Combining Retrieval-Augmented Generation and Few-Shot Learning for Model Synthesis of Uncommon DSLs

Nils Baumann[1], Juan Sebastian Diaz[1], Judith Michael [1], Lukas Netz [1], Haron Nqiri[1], Jan Reimer[1], and Bernhard Rumpe [1]

Abstract: We introduce a method that empowers large language models (LLMs) to generate models for domain-specific languages (DSLs) for which the LLM has little to no training data on. Common LLMs such as GPT-4, Llama 2, or Bard are trained on publicly available data and thus have the capability to produce models for well-known modeling languages such as PlantUML, however, they perform worse on lesser-known or unpublished DSLs. Previous work focused on the usage of few-shot learning (FSL) to synthesize models but did not address or evaluate the potential of retrieval-augmented generation (RAG) to provide fitting examples for the FSL-based modeling approach. In this work, we propose a toolchain and test each building block individually: We use the MontiCore Sequence Diagram Language, which GPT-4 has minimal training data on, to assess the extent to which FSL enhances the likelihood of synthesizing an accurate model. Additionally, we evaluate how effectively RAG can identify suitable models for user requests and determine whether GPT-4 can distinguish between requests for a specific model and those for general information. We show that RAG and FSL can be used to enable simple model synthesis for uncommon DSLs, as long as there is a fitting knowledge base that can be accessed to provide the needed examples for the FSL approach.

Keywords: LLMs, RAG, DSLs, Few-Shot Learning, MDSE

1 Introduction

Tools like ChatGPT [Op] or Copilot [Mi] have enabled software engineers to easily outsource software development tasks to large language models (LLMs). Large language models such as GPT-4 [Op23] show an enormous potential to support modeling tasks [FFK23]. They are well suited to extrapolate from data they have been pre-trained on (*in-distribution-generalization*). GPT-4 can produce syntactically correct models in the PlantUML [Pl15] syntax [Cá23; Hä23], as the modeling language is part of its training data. However, if tasked with model creation for languages it was not trained on (*out-of-distribution generalization*) it will perform significantly worse or be unable to produce valid syntax [Wa23]. This poses a problem for any use cases in which an LLM should perform modeling tasks for modeling languages that are either uncommon, not published, or have been updated recently, and thus are not part of the training data of the LLM. Prompting an LLM with examples also known

1 Software Engineering, RWTH Aachen University, Aachen, Germany, nils.baumann@rwth-aachen.de; diaz@se-rwth.de; michael@se-rwth.de, https://orcid.org/0000-0002-4999-2544; netz@se-rwth.de, https://orcid.org/0000-0003-2013-2919; nqiri@se-rwth.de; jan.reimer@rwth-aachen.de; rumpe@se-rwth.de, https://orcid.org/0000-0002-2147-1966
This work is licensed under Creative Commons Attribution 4.0 International License http://creativecommons.org/licenses/by/4.0/, https://doi.org/10.18420/modellierung2024-ws-007

as *few-shot learning* (FSL) is an effective approach that has been shown to improve the capabilities of an LLM beyond the data sets it was pre-trained on [Br20].

In our approach, we concentrate on modeling tasks, which requires us to effectively teach the Large Language Model the specific grammar of the chosen modeling language. Wang et al. highlight a significant limitation of FSL: its inability to fully convey the grammar of a modeling language with a limited set of examples, especially when constrained by the context size of the LLM [Wa23]. To address this issue, we adopt a strategy of selectively choosing examples that are not only relevant to the desired modeling language but also to the specific use case. This approach helps in reducing the number and size of the examples needed within the LLM's context. Furthermore, we employ a technique known as '*retrieval-augmented generation*' (RAG) to assist the LLM in generating models in new modeling languages that it was not originally trained on [Iz22]."

Within this work, we tackle the following research question: *Can RAG and FSL support an LLM to produce models in a syntax it was not trained on?*

We propose an approach that uses retrieval augmented generation, to extract fitting models from a knowledge base and few-shot learning to create initial models for the modeler.

We introduce in Section 2 the background and present the target modeling language that we use as a running example. Next, we present an overview of our approach which is examined with two evaluations: Section 4 focuses on the improvement of modeling tasks for the given modeling language by using few-shot learning. Section 5 focuses on the capabilities of retrieval-augmented generation to extract the models needed for FSL from a knowledge base, enabling the LLM to create models for the given language. We discuss the results in Section 6. We summarize our findings in Section 7.

2 Background and Motivating DSL Example

We develop and maintain the language workbench MontiCore [HKR21]. Hence, there is a large collection of domain-specific languages that also need to be maintained and documented. Next to internal documentation, several publications provide insights into the definition of the developed languages [**GNM+20**; Ge20]. Within this paper, we take a closer look at the MontiCore Sequence Diagram Language [Ch22]. As of December 2023, the latest model of GPT-4 is unable to provide any specific Syntax and will state:
"*As of my last update in April 2023, I don't have the specific syntax details of the MontiCore Sequence Diagram Domain-Specific Language (DSL).*[...]".

Large Language Models

Recent years have seen the advent of an incredibly powerful tool in Large Language Models, based on the transformer architecture [Va17], which exploits large quantities of text to form generative models and enables a series of new problem solutions. Leading the charge is

OpenAI's GPT series, which achieved great popularity with the release of ChatGPT, whose free version is now powered by the GPT-3.5 model, with 175 billion parameters. This was soon followed by the GPT-4 model, which is rumored to have 1.5 trillion parameters and shows vastly improved results in several areas; while being roughly 10 times more expensive to use (through the OpenAI API). GPT-4's main competitors are Google's Bard (1.6 trillion parameters) and MetaAI´s LLaMA 2 (1.2 trillion parameters) which are similarly powerful, but have not seen widespread use in software engineering tasks [Ho23].

Even though they are trained on general data, LLMs exhibit surprising results on multiple tasks, particularly when prompts contain additional context that adds information for the LLM. A basic prompt is augmented with context which helps the model "understand" the problem and achieve better results. One such method is *few-shot learning* [Br20] in which a model is given some solved examples of the required task before asking it to continue with the desired question. This method requires no knowledge of the underlying architecture of the model or of machine learning in general.

LLMs exhibit certain pitfalls untrained users may fall into. Chief among them are *hallucinations*, completely made-up facts that the LLM will present as true in a confident fashion. Studies show that ChatGPT, for example, generates unverifiable information in about one in five queries [Li23]. These are often small details hidden inside otherwise true information given by the model, and so can be very hard to find, particularly if the user does not have deep knowledge of the area they have queried. Furthermore, LLMs cannot provide any justification for their answers, since they are obtained from statistical distributions. This means, in particular, they cannot source where their information comes from, making it difficult to cite or verify.

Retrieval-augmented generation [Le20] is a framework that aims to solve some of these problems by grounding the model in an external information repository. RAG works in two phases, First a retrieval phase in which algorithms search for query-relevant information in a knowledge base, which is then attached to the query. Second, in the generation phase, the LLM uses this augmented query to generate a user-readable answer with links to its sources.

MontiCore is a language workbench specialized in the engineering of modeling languages, with DSLs being a particular focus [HKR21]. It allows for an agile and modular definition of languages and language components, as well as providing tools for their analysis. It also provides tools that allow for the reuse of language features, making it an excellent tool for the building of language families [He23; JR23].

```
package correct;

sequencediagram allGrammarElements {

  // Interacting objects
  (c) o: Order;
  c: Customer;

  // Offer -> Production
  o -> c : trigger sendConfirmation();
  o <- c : return;
  o -> o : orderParts();
  assert state == Production;

  // Production -> Shipping
  o -> o : trigger shipItems();
  assert state == Shipping;

  // Shipping -> Payment
  o -> c : sendInvoice(sum);
  o <- c : return;
  assert state == payment;

  // Payment -> Complete
  assert state == complete;
  o -> Mail m = new Mail();
}
```

List. 1: Sequence diagram of a shipping process

```
package examples.correct;

sequencediagram bid {

  kupfer912:Auction;
  bidPol:BiddingPolicy;
  timePol:TimingPolicy;
  theo:Person;

  kupfer912 -> bidPol : validateBid(bid) {
    bidPol -> kupfer912 : return
         BiddingPolicy.OK;
  }
  kupfer912 -> timePol :
       newCurrentClosingTime(kupfer912,bid)
        {
    timePol -> kupfer912 : return t;
  }
  assert t.timeSec == bid.time.timeSec +
       extensionTime;
  let int m = theo.messages.size;
  kupfer912 -> theo : sendMessage(bm) {
    theo -> kupfer912 : return;
  }
  assert m + 1 == theo.messages.size;
}
```

List. 2: Sequence diagram of a bidding process

```
package examples.correct;

sequencediagram size {

  kupfer912:Auction;
  theo:Person;

  kupfer912 -> BidMessage bm = new
       BidMessage(...);
  let int m = theo.messages.size;
  kupfer912 -> theo : sendMessage(bm);
  theo -> kupfer912 : return;
  assert m + 1 == theo.messages.size;
}
```

List. 3: Sequence diagram of a bidding process

```
package examples.correct;

import very.very.deep.DeepType;

sequencediagram deepTypeUsage {

  // Am example for a very deep nested type
  d:very.very.deep.DeepType;
}
```

List. 4: MontiCore Sequence diagram

Sequence Diagram Language The MontiCore Sequence Diagram Language [HKR21] is a textual representation of the UML Sequence diagrams [UM01]. Similar to PlantUML, this DSL supports all common elements of Sequence diagrams and can be used to model all kinds of processes (cf. Figure 1). In List. 1 - List. 4 we present four MontiCore Sequence Diagrams that were used within our FSL approach. List. 1 represents a simple shop transaction in

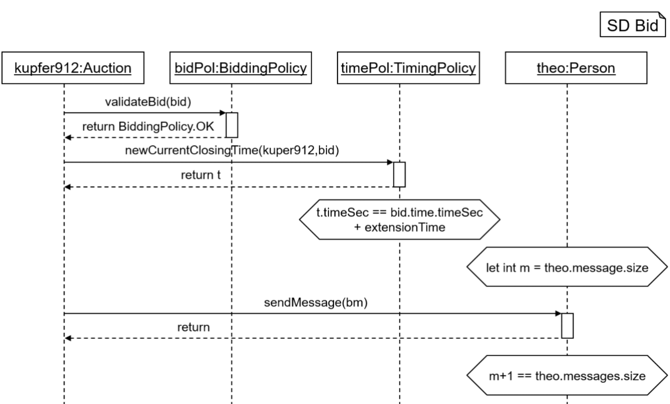

Fig. 1: Visualization of the sequence diagram presented in List. 2

which a customer orders and pays for a product. This model covers all grammar elements. List. 2 describes an auction process in a person 'theo' bids in a 'kupfer912' auction. List. 3 also relates to the auction in the previous listing, it describes a simple message transaction within the auction. The fourth model is an unusual sequence diagram as it only declares a very deep nested type. The four models were chosen as they cover a broad spectrum of grammar elements and their usages.

2.1 Related Work

LLMs in Software Engineering Recent advancements in chat-based LLM technology has made it very clear that this is a tool that will shape software engineering in the coming years. Several AI-assisted code-generation tools have been developed by the industry [Ye23], allowing for collaborative code-generation methods that show promising results [Do23]. Fully automatic code generation by users has also been studied [Fe23], as well as SE frameworks to improve performance on programming tasks [SCS23]. LLM code generation for DSLs has been also specifically targeted by using *grammar prompting* [Wa23]. In contrast to our study, these works focus on the generation of source code in well-established programming languages.

Retrieval-Augmented Generation While originally conceived simply as a way of augmenting queries with a specific domain, RAG has found diverse applications. It has been used as way to mitigate LLM hallucinations [To24], as part of a framework for debugging syntax errors [TLR23] and as a tool for generating knowledge graphs [Ji23]. Work is also being done to improve its generality via domain adaptation, which allows for open domain question-answering [Si22]. RAG has even seen use in other fields, for example for intelligent traffic monitoring [Zh23] and even writing biographies [FG22]. Unlike these works, which

mostly retrieve from textual sources, our approach uses both text documents and models of the DSL.

3 Proposed Approach

Current LLMs are well-suited to produce source code and models of known modeling languages, such as PlantUML [Cá23]. However, they perform badly or fail in producing source code or models of DSLs they have not been trained on [Fa23]. With the aim to mitigate this problem, we will, as a first study, investigate the suitability of few-shot learning to create syntactically correct DSL models in general. Then in a second study, we will employ an approach, as outlined in the following, that combines few-shot learning and RAG to retrieve examples relevant to the user query and uses them as few-shot examples for model generation. We choose to only use few-shot learning, instead of a textual syntax description of the target DSL, as this approach is directly extendable for further DSLs with more complex grammars by just adding the according few-shot examples to the database.

Initially, the knowledge database for the RAG approach is filled with models of the DSL and knowledge documents. Those need to be semantically encoded using an embedding model. In our approach, we decide on the *text-embedding-ada-002* model, as it is trained for retrieving documents matching a user request while encoding both in the same latent space. The knowledge documents contain information about certain domain-specific topics and are accordingly encoded. For the DSL models we need to adapt the embedding approach as we intend to find suited models for a natural language user request. In an attempt to create a natural language version of the models, a description is generated by a LLM which is then embedded. The quality of the description is enhanced by first asking the LLM to deduce a high-level description of the model's content, and afterward explaining the design principles that are used. This way, a broad search area is covered, enabling the user to specify a domain for the search, but also patterns. This is a semi-automated approach, as the descriptions are verified or adjusted manually before the embeddings are created. While during our study the automatically generated descriptions were sufficient, for more complex models the description might need extensive adjustment. This is a known limitation of the presented approach. Before entering the database the models undergo an automated validation process with a parser to ensure their syntactical validity. This is necessary since those models were not created for this approach, but originate from former projects and even student exercises. In the retrieval phase, the user query is augmented with the models that correspond to the descriptions with the closest embeddings. An example of a model-generated description without post-processing is shown in Figure 2.

When interacting with a LLM in a chat setting, a user may ask for various tasks throughout a conversation. In an attempt to create a dynamic interaction between user and LLM, we consider an approach that automatically predicts whether the user requests knowledge or triggers a modelling task. We create a specific prompt for this task that includes the existing LLM conversation, the user query and static few-shot examples. In this setting, the LLM

```
/* (c) https://github.com/MontiCore/monticore */

package examples.correct;

sequencediagram example {

  // Interacting objects
  o: Order;
  c: Customer;

  // Offer -> Production
  o -> c : trigger sendConfirmation();
  o <- c : return;
  o -> o : orderParts();
  assert state == Production;

  // Production -> Shipping
  o -> o : trigger shipItems();
  assert state == Shipping;

  // Shipping -> Payment
  o -> c : sendInvoice(sum);
  o <- c : return;
  assert state == payment;
  assert state == complete;
}
```

LLM

```
This sequence diagram models a process that involves an order being placed
by a customer and then going through several stages until it is completed.
The order object communicates with the customer object to trigger a
confirmation email and then orders the necessary parts for production.
After the parts are ordered, the order object triggers the shipping of the
items and then sends an invoice to the customer for payment. Once payment
is received, the order is marked as complete.

Types used in this sequence diagram include objects (Order and Customer),
messages (trigger, sendConfirmation, orderParts, shipItems, sendInvoice,
return), and assertions (state == Production, state == Shipping, state ==
payment, state == complete).

Programming concepts used in this sequence diagram include object-oriented
programming, message passing, and state management.
```

Fig. 2: Example of a MontiCore Sequence Diagram model that is converted to a natural language description using a LLM

serves as a classifier and can answer with the name of a category. If the user prompt should be augmented with few-shot examples consisting of sequence diagrams the classifying LLM should answer with 'code', if knowledge documents are to be retrieved the LLM is supposed to choose the 'documents' category. In the case that no semantic search is required to answer the user question, the category 'off' should be selected. Listing 4 shows an excerpt of the used few-shot examples along with their desired classes.

Example Workflow. A user initiates the interaction with the LLM by sending a request. The intent-classification system then determines whether the user query requires few-shot examples of sequence diagrams, knowledge documents or no additional information. In the case that information is needed, the request is then vectorized using the *text-embedding-ada-002* embedding model. This vectorized request is subsequently used to query the vector database, returning the top-k sequence diagram models or knowledge documents with respect to the cosine-similarity of the embeddings. In the case that sequence diagram models are retrieved, as the database only contains syntactically correct sequence diagrams, the user query is augmented with few-shot examples containing syntactically valid models. This augmented prompt is fed into a large language model to generate the requested model. Finally, the syntax of the produced model is validated with a parser to ensure the user is provided with a syntactically correct model.

4 Model Creation with Few-Shot Learning

In this study, we utilize OpenAI API-compliant chat inputs and multiple files containing valid sequence diagrams. These diagrams serve as examples for a LLM, which is then tasked with generating sequence diagrams for the LLM4Modelling Workshop submission process. An important parameter is the temperature, which has influence on the steepness of the probability distribution for each token generated by the LLM. A lower temperature

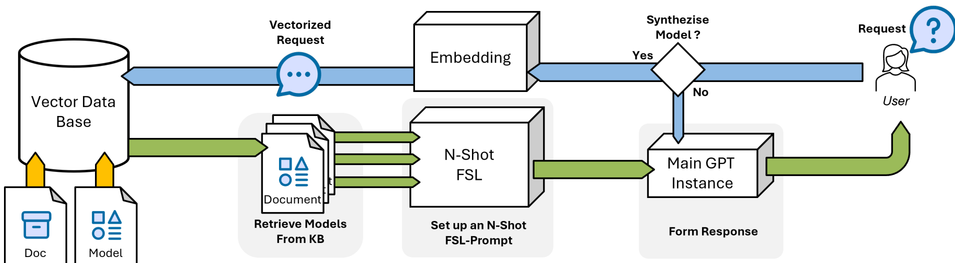

Fig. 3: Approach Description

Fig. 4: Static few-shot examples for identifying search mode

Fig. 5: Chat input in JSON format

leads to a steeper distribution and throughout the sampling to a more deterministic output. Usually, values between 0 and 1 are used, where 0 indicates that no sampling is applied (full deterministic output) and 1 produces the most diverse output over multiple runs. The artifacts are predominantly created with a temperature setting of 0.8, unless specified otherwise. Recognizing the term *Sequence Diagram* has strong semantic association with PlantUML in the LLM's training data, we refer to MontiCore Sequence Diagram Files as *sd files* to avoid misdirection. The LLM initially encountered errors, such as creating arrays of participants, which is not a part of the SD language specification. To rectify this, the LLM was explicitly instructed to avoid such constructs. This approach is indicated as bracket mitigation in our results table. The chat input as detailed in Listing 5, demonstrates a 3-shot approach with bracket mitigation. For zero-shot instances, where the LLM cannot inherently deduce the need for MontiCore-style sequence diagrams, we modified the chat input to explicitly request such diagrams. While the LLM did never produced entirely accurate models, almost all produced artifacts demonstrated close approximations. For GPT-3, it is observed that using more samples does not automatically guarantee better performance. While a 3-shot tends to lead to better results, a 4-shot setup on GPT-3.5 yields worse results. This is a repeatable observation in which, the LLM struggles to incorporate the whole context into the generation, due to its limitation in processable tokens. Other smaller LLMs exhibit worse similar issues, struggling to fully attend to large contexts, even though these contexts technically fit within the context size limitations. This hypothesis be can checked by reducing the context size of the larger models and comparing the results, as possible further work. GPT-4 does not necessarily produce more correct models with an increasing number of samples and, in some cases (3-Shot, 3-Shot (temp 0.2)) is outperformed by the smaller GPT-3.5. However, a qualitative difference is observed. While GPT-3 produces mostly correct PlantUML artifacts erroneously, GPT-4's incorrect artifacts are almost correct MontiCore sequence diagrams. The issue with GPT-4 is its tendency to invent language features that, while arguably sensible for a sequence diagram DSL, differ from or are not

implemented in the MontiCore version. In Listing 6, we clearly see output resembling more of a sequence diagram language extension than incorrect code. By instructing GPT-4 to avoid brackets and keep the code simple, it reduccontext lengthes the hallucination of new language features. Although this approach does not completely eliminate errors, with two small samples, GPT-4 outputs correct code 88% of the time. Making GPT-4 highly practical for this use case.

```
chair -> easyChair : assignReviewers(paper, [reviewer1, reviewer2, reviewer3]);
parallel {
  easyChair -> reviewer1 : requestReview(paper);
  easyChair -> reviewer2 : requestReview(paper);
  easyChair -> reviewer3 : requestReview(paper);
}
```

Fig. 6: Excerpt Illustrating Hallucinated Language Features: Arrays and Parallel Processing

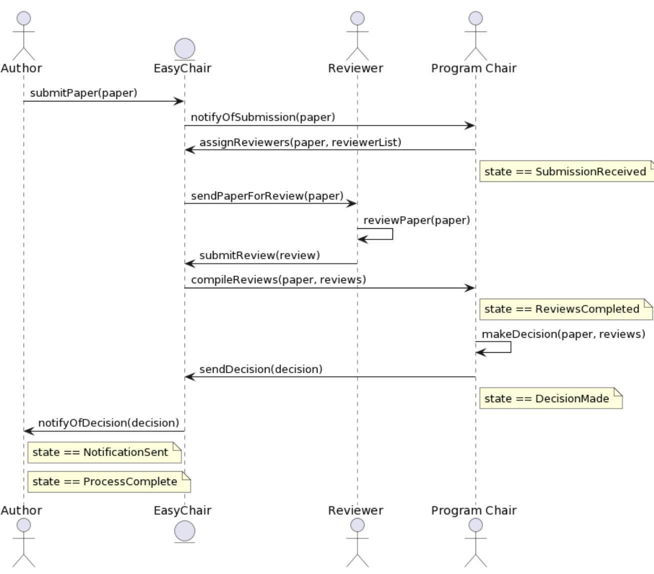

Fig. 7: LLM-generated artifact illustrating the sequence diagram for the submission process of the LLM4Modelling Conference.

5 Retrieval Augmented Generation for Models

Experimental Setup. As an initial step, we populated the knowledge base for the RAG approach with 19 MontiCore sequence diagrams. Initially, we verified the syntactical correctness of each diagram, with all examples passing the correctness test. In a next step, we generated descriptions for these diagrams, followed by a post-processing step where we manually fine-tuned the generated descriptions. While generating the natural language descriptions for the sequence diagrams, we observed that the LLM often generated a fitting

Test Run Name	Model	Samples	Error Percentage	Total Artifacts
0-Shot	GPT-3.5	N/A	100	100
1-Shot	GPT-3.5	Listing 1	65	100
2-Shot	GPT-3.5	Listing 1,2	71	100
3-Shot	GPT-3.5	Listing 1,2,3	44	100
4-Shot	GPT-3.5	Listing 1,2,3,4	85	100
0-Shot	GPT-4-1106	N/A	100	100
1-Shot	GPT-4-1106	Listing 1	63	100
2-Shot	GPT-4-1106	Listing 1,2	66	100
3-Shot	GPT-4-1106	Listing 1,2,3	66	100
3-Shot (temp 0.2)	GPT-4-1106	Listing 1,2,3	73	100
1-Shot (bracket mitigation)	GPT-4-1106	Listing 1	45	100
2-Shot (bracket mitigation)	GPT-4-1106	Listing 1,2	28	100
3-Shot (bracket mitigation)	GPT-4-1106	Listing 1,2,3	27	100

Tab. 1: Percentage of syntactically invalid models produced with few-shot learning.

description for the diagram, but also mentioned implementation details of the diagram. We then removed those details in the fine-tuning step. Finally, we embedded the fine-tuned descriptions, storing the original sequence diagrams and their corresponding description embeddings in a vector database. Additionally, we added the description in a metadata field of the database.

In a second phase we constructed the experiment for classifying user intent. For each of the three categories 'code', 'documents' and 'off' we curated two prompts and labeled them with the correct category. As an example, Listing 8 shows the used prompts for the 'code' category. As a LLM we used GPT-3 with a temperature setting of 0.8. For the evaluation we classified each of the prompts 100 times using the LLM and compared the answer to prompt label. For each category we computed the error percentage, indicating how often the LLM selects the wrong category.

Separately, we setup an evaluation only focusing on the performance of DSL model generation using the RAG approach to retrieve few-shot examples. We consider both GPT-3 and GPT-4 as LLMs with a temperature setting of 0.8. As a prompt template we used the template from Figure 5, same as in the previous study. We adjusted the number of models the RAG should retrieve and experimented with values from 1 to 4 example models. For each combination, we generated 100 LLM outputs resulting in 800 total artifacts. For each of the outputs we tested if the LLM generated a syntactically correct MontiCore sequence diagram.

Fig. 8: Prompts that require code examples

Experimental Results. The results for the user intent classification are depicted in Table 2 and show very low error percentages when the user query requires few-shot examples or knowledge documents. On the other side, if no semantic search is necessary we could observe

an error percentage of 29%, showing a weakness of this approach. In our experiments we could observe, when documents or few-shot examples are added when they are not needed, the LLM would then often reference the added information instead of answering the user question to the best of its abilities. However, for the other categories the LLM seems to be well suited and can be used to dynamically detect the user intend. It has to be noted, while 200 artifact were created for each category, they only consist of two different prompts indicating that further experiments are required to verify the results of this approach.

Category	Model	#Different Prompts	Error Percentage	Total Artifacts
code	GPT-3.5	2	0	200
documents	GPT-3.5	2	0	200
off	GPT-3.5	2	29	200

Tab. 2: Percentage of misclassified user queries within each category.

Results of the RAG approach for creating syntactically correct MontiCore sequence diagram models are depicted in Table 3. We can see a very high error percentage using GPT-3 for experiments that only retrieve 1 to 3 examples. The LLM even failed to produce a single syntactically correct sequence diagram using only 1 or 2 examples. This can be a result of the retrieved examples or the capabilities of the LLM. These results are also a lot worse than the few-shot experiments with manually curated examples. However, we can see a large performance jump when considering the 4-shot experiment. This can be the result of a very representative example added to the user prompt.

In comparison, the results of GPT-4 are a lot better than the GPT-3 results. Already exhibiting a decent error percentage at only one retrieved example. Another interesting observation is that while the GPT-3 1-shot results are a lot worse compared to the previous study, the same experiment with GPT-4 exhibits better results. This can indicate that GPT-4 can make better use of the example, while the retrieved example does not seem to be suited for GPT-3. For all GPT-4 experiments we can observe that the error percentage is lower compared to the same experiments (without bracket mitigation) in the previous study.

Test Run Name	Model	Error Percentage	Total Artifacts
1-Shot	GPT-3.5	100	100
2-Shot	GPT-3.5	100	100
3-Shot	GPT-3.5	96	100
4-Shot	GPT-3.5	73	100
1-Shot	GPT-4-1106	40	100
2-Shot	GPT-4-1106	49	100
3-Shot	GPT-4-1106	42	100
4-Shot	GPT-4-1106	29	100

Tab. 3: Percentage of syntactically invalid models produced with the RAG approach.

6 Discussion

We demonstrated that our method allows a LLMs to create models in a specific modeling language, even when it has minimal or no prior training data in that language. However, there are several challenges associated with this approach: PlantUML poses a challenge in generating less common DSLs, as its widespread use in UML modeling contributes significantly to the training data in the semantic cluster of UML-like code. Avoiding unwanted PlantUML output is feasible with specialized prompting, offering a low-effort, partially effective mitigation. However, in the long term, developers and users need alternative solutions to address the limitations imposed by working around PlantUML misgenerations. An intriguing approach for the Monticore DSL family, in particular, is Grammar Inference, which involves constraining the output of the LLM with a context-free grammar. Tools [BFV23] exist that support this functionality and enable other constraints, such as datatype or regex constraints on output. This effectively restricts the output to syntactically correct forms. Another approach is fine-tuning, which necessitates extensive datasets of high-quality data, typically created by experts. Such datasets are not easily accessible for lesser-known DSLs and can be expensive to compile. Nevertheless, the use of parameter-efficient fine-tuning methods can alleviate training costs. In our evaluations, we focused on one modeling language. Therefore, it is difficult to generalize and make statements about the performance of this approach to an arbitrary language. General statements about languages that have little in common with UML require further testing.

7 Conclusion

Our research demonstrated that by combining few-shot Learning and retrieval-augmented Generation, we can equip Large Language Models to handle modeling tasks in unfamiliar modeling languages. We tested this method by successfully creating models for a domain-specific language used in sequence diagram descriptions. This suggests that our approach could assist developers working with unpublished or less common DSLs. However, to establish the broader applicability of this toolchain, we need to conduct further tests with a more diverse range of DSLs

References

[BFV23] Beurer-Kellner, L.; Fischer, M.; Vechev, M.: Prompting Is Programming: A Query Language for Large Language Models. Proceedings of the ACM on Programming Languages 7/PLDI, pp. 1946–1969, 2023.

[Br20] Brown, T. B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; Agarwal, S.; Herbert-Voss, A.; Krueger, G.; Henighan, T.; Child, R.; Ramesh, A.; Ziegler, D. M.; Wu, J.; Winter, C.; Hesse, C.; Chen, M.; Sigler, E.; Litwin, M.; Gray, S.; Chess, B.; Clark, J.; Berner, C.; McCandlish, S.; Radford, A.; Sutskever, I.; Amodei, D.: Language Models are Few-Shot Learners./, arXiv:2005.14165, 2020, arXiv: 2005.14165 [cs.CL].

[Cá23] Cámara, J.; Troya, J.; Burgueño, L.; Vallecillo, A.: On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML. Software and Systems Modeling/, pp. 1–13, 2023.

[Ch22] Chair for Software Engineering, RWTH: Sequence Diagram - MontiCore Repository, https://github.com/MontiCore/sequence-diagram, Accessed: 2024-01-10, 2022.

[Do23] Dong, Y.; Jiang, X.; Jin, Z.; Li, G.: Self-collaboration Code Generation via ChatGPT, 2023, arXiv: 2304.07590 [cs.SE].

[Fa23] Fan, A.; Gokkaya, B.; Harman, M.; Lyubarskiy, M.; Sengupta, S.; Yoo, S.; Zhang, J. M.: Large language models for software engineering: Survey and open problems. arXiv preprint arXiv:2310.03533/, 2023.

[Fe23] Feng, Y.; Vanam, S.; Cherukupally, M.; Zheng, W.; Qiu, M.; Chen, H.: Investigating Code Generation Performance of ChatGPT with Crowdsourcing Social Data. In: 2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC). Pp. 876–885, 2023.

[FFK23] Fill, H.-G.; Fettke, P.; Köpke, J.: Conceptual modeling and large language models: impressions from first experiments with ChatGPT. Enterprise Modelling and Information Systems Architectures 18/, pp. 1–15, 2023.

[FG22] Fan, A.; Gardent, C.: Generating Full Length Wikipedia Biographies: The Impact of Gender Bias on the Retrieval-Based Generation of Women Biographies, 2022, arXiv: 2204.05879 [cs.CL].

[Ge20] Gerasimov, A.; Heuser, P.; Ketteniß, H.; Letmathe, P.; Michael, J.; Netz, L.; Rumpe, B.; Varga, S.: Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In (Michael, J.; Bork, D., eds.): Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers. CEUR Workshop Proceedings, Vienna, pp. 22–30, 2020, URL: http://www.se-rwth.de/publications/Generated-Enterprise-Information-Systems-MDSE-for-Maintainable-Co-Development-of-Frontend-and-Backend.pdf.

[Hä23] Härer, F.: Conceptual model interpreter for Large Language Models. arXiv preprint arXiv:2311.07605/, 2023.

[He23] Heithoff, M.; Jansen, N.; Kirchhof, J. C.; Michael, J.; Rademacher, F.; Rumpe, B.: Deriving Integrated Multi-Viewpoint Modeling Languages from Heterogeneous Modeling Languages: An Experience Report. In: 16th ACM SIGPLAN International Conference on Software Language Engineering. SLE 2023, ACM, pp. 194–207, 2023.

[HKR21] Hölldobler, K.; Kautz, O.; Rumpe, B.: MontiCore Language Workbench and Library Handbook: Edition 2021. Shaker Verlag, 2021, ISBN: 978-3-8440-8010-0.

[Ho23] Hou, X.; Zhao, Y.; Liu, Y.; Yang, Z.; Wang, K.; Li, L.; Luo, X.; Lo, D.; Grundy, J.; Wang, H.: Large Language Models for Software Engineering: A Systematic Literature Review, 2023, arXiv: 2308.10620 [cs.SE].

[Iz22] Izacard, G.; Lewis, P.; Lomeli, M.; Hosseini, L.; Petroni, F.; Schick, T.; Dwivedi-Yu, J.; Joulin, A.; Riedel, S.; Grave, E.: Few-shot learning with retrieval augmented language models. arXiv preprint arXiv:2208.03299/, 2022.

[Ji23] Jiang, X.; Zhang, R.; Xu, Y.; Qiu, R.; Fang, Y.; Wang, Z.; Tang, J.; Ding, H.; Chu, X.; Zhao, J.; Wang, Y.: Think and Retrieval: A Hypothesis Knowledge Graph Enhanced Medical Large Language Models, 2023, arXiv: 2312.15883 [cs.CL].

[JR23] Jansen, N.; Rumpe, B.: Seamless Code Generator Synchronization in the Composition of Heterogeneous Modeling Languages. In: 16th ACM SIGPLAN International Conference on Software Language Engineering. SLE 2023, ACM, pp. 163–168, 2023.

[Le20] Lewis, P.; Perez, E.; Piktus, A.; Petroni, F.; Karpukhin, V.; Goyal, N.; Küttler, H.; Lewis, M.; Yih, W.-t.; Rocktäschel, T.; Riedel, S.; Kiela, D.: Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks./, arXiv:2005.11401, 2020, arXiv: 2005.11401 [cs.CL].

[Li23] Li, J.; Cheng, X.; Zhao, X.; Nie, J.-Y.; Wen, J.-R.: HaluEval: A Large-Scale Hallucination Evaluation Benchmark for Large Language Models. In (Bouamor, H.; Pino, J.; Bali, K., eds.): Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. ACL, 2023.

[Mi] Microsoft Corporation: Bing, https://www.bing.com, Accessed: 2024-01-10.

[Op] OpenAI: OpenAI Chat, https://chat.openai.com, Accessed: 2024-01-10.

[Op23] OpenAI et al.: GPT-4 Technical Report, 2023, arXiv: 2303.08774 [cs.CL].

[Pl15] PlantUML: PlantUML: open-source tool that uses simple textual descriptions to draw UML diagrams./, 2015.

[SCS23] Siddiq, M. L.; Casey, B.; Santos, J. C. S.: A Lightweight Framework for High-Quality Code Generation, 2023, arXiv: 2307.08220 [cs.SE].

[Si22] Siriwardhana, S.; Weerasekera, R.; Wen, E.; Kaluarachchi, T.; Rana, R.; Nanayakkara, S.: Improving the Domain Adaptation of Retrieval Augmented Generation (RAG) Models for Open Domain Question Answering, 2022, arXiv: 2210.02627 [cs.CL].

[TLR23] Tsai, Y.; Liu, M.; Ren, H.: RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Models, 2023, arXiv: 2311.16543 [cs.AR].

[To24] Tonmoy, S. M. T. I.; Zaman, S. M. M.; Jain, V.; Rani, A.; Rawte, V.; Chadha, A.; Das, A.: A Comprehensive Survey of Hallucination Mitigation Techniques in Large Language Models, 2024, arXiv: 2401.01313 [cs.CL].

[UM01] UML, O.: Unified modeling language. Object Management Group 105/, 2001.

[Va17] Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; Polosukhin, I.: Attention Is All You Need./, arXiv:1706.03762, 2017, arXiv: 1706.03762 [cs.CL].

[Wa23] Wang, B.; Wang, Z.; Wang, X.; Cao, Y.; Saurous, R. A.; Kim, Y.: Grammar Prompting for Domain-Specific Language Generation with Large Language Models, 2023, arXiv: 2305.19234 [cs.CL].

[Ye23] Yetiştiren, B.; Özsoy, I.; Ayerdem, M.; Tüzün, E.: Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT, 2023, arXiv: 2304.10778 [cs.SE].

[Zh23] Zhang, J.; Ilievski, F.; Ma, K.; Kollaa, A.; Francis, J.; Oltramari, A.: A Study of Situational Reasoning for Traffic Understanding. In: 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. KDD '23, ACM, 2023.