

# Conceptual model interpreter for Large Language Models

Felix Härer<sup>1</sup>

<sup>1</sup>University of Fribourg, Boulevard de Pérolles 90, 1700 Fribourg, Switzerland

## Abstract

Large Language Models (LLMs) recently demonstrated capabilities for generating source code in common programming languages. Additionally, commercial products such as ChatGPT 4 started to provide code interpreters, allowing for the automatic execution of generated code fragments, instant feedback, and the possibility to develop and refine in a conversational fashion. With an exploratory research approach, this paper applies code generation and interpretation to conceptual models. The concept and prototype of a conceptual model interpreter is explored, capable of rendering visual models generated in textual syntax by state-of-the-art LLMs such as Llama 2 and ChatGPT 4. In particular, these LLMs can generate textual syntax for the PlantUML and Graphviz modeling software that is automatically rendered within a conversational user interface. The first result is an architecture describing the components necessary to interact with interpreters and LLMs through APIs or locally, providing support for many commercial and open source LLMs and interpreters. Secondly, experimental results for models generated with ChatGPT 4 and Llama 2 are discussed in two cases covering UML and, on an instance level, graphs created from custom data. The results indicate the possibility of modeling iteratively in a conversational fashion.

## Keywords

Large Language Model, Conceptual Model, Code Generation, Interpreter

## 1. Introduction

In recent years, Large Language Models (LLMs) have seen broad adoption due to the wide variety of successful applications utilizing transformers [1], enabling language-related tasks at higher abstraction levels, e.g., when detecting and describing images, performing audio tasks such as speech-to-text and voice cloning, or handling text for modeling topics, summarization, or translation [2]. Especially in the form of Generative Pre-trained Transformer (GPT) models, language analysis and generation capabilities evolved and today include programming languages [3, 4] in addition to first indications of support for domain-specific languages and conceptual modeling [5]. After the large-scale adoption of ChatGPT 4, gaining over 100 million active users within two months after its initial release in November 2022 [6], commercially available products such as ChatGPT 4 and the GitHub Copilot started to introduce further features for software development in 2023 [7]. Notably, the conversational generation of source code is now complemented by the ability to invoke external APIs using Plugins [8] and the

---

*ER2023: Companion Proceedings of the 42nd International Conference on Conceptual Modeling: ER Forum, 7th SCME, Project Exhibitions, Posters and Demos, and Doctoral Consortium, November 06-09, 2023, Lisbon, Portugal*


✉ [felix.haerer@unifr.ch](mailto:felix.haerer@unifr.ch) (F. Härer)

🌐 <https://www.unifr.ch/inf/digits/en/group/team/haerer.html> (F. Härer)

🆔 0000-0002-2768-2342 (F. Härer)



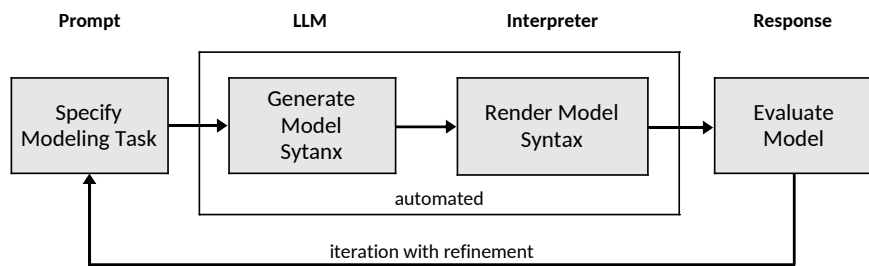
© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

execution of generated source code with a code interpreter [9]. In particular, OpenAI introduced a code interpreter that allows a user in a conversation to describe or specify a task for programming, automatically generating the source code, and automatically executing it instantly to provide output and computation results as a response. Examples demonstrated capabilities to interpret code fragments, e.g., calculating functions or creating plots in Python. The conversational approach also provides the potential to include code generation with instant feedback and step-wise refinement into the software development process, allowing for the iterative development of code fragments with instant feedback on execution results.

In this paper, the conversational approach with instant feedback and step-wise refinement is applied to conceptual models. For exploring the potential of this approach, the research objectives are (1.) to determine whether the approach can be realized with state-of-the-art commercial or open source LLMs and interpreters, and (2.) how a possible realization could be constructed in terms of an architecture. By exploratory research, these objectives are investigated through the construction of a prototype chat application.

Potentially, conceptual models may be created within a dialogue where a modeling task is specified fully or partially in natural language, textual syntax for models is then generated by the LLM and automatically rendered visually by interpreters, e.g., for PlantUML or Graphviz syntax. In addition, the interpreter is a first step towards specifying executable models for generating software through LLMs that can be executed directly in local environments or at the server side, e.g., generating step function models for cloud platforms and blockchains [10].



**Figure 1:** Iterative modeling process from a user perspective, generating and rendering the concrete syntax of a model in a conversational fashion.

Modeling with instant feedback allows for an iterative process with refinement as described in Figure 1. Such a process could be applied generally in many areas, e.g., in requirements and software engineering, where the generation of structure or behavior diagrams is above the level of source code. Further, it lowers the barrier to entry for modeling and design activities. While LLMs also lower the barrier to generating source code directly, the difficulty oftentimes lies in its evaluation; that is, to evaluate whether source code and the implied software design are suitable and fulfill the requirements.

The remainder of this paper is structured as follows. Section 2 introduces background on LLMs and related work on applying LLMs for conceptual modeling. In Section 3, the concept of a model interpreter and a possible realization architecture are outlined. Section 4 discusses experimental results of the prototype with prompts and generated models for ChatGPT 4 and Llama 2. Section 5 outlines the overall results and concludes.

## 2. Background and related work

### 2.1. Background

Generative Pre-trained Transformer models (GPT) are a class of Large Language Models (LLMs) that apply a specific architecture on the principle of predicting or completing text, progressing on a sequence of tokens.

An input sequence is given by a prompt and the context of a dialogue preceding it, where words are split into tokens, encoded, and represented by a vector according to an embedding in a high-dimensional vector space in the thousands of dimensions [3, 11, 1]. With an additional encoding of the position in the sequence, each token is applied to a transformer architecture to generate candidates for the next token with a probability distribution. Tokens pass sequentially through multiple transformer layers of transformer blocks, each consisting of attention and feed-forward network components. Each transformer block produces tokens and probabilities with increasing abstraction. Within a block, attention directs the focus to select tokens seen before within a similar context, based on similar attention at prior positions in the sequence. Attention mechanisms achieve this by calculating attention at a given position relative to other positions in the sequence [1], instead of relying only on co-occurrences. The output of a transformer block with tokens and a probability distribution is dependent on the weights of the feed-forward network and normalization. After passing all the layered transformer blocks, 96 in the case of GPT-3 [12], the next token is ultimately selected according to probability with a sampling algorithm [13] and appended to the sequence. Subsequently, the iterative token generation produces the response and becomes part of the context.

LLMs notably differ in terms of their supported context size, allowing for 4096 tokens in Llama 2 [14], 8192 in the standard version of GPT-4, and 32768 in its extended version<sup>1</sup>. Substantial differences also exist in the number of parameters applied by the training data with 7, 13, 34, or 70 billion parameters for Llama 2 variants [14], 175 billion in GPT-3.5 [3], and possibly trillions of parameters in GPT-4 [3]. Exact specifications, also concerning the transformer architecture, are unknown for the closed-source GPT-4 model. Further optimizations are applied for LLMs used in a conversational fashion, e.g. GPT-4 and Llama 2, to adapt responses to the style of conversational dialogues and to emphasize or suppress specific context by applying Reinforcement Learning from Human Feedback (RLHF) and fine-tuning [3].

### 2.2. Related work

While prior work on LLMs and modeling applications is scarce, first publications do exist for conceptual modeling, business process management, and software modeling, in addition to related findings for programming languages.

For conceptual modeling, the application of ChatGPT has been investigated before in a publication by Fill et al. [5] which suggested the generation of UML class diagrams in PlantUML syntax as well as creating ER, workflow, and Heraklit models in a custom syntax. Results further demonstrate the capability of GPT-4 to generate new models beyond examples potentially existing in training data, by issuing prompts for textbook-like case descriptions and abstract

---

<sup>1</sup><https://platform.openai.com/docs/models>

examples. This paper is inspired by this investigation and seeks to integrate multiple LLMs and interpreters in a more generalized way to explore their modeling capabilities towards further systematic evaluations in the future.

For Business Process Management (BPM), Vidgof et al. [15] note opportunities and challenges along the BPM lifecycle. In relation to modeling, discovery is a major topic for discovering process models through process mining. While LLMs can support discovery generally, e.g. from documentation, the generation of BPMN in XML format is noted as well as parsing existing XML process models with a LLM, opening the potential to query the LLM for knowledge on the syntax, the semantics of the process, and potential execution behavior. Regarding process implementation, BPMN models might be augmented with plain text, and accessed in a user-specific way through chatbots.

For software modeling, Cámara et al. [16] provide an experience report on using ChatGPT for UML modeling tasks in different notations. ChatGPT produced diagrammatic notation using characters, was found to support the PlantUML and USE notations “generally well”, and could also generate OCL expressions. For the test cases, ChatGPT seemed to handle the creation of classes with attributes in addition to associations, aggregations and compositions, simple inheritance and role names of association ends. Certain elements required explicit indication, e.g., enumerations, and results using abstracts and association classes were not acceptable. Further findings indicate results were generally correct, could exhibit small syntactic errors, had high variability for test cases and randomness over time, and depended on the domain understanding of ChatGPT. The paper also notes size limitations of about 8 to 10 classes given a single prompt and the possibility to construct larger models iteratively in further prompts.

Further limitations regarding syntax and semantics aspects were investigated for programming languages in a systematic study by Ma et al. [4]. It finds ChatGPT generally “excels at understanding code syntax (AST)” while struggling with semantics. In particular, it finds ChatGPT generally understands syntax, is capable of inference based on an abstract syntax tree (AST), and can perform static analysis. However, the understanding of semantics and dynamic behavior was found limited. For static analysis and semantic aspects, ChatGPT also seemed prone to hallucination, i.e., generating non-existent facts, in some cases.

### **3. Conceptual model interpreter**

This section outlines the overall concept together with requirements, leading to the discussion of the architecture for describing the system structure.

#### **3.1. Concept and requirements**

The concept of a model interpreter is first explored from a requirements perspective. This discussion aims at introducing the overall concept and stating requirements that need to be fulfilled in order to construct a corresponding architecture.

## Requirements

1. Conversational user interface. The application requires a conversational user interface, where a continuously ongoing dialogue is presented between the user, a LLM, and an interpreter. A user enters a textual prompt, describing a modeling task, that is appended to the dialogue and sent to a LLM. The generated LLM response in text form, potentially containing a concrete syntax of a modeling language, is appended to the dialogue. In case the response contains the concrete syntax of a known modeling language, it is sent to an interpreter. The execution result of the interpreter is appended to the dialogue in text form and, if a visual model could be rendered, in an image format.
2. LLM inference. For running LLM inference, the selection of a suitable LLM, parametrization, and the local or remote execution of the inference are required.
  - 2.1 LLM Selection. For supporting multiple open source and commercial LLMs, possibly differing in their capabilities of generating concrete modeling language syntax, a LLM needs to be selected for inference.
  - 2.2 LLM Parametrization. Depending on the selected LLM, setting hyperparameters might be required for inference, e.g., to influence the predictability and stability of responses for given prompts. Typically, open source LLMs provide fine-grained control over the sampling in the sequence of tokens, often depending on *temperature*, with lower values decreasing randomness when choosing the next token randomly from the candidates, *top\_k*, limiting the random choice of a token to the *k* most probable candidates, and *top\_p*, limiting a token to be chosen from the most probable candidates as far as they are, in sum, within probability *p* [13].
  - 2.3 Local LLM Inference. LLMs require a runtime component for running inference locally. Especially open source LLMs tend to be suitable for execution within a client application on end-user devices, depending on computation, storage, and memory requirements.
    - 2.3.1 LLM Runtime. A runtime is required for a specific LLM, depending on its architecture and the support of required software or hardware inference or acceleration components. For example, the open source runtime llama.cpp is a C and C++ implementation supporting well-known models such as Alpaca, Vicuna, Falcon, and Llama 2 by inference on x86 and ARM CPUs, e.g. by AVX instructions and the ARM instructions through the Metal framework, as well as on GPUs through CUDA<sup>2</sup>.
    - 2.3.2 LLM Files. LLMs might be provided in versions differing in formats, quantization, and overall model size, mostly due to the number of parameters that are typically in the billions. The runtime is required to support and manage LLMs and file formats given sufficient storage and memory capacity. With the released versions in 7, 13, and 70 billion parameters, Llama 2 ranges between 14 GB and 138 GB<sup>3</sup>.

---

<sup>2</sup><https://github.com/ggerganov/llama.cpp>

<sup>3</sup><https://huggingface.co/meta-llama>

- 2.4 Remote LLM Inference. LLMs that are available remotely, especially commercial products not available as open source software, require an API client connected to compatible remote servers.
  - 2.4.1 LLM API Client. Common APIs include the OpenAI API<sup>4</sup>, used for ChatGPT and many commercial and non-commercial LLMs, and APIs of cloud platforms such as the Replicate HTTP API<sup>5</sup>. Oftentimes, HTTP and REST are used with specialized bindings for common programming languages.
  - 2.4.2 LLM Server. Outside the client application, LLM servers provided, e.g., by OpenAI, Microsoft Azure, Replicate, or Google are part of the system architecture. Implications, at least to privacy, cost, and performance, need to be considered in this regard.
3. Interpreter. For interpreting models, the selection, parametrization, and a runtime are required.
  - 3.1 Interpreter Selection. To support the concrete syntax of multiple modeling languages in a textual format, a suitable interpreter needs to be selected. E.g., an interpreter for PlantUML<sup>6</sup> such as Plantweb<sup>7</sup>.
  - 3.2 Interpreter Parametrization. Depending on the interpreter, parameters might be required for determining the rendering layout or the output format. While a textual syntax is assumed for the input, the output might result in vector graphics such as SVG, raster graphics, or textual drawings [16].
  - 3.3 Interpreter Runtime. For rendering visual models of a specific modeling language, an interpreter supporting a concrete syntax in textual format is required. The interpreter needs to be capable of processing the textual input with low latency, rendering a visual model as an output.
4. Data Store. For assessing the results of different LLMs and interpreters over the course of multiple conversations, a data store is required. That is, the selected LLM, interpreter, LLM parameters, and interpreter parameters need to be stored for a specific conversation in addition to the requested prompts, generated responses, and rendered visual models.

## 3.2. Architecture

Based on the concept and requirements, Figure 2 shows components realizing a possible architecture of a client application. In the *Conversation* subsystem, the main control flows are initiated through the *Conversational User Interface*. According to the sequences in Figure 2, the user engages in the preparation of a LLM (1.1-1.5), the preparation of an interpreter (2.1-2.4), the interaction with the LLM (3.1-3.6), and the interaction with the interpreter (4.1-4.5).

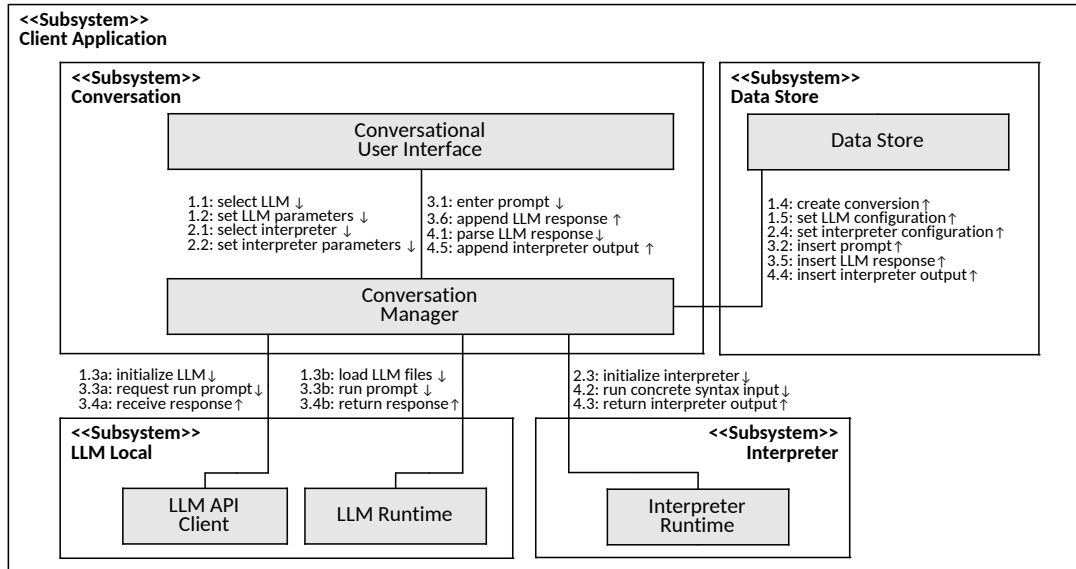
---

<sup>4</sup><https://platform.openai.com/docs/api-reference/chat>

<sup>5</sup><https://replicate.com/docs/reference/http>

<sup>6</sup><https://plantuml.com/>

<sup>7</sup><https://plantweb.readthedocs.io/>



**Figure 2:** Architecture describing the necessary components of a client application as UML Communication Diagram. The main control flows are indicated by numbered sequences.

After selecting and setting up the LLM (1.1-1.2), the *Conversation Manager* prepares the initiation of the conversation. Depending on the selection, either a server-side LLM is initialized through the *LLM API Client* (1.3a), or client-side LLM files are loaded (1.3b) with the *LLM Runtime*. Following the preparation of the LLM, the conversation and its configuration are recorded in the *Data Store* (1.4-1.5). In a similar way, an interpreter is selected and set up (2.1-2.2) with the *Conversation Manager*. The interpreter is initialized by the *Interpreter Runtime* (2.3), assumed running locally, and with its configuration recorded in the *Data Store* (2.4).

The user starts interacting with a LLM by entering a prompt that is sent to the *Conversation Manager* (3.1) and inserted in the *Data Store* (3.2). Either, a server-side LLM is requested to run the prompt with the *LLM API Client* (3.3a) and returns the received response (3.4a). Alternatively, a client-side LLM runs the prompt directly (3.3b) and returns the response (3.4b). The response is inserted in the *Data Store* (3.5) and appended in the *Conversational User Interface* (3.6).

For interacting with the interpreter, an appended response is parsed by the *Conversation Manager* to detect a supported modeling language syntax (4.1). The selected interpreter is executed by running it in the *Interpreter Runtime* (4.2) with the detected syntax as input. The output is returned to the *Conversation Manager* (4.3), inserted in the *Data Store* (4.4), and appended to the dialogue in the *Conversation Manager* (4.5).

## 4. Results of initial experiments

This section discusses first experimental results of executing a prototype implemented for ChatGPT 4 and Llama 2 with two test cases covering the generation and interpretation of models in PlantUML and Graphviz syntax. Figure 3 shows the beginning of a dialogue in the user interface of the application, implemented as a feasibility demonstration in Python 3.11<sup>8</sup>.

<sup>8</sup><https://github.com/fhaer/llm-cmi>



## 4.1. PlantUML test case

At the beginning of the dialogue in Figure 3, the user-provided prompt describes the test case for generating a UML class diagram according to a scenario. Without an explicit specification of the diagram, it is the task of the LLM to create classes and relationships such as orders, order items, and customers. The aim is to observe how the case is conceptualized by GPT-4 and Llama 2.

The screenshot displays the user interface for an LLM application. On the left, the 'LLM Settings' panel is visible, showing the model set to 'OpenAI/gpt-4'. Below this, there are sliders for 'temperature' (set to 0.20), 'top\_p' (set to 0.90), 'frequency\_penalty' (set to 0.00), and 'presence\_penalty' (set to 0.00). At the bottom of the settings panel, the 'Interpreter Settings' section shows the interpreter set to 'Plantweb/PlantUML'. On the right, the dialog area shows a user prompt: 'Create a class diagram in PlantUML code according to the following description. An order consists of an order header and one or more order itmes. Each order has an id and a timestamp. In an order header, a customer is referenced in addition to a billing address, a shipping address and a payment method. For the customer, an id as well as first name, last name and a registration timestamp are known in addition to a billing address and potentially multiple shipping addresses.' Below the prompt, a status bar indicates 'Running Inference: OpenAI/gpt-4 ...'. The response from the LLM is a PlantUML class diagram: 

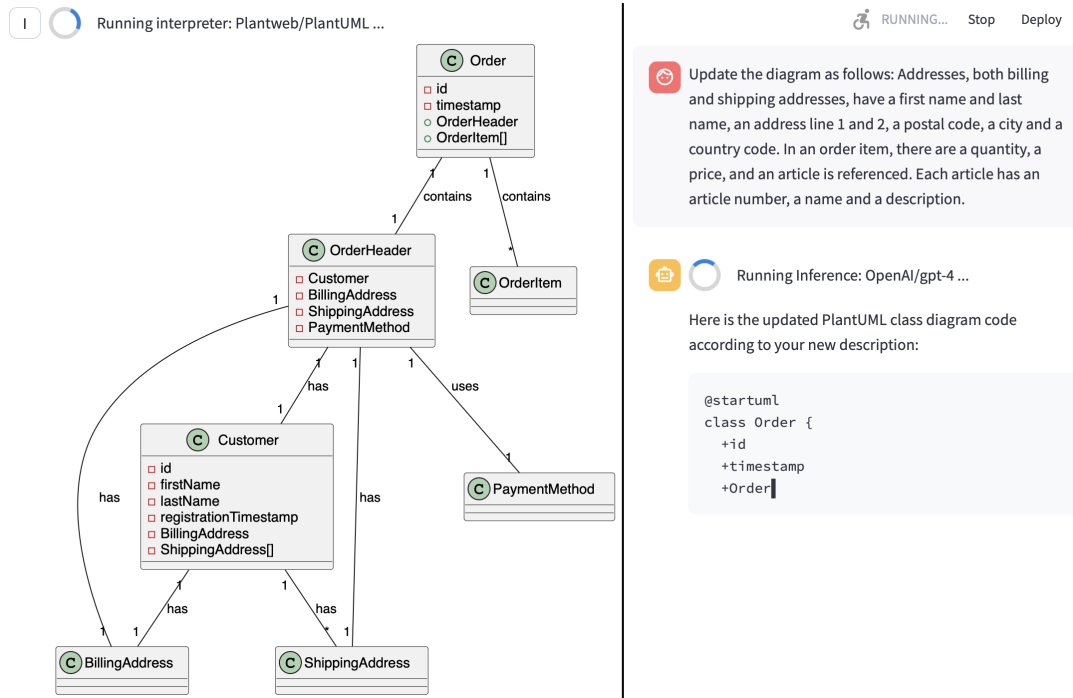
```
@startuml
class Order {
+id
+timestamp
+OrderHeader
+OrderItems[]
}
class OrderHeader
```

**Figure 3:** User interface of the prototype showing the PlantUML test case with the settings (left) and dialog (right) for GPT-4. PlantUML code is generated by GPT-4 and rendered as shown in Figure 4.

At first, GPT-4 is run using the settings visible on the left-hand side in Figure 3, resulting in the diagram in Figure 4. GPT-4 created syntactically correct PlantUML code and recognized classes, attributes, and relationships with multiplicities. All classes and relationships seem appropriate for the scenario, including the classes `OrderItem`, `PaymentMethod`, `BillingAddress`, and `ShippingAddress`, for which no specific properties or attributes were described. Possibly, `BillingAddress` and `ShippingAddress` could be further generalized. Attributes use list notation, recognizing the multiplicities correctly, and specify visibility, which is always private except for `OrderHeader` and `OrderItem[]` in the `Order` class for no apparent reason. While the spelling is generally appropriate for software classes, the capitalization of attributes referencing classes seems slightly out of place.

Figure 4 shows the continuation of the dialogue on the right-hand side, requesting to extend the diagram with articles and further descriptions of order items and addresses. After re-generating the PlantUML syntax and rendering, the class diagram shown in Figure 5 is the final result of the test case. A new class `Article` with attributes and further attributes in existing classes were created according to the description. Furthermore, GPT-4 introduced a class `Address` with





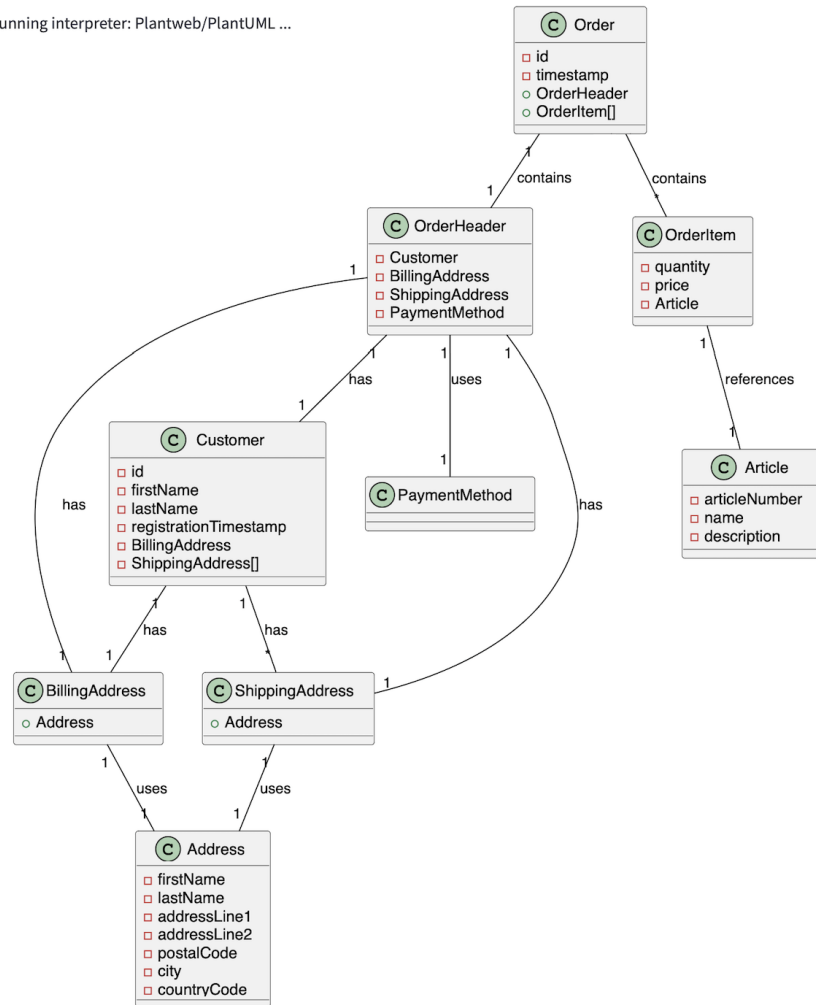
**Figure 4:** Continuation of the dialogue from Figure 3 showing the rendered class diagram (left) and an update requested by the user. After re-generation of the syntax, the rendering in Figure 5 results.

corresponding attributes and referenced it in the classes `BillingAddress` and `ShippingAddress` by attributes. In this way, the duplication of the address attribute was prevented without a generalization, using composition rather than inheritance. Variability in the responses could be observed, e.g. distinguishing between the two types of addresses in different ways and using compositions instead of associations. In a few cases, additional data types and simple operations such as getter methods were additionally created.

For Llama 2, the test case is applied as before with custom settings (Appendix, Figure A.1). Also for this LLM, the result is syntactically correct PlantUML code for a class diagram, however, only classes were generated without any relationships. When asked to add relationships or associations, the LLM either did not add them or added incorrect syntax to the code.

Figure 6 shows the classes with attributes, including data types. Classes were generated similarly to GPT-4, except for the `Address` class which represents both billing and shipping addresses with corresponding attributes in `OrderHeader` and `Customer`. While there is no distinction on a type level and no generalization, this design also prevents the duplication of attributes. The attributes created by GPT-4 were also generated by Llama 2, representing the description correctly. However, Llama 2 also engaged in hallucination and added attributes to the `OrderItem`, `PaymentMethod`, and `Address` classes where the description did not indicate any specific details. Data types were correctly recognized for attributes referencing other classes and denote list types for referencing multiple classes when required by the scenario.

The dialogue is continued as before (Figure 4), still not producing relationships (Appendix, Figure A.2). An article class was added correctly, including attributes. The scenario introduced



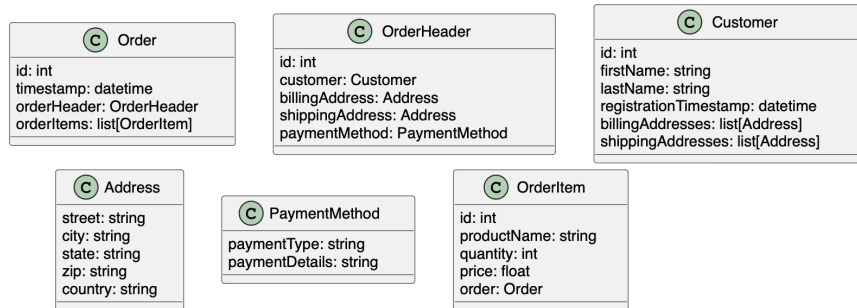
**Figure 5:** Continuation of the dialogue from Figure 4, showing the updated class diagram.

further details for addresses and order items, requiring new attributes. These were added only, it seems, if they did not conflict with existing ones. E.g., no address line and postal code attributes were added to Address which already had street and zip attributes due to prior hallucination. Variability could be observed in slight changes to the chosen attributes and data types.

#### 4.2. Graphviz test case

In the Graphviz test case, a description for generating a directed graph from user-provided data is given to GPT-4 and Llama 2 as described in Figure 7. The aim is to observe how the LLMs construct graphs on an instance level from custom data with syntax and formatting instructions.

GPT-4 generated syntactically correct Graphviz code and created the graph as specified by the data. For the nodes, further formatting instructions for the shapes and a custom numbering format for labels were correctly implemented. As requested, edges were correctly visualized by their weight using the width; however, the weight attribute should have been specified



**Figure 6:** Rendering generated from the PlantUML test case with Llama 2 using the settings specified in the Appendix (Figure A.1). Notably, the LLM did not generate relationships.

as well. For Llama 2, Figure 8 shows the result with the settings applied before (Appendix, Figure A.1). The Graphviz code was also generated correctly, in terms of the syntax and the graph specified by the data. Compared to GPT-4, node names and labels are swapped since the specified numbering format was applied to the node names instead of the labels. Further, the shapes are missing the style attribute. Regarding the edges, the weight attributes were correctly set without the requested visualization of the width. Unnecessary edge labels denoting the graph data were additionally generated. Generally, the data and its syntax were recognized correctly by GPT-4 as well as Llama 2 with relatively low variability.

Consider the following data stored in triples using the format (s,t,w) and create a directed graph in Graphviz syntax. For each triple, create a directed edge from node s to node t with weight w. Show nodes as rectangles with rounded corners and add labels numbering them N1, N2 and so on. Adjust the width of edges according to the weight. ("3e4a8f1d","5b6c7d2e",1), ("3e4a8f1d","9a8b7c6d",2), ("3e4a8f1d","2f4a6e8d",3), ("5b6c7d2e","9a8b7c6d",2), ("9a8b7c6d","5b6c7d2e",1), ("9a8b7c6d","2f4a6e8d",2), ("2f4a6e8d","3e4a8f1d",3)

Running Inference: OpenAI/gpt-4 ...

Here is the Graphviz syntax for the directed graph based on your data:

```

digraph G {
  graph [splines=polyline]
  node [shape=box, style=rounded]

  "3e4a8f1d" [label="N1"]
  "5b6c7d2e" [label="N2"]
  "9a8b7c6d" [label="N3"]
  "2f4a6e8d" [label="N4"]

  "3e4a8f1d" -.-> "5b6c7d2e" [penwidth=1]
  "3e4a8f1d" -.-> "9a8b7c6d" [penwidth=2]
  "5b6c7d2e" -.-> "9a8b7c6d" [penwidth=2]
  "9a8b7c6d" -.-> "2f4a6e8d" [penwidth=1]
  "9a8b7c6d" -.-> "2f4a6e8d" [penwidth=2]
  "2f4a6e8d" -.-> "3e4a8f1d" [penwidth=3]
}

```

```

"3e4a8f1d" -.-> "2f4a6e8d" [penwidth=3]
"5b6c7d2e" -.-> "9a8b7c6d" [penwidth=2]
"9a8b7c6d" -.-> "5b6c7d2e" [penwidth=1]
"9a8b7c6d" -.-> "2f4a6e8d" [penwidth=2]
"2f4a6e8d" -.-> "3e4a8f1d" [penwidth=3]
}

```

In this syntax, `digraph G` declares a directed graph named `G`. The `node` line sets the default shape and style for all nodes. Each node is then declared with its unique identifier and a label. The `-.->` operator creates a directed edge between two nodes, and the `penwidth` attribute sets the width of the edge according to the weight.

Running interpreter: Plantweb/Graphviz ...

```

graph TD
  N1((N1))
  N2((N2))
  N3((N3))
  N4((N4))
  N1 -- 1 --> N2
  N1 -- 2 --> N3
  N2 -- 2 --> N3
  N3 -- 1 --> N4
  N3 -- 2 --> N4
  N4 -- 3 --> N1
  
```

**Figure 7:** Graphviz test case executed with GPT-4 with the resulting rendering.

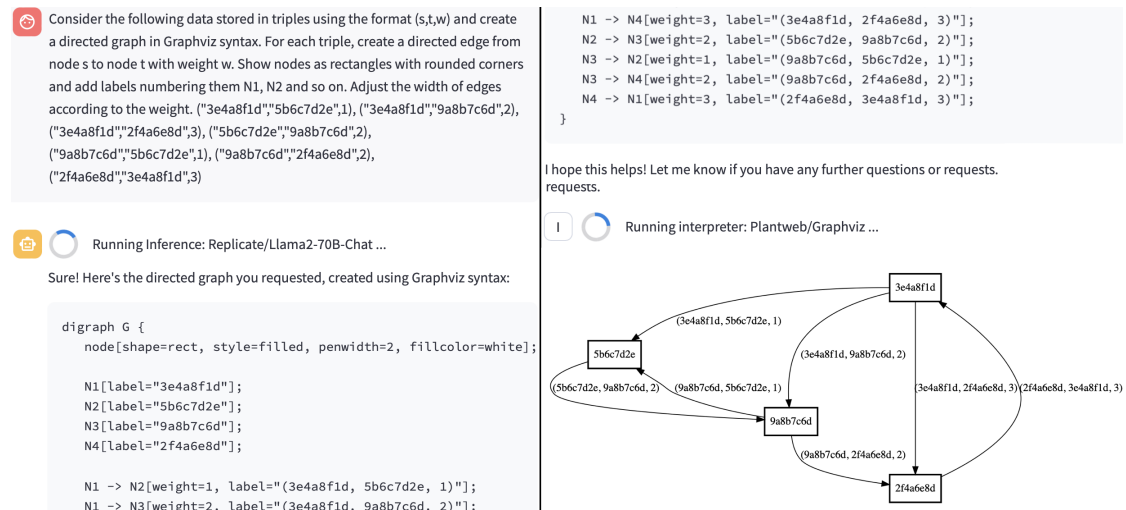


Figure 8: Graphviz test case executed by Llama 2 with the resulting rendering.

### 4.3. Discussion

The syntax for PlantUML and Graphviz was generally created correctly by GPT-4 and Llama 2. GPT-4 utilized the syntax to a greater extent and produced more comprehensive solutions with greater complexity, also considering details recognized from the case descriptions. For Llama 2, it remains unclear whether missing elements were unknown in their syntax or not recognized, especially in the PlantUML test case where relationships had to be inferred and were missing. In this test case, the LLM was also prone to hallucination. When given a syntax description and custom data in the Graphviz test case, both Llama 2 and GPT-4 were able to recognize it and rendered graphs on an instance level. Due to the chosen phrasing and parameters, variability was generally low, however, changes as described in the results could still be observed. As expected, parameters of the sampling influenced variability, i.e., temperature, top\_p, and top\_k. Concerning the prototype, limitations exist in its current implementation stage, not realizing the architecture in full regarding compatibility with APIs and local LLMs beyond GPT-4 and Llama 2.

## 5. Conclusion

In this paper, the application of a conceptual model interpreter for LLMs was explored through the creation of an architecture and prototype using GPT-4 and Llama 2 for generating and rendering models in UML and Graphviz syntax. Results encompass (1.) the components of the architecture compatible with state-of-the-art LLMs and interpreters, and (2.) initial experimental results, demonstrating the creation of models in correct syntax for GPT-4 and Llama 2 with major advantages in correctness, recognized details, and comprehensiveness for GPT-4. Especially for GPT-4, the results show that modeling iteratively in a conversational dialogue could be practical, however, further systematic evaluations need to be conducted. Future research will apply the interpreter for these evaluations with additional commercial and open source LLMs.

## Acknowledgments

This work is partially supported by the Swiss National Science Foundation project Domain-Specific Conceptual Modeling for Distributed Ledger Technologies [196889].

## References

- [1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, in: Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS'17, Red Hook, NY, USA, 2017.
- [2] P. Xu, X. Zhu, D. A. Clifton, Multimodal learning with transformers: A survey, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45 (2023).
- [3] P. P. Ray, Chatgpt: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope, *Internet of Things and Cyber-Physical Systems* 3 (2023) 121–154. doi:<https://doi.org/10.1016/j.iotcps.2023.04.003>.
- [4] W. Ma, S. Liu, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, Y. Liu, The scope of chatgpt in software engineering: A thorough investigation, *CoRR abs/2305.12138* (2023). doi:10.48550/arXiv.2305.12138.
- [5] H. Fill, P. Fettke, J. Köpke, Conceptual modeling and large language models: Impressions from first experiments with chatgpt, *Enterp. Model. Inf. Syst. Archit. Int. J. Concept. Model.* 18 (2023) 3. doi:10.18417/emisa.18.3.
- [6] A. R. Chow, How chatgpt managed to grow faster than tiktok or instagram, *Times Online* (2023). URL: <https://time.com/6253615/chatgpt-fastest-growing/>, accessed on 2023-08-30.
- [7] M. Wermelinger, Using github copilot to solve simple programming problems, in: Proceedings of the 54th ACM Technical Symposium on Computer Science Education, SIGCSE 2023, ACM, New York, NY, USA, 2023. doi:10.1145/3545945.3569830.
- [8] OpenAI, ChatGPT Plugins, Technical Report, 2023. URL: <https://platform.openai.com/docs/plugins/getting-started/openapi-definition>, accessed on 2023-08-30.
- [9] E. Davis, S. Aaronson, Testing GPT-4 with wolfram alpha and code interpreter plug-ins on math and science problems, *CoRR* (2023). doi:10.48550/arXiv.2308.05713.
- [10] F. Härer, Scalable model-based decentralized applications in the cloud using certificates and blockchains, in: 2023 IEEE International Conference on Public Key Infrastructure and its Applications (PKIA), 2023, pp. 1–8. doi:10.1109/PKIA58446.2023.10262768.
- [11] D. Rothman, Transformers for Natural Language Processing. Second Edition, Packt Publishing, O'Reilly Media, Birmingham, UK, 2022.
- [12] M. Isaev, N. McDonald, R. Vuduc, Scaling infrastructure to support multi-trillion parameter LLM training, in: Architecture and System Support for Transformer Models (ASSYST) held at the International Symposium on Computer Architecture 2023 (ISCA 2023), 2023.
- [13] M. Labonne, Decoding Strategies in Large Language Models, Technical Report, 2023. URL: [https://mlabonne.github.io/blog/posts/2023-06-07-Decoding\\_strategies.html](https://mlabonne.github.io/blog/posts/2023-06-07-Decoding_strategies.html), accessed on 2023-08-30.
- [14] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. Canton-Ferrer, M. Chen, G. Cucurull,

D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, T. Scialom, Llama 2: Open foundation and fine-tuned chat models, CoRR abs/2307.09288 (2023). doi:10.48550/arXiv.2307.09288.

- [15] M. Vidgof, S. Bachhofner, J. Mendling, Large language models for business process management: Opportunities and challenges, in: Business Process Management Forum - BPM 2023 Forum, Utrecht, The Netherlands, volume 490 of *Lecture Notes in Business Information Processing*, Springer, 2023. doi:10.1007/978-3-031-41623-1\_7.
- [16] J. Cámara, J. Troya, L. Burgueño, A. Vallecillo, On the assessment of generative AI in modeling tasks: an experience report with chatgpt and UML, *Softw. Syst. Model.* 22 (2023).

## Appendix

The screenshot shows the Llama 2 settings interface with the following values:

- temperature: 0.20
- top\_p: 0.01
- max\_new\_tokens: 4096
- min\_new\_tokens: -1
- top\_k: 50

Figure A.1: Settings used in the PlantUML and Graphviz test cases with Llama 2.

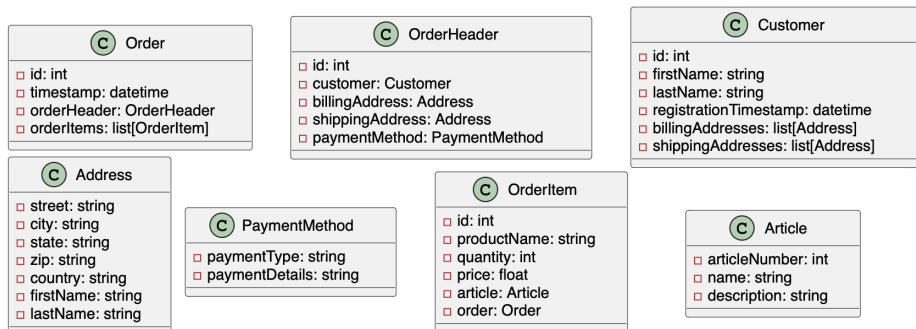


Figure A.2: PlantUML rendering from Llama 2, after requesting an update of the initial diagram.