Towards an In-Context LLM-Based Approach for Automating the Definition of Model Views

James Pontes Miranda, Hugo Bruneliere, Massimo Tisi, Gerson Sunyé

Towards an In-Context LLM-Based Approach for Automating the Definition of Model Views

James William Pontes Miranda
Hugo Bruneliere
Massimo Tisi
IMT Atlantique, LS2N (UMR CNRS 6004)
Nantes, France
firstname.lastname@imt-atlantique.fr

Gerson Sunyé
Nantes Université, LS2N (UMR CNRS 6004)
Nantes, France
gerson.sunye@univ-nantes.fr

Abstract

In the Model-Driven Engineering (MDE) of complex systems, multiple models represent various systems' aspects. In practice, these models are often unconnected and specified using different modeling languages. Model view solutions can be employed to automatically combine such models. However, writing model view definitions is not trivial. When modeling languages are semantically distant and/or have a large number of concepts, it can quickly become difficult to manually identify the language elements to be selected, associated, or queried to build a model view. As a solution, this paper proposes an in-context Large Language Model (LLM)-based approach to assist engineers in writing model-view definitions. Notably, we rely on LLMs and Prompt Engineering techniques to automatically generate drafts of model-view definitions by providing as input only minimal information on the modeling languages to be combined. We implemented our approach by integrating the EMF Views solution for model views with the LangChain framework for LLM-based applications. To this end, we tailored LangChain to handle EMF metamodels. We validated our approach and implementation on a set of model views originally specified either in VPDL, the ViewPoint Definition Language of EMF Views, or as ATL model-to-model transformations. We compared these original model view definitions with the ones we automatically generated. The obtained results show the feasibility and applicability of our approach.

CCS Concepts: • **Software and its engineering** → *Software development techniques*; System modeling languages; • **Computing methodologies** → Learning from demonstrations; **Modeling methodologies**.

Keywords: Model-driven engineering, Modeling languages, Model views, Large language models, Prompt engineering.

SLE '24, October 20–21, 2024, Pasadena, CA, USA

ACM Reference Format:
James William Pontes Miranda, Hugo Bruneliere, Massimo Tisi, and Gerson Sunyé. 2024. Towards an In-Context LLM-Based Approach for Automating the Definition of Model Views. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE '24), October 20–21, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3687997.3695650

1 Introduction

When engineering complex systems, the separation of concerns and the fragmentation of information among stakeholders are major challenges [3, 13]. In order to mitigate these challenges, Model-Driven Engineering (MDE) promotes the use of multiple models as essential artifacts of the system engineering process. The overall objective is to support architects, engineers, and other stakeholders, in performing their activities in a more efficient way. To this end, these actors often have to combine and navigate together the various interrelated system models. These models can be defined in different modeling languages dealing with varied system aspects, for example, the Unified Modeling Language (UML) [37], System Modeling Language (SysML) [36], Business Process Modeling Notation (BPMN) [35], or various Domain-Specific Languages (DSLs) [32].

Model view solutions are suitable for combining and navigating such heterogeneous models more transparently [11]. They allow building views over one or several existing models which potentially conform to different metamodels, i. e. that are expressed in different modeling languages. There are several more or less automated ways of creating model views. They often rely on the definition of model view via a DSL and/or query language. However, when manually writing these model view definitions, it can be difficult to identify the language elements to be selected, associated, or queried. This is notably the case when the concerned modeling languages are either large or semantically distant. Thus, automatically generating model view definitions is challenging since it requires a certain level of understanding and reasoning on the input metamodels (i. e. modeling languages).

As a potential solution, different Machine Learning (ML) approaches have already been proposed to improve the support for model management operations [6, 23, 42, 46]. In particular, Large Language Models (LLMs), such as BERT [24] and GPT-3 [10], have demonstrated their capability in code generation [34, 47]. In the MDE community, LLMs have also been used for automating complex modeling tasks [17, 22] and providing recommendations [17].

In this paper, we propose an in-context LLM-based approach to assist engineers in writing model-view definitions. In particular, we automatically generate drafts of model-view definitions by providing as input only minimal information on the modeling languages to be combined. We want to achieve this by using off-the-shelf LLMs: we do not want to perform any costly additional training on the LLM, even if the LLM has not been originally trained on the model-view definition language. Thus, we query the LLM with punctual questions about the structure of the view, and we combine the LLM answers to generate the model view definition programmatically. The approach is completely in-context, i. e. it relies exclusively on Prompt Engineering (PE) techniques in order to improve reasoning capabilities [5], composability, and to enable tool-augmentation [26].

The objective of our approach is to avoid the engineers starting from scratch when developing views, by providing them with a skeleton of the specification "for free" that they can then update and refine. Moreover, they do not need to provide any training data set, as very often challenging to obtain and prepare. We developed a first implementation of our approach as an LLM-powered application enhancing the capabilities of the EMF Views model-view solution [14]. To this end, we leveraged the LangChain open-source framework for developing applications powered by LLMs [29]. We validated our approach by applying it on a selected set of model views. These model views, coming from the literature and open resources, are originally specified either in the ViewPoint Definition Language (VPDL) of EMF Views or as ATL model-to-model transformations. We evaluate the relevance of the generated model view definitions, by comparing them with the original ones. The results we obtained already show the feasibility and applicability of our approach.

This paper is structured as follows. Section 2 introduces the background of our work. Based on this, Section 3 motivates our work via a running example. Then, Section 4 presents the proposed approach, while Section 5 describes its current implementation. Section 6 explains the experiments we performed and the results of our evaluation. Finally, Section 7 discusses the related work before Section 8 concludes the paper by opening on the next steps of our work.

2 Background

This section introduces key notions of the two main areas covered by our work, namely model views and LLMs.

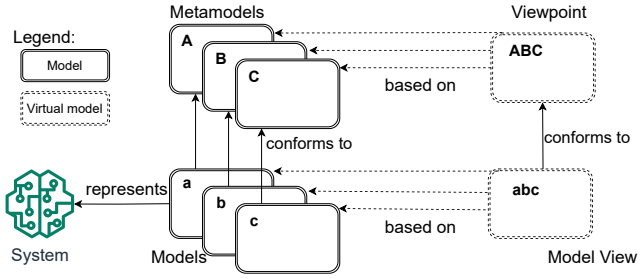

Figure 1. Main concepts of model views [33].

2.1 Model Views

As introduced in Section 1, our work currently targets the EMF Views solution [12], that uses the VPDL language to specify viewpoints and build corresponding views. In this context, Figure 1 highlights the main concepts of model views. A given system can be described by various models that potentially conform to different metamodels, i. e. expressed in different modeling languages. At the metamodel level, a viewpoint determines which concepts and properties from the contributing metamodels should be included or excluded in the corresponding views. It also expresses how these concepts should be interconnected, i. e. with which rules. At the model level, a view combines a given set of contributing models according to this viewpoint. A specificity of EMF Views is that it materializes both viewpoints and views as virtual metamodels and models, respectively. A virtual model (respectively, metamodel) only points to elements from the original models (respectively, metamodels), thus preventing unnecessary information duplication.

VPDL, provided along with EMF Views, is a textual SQL-like language for writing model view definitions. A VPDL file expresses a viewpoint, i. e. it defines the concepts and properties from the contributing metamodels that need to be selected, associated, or queried (and how). Then, EMF Views take this VPDL file as input to build corresponding model views on given sets of contributing models. An example of a VPDL file is provided later in Listing 1. Despite the relative simplicity of the textual concrete syntax, challenges arise when VPDL users do not have sufficient knowledge of the concerned modeling languages, or when these modeling languages are large and complex.

2.2 Large Language Models (LLMs)

LLMs are neural networks with the transformer architecture [44], pre-trained on massive textual content corpora and specifically tailored for text completion. Given textual inputs, i. e. *prompts*, they generate corresponding text outputs in a probabilistic manner.

2.2.1 Prompt Engineering. Prompts are the main inputs of LLMs, besides the trained weights and hyper-parameters.

PE is an essential technique to interface with LLMs by systematically designing and optimizing the inputs to guide the responses [19]. Notably, it allows orienting the LLM's behavior to achieve better results without modifying its internal weights. PE is empirical, and its efficiency can vary across different LLMs and targeted tasks. Thus, it demands various experiments and the use of well-studied heuristics [19].

With chat interfaces like ChatGPT[1], the widely used LLM of OpenAI, specific approaches are used. For example, a role is established and indicates how to act by following instructions with step-by-step explanations[2]. In this paper, we apply two PE techniques: *Few-shot Learning* [50] and *Chain-of-Thought prompting* [45]. Few-shot learning involves providing a series of high-quality demonstrations, each containing both the input and the desired output for the target task. From these examples, the obtained results are typically more accurate w.r.t. to prompts with no examples (i. e. *Zero-shot Learning*) [50]. Instead, Chain-of-thought (CoT) prompting involves creating the prompt with a series of short sentences that outline the reasoning process step-by-step [45].

2.2.2 Fine-tuning and RAG.

The performance of off-the-shelf LLMs on a given task is strongly dependent on how much the task is covered by their training dataset [18]. To extend the application of LLMs to tasks that require additional task-specific knowledge, the two most common techniques are fine tuning and retrieval-augmented generation (RAG).

Fine-tuning enhances an LLM, already pre-trained on a vast and diverse corpus of text, by additional training on new task-specific content. It refines the LLM model with specialized datasets relevant to the targeted task [40]. Retrieval Augmented Generation (RAG) enhances the standard LLM response for specific contextual data. It allows the injection of such data for the targeted task by indexing it in a vector database, and making it directly accessible by the LLM [31].

Both techniques show promising results. Still, fine-tuning demands a large dataset of examples and high computational resources [31]. While more accessible, RAG applications still need a fairly large dataset and an infrastructure for the retrieval process [27]. The availability of public dataset is a well-known problem in MDE and, especially for view definition, not many examples are publicly available. Thus, in this paper we do not use any of these techniques, and we study a solution that works directly on off-the-shelf LLMS.

2.2.3 LangChain and LLM Tool-augmentation.

LLMs already come with a rich ecosystem of frameworks such as Llamaindex[3], DSpy[4] or LangChain[5]. LangChain is an open-source framework designed to develop composable applications powered by LLMs. Its main goal is to simplify the life-cycle of LLM-based application. LangChain provides a structured approach to chaining together multiple LLM calls and managing the flow of data through various stages of processing. Technically, the framework consists of several open-source libraries that can be combined to create all the components of the LLM-powered app. This notably includes well-crafted prompts, support for different LLM models, as well as some auxiliary components like parsers and third-party integrators with external tools. Composability and tool-augmentation are important features [28, 38]. As LangChain already demonstrated its capabilities regarding them [43], we decided to use this framework in our work.

3 Running Example

This section presents our running example for the rest of the paper, a simple model view called *Book-Publication*. It comes from the EMF Views user guide[6] where it is used to explain EMF Views and VPDL. We selected this running example because its contributing metamodels (i. e. modeling languages) are very simple, but the view definition itself contains not-so-trivial associations.

Figure 2 shows the two `book` and `publication` metamodels, in graphical and textual format (in PlantUML[7]). `Books` have `titles` and `authornames` and contain `Chapters` that have their own `title` and `nbPages`. `Publications` are more general than books, and contain a `title`, an `author`, a `publisher` and a `publication year`.

```
1  create view publicationsAndBooks as
2  select publication.Publication.*,
3      book.Book.*,
4      book.Chapter.title,
5      publication.Publication join book.Chapter as
           ↪ firstChapter,
6      publication.Publication join book.Chapter as
           ↪ bookChapters
7  from 'http://publication' as publication,
8      'http://book' as book
9  where s.title = t.eContainer().title and
10     t = t.eContainer().chapters.first() for firstChapter,
11     s.title = t.eContainer().title for bookChapters
```

Listing 1. Example of a standard VPDL file.

Listing 1 shows our example view expressed in the VPDL language. The `select` part is used to define which concepts and properties from the `book` and `publication` metamodels have to appear in the view, i. e. `Publications` and `Books` with all their properties (`*`), `Chapters` with only their `title`. It also introduces new inter-model associations, i. e. the `firstChapter` and `bookChapters` relations between the `Publication` concept from the `publication` metamodel and the `Chapter` concept from the `book` metamodels. The

[1] https://chatgpt.com/
[2] https://platform.openai.com/docs/guides/prompt-engineering/
[3] https://www.llamaindex.ai/
[4] https://dspy-docs.vercel.app/
[5] https://www.langchain.com/
[6] https://www.atlanmod.org/emfviews/manual/user.html
[7] https://plantuml.com/

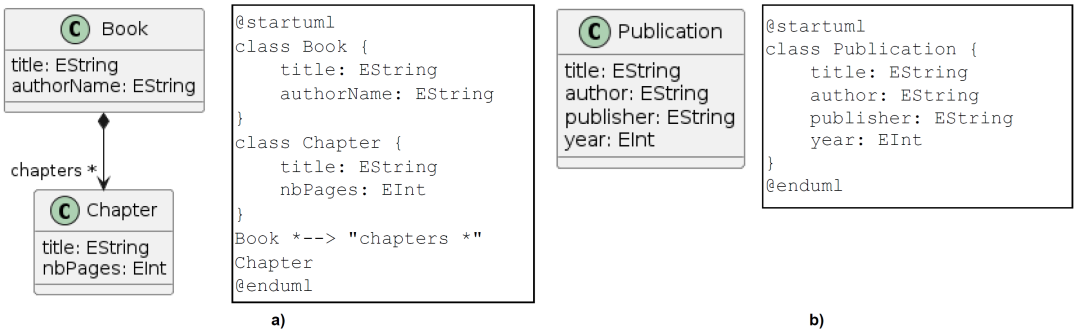

Figure 2. Running example's metamodels in graphical and PlantUML format: a) book and b) publication.

`from` part allows users to declare the contributing metamodels, i. e. `book` and `publication`. Finally, the `where` part contains OCL-like expressions specifying matching rules for the new inter-model associations, i. e. for `firstChapter` and `bookChapters`. E.g. for `firstChapter`, an OCL rule checks the inclusion of the first chapter of a book in the corresponding publication. Note that the variables `s` and `t` denote the source and target element of the association (respectively).

To summarize, a VDPL file includes the three main parts of a model-view definition:

- A list of concepts and properties to be selected from the contributing metamodels (in the `select` part).
- A list of new relations to be created between the contributing metamodels (also in the `select` part).
- Queries describing how to compute these new relations on contributing models (in the `where` part).

Our objective is to be able to automatically generate such a model view definition by providing only limited input information. To this end, Listing 2 shows a possible user prompt we could consider for our running example.

```
"""
These two metamodels represent collections of
books, but they have a different level of detail
.
While the book metamodel details each chapter,
the publication has more information about the
publisher and publishing date.
I want to select all the information regarding
publications, but I also wish to access the book
 chapters through the publication.
"""
```

Listing 2. Possible simple user prompt for generating a corresponding model view specification.

As a result, Listing 3 shows a model view definition draft, which can be generated automatically from the previous prompt by using our framework.

```
create view PublicationAndBook as
select Book.Book.*,
Book.Chapter.*,
Publication.Publication.*,
Book.Book join Publication.Publication as
    ↪ BookToPublication,
Book.Chapter join Publication.Publication as
    ↪ ChapterToPublication,

from 'http://book' as Book,
    http://publication' as Publication

where `Combine Book and Publication based on the title and
    ↪  authorName/author attributes. This allows
    ↪ linking books to their corresponding publications
    ↪ .`
     for BookToPublication
`Combine Chapter and Publication by associating chapters
    ↪ with their parent book's publication. This allows
    ↪  accessing chapters through the publication of
    ↪ the book they belong to.`
     for ChapterToPublication
```

Listing 3. Possible VPDL file resulting from the generation made with the prompt in Listing 2.

As visible in Listing 3, the approach aims at generating a draft of the `create`, `select` and `from` parts of the model view definition. Moreover, it gives a natural language description of the content of the expressions to develop in the `where` part. We want to generate a syntactically correct draft of the model view definition. We also want to guarantee correct references to classes and properties of the original metamodels.

While the generated draft may contain semantic inconsistencies, it provides a useful practical starting point for the engineer in charge of writing the model-view definition. It partially relieves the developer from identifying which classes and properties of the original metamodels should be included in the view, and which classes should be connected by inter-model associations. The queries (`where` part) are left to the developers. Still, the provided textual guidance can support them to be more efficient when writing the queries.

4 Approach

As explained in Section 2, the approach we propose relies on the composition of several LLM-based components without requiring any particular fine-tuning. Users only provide minimal information as input, e. g. the metamodels contributing to the view, to be able to automatically obtain a draft of a corresponding model view definition (cf. Section 3) that they can then update as they wish. More generally, the decomposition of the problem to be solved (i. e. model views description) into a set of sub-problems, and their resolution via an integrated chain of corresponding LLM-based components, is a key contribution of our approach that could also be extended to other modeling problems.

4.1 Overview

Figure 3 provides an overview of our proposed approach.

In EMF Views, the contributing metamodels are serialized in Ecore/XMI. However, because of their training, off-the-shelf LLMs are more efficient for human-readable textual formats. Thus, we decided to use *PlantUML class diagrams as the representation format for metamodels*. It is a popular format supposedly included in LLM training sets and whose usage for LLMs was already experimented in [22]. To this end, the first step converts *Ecore Metamodels* into equivalent PlantUML class diagrams. As an illustration, Figure 2 shows the two contributing metamodels of our running example (i. e. `book` and `publication`) converted to PlantUML.

Then, the *PlantUML metamodels* are injected, together with the user-provided prompt-like *User's View Description*, into a specific *Prompt Template* created to solve a specific part of the decomposed problem. For our model view definition problem, we considered three complementary sub-problems. These sub-problems directly correspond to the main parts of the definition as introduced in Section 3: `SELECT`, `JOIN` (associate), and `WHERE` (query). As a consequence, in our case, we need to have three different well-crafted *Prompt Templates* resulting in the LLM calling chain being executed at least three times.

For each *Prompt Template*, the performed actions are similar. First, a *Prompt* is generated from the concerned *Prompt Template*. The LLM is then directly called with this *Prompt* provided as input. As a result, it produces a corresponding textual *Raw Ouptut*. This *Raw Ouptut* is parsed to validate it and produce the textual *Parsed Output* in the expected format. This *Parsed Output* can be stored and, when required, reused as a complementary input to another iteration of the whole chain of actions. In our context, the validation process carried out by the parse operation is performed by specialized tools that deal with EMF models (e. g., PyEcore or the Java EMF API). If the output is not validated, then the LLM is asked for a new solution. This is an example of tool augmentation to enhance the output quality.

Finally, the *Parsed Outputs* resulting from the different iterations (three in our case) are combined to generate the content of the target *Model Views Specification* (definition) textual file, in the VPDL language in our case. For this final step, we do not use the LLM (since we do not assume it to be familiar with the VDPL syntax) but a standard code generator. The final resulting file is meant to be manually checked and eventually revised by the engineer before being provided as input to the Model View solution.

4.2 Focus on Prompt Templates

As presented earlier, *Prompt Templates* are key artifacts of the proposed approach and related process. Listing 4 shows one of these templates, corresponding to the JOIN (associate) iteration in our approach. This example notably illustrates in practice how such templates are structured and what kind of information they contain. Note that the two other prompts for the `SELECT` and `WHERE` (query) iterations in our approach are also provided online (cf. Section 5). Overall, they follow the same structure and organization of information.

```
"""
You are now a PlantUML analyst that find
relations between classes from two metamodels.
# TASK
Your task is to analyze the input metamodel and
the view description and define a list of
relations between the metamodels' classes.
The classes are always combined in pairs, being
one coming from the first metamodel and the
other coming from the second metamodel.
Classes can be combined when they represent the
same domain object or when they are
complementary classes, which means that one can
be extended with the attributes of the other.

Other possible reason for combination is when
the view description includes explicit
attributes from one metamodel that should appear
 in the other.

Your answer should be a valid JSON list of
dictionaries where each dictionary entry
represents a relation.
It should be a list even when it contains just
one relation.
Each relation always contains precisely one
class coming from each metamodel.
In your response, the classes are always in
order: the first class comes from the first
metamodel, and the second class comes from the
second metamodel.

# OUTPUT DATA FORMAT
{format_instructions}

# RULES
When generating the JSON response, you should
follow these rules:
```

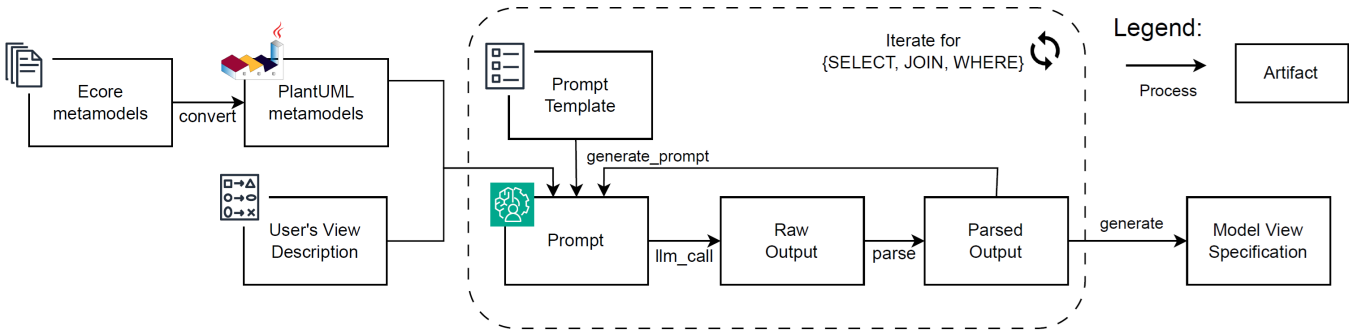

Figure 3. Overview of the proposed approach.

```
- Only use class names that exist in the metamodels. Never include classes that are not in the metamodels
- The relation's name can be any string, but it should be unique and meaningful for each relation.

# STEP BY STEP PROCESS
1. Identify all the classes from the first metamodel.
2. Identify all the classes from the second metamodel.
3. Given the metamodels and their classes, combine the elements in pairs when the selected classes represent the same domain object in each metamodel.
4. Given the metamodels and their classes, combine the elements in pairs when some selected class in the second metamodel can be complemented by some chosen class on the first metamodel and vice-versa.
5. Analyse the view description to find out other potential relations.
6. Ensure that the classes are combined in pairs, one from each metamodel.
7. Ensure that the relation's name is unique and meaningful.
8. Ensure that all the classes exist in the PlantUML metamodels.
9. Create the JSON array with the combination pairs.
10. Provide the answer.

# EXAMPLE
Given the following metamodels and view description:
View description: "The view  should conatins the name, and email from the Customer and also the name of the item bought by they."
Metamodel 1:
@startuml

    class Customer {{
        +int id
        +String name
        +String email
        +String deliveryAddress
    }}

    @enduml
Metamodel 2:
@startuml

    class Item {{
        +int id
        +String name
        +String category
    }}

    class Order {{
        +int orderId
        +String orderNumber
        +Date orderDate
        +Date creationDate
        +String currentOrderStatus
        +String customerName
    }}

    @enduml

The result Relations should be:
{{
    "relations": [
        {{
            "name": "itemBoughtByCustomer",
            "classes": [
                "Customer",
                "Item"
            ]
        }}
    ]
}}

You can think step-by-step, but your final answer should contain only the valid JSON and nothing else. Exclude any explanation or delimiter from the final response.

# INPUT
View description: {view_description}
Metamodel 1: {meta_1}
Metamodel 2: {meta_2}
"""
```

Listing 4. Python f-string used as prompt template in the JOIN step.

To design the templates and corresponding prompts, we implement the CoT approach, using few-shot examples for the format instructions (cf. Section 2). As presented in Listing 4, our templates follow a structure that contains a role definition (line 2), the task definition (line 4), the task downstream explanation (lines 5 to 9), the desired output format (line 16, to be replaced at runtime by the explanations on the expected JSON-like format), and finally, the step-by-step execution of the task (directly implementing the CoT approach).

Building a relevant prompt template is an empirical process that involves several attempts, based on try-and-error calls to the LLM until reaching the target results. However, it is possible to benefit from the Prompt Engineering best practices coming both from academia [19, 48] and LLM provider guidelines. Finally, we want to highlight the central role of the prompt templates in our approach. They are actually provided for free and immediately usable, for model views over any modeling language. As a result, engineers do not need to edit them nor have any direct interaction with the LLM if they do not want to.

5 Implementation

To validate our proposed approach, we created a prototype implementation intended to produce model-view definitions as VPDL files for EMF Views. Globally, the current implementation relies on the combination of the LangChain framework for chaining our actions, with the PyEcore library[8] to handle EMF models, and the GPT LLM models as made available through the OpenAI API.[9] The prototype source code, as well as all the related resources, are available on Zenodo[10].

Figure 4 shows an overview of the current technical implementation of our approach. To start this, the `LangChain` package displays the main composable components of the framework that are relevant in our case. A `PromptTemplate` is a runnable component that manages prompt structures, by incorporating multiple documents and variables to generate prompt content dynamically. A `LLMModel` is another runnable component that configures the properties of an LLM, including its temperature that influences the creativity and variance of the outputs. A `OutputParser` is also a runnable component that handles the parsing and validation of model outputs, thus ensuring adherence to the required formats and supporting retry strategies for self-reflection. In addition, a `Tool` is any kind of external piece of software. Finally, a `Chain` is the interface for invoking a sequence of task-specific runnable components and tools.

As described in Section 4, we created a specific prompt template for each iteration of the process we follow. These are the primary inputs for creating LangChain `PromptTemplate` instances. These instances are in charge of injecting of the format instructions in JSON Schema (the default format in LangChain), the conversion of the Ecore metamodels into their PlantUML equivalents, and the calls to the LLM. The full trace can be checked online[11].

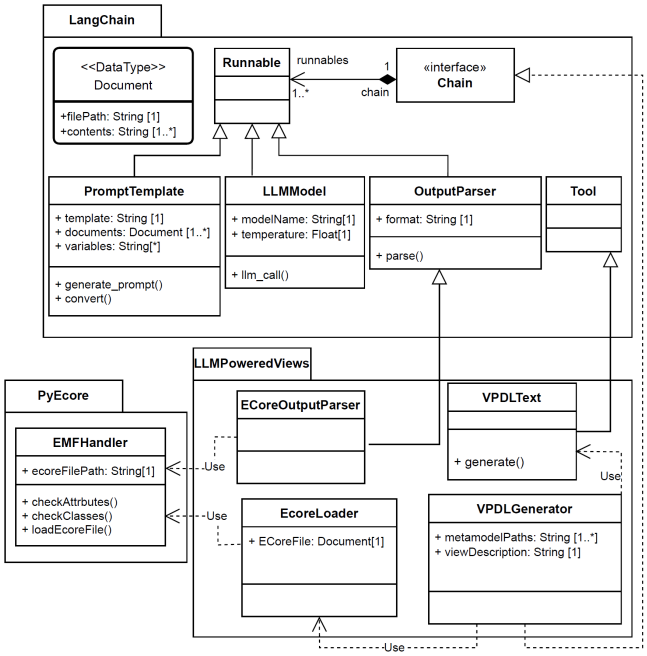

Figure 4. Overview of the technical implementation of our proposed approach.

The `LLMPoweredViews` package displays the model view-specific components we developed to refine and complement the LangChain ones. The `VPDLGenerator` is the integration `Chain` component in charge of collecting the required inputs (i. e. the metamodel paths and *View description*) and chaining all the necessary `Runnable` and `Tool` components. The `EcoreLoader` component is responsible for loading the *Ecore metamodels* to be used in *Prompt Templates*. The `EcoreOutputParser` is an `OutputParser` checking that classes and attributes returned by the LLM *Raw Outputs* are present in these input *Ecore metamodels*. In practice, it parses these outputs and repeatedly calls the LLM again until the *Parsed Outputs* are valid. Finally, `VPDLText` is a `Tool` component for generating the final target *Model View Specification*, as a VPDL textual file, from the *Parsed Outputs* produced by SELECT, JOIN, and WHERE iterations in our approach.

`EcoreLoader` and `EcoreOutputParser` use handlers from the `PyEcore` package. In terms of LLMs, we currently use GPT-4o as we generally observed better performance with it compared to GPT-3.5 and Mistral[12]. However, our approach and its implementation (thanks to LangChain) are flexible

[8] https://github.com/pyecore
[9] https://platform.openai.com/docs/models
[10] https://doi.org/10.5281/zenodo.13712247
[11] https://smith.langchain.com/public/716a3e84-d344-42e8-bf82-5b337a8b7d9b/r
[12] "mistral-large-latest" provided by Mistral: https://docs.mistral.ai/

in this respect, as they allow choosing between different OpenAI models and other LLMs.

6 Evaluation

To evaluate our implementation, we defined a dedicated benchmark using actual model-view definitions from the literature (Section 6.1) and open-source model-to-model transformations (Section 6.2). In terms of LLM parameters, we opted for a default temperature of 0 to be as close as possible to a deterministic behavior (and thus results). Still, the use of an LLM introduces inherent unpredictability due to their non-deterministic nature. This can notably impact the consistency of the generated results across different runs. In practice, the evaluation was performed using LangSmith[13], a DevOps platform dedicated to the tracing and evaluation of LLM-based applications built with LangChain.

To better assess our results, we compare them with the ones obtained via a baseline solution for LLM-based generation of model views. We simply asked ChatGPT (the ready-to-use version of the OpenAI LLM encapsulated in a chat interface) to produce the views in well-known languages (Query/View/Transformation - QVT for views, ATL for transformations). To this end, we considered 2 simple prompts tailored for ChatGPT and including the same inputs than in our experiments (cf. Listing 5 and Listing 6 respectively).

```
"""
Given the view description and the following
PlantUML metamodels, please give to me the view
definition written in QVT.

View description: {view_description}
Metamodel 1: {meta_1}
Metamodel 2: {meta_2}
"""
```

Listing 5. Simple prompt for generating views in QVT.

```
"""
Given the transformation description and the
following PlantUML metamodels, please give me
the ATL code for the transformations.

Transformation description: {
transformation_description}
Metamodel 1: {meta_1}
Metamodel 2: {meta_2}
"""
```

Listing 6. Simple prompt for generating views as ATL transformations.

Since ChatGPT is a general-purpose application trained and fine-tuned for human-machine interaction, our prompts included some conversational constructs (e. g. "please" and "give me") together with our minimal inputs. The detailed prompts and results of a simple query in the ChatGPT platform were collected in the same repository. This collection acts as an experiment journal containing the pair prompt/-completion and also a link to access the recorded chat[14]. To summarize, we compared three solutions: (i) one manually developed by authors from the related works used as a reference for the comparison, (ii) one generated by our approach, and (iii) one produced by ChatGPT as our baseline solution.

6.1 Reproducing existing model views

Table 1 shows the four model views we considered in the first part of our evaluation. They have been selected for their heterogeneity in terms of contributing modeling languages (i. e. metamodels), and for their varying levels of complexity in terms of mappings.

For each model view, Table 1 displays an identifier (ID), a high-level description, the name of the two contributing metamodels, and its source (from literature or other sources). Note that we did not write the model-view textual descriptions. Instead, we directly extracted small explanations of the desired output from the source document (e. g. research article or documentation).

Concerning model-views *3* and *4* in particular, we slightly adapted the descriptions from the sources, since these model-views initially concerned more than two contributing metamodels. These adaptations were due to the publicly available context window size of the GPT-4o LLM, which prevented us from considering numerous large metamodels within a single model view.

Overall, for model views in VPDL, the performed experiments aimed at evaluating:

- How effectively our approach automatically identifies possible relations between classes from two contributing metamodels – `JOIN` (associate).
- How accurately our approach automatically identifies relevant attributes for each class of a given relation – `SELECT`.
- Which level of quality and understandability are exhibited by the automatically generated model view definition – `SELECT`, `JOIN` (associate), `WHERE` (query).

6.2 Inferring semantic equivalence

In the second experimentation, we focus on a specific but particularly important kind of view: the views concerning two existing models (conforming to different metamodels) and connections between *semantically equivalent* elements from these models. The *SELECT* part of such a view is trivial, since we always select all classes and attributes of the corresponding metamodels. Our objective is to leverage the LLM

[13] https://www.langchain.com/langsmith

[14] The links are maintained by OpenAI. We cannot ensure for how much time they will keep it.

Table 1. Evaluated Model Views in VPDL.

ID	Model View Description	Metamodel 1	Metamodel 2	Source
V1	"The Book metamodel has details about each chapter, while the Publication has more information about the publisher and publishing date... [3 lines]"	Book	Publication	EMF Views manual
V2	"The considered view combines... an Architecture model... a Requirement model... [4 lines]"	contentfwk	ReqIF	Example view in [14]
V3	"The views allows to follow the evolution of a engineering system. It shows different versions of the same system... [3 lines]"	caex	ecoreXES	Example view in [16]
V4	"The view aggregates all the models seen so far. This allows the system engineer to transparently point to the relevant information... [11 lines]"	Traceability	B	EMF Views[15] Example

to infer the *JOIN*, i. e. the identification of the semantically equivalent classes in the two contributing metamodels.

To this end, we consider model-to-model transformations in ATL coming from existing work. Each transformation specifies how to translate models conforming to a source metamodel into models conforming to a target metamodel, by means of transformation rules. Each rule defines *correspondences* between elements of the source model (i. e. an instance of rule source pattern) and elements of the target model (i. e. an instance of the rule target pattern) considered as semantically equivalent to the source ones. We want to build a view that contains the full source model, the full target model, and inter-model association between corresponding elements in the two models (i. e., instances of source patterns and target patterns of the same rule application).

Table 2 shows the five model-to-model transformation we considered in our evaluation. They have been selected from the ATL Transformations Zoo[16] considering their diversity in terms of contributing modeling languages (i. e. metamodels) and the domains they cover. The descriptions are directly extracted from the documentation of the transformation.

6.3 Obtained Results

Table 3 and Table 4 show the quantitative results of our evaluation using a 1-shot prompt template for the model views in VPDL from Table 1. They display the detailed results for the predicted relations (`JOIN`) and the predicted attributes (`SELECT`), respectively. The **Our approach** columns indicate the means of three consecutive executions of the evaluation by using precisely the same inputs and configuration. The **ChatGPT** columns indicate the results of a single execution of our baseline solution, for comparison purposes.

Similarly, Table 6 show the corresponding results for the model views as ATL transformations from Table 2. However, as explained before, this only concerns the predicted relations (`JOIN`) in that case.

Overall, *Quantitative Evaluation* concerns the fully automated evaluation performed thanks to standard algorithms provided by the LangChain ecosystem and corresponding customized functions. *Reference* corresponds to the number of considered relations between classes and selected properties in the VPDL case, and of considered relations between classes in the ATL case. This represents our expected results. *Precision* is a standard metric providing the ratio of the number of relevant items retrieved/predicted/matched based on the total number of retrieved items. It measures the accuracy of the retrieved items. *Recall* is the ratio of the number of relevant items retrieved based on the total number of relevant items in the reference. It measures the completeness of the retrieval. Finally, *Syntactic Correctness* is the percentage of generated code that is correct from a syntactic point of view.

For the baseline solution, we considered a rough approximation since it was necessary to make assumptions. By default, the code generated by ChatGPT was not VPDL or QVT code. Instead, it used a language hallucinated by ChatGPT. Note that some results are indicated as non-available (N/A) when the generated code was almost completely irrelevant.

Qualitative Evaluation concerns the one-to-one comparison between the final outputs of our approach and the expected outputs (i. e. the reference model views). This is a manual evaluation of the overall quality of the obtained results by experts in the VPDL and ATL languages. Table 5 shows our qualitative analysis for the model views in VPDL from Table 1. *Matched Rules* indicates The overall quality (manually assessed) of the generated textual explanation for each identified relation. It can be *Good* (the engineer directly understands the semantics of the relation), *Satisfactory* (it requires her some effort), or *Inadequate* (it is very or too difficult for her) – `WHERE` (query). *Human Judge* indicates the overall quality (manually assessed) of the whole generated output, i. e. a VPDL file or a set of ATL relations. We use the same classification as from the previous metric – `SELECT`, `JOIN` (associate), `WHERE` (query). *LLM Judge* indicates the overall quality (LLM assessed) of the whole generated output, i. e. a VPDL file or a set of ATL relations. Using the same setup, the LLM receives an extra prompt to give a score from 1 to 10 concerning the generated output. 1 means that it demands a huge effort to transform the output into the reference, and 10 means that this transformation can be very easily done. Although not a standard practice yet, the use of LLM-as-judge becomes more common [51] – `SELECT`, `JOIN` (associate), `WHERE` (query).

[16] https://eclipse.dev/atl/atlTransformations/

Table 2. Evaluated Model Views as Model-to-Model Transformations in ATL.

ID	Transformation Description	Metamodel 1	Metamodel 2
T1	"The BibTeXML to DocBook example describes a transformation of a BibTeXML model to a DocBook-composed document. BibTeXML is an XML-based format... [5 lines]"	BibTeX	DocBook
T2	"The Class to Relational example describes the simplified transformation of a class model to a relational database schema. [1 line]"	Class	Relational
T3	"The "Families to Persons" transformation describes a simple model transformation example... [2 lines]"	Families	Persons
T4	"RSS is a format for syndicating news and the content of news-like sites. Atom is an XML-based file format intend... [4 lines]"	ATOM	RSS
T5	"This transformation presents a basic example where a tree is transformed into a list... [2 lines]"	List	Tree

Table 3. Quantitative evaluation - VPDL matching relations between classes using 1-shot prompt template.

ID	Reference	Our approach			Baseline solution (ChatGPT)		
		Precision	Recall	Syntactic Correctness	Precision	Recall	Syntactic Correctness
V1	2	0.50	0.50	100 %	0.00	0.00	0 %
V2	1	0.02	0.50	100 %	N/A	N/A	0 %
V3	1	0.00	0.00	100 %	0.16	1.00	0 %
V4	1	0.00	0.00	100 %	N/A	N/A	0 %

Table 4. Quantitative evaluation - VPDL matching properties using 1-shot prompt template.

ID	Reference	Our approach			Baseline solution (ChatGPT)		
		Precision	Recall	Syntactic Correctness	Precision	Recall	Syntactic Correctness
V1	8	0.58	0.58	100 %	0.66	1.00	0 %
V2	8	0.38	0.38	100 %	N/A	N/A	0 %
V3	52	0.15	0.15	100 %	0.00	0.00	0 %
V4	12	0.54	0.54	100 %	N/A	N/A	0 %

Table 5. Qualitative evaluation - Assessment for the matched rules (i. e. WHERE).

ID	Match Rules	Human judge	LLM judge
V1	Yes	Good	3
V2	Yes	Satisfactory	2.5
V3	No	Good	2
V4	No	Inadequate	2

Table 7 shows similar metrics assessing the quality of the results for the model views expressed as ATL transformations (cf. Table 2). In this ATL case, only *Human Judge* and *LLM Judge* were considered as it is not trivial to go from the list of predicted relations to the final ATL code used as reference.

6.4 Analysis of the Results

Concerning the *Quantitative Evaluation*, the *Precision* and *Recall* vary from 0.00 (no relevant prediction) to 0.58 (a decent number of relevant predictions) depending on the cases. Overall, our approach performs better when trying to predict selected properties (`SELECT`) than when trying to predict potential relations (`JOIN`). This seems logical since identifying semantic relations between concepts is a more challenging task. Still, for the relations (`JOIN`), our approach is currently more efficient in the ATL case than in the VPDL one. This could be explained by the fact that the LLM is by default more knowledgeable about the notion of model-to-model transformation (and ATL) than about the notion of model views (and VPDL). This is coherent with our choice of using an in-context approach in order to avoid having to perform a pre-training phase (for both VPDL and ATL).

When compared to the baseline solution, our approach does not currently perform systematically better in terms of precision and recall. However, it always succeeded in providing an output at least suitable in terms of syntax. This is already valuable from an user perspective, compared to the baseline solution that was sometimes unable to provide actually exploitable code. Indeed, using standard ChatGPT requires the engineer to have a solid PE expertise in order to

Table 6. Quantitative evaluation - ATL matching relations between classes using 1-shot prompt template.

ID	Reference	Our approach			Baseline solution (ChatGPT)		
		Precision	Recall	Syntactic Correctness	Precision	Recall	Syntactic Correctness
T1	16	0.05	0.08	100 %	0.11	0.12	0 %
T2	6	0.30	0.38	100 %	0.33	0.33	0 %
T3	2	0.50	1.00	100 %	0.00	0.00	0 %
T4	3	0.00	0.00	100 %	0.5	0.33	0 %
T5	2	0.00	0.00	100 %	0.5	1.00	0 %

Table 7. Qualitative evaluation - Assessment for model-to-model transformations.

ID	Human judge	LLM Judge
T1	Satisfactory	3.00
T2	Satisfactory	3.00
T3	Satisfactory	2.33
T4	Satisfactory	2.67
T5	Satisfactory	2.00

improve the results and avoid hallucinations. A main objective of our approach is to completely hide this complexity to the regular engineer.

Concerning the *Qualitative Evaluation*, the *Matched Rules* and *Human Judge* metrics reveal that a majority of the obtained outputs are actually satisfactory from an engineer perspective. While still requiring human intervention, the generated drafts of model view definitions appear to be relevant starting points. The *LLM Judge* scores, that globally range from 2 to 3, provide a complementary perspective on the possibility of transforming relatively easily the obtained outputs into the reference code. This indicates that, while not always very close to the expected output, the desired model view definitions can actually be derived by considering a reasonable number of modifications.

To summarize, results we obtained so far demonstrate that our approach already manages to fully automatically generate exploitable model view definitions. In this sense, it appears to be relevant in comparison to a baseline solution relying on standard ChatGPT. While still improveable, the obtained results show the feasibility and applicability of the proposed approach and its current implementation. The main intent of the presented work is notably to show that it is possible to automatically provide relatively relevant inputs to the engineer without any pre-training.

7 Related work

To contextualize our contributions, we discuss the related work on LLMs applied to software engineering and more particularly to MDE. We also discuss other textual model view definition approaches, with a focus on automated solutions.

7.1 LLMs for Software Engineering and MDE

LLMs have seen rapid development over recent years, evolving from simple text-to-text language problems [40] models to complex architectures capable of understanding and generating human-like text [10]. Due to their capabilities, LLMs have been mostly applied in SE to code-related tasks [7, 25]. This notably includes code generation, repair, completion, debugging, and testing. Besides code, LLMs have also been applied to deal with SE processes [4, 9] for instance.

In the context of MDE, LLMs are primarily applied to provide more advanced model recommendation and generation capabilities. For example, existing approaches intend to use LLMs in order to propose design recommendations when metamodeling in general [46] or more specifically for UML [17]. Overall, various analysis of the relevance and performance of LLMs for supporting MDE activities show promising results [20, 22].

Closer to our view definition context, there is a long history of solutions for dealing with text-to-SQL generation [30]. These notably includes the use of ML techniques such as in the TaBERT LM pre-trained on (semi-)structured tables [49] for example. This kind of approach is different from ours since we do not want to pre-train the LLM.

More similar to our approach, a more recent solution investigates the use of LLMs and PE (precisely few-shot prompting) to explore the text-to-SQL capabilities of the GPT-family models [39]. We have been inspired by such an approach that we adapted in our particular context. Also, quite recently, researchers used both a specialized LLM for code (cf. Codex) and a general purpose LLM (GPT-4) to generate Object Constraint Language (OCL) code [1, 2]. In both cases, the LLMs were capable of directly generating relevant code. Although the precise information of the datasets used for trained closed-source models like the GPT ones does not exist, a cursory search on GitHub can reveal the amount of public code available[17] Compared to our VPDL case, the existing public code base (for both OCL and SQL) is very

[17] ~6 k OCL files and ~163 k SQL files. Searched on June 2024 with the query: https://github.com/search?q=path%3A.LANGUAGE_EXTENSION+context&type=code.

significantly larger. As a result, we cannot expect an off-the-shelf LLM to generate VPDL code as it can already generate SQL or OCL (for example).

7.2 Textual Definition of Model Views

Some approaches propose to write model view definitions using textual notations. This is the case of the EMF Views solution via the VPDL language we consider in this paper [14]. However, it is important to note that EMF Views also provide two alternative ways of defining a model view: 1) programmatically via a dedicated API, 2) automatically by computing (for instance by model transformation) a weaving model containing the view specific information. Similarly to EMF Views and VPDL, the ModelJoin solution proposes a different DSL to specify model views relying on a metamodel generator and corresponding higher-order transformations [15].

Following a different approach, other researchers proposed a solution for partially automated view creation based on existing source code [8]. Contrary to our approach, this is performed without an actual model view definition and is not intended to deal with modeling languages. In the same vein, the VIATRA framework also allows to create view-like artifacts without providing an explicit textual definition [21]. However, this work is quite specific to UML models while we target different modeling languages.

To the best of our current knowledge, there is no real LLM-based solutions for model view definition in the literature. As we have seen, existing automation approaches mostly focus on particular kinds of models/artifacts rather than on the possible support for multiple modeling languages. The approach and implementation proposed in this paper intend to be a first step in this direction.

8 Conclusions and Future Work

In this paper, we presented an in-context LLM-based approach to support engineers in writing their model view definitions by providing only limited information as input. The main objective is to prevent them from starting from scratch when dealing with such a task, independently from the modeling languages contributing to the view. To achieve this, we proposed to adapt and combine state-of-the-art PE techniques in our MDE context. The current implementation of our approach notably relies on the LangChain integration framework, GPT LLM, PyEcore library, and newly defined EMF Views-specific components. We validated the proposed approach and implementation by considering different model views, specified as VPDL files or ATL model-to-model transformations. The results we obtained already demonstrate the feasibility and applicability of our approach.

However, there is still room for improvement regarding various aspects of our approach and its current implementation. Thus, we plan to explore different complementary research directions in the future.

First of all, we could start by extending the already performed evaluation. In practice, we could experiment with our approach and implementation on more examples of model views specified in VPDL, such as ATL transformations, or ideally in other ways. However, while evaluating our approach, we encountered some difficulties finding relevant model view data sets outside of the EMF Views and ATL worlds. To overcome this in the future, building a larger and more generic data set could be a useful contribution to the community (notably for benchmark purposes).

Moreover, the work on the prompt templates themselves could be continued without having to actually alter the rest of the approach and implementation. As we have been able to observe, PE is an empirical discipline that often requires many trial-and-error iterations. Thus, we may be able to further improve the current results just by modifying our prompt templates in different ways. Complementary to this, the application of other PE techniques (e. g. in addition to Chain-of-Thoughts) could be studied and then evaluated.

Another direction for potential improvement would consist in evolving the architecture of the approach itself. The current version of our approach is basically relying on a single LLM-agent. However, multi-agent collaboration is gaining significant momentum in the AI/LLM community, and is already demonstrating interesting capabilities compared to single-agent solutions [41]. As a consequence, we could envision the integration of our single approach/agent together with other more specialized agents and tools (e.g. for model querying or semantic mapping).

In addition to improving the quality of the results, a direct benefit of a multi-agent approach could be a better integration of the human in the loop if we consider it relevant in the future. Indeed, our current approach is voluntarily designed to allow engineers to provide only the initial inputs. It then works as a black box until the model view definition is generated as output. In a possible alternative version of our approach, we could 1) collect intermediate inputs from the engineers (e.g. in chat mode) and 2) consider these inputs for the different internal iterations regarding the three main parts of view. Thanks to such a more interactive approach, we may be able to obtain comprehensive model view definitions that better correspond to the engineers' wills.

Acknowledgments

This work was partially funded by the AIDOaRt European project, an ECSEL Joint Undertaking (JU) project under grant agreement No. 101007350, and by a complementary grant from the French region Pays de la Loire.

References

[1] Seif Abukhalaf, Mohammad Hamdaqa, and Foutse Khomh. 2023. On Codex Prompt Engineering for OCL Generation: An Empirical Study. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 148–157. https://doi.org/10.1109/MSR59073.2023.

00033 ISSN: 2574-3864.

[2] Seif Abukhalaf, Mohammad Hamdaqa, and Foutse Khomh. 2024. PathOCL: Path-Based Prompt Augmentation for OCL Generation with GPT-4. http://arxiv.org/abs/2405.12450 arXiv:2405.12450 [cs].

[3] Wasif Afzal, Hugo Bruneliere, Davide Di Ruscio, Andrey Sadovykh, Silvia Mazzini, Eric Cariou, Dragos Truscan, Jordi Cabot, Abel Gómez, Jesús Gorroñogoitia, Luigi Pomante, and Pavel Smrz. 2018. The MegaM@Rt2 ECSEL project: MegaModelling at Runtime – Scalable model-based framework for continuous development and runtime validation of complex systems. *Microprocessors and Microsystems* 61 (Sept. 2018), 86–95. https://doi.org/10.1016/j.micpro.2018.05.010

[4] Massoud Alibakhsh. 2023. Challenges of Integrating LLMs Like ChatGPT with Enterprise Software and Solving it with Object Messaging and Intelligent Objects as a New Software Design Paradigm. In *2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)*. 313–317. https://doi.org/10.1109/CSCE60160.2023.00054

[5] Guangsheng Bao, Hongbo Zhang, Linyi Yang, Cunxiang Wang, and Yue Zhang. 2024. LLMs with Chain-of-Thought Are Non-Causal Reasoners. https://doi.org/10.48550/arXiv.2402.16048 arXiv:2402.16048 [cs].

[6] Angela Barriga, Rogardt Heldal, Adrian Rutle, and Ludovico Iovino. 2022. PARMOREL: a framework for customizable model repair. *Software and Systems Modeling* 21, 5 (Oct. 2022), 1739–1762. https://doi.org/10.1007/s10270-022-01005-0

[7] Adna Beganovic, Muna Abu Jaber, and Ali Abd Almisreb. 2023. Methods and Applications of ChatGPT in Software Development: A Literature Review. *Southeast Europe Journal of Soft Computing* 12, 1 (May 2023), 08–12. http://scjournal.ius.edu.ba/index.php/scjournal/article/view/251 Number: 1.

[8] Artur Boronat. 2019. Code-First Model-Driven Engineering: On the Agile Adoption of MDE Tooling. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 874–886. https://doi.org/10.1109/ASE.2019.00086 ISSN: 2643-1572.

[9] Sebastian G. Bouschery, Vera Blazevic, and Frank T. Piller. 2023. Augmenting human innovation teams with artificial intelligence: Exploring transformer-based language models. *Journal of Product Innovation Management* 40, 2 (2023), 139–153. https://doi.org/10.1111/jpim.12656 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/jpim.12656.

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[11] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. 2019. A feature-based survey of model view approaches. *Software & Systems Modeling* 18, 3 (June 2019), 1931–1952. https://doi.org/10.1007/s10270-017-0622-9

[12] Hugo Bruneliere, Florent Marchand de Kerchove, Gwendal Daniel, Sina Madani, Dimitris Kolovos, and Jordi Cabot. 2020. Scalable model views over heterogeneous modeling technologies and resources. *Software and Systems Modeling* 19, 4 (2020), 827–851.

[13] Hugo Bruneliere, Vittoriano Muttillo, Romina Eramo, Luca Berardinelli, Abel Gómez, Alessandra Bagnato, Andrey Sadovykh, and Antonio Cicchetti. 2022. AIDOaRt: AI-augmented Automation for DevOps, a model-based framework for continuous development in Cyber–Physical Systems. *Microprocessors and Microsystems* 94 (Oct. 2022), 104672. https://doi.org/10.1016/j.micpro.2022.104672

[14] Hugo Bruneliere, Jokin Garcia Perez, Manuel Wimmer, and Jordi Cabot. 2015. EMF Views: A View Mechanism for Integrating Heterogeneous Models. https://doi.org/10.1007/978-3-319-25264-3_23

[15] Erik Burger, Jörg Henss, Martin Küster, Steffen Kruse, and Lucia Happe. 2016. View-based model-driven software development with ModelJoin. *Software & Systems Modeling* 15, 2 (May 2016), 473–496. https://doi.org/10.1007/s10270-014-0413-5

[16] Johan Cederbladh, Luca Berardinelli, Hugo Bruneliere, Antonio Cicchetti, Mohammadhadi Dehghani, Claudio Di Sipio, James Pontes Miranda, Abbas Rahimi, Riccardo Rubei, and Jagadish Suryadevara. 2024. Towards Automating Model-Based Systems Engineering in Industry - An Experience Report. In *The 18th Annual IEEE International Systems Conference (SYSCON 2024)*. Montreal, Canada. https://hal.science/hal-04448172

[17] Meriem Ben Chaaben, Lola Burgueño, and Houari Sahraoui. 2023. Towards using Few-Shot Prompt Learning for Automating Model Completion. In *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 7–12. https://doi.org/10.1109/ICSE-NIER58687.2023.00008 ISSN: 2832-7632.

[18] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. 2024. A Survey on Evaluation of Large Language Models. *ACM Transactions on Intelligent Systems and Technology* 15, 3 (March 2024), 39:1–39:45. https://doi.org/10.1145/3641289

[19] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. 2024. Unleashing the potential of prompt engineering: a comprehensive review. https://doi.org/10.48550/arXiv.2310.14735 arXiv:2310.14735 [cs].

[20] Kua Chen, Yujing Yang, Boqi Chen, José Antonio Hernández López, Gunter Mussbacher, and Dániel Varró. 2023. Automated Domain Modeling with Large Language Models: A Comparative Study. (July 2023). https://doi.org/10.5281/ZENODO.8118642 Publisher: Zenodo Version Number: v5.

[21] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varro. 2002. VIATRA - visual automated transformations for formal verification and validation of UML models. In *Proceedings 17th IEEE International Conference on Automated Software Engineering,*. IEEE, Edinburgh, UK, 267–270. https://doi.org/10.1109/ASE.2002.1115027

[22] Javier Cámara, Javier Troya, Lola Burgueño, and Antonio Vallecillo. 2023. On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML. *Software and Systems Modeling* 22, 3 (June 2023), 781–793. https://doi.org/10.1007/s10270-023-01105-5

[23] MohammadHadi Dehghani, Shekoufeh Kolahdouz-Rahimi, Massimo Tisi, and Dalila Tamzalit. 2022. Facilitating the migration to the microservice architecture via model-driven reverse engineering and reinforcement learning. *Software and Systems Modeling* 21, 3 (June 2022), 1115–1133. https://doi.org/10.1007/s10270-022-00977-3

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[25] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. https://doi.org/10.48550/arXiv.2310.03533 arXiv:2310.03533 [cs].

[26] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided Language Models. In *Proceedings of the 40th International Conference on Machine Learning*. PMLR, 10764–10799. https://proceedings.mlr.press/v202/gao23f.html ISSN: 2640-3498.

[27] Samira Ghodratnama and Mehrdad Zakershahrak. 2024. Adapting LLMs for Efficient, Personalized Information Retrieval: Methods

and Implications. In *Service-Oriented Computing – ICSOC 2023 Workshops*, Flavia Monti, Pierluigi Plebani, Naouel Moha, Hye-young Paik, Johanna Barzen, Gowri Ramachandran, Devis Bianchini, Damian A. Tamburri, and Massimo Mecella (Eds.). Springer Nature, Singapore, 17–26. https://doi.org/10.1007/978-981-97-0989-2_2

[28] Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Paul Saldyt, and Anil B. Murthy. 2024. Position: LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks. https://openreview.net/forum?id=Th8JPEmH4z

[29] Aarushi Kansal. 2024. LangChain: Your Swiss Army Knife. In *Building Generative AI-Powered Apps: A Hands-on Guide for Developers*. Springer, 17–40.

[30] George Katsogiannis-Meimarakis and Georgia Koutrika. 2023. A survey on deep learning approaches for text-to-SQL. *The VLDB Journal* 32, 4 (July 2023), 905–936. https://doi.org/10.1007/s00778-022-00776-8

[31] Zheng Liu, Yujia Zhou, Yutao Zhu, Jianxun Lian, Chaozhuo Li, Zhicheng Dou, Defu Lian, and Jian-Yun Nie. 2024. Information Retrieval Meets Large Language Models. In *Companion Proceedings of the ACM on Web Conference 2024*. ACM, Singapore Singapore, 1586–1589. https://doi.org/10.1145/3589335.3641299

[32] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.

[33] James Miranda, Hugo Bruneliere, Massimo Tisi, and Gerson Sunyé. 2024. Integrating the Support for Machine Learning of Inter-Model Relations in Model Views. *The Journal of Object Technology* (July 2024), 1–14.

[34] Mohamed Nejjar, Luca Zacharias, Fabian Stiehle, and Ingo Weber. 2024. LLMs for Science: Usage for Code Generation and Data Analysis. https://doi.org/10.48550/arXiv.2311.16733 arXiv:2311.16733 [cs].

[35] Object Management Group (OMG). 2024. The Business Process Model and Notation (BPMN). https://www.bpmn.org/ Last accessed 24 June 2024.

[36] Object Management Group (OMG). 2024. The Systems Modeling Language (SysML). https://sysml.org/ Last accessed 24 June 2024.

[37] Object Management Group (OMG). 2024. The Unified Modeling Language (UML). https://www.uml.org/ Last accessed 24 June 2024.

[38] Oleksiy Ostapenko, Zhan Su, Edoardo Maria Ponti, Laurent Charlin, Nicolas Le Roux, Matheus Pereira, Lucas Caccia, and Alessandro Sordoni. 2024. Towards Modular LLMs by Building and Reusing a Library of LoRAs. http://arxiv.org/abs/2405.11157 arXiv:2405.11157 [cs].

[39] Rajaswa Patil, Manasi Patwardhan, Shirish Karande, Lovekesh Vig, and Gautam Shroff. 2023. Exploring Dimensions of Generalizability and Few-shot Transfer for Text-to-SQL Semantic Parsing. In *Proceedings of The 1st Transfer Learning for Natural Language Processing Workshop*. PMLR, 103–114. https://proceedings.mlr.press/v203/patil23a.html ISSN: 2640-3498.

[40] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[41] Zachary Schillaci. 2024. LLM Adoption Trends and Associated Risks. In *Large Language Models in Cybersecurity: Threats, Exposure and Mitigation*, Andrei Kucharavy, Octave Plancherel, Valentin Mulder, Alain Mermoud, and Vincent Lenders (Eds.). Springer Nature Switzerland, Cham, 121–128. https://doi.org/10.1007/978-3-031-54827-7_13

[42] Xiangru Tang, Zhihao Wang, Jiyang Qi, and Zengyang Li. 2019. Improving code generation from descriptive text by combining deep learning and syntax rules. *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE* 2019-July (2019), 385–390. https://doi.org/10.18293/SEKE2019-170 ISBN: 1891706489.

[43] Oguzhan Topsakal and Tahir Cetin Akinci. 2023. Creating Large Language Model Applications Utilizing LangChain: A Primer on Developing LLM Apps Fast. *International Conference on Applied Engineering and Natural Sciences* 1, 1 (July 2023), 1050–1056. https://doi.org/10.59287/icaens.1127

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[45] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems* 35 (Dec. 2022), 24824–24837. https://proceedings.neurips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[46] Martin Weyssow, Houari Sahraoui, and Eugene Syriani. 2022. Recommending Metamodel Concepts during Modeling Activities with Pre-Trained Language Models. *Software and Systems Modeling* 21, 3 (June 2022), 1071–1089. https://doi.org/10.1007/s10270-022-00975-5 arXiv:2104.01642 [cs].

[47] Weizhe Xu, Mengyu Liu, Oleg Sokolsky, Insup Lee, and Fanxin Kong. 2024. LLM-enabled Cyber-Physical Systems: Survey, Research Opportunities, and Challenges. (May 2024). https://par.nsf.gov/biblio/10499418-llm-enabled-cyber-physical-systems-survey-research-opportunities-challenges Publisher: International Workshop on Foundation Models for Cyber-Physical Systems & Internet of Things (FMSys).

[48] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. *International Conference on Learning Representations (ICLR)* (Jan. 2023). https://par.nsf.gov/biblio/10451467-react-synergizing-reasoning-acting-language-models

[49] Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (Eds.). Association for Computational Linguistics, Online, 8413–8426. https://doi.org/10.18653/v1/2020.acl-main.745

[50] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate Before Use: Improving Few-shot Performance of Language Models. In *Proceedings of the 38th International Conference on Machine Learning*. PMLR, 12697–12706. https://proceedings.mlr.press/v139/zhao21c.html ISSN: 2640-3498.

[51] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. http://arxiv.org/abs/2306.05685 arXiv:2306.05685 [cs].