Software and Systems Modeling
https://doi.org/10.1007/s10270-024-01206-9

EXPERT VOICE

Towards standarized benchmarks of LLMs in software modeling tasks: a conceptual framework

Javier Cámara[1] · Lola Burgueño[1] · Javier Troya[1]

Received: 24 July 2024 / Revised: 26 July 2024 / Accepted: 27 July 2024

Abstract
The integration of Large Language Models (LLMs) in software modeling tasks presents both opportunities and challenges. This Expert Voice addresses a significant gap in the evaluation of these models, advocating for the need for standardized benchmarking frameworks. Recognizing the potential variability in prompt strategies, LLM outputs, and solution space, we propose a conceptual framework to assess their quality in software model generation. This framework aims to pave the way for standardization of the benchmarking process, ensuring consistent and objective evaluation of LLMs in software modeling. Our conceptual framework is illustrated using UML class diagrams as a running example.

Keywords Modeling · LLMs · Benchmarking

1 Introduction

The field of software engineering has recently witnessed a significant paradigm shift with the introduction of Large Language Models (LLMs). These have shown remarkable proficiency in a range of tasks that go from programming to testing, including software modeling [1, 2]. However, despite their potential, the adoption of LLMs in software engineering tasks (and in particular, in software modeling) raises critical questions about the quality and reliability of the outputs they produce [3]. The intricate nature of software design, characterized by a need for precision, demands a high degree of accuracy from any automated tool employed in the process.

In light of this, there emerges a pressing need for robust methodologies to standardize the assessment of the quality of outputs generated by LLMs in software engineering tasks [4]. For instance, Austin et al. proposed two benchmarks for program synthesis with LLMs [5], and Xu et al. have systematically analyzed how different LLMs can complete and synthesize code [6]. However, benchmarking approaches for LLMs that focus on artifacts at a higher level of abstraction are so far underexplored.

This Expert Voice focuses on one such crucial area - the modeling of software systems. The benchmarking of LLMs in this context requires a formalized and systematic approach to assess the artifacts produced. Such formalization is pivotal to developing a reference framework against which the quality of different LLM outputs can be measured and compared. By establishing general guidelines for standardized metrics and evaluation protocols, this paper aims to lay the groundwork for consistent and objective benchmarking by proposing a conceptual reference framework, ensuring that the integration of LLMs in software modeling not only enhances efficiency but also upholds quality standards in software development. We illustrate our conceptual framework using UML class diagrams as a running example.

In the remainder of this paper, we first discuss challenges of benchmarking LLMs in software modeling (Sect. 2) and follow with an overview of the reference framework we propose to address them (Sect. 3). The paper ends with some concluding remarks.

Communicated by Bernhard Rumpe.

✉ Javier Cámara
jcamara@uma.es

Lola Burgueño
lolaburgueno@uma.es

Javier Troya
jtroya@uma.es

[1] ITIS Software, Universidad de Málaga, Blvd. Louis Pasteur, Málaga 29071, Spain

2 Challenges

The benchmarking of LLMs for software modeling presents multiple challenges rooted in the inherent complexity of the task. These include the diversity of valid solutions that a single problem can yield, the non-deterministic nature of LLM outputs, and the variation in effectiveness of different prompt strategies.

2.1 Solution space variability

One of the primary challenges in benchmarking LLMs for software modeling is the inherent diversity of valid solutions to a given problem. Unlike tasks with singular correct answers (or tasks, such as programming, where multiple valid solutions exist, but these can be systematically tested for quality), software modeling often admits multiple equally valid representations that satisfy the specified requirements. This multiplicity complicates the establishment of benchmarks for evaluating the quality of models produced by LLMs. To address this issue, we must develop methodologies that can represent this variability, enabling the quantification of quality and comparison of solutions within a broad solution space. This entails designing representations of the valid solution space, as well as metrics that can accommodate different, yet correct approaches to modeling.

2.2 Non-determinism of LLM output

A significant challenge in benchmarking LLMs for software modeling is the intended non-determinism of their outputs. When prompting an LLM, the response is generated in two steps. First, the LLM processes the prompt and generates a probability distribution over possible tokens (words) that are likely to be a good response. This process is deterministic. Second, from this distribution the response is generated following a decoding strategy. The most straightforward strategy would be to select the token with a higher probability (and the response would be deterministic). However, to make LLMs sound more "human", it is common to follow a stochastic sampling (controlled by hyperparameters such as temperature, top-k and top-p [7]). The current most powerful LLMs follow the latter strategy, which means that, for identical inputs, LLMs will generate varying outputs across different executions. This complicates the assessment of output quality. To tackle this issue, it is essential to devise benchmarks that not only measure the quality of individual outputs but also quantify the model's reliability in maintaining a certain quality level over multiple runs. Incorporating measures of consistency into our evaluation criteria will enable a more comprehensive understanding of an LLM's performance, highlighting not just its ability to generate high-quality models but also its predictability and dependability as a tool in software engineering processes.

2.3 Prompting strategy dependency

The effectiveness of LLMs significantly hinges on the prompting strategies employed. Different prompt formulations can elicit varying levels of quality in the output of a given LLM. This poses a challenge because a fair benchmark should be capable of distinguishing the effectiveness of different LLMs across a spectrum of prompting techniques. This requires a systematic exploration of prompt strategies to identify which are most suited to generating high-quality software models, facilitating a more informed comparison of LLM capabilities.

A common thread among these challenges is the necessity for a flexible and nuanced approach to evaluation. Hence, an approach for benchmarking LLMs in software modeling should meet several key requirements to effectively address the identified challenges: (i) it must be adaptable to account for the diversity of valid solutions and capable of evaluating outputs under varying conditions; (ii) it should ensure reliability by assessing consistency across multiple executions, capturing the non-deterministic nature of LLM outputs, and (iii) it needs to be able to evaluate effectiveness across various prompt strategies.

In the following section, we describe our vision for the class of approach that satisfies these requirements.

3 Conceptual framework

The benchmarking process for LLMs in software modeling, as depicted in Fig. 1, involves a structured approach that integrates both human-driven specification, as well as automated execution and analysis.

Human-driven specification (Sect. 3.2). Initially, the process begins with the human-driven specification phase, comprising prompt specification, solution space specification, and the definition of metrics and scoring scheme. This phase ensures that the LLM's tasks are clearly defined, alongside the criteria for evaluating the generated outputs. This phase consists of three different activities:

(1a) *Prompt specification*. This activity consists in constructing the prompt workload that LLMs will be subject to during the benchmark process, and may involve the exploration and use of different prompt strategies that are appropriate to the problem at hand, to address Challenge 2.3.

(1b) *Solution space specification*. In this activity, designers build a structured specification of acceptable solutions to the prompts incorporated into the prompt workload. To address Challenge 2.1 in an effective manner, this will involve the systematic definition of possible variation points (e.g., through the use of feature models),

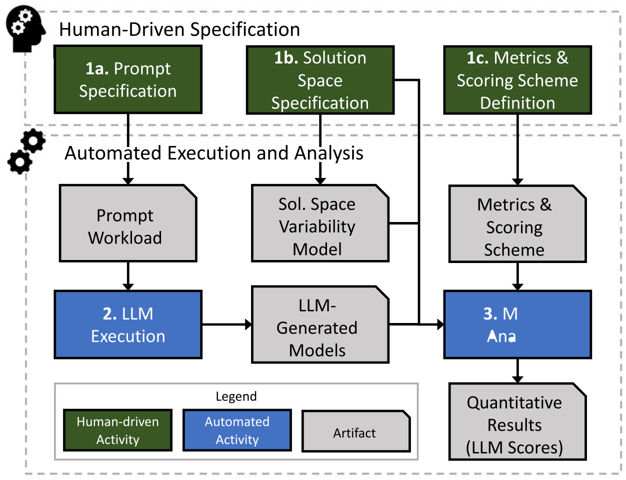
Human-Driven Specification
1a. Prompt Specification
1b. Solution Space Specification
1c. Metrics & Scoring Scheme Definition
Automated Execution and Analysis
Prompt Workload
Sol. Space Variability Model
Metrics & Scoring Scheme
2. LLM Execution
LLM-Generated Models
3. M Ana
Quantitative Results (LLM Scores)
Legend
Human-driven Activity
Automated Activity
Artifact

Fig. 1 Overview of LLM benchmarking for modeling

in order to capture the nuances of different modeling approaches to obtain valid solutions.

(1c) *Metrics and scoring scheme definition*. To properly quantify the quality of models generated by LLMs during the benchmark process, designers will need to define a set of metrics concerning different dimensions, such as completeness, syntactic correctness, as well as semantic consistency of the models [3]. These metrics must be accompanied by a set of scoring rules that will map characteristics of valid solutions to score contributions across the different dimensions of concern, based on a set of general scoring criteria.

Automated execution and analysis (Sect. 3.3). Following the human-driven specification phase, the process transitions into automated execution and analysis:

(2) *LLM execution*. Here, the specified prompt workload is fed into the *LLMs under Benchmark (LLMuB)* for execution, resulting into different sets of models (e.g., per prompt strategy) that will be used as part of the input provided to the next activity. To address the nondeterminism in LLM output (Challenge 2.2), multiple runs of each task in the workload may be required.

(3) *Model analysis*. In this activity, the LLM-generated models during the execution activity are analyzed based on the predefined metrics, scoring scheme, and solution space variability model. Model analysis provides the final output of the process, which is the set of quantitative results that reflect the LLM scores against the specified scoring scheme.

In the remainder of the section, we delve into the details of each of these phases and activities, illustrating them with a running example that focuses on UML class modeling.

3.1 Running example

The diagrams in Fig. 2 display alternative versions of the domain model that we will be using as running example to illustrate our vision. Given a prompt, the LLMuB should create a domain model analogous to any of these variants.

In the running example, we can see different variation points across all diagrams. One is in the Service class, whose type can be Maintenance or Repair. In diagram a), this is modeled with an enumeration. Another possibility, shown by diagram b), is to remove the type attribute and add two subclasses to Service: MaintenanceService and RepairService.

Another variation point is in the specialization of the OfficialGarage class. This class is a specialization of Garage because it has an additional warranty attribute, and it is modeled with two classes in diagram a). Another possibility would be to have three classes like diagram c) shows. One is a Garage abstract class with an address attribute. This class has two specializations: a NormalGarage class and the OfficialGarage class, the latter with the warranty attribute.

Since natural language is not as precise and unambiguous as a modeling language, a designer could also interpret the specification in a way that slightly changes the semantics of the diagram. For instance, diagram d) shows the case in which the Service class is modeled as an association class between Car and Garage, changing the semantics because making this modeling choice, we can only have one Service for each pair of {Car, Garage}.

3.2 Human-driven specification

The human-driven specification stage consists of three different activities, which are discussed in the following.

3.2.1 Prompt specification

Each prompt consists of a natural language description of a domain model, i.e., the domain specification in natural language, from which the LLMs should be able to derive a valid domain model. However, there are different aspects that have to be considered when specifying prompt workloads [8].

First, the prompt workload should cover the semantic domains that the LLMuB are designed / trained for. This will allow the assessment of their semantic capabilities. While this aspect is relevant for general LLMs, it will not apply to domain-specific LLMs, for which only prompts focusing on the specific domain they address need to be provided.

Second, while zero-shot prompting [9] is the most straightforward technique where only the domain specification of the model to build is provided, in many occasions it falls short when dealing with complex domains. To address this, since the emergence of LLMs, numerous prompting techniques

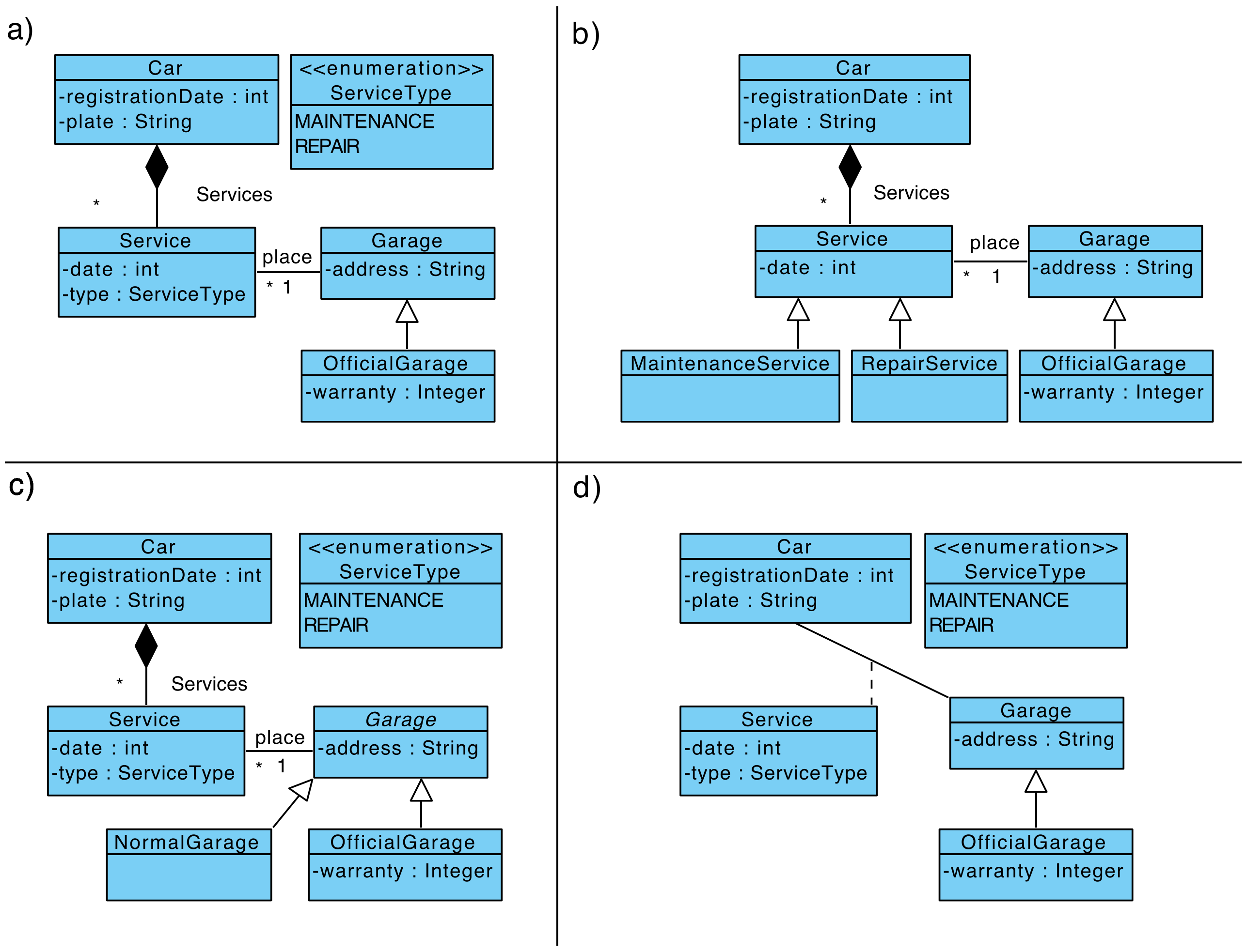

Fig. 2 Domain model of our running example

have appeared such as few-shot prompting, and chain-of-thought prompting.[1]

Some of the strategies pose their own challenges. For instance, few-shot prompt learning requires to provide the LLMs with some shots (i.e., examples) consisting of pairs {specification, output model} to enable in-context learning and guide the LLM to better performance. The fact that outputs need to be provided implies that the input domain models need to be encoded as text so that LLMs can deal with them.[2]

Listing 3 shows a sample prompt for the domain model of our running example that employs a few-shot prompting strategy. It includes two simple shots (in blue) and the domain specification (black). If we were to follow a zero-shot prompting strategy, only the text in black would be included. To represent the shots, we have used the PlantUML notation because it offers both a textual and a graphical representation for diagrams. LLMs are better suited to process the textual representation (e.g., used for the few-shot prompting strategy), while for humans, understanding the graphical representation is easier.

In our example, we can expect that the same LLM will perform differently with input from a zero-shot prompting strategy, compared to a few-shot strategy (that includes both the blue and black parts of the prompt).

Generalizing from our example, determining what are the best prompts/strategies to include in a given benchmark will strongly depend on its objectives (e.g., identifying areas for improvement, comparing quality of outputs, etc.) and vary from case to case. However, the two orthogonal aspects that must always be considered are whether a benchmark workload should: (i) provide prompt specifications covering different domains, and (ii) be formulated using various prompting strategies (Fig. 3).

[1] https://platform.openai.com/docs/guides/prompt-engineering

[2] Multi-modal LLMs are out of the scope of this paper.

Prompt:

Q: Banks have accounts, which register the account number and date in which it was opened.
A:
@startuml
class Bank {}
class Account {
number : Integer
openingDate : Integer
}
Bank ∗ −− Account
@enduml
Q: Hospitals are a special type of Medical Centers
A:
@startuml
MedicalCenter <|−− Hospital
@enduml
Q: We will have cars for which we would like to store their registration date as well as their plate. Cars may have services. For each service, we register its date and the type of service. We can have two types of services: maintenance and repair. Each service takes place in a particular garage, and all garages are located in an address. We have a special type of garage, namely an official garage, which offers a specific warranty.

Fig. 3 Example of prompts for our running example

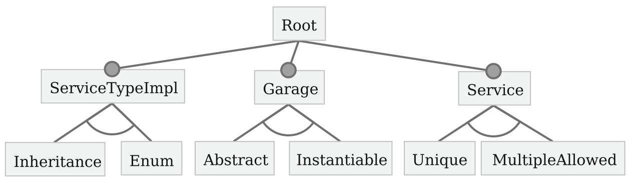

Fig. 4 Variability points expressed as a feature model

3.2.2 Solution space specification

During this stage, benchmark developers must produce a specification that enables capturing the inherent variability of the solution space. One of the ways in which this can be achieved is through the use of feature models. Figure 4 shows a sample feature model that captures the coarse-grained points of variability in our running example.

In the first level of the tree, below the root node, we have the `ServiceTypeImpl` node that specifies that the service type in our running example domain can be captured either via inheritance (Fig. 2b) or as an enumeration (Fig. 2a, c, d). Node `Garage` captures the design choice of making the `Garage` class `Abstract` (Fig. 2c) or `Instantiable` (Fig. 2a,b,d). Node `Service` captures the variation point that allows a semantic variation in which a unique (materialized in variants like the one displayed in Fig. 2d) or multiple services are allowed. For the sake of brevity, we are not representing lower levels in the tree that would incorporate e.g., multiple variation points capturing the alternative ways in which variants that incorporate `Unique` could be represented.

3.2.3 Metrics and scoring scheme definition

Assessing the quality of the outputs produced by the LLMuB will require both *metrics* to quantify the different dimensions of concern (in areas such as completeness, syntactic and semantic correctness), as well as a *scoring scheme* that enables mapping model instances produced by the LLMuB to scores. We posit that both metrics and scoring schemes can be derived via refinement from a general set of *scoring criteria* that can be applied to the modeling problem at hand.

Let us illustrate these concepts in the context of our running example. Table 1 includes a set of scoring criteria (adapted from [3]) that are common to all UML class diagrams. While these scoring criteria provide a general notion of interesting aspects to measure in class diagrams produced by LLMuB, they are not directly usable within the context of a specific modeling problem. Hence, there is a need to refine these criteria into sets of rules (i.e., a scoring scheme) that enables us to map characteristics of models to metrics. Let us take as an example criterion `CPT1` in Table 1. In the context of our problem, having all the necessary classes means that we will require having classes `Car`, `Service`, and `Garage` and `Official Garage`. While `MaintenanceService` and `RepairService` are classes required in some class diagram variants, others do not require their presence.

In this context, Fig. 5 illustrates a potential notation that hinges upon OCL and can be used to capture scoring rules for class diagrams in our example problem.[3] The proposed notation captures scoring rules as sets of pairs `(scope, assignments)`, where:

- **scope** designates a subtree of the solution space variability model to which `assignments` in the tuple apply (cf. Fig. 4). For instance, in line 1 the scope corresponds

[3] Note that these OCL expressions assume that models conform to a simplified UML metamodel.

Table 1 General scoring criteria for UML class diagrams

Cat.	**ID**	**Description**
Completeness	CPT1	All necessary classes are included
	CPT2	Classes include all necessary attributes
	CPT3	Classes include all necessary operations
	CPT4	All necessary relations are included
	CPT5	Multiplicities are included in relations
	CPT6	Relations names are included
	CPT7	Role names are included
Syntactic Correctness	SYC1	Class attributes are correctly represented
	SYC2	Class operations are correctly represented
	SYC3	Simple associations are correctly represented
	SYC4	Aggregation relations are correctly represented
	SYC5	Composition relations are correctly represented
	SYC6	Generalisation relations are correctly represented
	SYC7	Relation multiplicities are correctly represented
	SYC8	Roles are correctly represented
Semantic Correctness	SEC1	Class names are appropriate
	SEC2	Attribute names are appropriate
	SEC3	Operation names are appropriate
	SEC4	Relation names are appropriate
	SEC5	Multiplicities are correct
	SEC6	Role names are appropriate
	SEC7	There are no redundant classes
	SEC8	There are no redundant attributes
	SEC9	There are no redundant operations
	SEC10	There are no redundant relations

```
1. scope [Root]
2.  context Model
3.  [inv GarageClassExists: self.elements->select(oclIsKindOf(Class) and match(’Garage’))->size() > 0] [M-CPT1:20]
4.  ...
5.  [inv ServiceClassExists: self.elements->select(oclIsKindOf(Class) and match(’Service’))->size() > 0] [M-CPT1:20]
6.  ...
7. scope [Root->ServiceTypeImpl->Inheritance]
8.  context Model
9.  [inv RepairServiceClassExists: self.elements->select(oclIsKindOf(Class)
    and match(’RepairService’))->size() > 0] [M-CPT1: 10]
10. [inv MaintenanceServiceClassExists: self.elements->select(oclIsKindOf(Class)
    and match(’MaintenanceService’))->size() > 0] [M-CPT1:10]
11. ...
12. scope [Root->ServiceTypeImpl->Enum]
13.   context Service
14.   [inv ServiceTypeAttributeCheck:
          self.features->exists(a |
              a.oclIsKindOf(Attribute) and
              match (a.name, ’type’) and
              a.type.oclIsKindOf(Enumeration) and
              a.type.oclAsType(Enumeration).ownedLiteral->includesAll(match(Set{’MAINTENANCE’, ’REPAIR’})))] [M-CPT1:10]
```

Fig. 5 Illustration of potential scoring rules for class inclusion (CPT-1, Table 1) in our running example

to the `Root` node of the feature model, indicating that the assignments included before the next `scope` statement apply to all class diagram variants. In contrast, scope `Root->ServiceTypeImpl->Inheritance` in line 7 indicates that assignments are valid only for variants that employ inheritance to represent service types in our modeling problem.

- **assignments** are pairs `(exp, A)`, where `exp` is an OCL-like expression that captures a property of the model, and `A` is a set of pairs that map score contributions

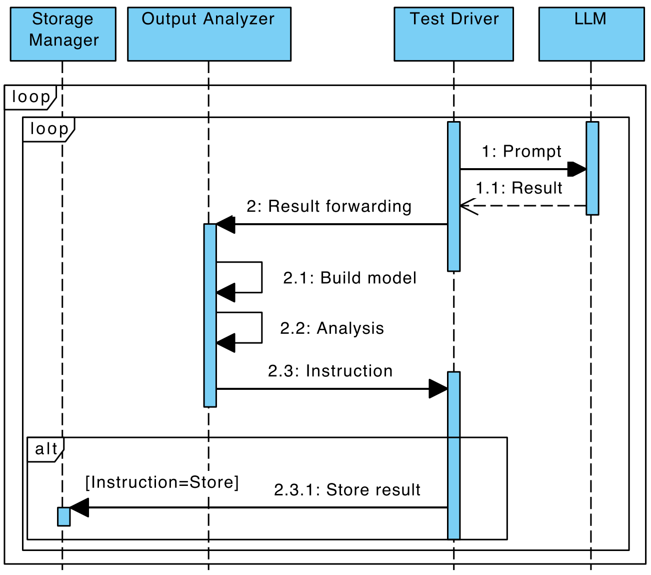

Fig. 6 LLM execution protocol overview

to metrics. Line 3, for instance, shows an expression that checks whether a class `Garage` exists in the model. If that is indeed the case, there is an increment of 20 in the value of metric `M-CPT1` associated with general scoring criteria `CPT1`. Although the example illustrates the case where the assignment impacts a single metric, in general multiple metrics may be affected by each assignment.

3.3 Automated execution and analysis

The execution and analysis phase of LLM benchmarking for modeling tasks consists of automatically executing the LLMuB through the workload, and analyzing the results obtained.

3.3.1 LLM execution

Figure 6 illustrates the execution protocol for an LLM. The outer loop corresponds to the iteration over the different prompt strategies included in the workload. The inner loop captures the cycle in which the `Test Driver` (i.e., the software component in charge of orchestrating the execution process) sends first a prompt to a LLMuB and waits for its results. Once these are obtained, they are forwarded to the `Output Analyzer`, which is in charge of parsing them, building the actual model and indicating the test driver whether further prompts are to be sent to the LLMuB (multiple interactions may be required for certain prompt strategies such as *iterative prompting*, i.e., when the LLMuB is prompted with follow-up prompts based on the output of an initial prompt). When the output analyzer determines that no further interactions are required, it instructs the test driver to store the result obtained from the LLMuB during the last iteration into a `Storage Manager`, which holds the information that will be processed during model analysis.

3.3.2 Model analysis

The last activity of our proposed LLM benchmarking process entails assigning scores to the models produced during the LLM execution stage. This is achieved by systematically evaluating the models against the set of rules contained in a scoring scheme, which are described in the terms illustrated in Sect. 3.2.3.

We posit that systematic evaluation of a model against the rules in a scoring scheme and the aggregation of total scores for a given metric can be performed based on the structure of the feature model.

We illustrate this point following our running example: Figs. 7 and 8 depict two class diagrams produced by GPT-4 in PlantUML notation for the zero-shot and two-shot prompt, respectively, displayed in Listing 3. Both diagram variants include all mandatory classes required for any complete solution to this modeling problem (i.e., `Car`, `Service`, `Garage`, and `OfficialGarage`), but while the zero-shot prompt variant employs an attribute `Type` and an enumeration to capture the type of `Service`, the two-shot variant employs inheritance.

Table 2 illustrates the scoring of the two class diagram variants with respect to `M-CPT1` and the rules shown in Fig. 5. The first four rows correspond to the rules that check the inclusion of mandatory classes (scope `Root` in the feature model).[4] If we focus on `GarageClassExists`, we

[4] Although not included in Fig. 5 for the sake of brevity, we assume rules analogous to `GarageClassExists` for classes `Car` and `OfficialGarage`.

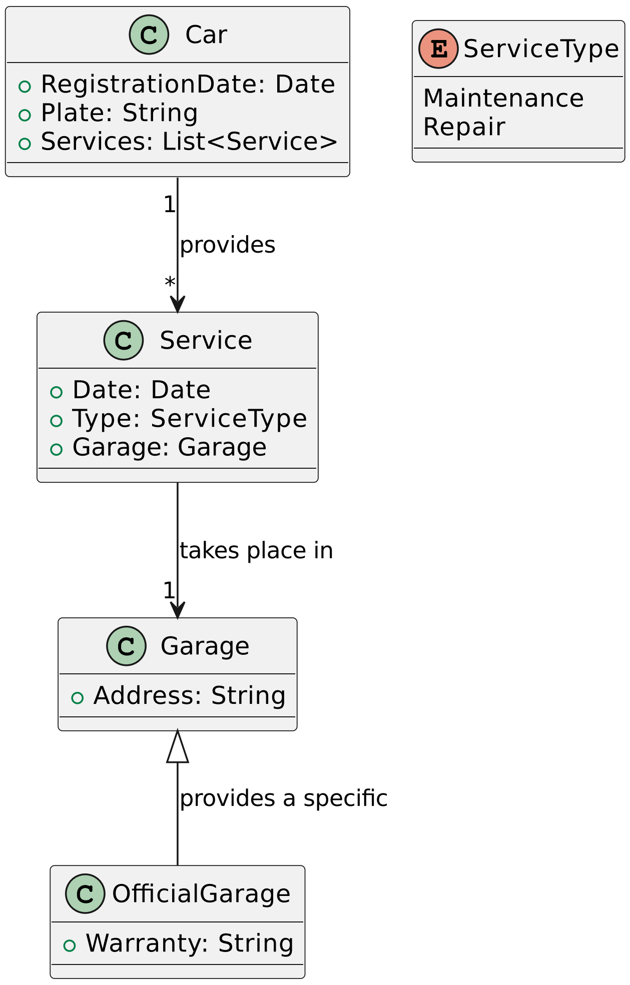

Fig. 7 GPT-4 response to the zero-shot prompt

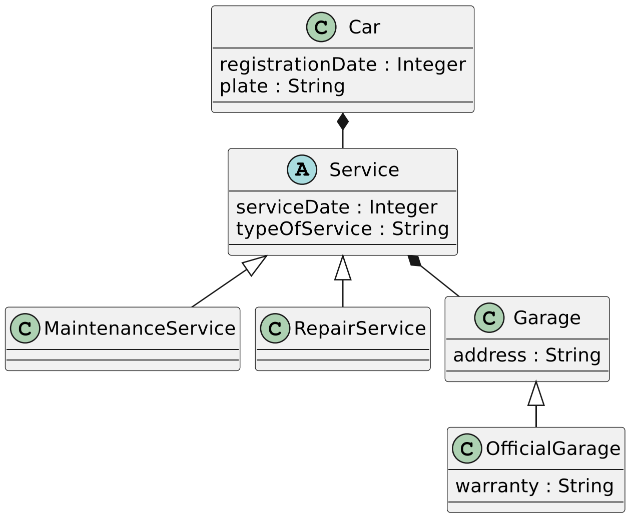

Fig. 8 GPT-4 response to the two-shot prompt

can observe that the OCL-like expression does not employ the equality operator to check the name of the class. Instead, a "`match`" function is employed to abstract a semantic check mechanism that determines whether the name of the class generated by the LLMuB is equivalent to the intended class in the solution space. Note that the approach we are proposing is transparent to this semantic check, which might involve the use of ontologies or semantic distance measures and thresholds, which have been employed, for instance, in works that target the automated grading of UML class diagrams (e.g., [10]).

The bottom three rows of Table 2 show rules that are within the common scope of `Root->ServiceTypeImpl` in the feature model tree. In these rows, the two class diagram variants differ in scoring because the zero-shot variant satisfies both `MaintenanceServiceClassExists` and `RepairServiceClassExists` (aggregated `M-CPT1` score: 20), whereas the two-shot variant satisfies the rule `ServiceTypeAttributeCheck`, which scores only 10 in `M-CPT1` because the rule assumes that this modeling of service type is less appropriate, compared to inheritance, in the context of the modeling problem at hand. Note that no class diagram should in principle satisfy all three rules, given that the satisfaction of rule `ServiceTypeAttributeCheck` is mutually exclusive with the satisfaction of the other two. Having this mutual exclusivity captured as part of the feature model (Fig. 4 – nodes `Enum` and `Inheritance`) provides an opportunity for benchmark designers to explicitly check and handle anomalous situations in which, e.g., a class diagram employs both an enumeration and inheritance to capture service types.

4 Concluding remarks

In this Expert Voice we have discussed the need for benchmarking LLMs in software modeling, the main challenges it poses, and our vision to move forward in this area. Although we have focused on class diagrams to illustrate our ideas, we believe that the general principles and strategies we have outlined are broadly applicable to a wide range of modeling tasks that go beyond structural modeling in UML. Instantiating this benchmarking approach to other modeling domains will entail adapting our conceptual framework's elements to fit specific classes of models and languages.

Of course, the community does not have to start from scratch and there are already efforts upon which the community can build to support advances in LLM benchmarking for modeling, such as those that exist on automated UML model evaluation (e.g., [10–12]), which can be adapted and extended to fit this new context.

LLMs are being progressively incorporated into many modeling-related tasks and processes, with outputs that will

Table 2 M-CPT1 scores for model variants generated by GPT-4: Zero-Shot (ZS –Fig. 7) vs. Two-Shot (TS – Fig. 8)

Rule	M-CPT1 (ZS)	M-CPT1 (TS)
`CarClassExists`	20	20
`GarageClassExists`	20	20
`OfficialGarageClassExists`	20	20
`ServiceClassExists`	20	20
`MaintenanceServiceClassExists`	0	10
`RepairServiceClassExists`	0	10
`ServiceTypeAttributeCheck`	10	0
Total	90	100

make it into production environments and provide support in real-world scenarios. As a community, it is our responsibility to be ready to assess the quality of the outputs generated by such processes, thus helping to pave the way to a future in which productivity does not have to be traded by quality.

Acknowledgements Work partially funded by the Spanish Government (FEDER, Ministerio de Ciencia e Innovación–Agencia Estatal de Investigación) under projects TED2021-130523B-I00 and PID2021-125527NB-I00, and Universidad de Málaga under project B1-2022_81.

Funding Funding for open access publishing: Universidad Málaga/CBUA

References

1. 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 (2023)
2. 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)
3. Cámara, J., Troya, J., Burgueño, L., Vallecillo, A.: On the assessment of generative AI in modeling tasks: an experience report with chatgpt and UML. Softw. Syst. Model. **22**(3), 781–793 (2023). https://doi.org/10.1007/S10270-023-01105-5
4. Ozkaya, I.: Application of large language models to software engineering tasks: Opportunities, risks, and implications. IEEE Software **40**(3), 4–8 (2023). https://doi.org/10.1109/MS.2023.3248401
5. Austin, J., Odena, A., Nye, M.I., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C.J., Terry, M., Le, Q.V., Sutton, C.: Program synthesis with large language models. CoRR abs/2108.07732, (2021). https://arxiv.org/abs/2108.07732
6. Xu, F.F., Alon, U., Neubig, G., Hellendoorn, V.J.: A systematic evaluation of large language models of code. In: Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (ACM, 2022), p. 1-10
7. 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. CoRR abs/2005.14165, (2020). https://arxiv.org/abs/2005.14165
8. G. Marvin, N. Hellen, D. Jjingo, J. Nakatumba-Nabende, Prompt Engineering in Large Language Models. In: Data Intelligence and Cognitive Informatics (Springer, 2024), pp. 387–402
9. Kojima, T., Gu, S.S., Reid, M., Matsuo, Y., Iwasawa, Y.: Large language models are zero-shot reasoners. Adv. Neural Inf. Process. Syst. **35**, 22199–22213 (2022)
10. Bian, W., Alam, O., Kienzle, J.: Automated Grading of Class Diagrams. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C) (2019), pp. 700–70https://doi.org/10.1109/MODELS-C.2019.00106
11. Bian, W., Alam, O., Kienzle, J.: Is automated grading of models effective? assessing automated grading of class diagrams. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. (Association for Computing Machinery, New York, NY, USA, 2020), MODELS '20, p. 365-376
12. Hasker, R.: Umlgrader: an automated class diagram grader. J. Comput. Sci. Coll. **27**, 47–54 (2011)

Javier Cámara is Associate Professor of Computer Science at the University of Málaga and Honorary Visiting Fellow at the Department of Computer Science, University of York. His current research interests include self-adaptive and autonomous systems, software architecture, formal methods, as well as cyber-physical and AI systems. He received his European PhD with honors from the University of Málaga in 2009. For more information, contact him at jcamara@uma.es or visit https://javier-camara.github.io/.

Lola Burgueño is an Associate Professor at the University of Málaga, Spain. Her research interests lie in the fields of software engineering and model-based software engineering. She has made contributions to the application of artificial intelligence techniques to improve software development processes and tools, uncertainty management during the software design phase, and model-based software testing, among others. For more information, please visit http://lolaburgueno.github.io.

Javier Troya is Associate Professor at the University of Málaga, Spain. Before, he was Assistant Professor at the University of Seville, Spain (2016-2020), and a post-doctoral researcher in the TU Wien, Austria (2013-2015). He obtained his International PhD with honors at the University of Malaga, Spain (2013). His current research interests include MDE, Software Testing and Digital Twins. For more information, please visit https://javiertroyauma.github.io/.