S* achieves 20% absolute increases on LiveCodeBench v2 across a wide variety of models like differently sized Qwen 2.5 Coder models, R1-distill models and GPT-4o-mini. Only the previously best model tested, o1-mini, did benefit much less with only 8.6% absolute gain.

One of the first to evaluate the use of LLMs in

model driven development are (Fill et al., 2023). They use ChatGPT to create ER, UML, BPMN and Heraklit diagrams with reasonable success. Puranik et al analyze the possibility of using NLP techniques and other AI methods to construct UML diagrams from textual descriptions of the requirements for the desired software (Puranik et al., 2024).

Sadik et al. research the other way around and use GPT-4 to generate Java and Python code to run within the JADE or PADE multi-agent system frameworks based on UML diagrams and use either OCL alone or OCL and FIPA-ontologies to constrain the generated code. They use cyclomatic complexity as a measure of quality for the generated code and find that there is an average of four bugs per class (7 classes in their use case) in the GPT-4 generated code. Most of the bugs were easily fixable missing library imports.

The closest existing publication to our endeavor is from Netz, Michael and Rumpe (Netz et al., 2024). They develop CD4A, a DSL in a Java-like syntax, and use it together with LLMs to generate running Web applications. The LLM is used to generate CD4A syntax in a few-shot approach, but GPT-4 is less successful in that than in generating PlantUML code. After that, MontiGem is used to generate the code for the Web application. The claim of a “running Web application” implies some functionality, which is not given by the automated approach. The resulting code still needs business logic implemented by human developers, but generates a runnable Web UI as needed by data-centric enterprise information systems.

5 CONCLUSION

The multi-paradigmatic approach presented in this paper integrates a triad of software development paradigms, consisting of modeling, prompting, and coding. It maximizes *determinism* (which is guaranteed by coding and modeling), *abstractness* (which are provided by both modeling and prompting), and *dynamism* (which is difficult to achieve in many static modeling approaches).

By combining the rigor of MDD with the accessibility of LLMs, we provide a powerful tool for accelerating software development while ensuring traceability. When combined with support technologies such as version control, continuous integration, and automated testing, our approach can streamline the entire software development life-cycle.

The running case study has demonstrated that it is possible to combine the strengths of structured modeling with the generative power of LLMs, while the user may locally decide for a development paradigm

(modelling, prompting, coding) suiting the current sub-problem. Our approach allows developers to experiment with different prompts and LLMs to find the best solution for their specific needs. The resulting code is consistent with the static model structure and traceable to the original prompts.

Future work is motivated by the discussion. We are planning to address the life-cycle management of the class model and its derived artifacts. As already mentioned in the discussion, code added after the initial generation, either manually or via LLM prompting, should be propagated back into the model. This would eventually allow for a round-trip engineering process, where the model is the central artifact and the code is derived from it in a preferably deterministic way while its traceability (e.g. its original prompt) is maintained. Furthermore, integrations with further AI backends in addition to *llama.cpp* and *Ollama* are desirable. Last, *global prompts* attached to packages or classes may overcome the problem of *domain unspecificity* by guiding the LLM in generating *platform-specific* target code, obviating the need of heavyweight MDD-like approaches such as domain specific languages or code generation templates.

REFERENCES

- Blinn, A., Li, X., Kim, J. H., and Omar, C. (2024). Statistically Contextualizing Large Language Models with Typed Holes. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):468–498.
- Buchmann, T., Peinl, R., and Schwägerl, F. (2024). White-box llm-supported low-code engineering: A vision and first insights. In *Proc. 27th Int'l Conf Model Driven Engineering Languages and Systems, MODELS Companion '24*, page 556–560. ACM.
- Buchmann, T. and Schwägerl, F. (2025). Bridging UML and Code in Education: A Textual Modeling Language for Teaching Object-Oriented Analysis and Design. To appear, currently under review.
- Burgueño, L., Di Ruscio, D., Sahraoui, H., and Wimmer, M. (2025). Automation in Model-Driven Engineering: A look back, and ahead. *ACM Transactions on Software Engineering and Methodology*, page 3712008.
- Chen, Mark et al. (2021). Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs].
- Cheng, Yao et al. (2024). FullStack Bench: Evaluating LLMs as Full Stack Coders. arXiv:2412.00535 [cs].
- Dinu, M., Leoveanu-Condrei, C., Holzleitner, M., Zellinger, W., and Hochreiter, S. (2024). SymbolicAI: A framework for logic-based approaches combining generative models and solvers. arXiv:2402.00854 [cs].
- Fill, H., Fettke, P., and Köpke, J. (2023). Conceptual modeling and large language models: Impressions from first experiments with chatgpt. *Enterp. Model. Inf. Syst. Archit. Int. J. Concept. Model.*, 18:3.
- Fowler, M. and Parsons, R. J. (2010). *Domain-Specific Languages*. Addison-Wesley Professional.
- Frankel, D. S. (2003). *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley Publishing, Indianapolis, IN.
- Guo, Daya et al. (2025). DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs].
- He, J., Treude, C., and Lo, D. (2025). LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision and the Road Ahead. *ACM Transactions on Software Engineering and Methodology*.
- Hui, B. et al. (2024). Qwen2.5-Coder Technical Report. arXiv:2409.12186 [cs].
- Jain, Naman et al. (2024). LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. arXiv:2403.07974 [cs].
- Li, Dacheng et al. (2025). S*: Test Time Scaling for Code Generation. arXiv:2502.14382 [cs].
- Lyu, M. R., Ray, B., Roychoudhury, A., Tan, S. H., and Thongtanunam, P. (2024). Automatic programming: Large language models and beyond. *ACM Transactions on Software Engineering and Methodology*.
- Mirzadeh, S., Alizadeh, K., Shahrokhi, H., Tuzel, O., Bengio, S., and Farajtabar, M. (2024). Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. *CoRR*, abs/2410.05229.
- Netz, L., Michael, J., and Rumpe, B. (2024). From natural language to web applications: using large language models for model-driven software engineering. In *Modellierung 2024*, pages 179–195.
- OMG (2017). *Unified Modeling Language (UML)*. Object Management Group, Needham, MA, formal/2017-12-05 edition.
- Puranik, B. S., Sonawane, A., Jose, J., Chavan, S., and Patil, Y. (2024). Enhancement of Model Driven Software Development using AI. In *4th Asian Conference on Innovation in Technology (ASIANCON)*, pages 1–5.
- Saba, W. S. (2023). Stochastic LLMs do not Understand Language: Towards Symbolic, Explainable and Ontologically Based LLMs. In *Conceptual Modeling*, volume 14320, pages 3–19. Springer Nature Switzerland.
- Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, Boston, MA, 2nd edition.
- Tian, M., Gao, L., Zhang, S., Chen, X., Fan, C., Guo, X., Haas, R., Ji, P., Krongchon, K., and Li, Y. (2025). Sci-code: A research coding benchmark curated by scientists. *Advances in Neural Information Processing Systems*, 37:30624–30650.
- Völter, M., Stahl, T., Bettin, J., Haase, A., and Helsen, S. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.
- Zelikman, E., Harik, G. R., Shao, Y., Jayasiri, V., Haber, N., and Goodman, N. (2024). Quiet-star: Language models can teach themselves to think before speaking. In *First Conference on Language Modeling*.