The impact of Software Model Slicing on Software Model Completion with Large Language Models

Alisa Welter
Saarland University
Saarbrücken, Germany
welter@cs.uni-saarland.de

Benedict Bliem
Saarland University
Saarbrücken, Germany
bebl00001@stud.uni-saarland.de

Omer Iqbal
Saarland University
Saarbrücken, Germany
omiq00001@stud.uni-saarland.de

Sven Apel
Saarland University
Saarbrücken, Germany
apel@cs.uni-saarland.de

Abstract

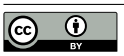
Large language models (LLMs) have recently shown promising results in supporting software model completion. Despite this, an open issue is what to provide as context to the LLM and how that influences completion correctness. We refer to the selection of model parts given as context to the LLM as *software model slicing* for LLM-based software model completion. Improving completion correctness directly benefits modelers by reducing manual corrections and additional prompting, while slicing models into smaller parts lowers token usage, reducing inference time.

In this paper, we formalize model slicing in a task-independent, metamodel-agnostic manner based on graph representations of software models and instantiate different representative slicing approaches, including radius-based, modularization-based, and LLM-based slicing. Thereupon, we systematically evaluate the influence of model slicing and compare the different approaches against identity and random baselines on 1,000 real-world models from two public datasets using open-source LLMs. Our results show that appropriate model slicing significantly improves completion correctness while simultaneously reducing token usage across several structural and semantic evaluation metrics. This is particularly noteworthy as reducing context might be expected to remove relevant information and degrade performance. In particular, radius-based and modularization-based slicing outperform both the identity and random baselines. These findings establish model slicing as a key factor in LLM-based model completion and provide guidance for effective context selection for other modeling tasks.

CCS Concepts

• **Software and its engineering → Model-driven software engineering**; **Software design engineering**; **Software evolution**.

ACM Reference Format:
Alisa Welter, Benedict Bliem, Omer Iqbal, and Sven Apel. 2026. The impact of Software Model Slicing on Software Model Completion with Large Language Models. In *ACM/IEEE 29th International Conference on Model Driven Engineering Languages and Systems (MODELS 2026), October 04–09, 2026, Málaga, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3822455.3830319



MODELS 2026, Málaga, Spain

ACM ISBN 979-8-4007-2809-9/26/10
https://doi.org/10.1145/3822455.3830319

1 Introduction

Software models can grow substantially in size, particularly in real-world industrial settings [73]. For instance, an industrial case study from the railway domain reports models containing, on average, more than 20 000 elements [73]. Advancements in large language models (LLMs) have recently made it possible to support and partially automate tasks in model-driven software engineering, for example, by assisting with model creation, evolution, and maintenance [17–19, 31, 74, 75, 77].

While initial drafts of software models generated by LLMs from specifications are often incomplete [17, 20, 30, 33], models also evolve as systems are extended and maintained, making incompleteness a recurring issue. *Software model completion* addresses this problem by inferring missing elements and relations from partially specified models. Although LLMs already perform well in software model completion on smaller models in generic domains by providing the partial model as input to the LLM [18], limited context size becomes a significant challenge when scaling to large-scale, domain-specific models [26, 74]. Even though there exists work in machine learning on extending the context lengths of an LLM, it has been shown that, even if large or theoretically sufficient context windows are available, LLM performance degrades as input length increases, indicating that sheer input size can hinder the effective use of task-relevant information [29, 44, 51]. With this in mind, the question arises which parts of a software model are relevant to provide as context for model completion with an LLM. We refer to this problem henceforth as *software model slicing* [67, 74, 77] for LLM-based software model completion.

Prior work has addressed the limited context challenge only implicitly. Some approaches select task-specific subsets of the software model as LLM input, for example, providing class names for class name completion [18, 19]. Other approaches adopt task-independent strategies, for example, by operating on change-based graph representations derived from historical model data [74]. Other techniques restrict the context to a local neighborhood around the user's current working position [77].

In any case, the impact of software model slicing on model completion performance, that is, the correctness of the model elements predicted by the LLM, generally remains a crucial, open question. This is particularly relevant because improved completion performance can benefit modelers: More accurate recommendations reduce manual corrections and make modeling faster and easier. Furthermore, slicing can enable the modeling support for large, real-world, industry models that are otherwise too large to be processed by current LLM context limits, and operating on smaller slices can reduce inference cost and improve response time.

We therefore set out to investigate the impact of model slicing on model completion performance and systematically compare different task-independent and metamodel-agnostic slicing approaches for software models on two large, real-world, publicly available datasets. For this purpose, we operate on unified graph representations of software models as proposed in the literature [67, 74] and formally define a set of representative software model slicing approaches that build on and extend existing slicing techniques, including LLM-based slicing, which delegates the selection of relevant model elements to an LLM query, radius-based slicing, which includes local neighborhoods, and a modularization-based approach.

We evaluate the impact of model slicing, in general, and the performance of individual slicing approaches, in particular, using various correctness metrics that capture different aspects of the suggested model elements, such as structural correctness (e.g., whether a predicted model element has the correct relations to other model elements) and semantic correctness (e.g., whether the predicted element matches or is semantically similar to the expected one).

We further compare the slicing approaches to two baselines: identity slicing, which provides the complete software model without removing any elements, and random slicing, which selects model elements at random to form a slice. Importantly, the slicing approaches that we consider are, in principle, applicable to software models of arbitrary types through the underlying graph-based representation, independent of specific datasets, and can thus be easily extended. We systematically compare the slicing approaches on a set of 1,000 software models from two large datasets, the real-world RepairVision dataset [62, 63] and ModelSet dataset [52]. The datasets contain real-world Ecore and UML models, covering multiple domains within model-driven engineering.

We find that model slicing indeed significantly improves model completion performance, with smaller slices consistently yielding higher accuracy across metrics, datasets and LLMs. This is particularly noteworthy, as one might expect that reducing context removes relevant information; instead, our results indicate that carefully selecting relevant context is more important than providing larger inputs. In particular, radius-based and modularization-based slicing outperform both baseline approaches and LLM-based slicing.

Note that, being uniform, metamodel-independent approaches, our example slicing approaches can be applied to different software models. This opens up opportunities for further extensions, including the development of additional slicing approaches. At the same time, our systematic comparison provides insights for research on model completion and other modeling tasks by clarifying how slicing influences model completion performance.

In summary, we make the following contributions[1]:

(1) We extend a graph-based formalism of software models with a definition of slicing criteria from the literature [74, 77], resulting in a task-independent and metamodel-agnostic problem formulation.
(2) We express five different concrete slicing approaches, including radius-based, modularization-based, and LLM-based slicing in this formalism.
(3) We systematically evaluate and compare the different slicing approaches on two real-world datasets using two open-source LLMs and a comprehensive set of evaluation metrics, including BLEU score, embedding-based and TF–IDF-based cosine similarity, as well as type and structural correctness.
(4) Our comparison provides practical guidelines to reduce refinement effort, token usage, and cost, while clarifying the isolated impact of slicing on completion performance.

[1]The paper artifacts and additional material are available on the supplementary Website (https://github.com/se-sic/ModelCompletionSlicing).

2 Related Work

In this section, we provide an overview of existing work on (LLM-based) model completion and generation, as well as model slicing approaches that mainly focus on producing well-formed slices for manual inspection, rather than selecting relevant context for LLMs-based model completion.

2.1 Model Completion

Early approaches on software model completion leveraged pattern catalogs by detecting changes via pattern- or graph-matching and subsequently applying the corresponding missing elements [24, 41, 42, 45, 46, 50, 60]. These approaches depend on domain-specific pattern catalogs that require manual creation and maintenance.

Early efforts to integrate natural language processing techniques into modeling support rely on semantic representations, such as knowledge bases, semantic networks [1, 2], clustering [32], and word embeddings [16, 53], to capture similarity and recommend model elements. More recently, deep-learning models have been adapted to software modeling tasks, for example, by using encoder-decoder architectures to suggest model element types [26, 27]. Other approaches provide similar examples of models through collaborative filtering [25] and similarity-based filtering [28], but ultimately rely on users to apply the final model completion [3]. Some approaches instead learn from existing modeling data. These include transformer-based models trained from scratch to suggest metamodel concepts [78], as well as fine-tuned language models for recommending metamodeling elements in textual DSLs [23].

More recent approaches employ LLMs without task-specific fine-tuning, relying instead on knowledge acquired during large-scale, general training [18, 19, 74, 77]. Chaaben et al. [18, 19] use GPT-3 to suggest new model class names, attributes, and associations by providing examples from unrelated domains. For prompting, the model is sliced by applying a task-specific slicing approach, e.g., for class name completion, they select randomly between two and four pairs of related classes, enclose the class names in square brackets, and include them in the prompt.

Apvrille and Sultan [6] use ChatGPT to complete SysML models by providing partial models together with JSON-encoded domain descriptions. Syntactic errors are fed back to ChatGPT in an iterative loop. To keep prompts manageable, the model completion is split into small, task-specific steps (e.g., actors, use cases, relations).

Other approaches adopt task-independent strategies and concentrate on the neighborhood of the most recently changed element [74, 77]. For example, Tinnes et al. [74] focus on the localized area by operating on difference graphs derived from historical model evolution data. In addition to Chaaben et al. [19] they incorporate

domain-specific context through similarity-based few-shot retrieval from the software model repository.

Welter et al. [77] propose a multi-location model completion approach, alternating between local LLM-based model completion where a radius-based slicing is used and a global embedding-based neural network that predicts additional locations requiring modification. Overall, however, the impact of software model slicing (and possible variants thereof) on model completion performance has not been studied systematically and remains an open question.

2.2 Model Generation

While model completion focuses on changing and extending a given software model, model generation focuses on the generation of software models directly from natural language descriptions [17, 20, 75], requirements [7, 20, 31, 33, 35, 71, 79], user stories [43], or images of UML diagrams [22]. However, existing approaches often produce only partially correct models on the first attempt [17, 20, 30, 33]. Generated models often require substantial iterative refinement, limiting scalability to large real-world models and motivating further refinement via model completion [30, 33].

2.3 Slicing for Software Models

Earlier work on software model slicing primarily aimed to trace requirements to model elements and to extract relevant parts of models for a given task to reduce manual inspection effort, improving model understanding, and better support manual maintenance [4, 15, 47, 48, 67]. While several model slicing approaches have been proposed since then, however, most are limited to specific model types or executable models [4, 47, 48]. As a result, these approaches do not readily transfer to models with different or non-executable semantics [67]. Blouin et al. [15] and Pietsch et al. [67] summarize some of the few generally applicable model slicing approaches that support the generation of new, well-formed models from slices. The actual slicing criterion is defined by the user.

In the context of LLM-based model slicing, only limited prior work exists. Luitel et al. [55] use an LLM to extract requirement-relevant slices from textualized Simulink models; however, their approach is type-dependent and primarily designed for human analysis rather than automated model completion.

However, no systematic comparison between different slicing approaches exists, particularly with respect to their impact on LLM performance in software model completion.

2.4 Supporting Evolution and Consistency

Further related work includes approaches that identify relevant parts of models, but target different goals. This includes change impact analysis and traceability across models, requirements, documentation, and code [5, 8, 34, 57] and metamodel co-evolution [21, 37, 65, 66]. Closely related is also model repair [12, 56], which focuses on automatically correcting inconsistencies in software models. Most approaches rely on graph transformation rules or OCL-like constraints to automatically repair models [49, 59, 61, 62], while others organize fixes (additionally) into repair trees [58, 68].

From a machine learning perspective, some model repair approaches use reinforcement learning, where rewards are based on achieving consistency and improving model quality [10, 11, 38]. While these works pursue different goals and we do not explicitly investigate slicing for these tasks, our results may provide valuable insights for tasks in these domains.

3 Metamodel-Independent Slicing for LLM-Based Model Completion

This section presents the formalization of software models based on prior work and further introduces slicing criteria examples that are serialization-independent, metamodel-independent and task-independent.

3.1 Preliminaries

We represent software models as graphs to provide a unified abstraction across diverse modeling formats and model types, as is common in the literature [39, 40, 54, 67, 72, 74]. In particular, we adopt the definition by Tinnes et al. [74], ensuring that all relevant information, such as types, is preserved, for example, as node attributes. Consequently, our slicing criteria generalize to semantically and syntactically rich models such as UML. In fact, any artifact that can be represented as a typed graph is in scope.

Definition 3.1 (Software model graph)**.** An abstract syntax graph G_m of a software model m is an attributed graph typed over an attributed type graph TG, which is defined by a metamodel TM. The type graph TG guarantees that all elements adhere to the structural and semantic constraints specified by TM.

The abstract syntax graph G_m is represented as a labeled directed graph $G_m = (V, E, l)$ over an alphabet L, with nodes V, directed edges $E \subseteq V \times V$, and a labeling function $l : V \cup E \to L$ [74]. An example is given in Figure 1.

We define slicing of software models similarly to definitions in the literature [55, 67, 77].

Definition 3.2 (Slicing criterion)**.** Given a software model m, the slicing criterion ς is defined as $\varsigma(G_m) \to C_m$, where m is a given model, and C_m is a set of graph elements.

For LLM-based software model completion, slicing is used to provide a suitable context for the LLM by selecting a subset of existing model elements $C_m \subseteq V_m \cup E_m$. An example is given in Figure 1. Furthermore, additional context in the form of task-specific, external knowledge (e.g., descriptions or requirements), depending on the available data, is conceivable but not further relevant for this paper.

3.2 Focus-driven Model Slicing

Initial drafts of software models are barely complete [17, 20] and therefore need to be iteratively extended and refined, with modelers typically focusing on small parts of the model at a time.

Recent advances in the model domain [6, 77] as well as in the code domain [9, 76, 80, 81] follow this principle by proposing iterative approaches that operate on localized contexts in each iteration step. For example, iterative line-level code completion focuses on the context around the developer's current cursor.

Motivated by this observation, we introduce the notion of *focus-driven model slicing*, where slicing is repeatedly applied around the current user focus in each iteration to dynamically refine the working context for the LLM during model completion (see Figure 2).

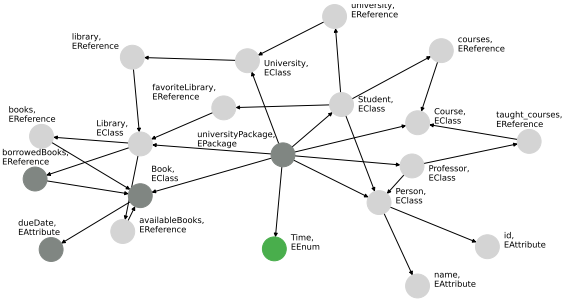


Figure 1: Example of an abstract syntax graph G_m of a model, including element names and types corresponding to the metamodel. Current user focus is highlighted in green. An example slice C'_m is highlighted dark grey and green.

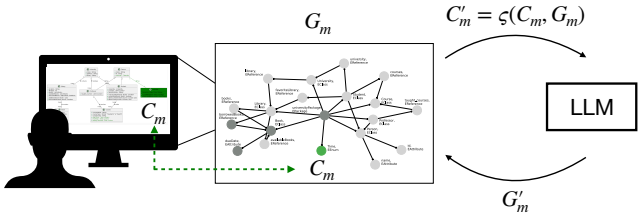


Figure 2: Visualization of focus-driven model slicing. Starting from the current user focus C_m and the abstract model graph G_m, a focus-driven slicing criterion ς is applied to derive the slice used as input for LLM-based model completion.

Definition 3.3 (Focus-driven model slicing)**.** Given a slicing criterion ς, an abstract syntax graph G_m of a software model, and a previous focus area C_m, focus-driven model slicing is defined as $\varsigma(C_m, G_m) \rightarrow C'_m$, where C_m is the current set of relevant graph elements and C'_m is a set of graph elements.

More specifically, we consider slicing criteria $\varsigma(C_m, G_m)$ that derive slices C'_m from m as a preprocessing step, which serve as an input to an LLM for software model completion. If a completion fails or needs revision, this is handled by the completion approach itself. Existing approaches already support such iterations [6]. A second form of iteration can occur after a successful completion, as models often require multiple changes, possibly at different locations. In such cases, new locations are predicted by existing multi-location completion approaches [77], and slicing is applied to each location accordingly.

3.3 Slicing Criteria: Example Approaches

Following prior work, our formulation of slicing criteria operates on a graph-based representation of software models and is therefore not restricted to a specific metamodel [74]. In the following, we formalize graph-based slicing criteria from the literature [18, 19, 74, 77]. To enable a fair and generalizable comparison, this work focuses on slicing criteria that satisfy the following properties:

- *Serialization-independent:* The slicing approach operates on the abstract model representation and is independent of concrete serializations (e.g., XMI syntax).
- *Metamodel-independent:* The slicing approach does not rely on language-specific modeling constructs and can be applied across different metamodels.
- *Task-independent:* The slicing approach does not depend on a specific, more fine-grained completion task (e.g., attribute generation or class name prediction).

Identity Slicing. The identity slicing criterion returns the input model m unchanged.

$$\varsigma_{\text{id}}(C_m, G_m) = G_m.$$

Identity slicing serves as a baseline and allows us to quantify the effect of any form of context reduction introduced by other slicing approaches.

Random Slicing. Random slicing ς_{rand} selects a subset of model elements from G_m independently of C_m. Random slicing serves as a baseline to compare criterion-driven slicing against uninformed context reduction.

Radius-based Slicing. Radius-based slicing selects model elements based on their graph distance $r \in \mathbb{N}$ to an initial set of relevant elements C_m.

$$\varsigma_{\text{rad}=r}(C_m, G_m) = V'_m \cup E'_m$$

with

$$V'_m = \{\, x \in V_m \mid \exists\, c \in C_m : \text{dist}_m(c, x) \leq r \,\},$$
$$E'_m = \{\, (u, v) \in E_m \mid u, v \in V'_m \,\},$$

where dist_m is the shortest-path distance in G_m. Similar concepts exist in the literature: Tinnes et al. [74] introduce simple change graphs as a slicing criterion, while Welter et al. [77] employ radius-based slicing.

Louvain-based Slicing. Louvain slicing uses the Louvain method of community detection [14] applied to the model graph by optimizing modularity, grouping elements such that intra-community connectivity is maximized relative to connections between communities.

Given the elements C_m, the slicing criterion selects all elements that belong to the same communities:

$$\varsigma_{\text{lou}}(C_m, G_m) = V'_m \cup E'_m,$$

with

$$V'_m = \bigcup_{c \in C_m} \text{community}(c),$$
$$E'_m = \{(u, v) \in E_m \mid u, v \in V'_m\},$$

where $\text{community}(c)$ denotes the graph community assigned to element c.

LLM-based Slicing. LLM-based slicing relies on a semantic grouping of model elements obtained via a large language model. The names of all elements in the model are provided to an LLM, which groups them into clusters[2].

[2]Note that clustering in general groups elements based on a feature-space representation, e.g., it operates on semantic metalevel representations of graph elements. In contrast, community detection operates directly on the graph structure and identifies groups based on connectivity [13, 14] .

Given an initial set of relevant elements C_m, the slicing criterion then selects all model elements that belong to the same clusters as the elements in C_m.

$$\varsigma_{\text{clust}}(C_m, G_m) = V'_m \cup E'_m,$$

with

$$V'_m = \bigcup_{c \in C_m} \text{cluster}(c), \qquad E'_m = \{(u, v) \in E_m \mid u, v \in V'_m\},$$

where $\text{cluster}(c)$ denotes the cluster assigned by the LLM to the class associated with element c. By operating on semantic clusters rather than purely structural connections, we capture high-level conceptual relationships that may not be explicitly represented in the model graph. A related concept is introduced by Chaaben et al. [19], who include a small number of pairs of related classes, written in square brackets, as contextual input to the LLM prompt.

4 Evaluation

We empirically evaluate the impact of model slicing on model completion performance and systematically compare the introduced slicing approaches using two datasets. In this section, we outline our research questions, describe the evaluation setup and data used, and present the results.

4.1 Research Questions

Prior work in machine learning suggests that restricting context of LLMs can improve performance [29, 44, 51]. Motivated by this observation, we investigate whether slicing software models into smaller subsets according to our slicing criteria introduced in Section 3.3 improves LLM-based model completion performance as compared to providing the complete software model to the LLM. We are further interested in how different slicing approaches affect model completion performance across different datasets and LLMs. All considered slicing approaches are metamodel-independent and operate on graph representations of software models. We systematically compare diverse slicing approaches on two datasets and with two different LLMs. We include random slicing as a baseline to distinguish meaningful context selection from arbitrary reductions.

RQ 1 *Does slicing software models improve the performance of LLM-based software model completion?*

Further, we investigate specifically the relationship between context size and model completion performance. While slicing reduces the number of tokens provided to the LLM, potentially lowering inference cost and latency, it may also remove relevant contextual information, decreasing performance.

RQ 2 *How does the slice size influence model completion performance?*

4.2 Datasets

To answer our research questions, we use two commonly established datasets.

RepairVision. The RepairVision dataset [62, 63] consists of versioned modeling projects and is essential for our study, as it provides ground-truth information on changes that were actually performed by modelers in real-world scenarios, rather than synthetic or simulated edits. The dataset comprises 41 modeling projects with a total of 912 commits and each commit includes over 160 changes on average [77]. Exact dataset statistics are given in Table 1.

ModelSet. The ModelSet dataset [52] consists of labeled snapshots of software models and is widely used in model-driven engineering research [23, 26]. ModelSet lacks version histories, though, requiring the construction of model evolution sequences as a preprocessing step for our experiments, as commonly done in existing literature [18, 19].

Overall, the datasets contain real-world Ecore and UML models, covering multiple domains within model-driven engineering, including real-world applications (e.g., banking), technical areas (e.g., Petri nets), BPMN, and transformation languages/tools.

Table 1: Dataset statistics for ModelSet [52] and RepairVision [62, 63]. Values denote the median, with the interquartile range (25th–75th percentile) in parentheses, and the maximum value in brackets.

Dataset	# Graphs	# Nodes	# Edges
ModelSet	7534	96 (56–164) [2993]	220 (124–379) [7389]
RepairVision	1249	284 (155–504) [14874]	507 (267–871) [22851]

4.3 Data Preparation

First, each software model is transformed into a unified graph representation and serialized into JSON following our definitions in Section 3.1.

Second, to construct partial models from the ModelSet dataset, we randomly select a target element as ground truth and remove it from the original model graph. This process yields pairs including the incomplete model graph G_m and ground truth graph elements y' that are to be correctly completed by the LLM. Focusing on a single ground truth element allows us to isolate the effect of slicing on completion performance, controlling other factors that could influence completion performance such as varying complexity of the task when multiple elements are missing. Note that this step is not strictly necessary for the RepairVision dataset, as ground-truth information is already included in the data. Nevertheless, to ensure comparability, we randomly select one element per model from the ground truth information, if multiple ground-truth elements are available (e.g., due to several changes per commit).

We consider an iterative setting in which slicing is repeatedly applied around the current user focus to dynamically refine the context provided to the LLM [77]. In this setting, the user focus C_m corresponds to one of the neighbors of the ground truth element in the original model graph. The focus element C_m is explicitly marked, while the actual ground truth is removed.

4.4 LLM Inference

Given a partial software model G_m and the ground truth elements y', we formulate a prompt p that includes a task description and the

respective slice C'_m computed via the slicing criterion $\varsigma(C_m, G_m) \rightarrow C'_m$ and the focus area C_m, which is explicitly highlighted within the slice, following the setup proposed by Tinnes et al. [74, 77]. The prompt p is then provided to the respective LLM, which produces the response y. The full prompt as well as an example for y are given on the supplementary Website.

Instruction tuning. In preliminary experiments, we used a short prompt instructing the LLM to predict the missing element. However, the LLM frequently failed to follow the task and instead produced descriptions of the model or other unrelated outputs.

To address this problem, we first introduced a more structured prompt that clearly specifies the input format, domain constraints, prediction task, and required output format. Second, we added a short reminder after the inserted software model summarizing the key instructions (e.g., returning only JSON without explanations).

Since the structures of the ModelSet and RepairVision datasets differ slightly, separate example models were included in the prompt for few-shot learning. This improved prompt significantly reduced the number of unusable predictions.

Note that the instruction explicitly states that the completion should be performed around the focus area C_m. Since we consider only a single ground-truth element, this constraint is necessary to ensure a fair evaluation. Without explicitly guiding the LLM towards the focus area, larger slices would introduce more potential completion locations, increasing the number of plausible but non-matching predictions. This would artificially penalize larger slices, as multiple conceivable completions may exist while y is only compared to one ground truth y' in the evaluation.

4.5 Experimental Setup

For both research questions, the experimental setup is as follows.

From each of our two datasets, RepairVision and ModelSet, we randomly sample 500 models from the datasets. This ensures a manageable evaluation size for LLM inference and computational resources, while still obtaining a sufficiently large sample to draw statistically meaningful conclusions. Afterwards, the data are preprocessed according to Section 4.3 to obtain a partial model and ground truth with some additional information on the focus area C_m. Afterwards, the respective slicing criteria $\varsigma(C_m, G_m) \rightarrow C'_m$ are applied to our preprocessed datasets (see Figure 2). The resulting slice is then provided to the LLM together with a prompt (see Section 4.4). We evaluate multiple LLMs to assess whether the observed results depend on the choice of LLM. In particular, we consider two openly available LLMs, gpt-oss-20b (context window of ~128k tokens[3]) and LLaMA-3.1-8B (128k context window [36]), and compare their performance across all experiments.

4.6 Performance Metrics

To assess the correctness of the model elements predicted by the LLM, we employ a set of evaluation metrics to capture different aspects of correctness. Specifically, we compare the LLM output y with the corresponding ground truth y', whose extraction is described in Section 4.3. An example of y' is given in Figure 3. The LLM output y is desired to match y' as closely as possible.

[3]https://developers.openai.com/api/docs/models/gpt-oss-20b

The ground truth y' consists of at most one new node and its connecting edges. The motivation behind this setup is to isolate the influence of slicing from other factors, such as the number of elements to be completed. In this setup, checking the correctness of the generated modeling graphs becomes straightforward and reduces to a direct comparison between y and y'. Since only a single ground truth element is considered, no ambiguity arises in matching predicted graph elements to multiple ground truth elements, which would otherwise require additional alignment strategies and graph matching.

```
{ "nodes": [{
        "id": "30",
        "name": "B"
        "eClass": "Class",
        "isfocusedNode": "False"
    }],
  "links": [{
        "source": "27",
        "target": "30",
        "key": "0"
    }]
}
```

Figure 3: Simplified example of ground truth y'.

We employ a set of different evaluation metrics capturing different aspects of correctness, including syntactic, semantic, and structural properties of the generated model completion candidates. All selected metrics are well-established in the literature and can be computed automatically, enabling a reproducible evaluation. Since we additionally only compare the prediction y against a single ground truth y', the results provide a conservative estimate of model completion performance, as other correct predictions may be conceivable, while enabling a scalable evaluation with minimal manual effort. Specifically, we consider BLEU score, cosine similarity of embeddings, and TF-IDF cosine similarity. In addition, we evaluate format, type correctness and structural correctness.

Valid JSON (V_{json}). To assess whether slicing affects the ability of the LLM to produce outputs in the expected format, we measure whether the prediction can be parsed as valid JSON. Minor deviations from the format, such as missing closing brackets, are corrected during post-processing.

Valid Graph (V_{graph}). To assess whether slicing affects the ability of the LLM to generate structurally correct graphs, we measure whether a valid JSON prediction conforms to the expected graph schema, i.e., contains required fields such as `nodes` and `links` with appropriate identifiers.

BLEU score (bleu). We use BLEU score [64] as a measure of content overlap between the prediction y and the ground truth y'. It computes the n-gram overlap between representations. Since the task involves completing a single node and its connected edges, graph comparison reduces to a string-based comparison. Both representations are tokenized by removing punctuation (e.g., brackets and commas). We use $n = 1$, such that the score captures the fraction of tokens in the prediction that appear in the ground truth.

Embedding-Based Cosine Similarity ($\cos_\phi$). While y and y' may differ in their exact wording, resulting in a lower BLEU score, they can still express the same intent; embedding-based cosine similarity captures this semantic equivalence between y and y'. Both strings are preprocessed into a canonical form. The resulting strings are then encoded using a pre-trained BERT-based sentence embedding model (all-MiniLM-L6-v2), resulting in:

$$\cos_\phi(y, y') = \frac{\phi(y) \cdot \phi(y')}{\|\phi(y)\| \cdot \|\phi(y')\|},$$

with $\phi(\cdot)$ denoting the embedding function.

TF–IDF Cosine Similarity ($\cos_{\text{tfidf}}$). We use TF–IDF cosine similarity to emphasize informative tokens and reduce the influence of correctly predicting common terms, making the comparison more sensitive to meaningful differences between y and y'. Both strings are preprocessed into a canonical form. The TF–IDF vectors are constructed as follows [69]:

$$t(y) = \left[\text{tfidf}(t_1, y),\ \text{tfidf}(t_2, y),\ \ldots\right],$$

with

$$\text{tfidf}(t, d) = \text{tf}(t, d) \cdot \log \frac{N}{\text{df}(t)},$$

where $\text{tf}(t, d)$ denotes the term frequency of term t in document d, $\text{df}(t)$ denotes the number of documents in the corpus containing term t, and N denotes the total number of documents in the corpus. In our case: $N = 2$. The similarity is then computed as the cosine between the resulting TF–IDF representations, yielding:

$$\cos_{\text{tfidf}}(y, y') = \frac{t(y) \cdot t(y')}{\|t(y)\| \cdot \|t(y')\|},$$

where $t(\cdot)$ denotes the TF–IDF vector representation.

Type Correctness (type). While the previous metrics focus on content, we further compute *type correctness* [74], indicating whether the predicted element in the generated output y has the same type as the corresponding element from the ground truth y'.

Structural Correctness (struct). We compute structural correctness to assess whether the predicted elements maintain the same connectivity as the ground truth y'. If the ground truth contains one missing node, we evaluate whether the predicted node is connected to the correct neighboring nodes. Not having enough edges or predicting too many edges is penalized.

4.7 Results

RQ1. We report average overall model completion performance for each context-aware slicing criteria introduced in Section 3.3. For statistical significance, we apply a paired one-sided Wilcoxon signed-rank test comparing each slicing approach with identity slicing. The test is performed over all model completion tasks (1,000 models evaluated with two different LLM calls each).

Furthermore, we additionally compare slicing approaches across LLMs and datasets using multiple evaluation metrics, we report format correctness in Figure 4. Overall, most generated graphs (i.e. completed models) conform to the JSON schema across slicing approaches. The identity slicing baseline achieves moderate correctness for LLaMA (over 75% correct), but below 58% of the predictions are format correct according to the JSON format for GPT. Random slicing and LLM-based slicing also perform well in most cases, with a noticeable drop for GPT on the RepairVision dataset. In contrast, radius-based and louvain-based slicing consistently yield the highest correctness scores across both datasets and models. Regarding graph correctness, the identity slicing approach performs worst. Random slicing and LLM-based slicing achieve moderate performance, while louvain-based and radius-based slicing yield comparatively high correctness.

Table 2: Average performance metric values per slicing approach. * ($p < 0.05$) and ** ($p < 0.01$) indicate that the respective approach performs significantly better than identity slicing (paired one-sided Wilcoxon signed-rank test).

Approach	bleu	$\cos_\phi$	$\cos_{\text{tfidf}}$	struct	type
Identity	0.107	0.667	0.229	0.014	0.017
Random	0.165^{**}	0.724^{**}	0.274^{**}	0.010	0.047^{**}
LLM-based	0.184^{**}	0.726^{**}	0.288^{**}	0.041^{**}	0.049^{*}
Louvain-based	0.289^{**}	0.795^{**}	0.389^{**}	0.131^{**}	0.256^{**}
Radius-based	0.337^{**}	0.813^{**}	0.440^{**}	0.171^{**}	0.378^{**}

All other correctness metrics in Table 3 assume a valid JSON schema; invalid outputs receive a score of 0. Table 3 reports the overall average results, highest values are highlighted. In most cases, radius-based slicing achieves the highest scores, particularly on the RepairVision dataset. The louvain-based approach performs competitively. The BLEU score, structural correctness, and type correctness metrics are generally low, whereas the embedding-based cosine similarity scores are comparatively high.

To compare the different approaches, we apply a paired one-sided Wilcoxon signed-rank test with $\alpha = 0.01$. A result is considered statistically significant only if an approach significantly outperforms all other approaches for every metric, dataset, and LLM. In particular, this means that we compute statistical significance separately for each combination of dataset, LLM, and metric, and only report a result significant if all comparisons are significant. More detailed results are provided in our supplementary Website.

Graph-aware slicing methods (louvain-based slicing and radius-based slicing) show the most significant improvements across all datasets (RepairVision and ModelSet) and LLMs, consistently outperforming LLM-based slicing, the identity slicing baseline, and random slicing. Radius-based slicing outperforms louvain-based slicing on the RepairVision dataset. However, LLM-based slicing does not have a statistically significant advantage over either random slicing or the identity slicing across all metrics and datasets.

RQ2. We are further interested in whether slice size, measured in token counts, influences model completion performance. To analyze this effect, we group instances into logarithmically spaced token count bins and compute the median performance within each bin. Logarithmic binning is commonly used for heavy-tailed data such as token counts, to avoid over-representing dense regions at smaller values. This allows us to visualize performance trends across increasing input sizes, as shown in Figure 5.

We report results for three semantic similarity metrics: BLEU score, TF-IDF and embedding-based cosine similarity. Binary metrics such as structural and type correctness are omitted from this analysis, as they are less informative. We additionally compute

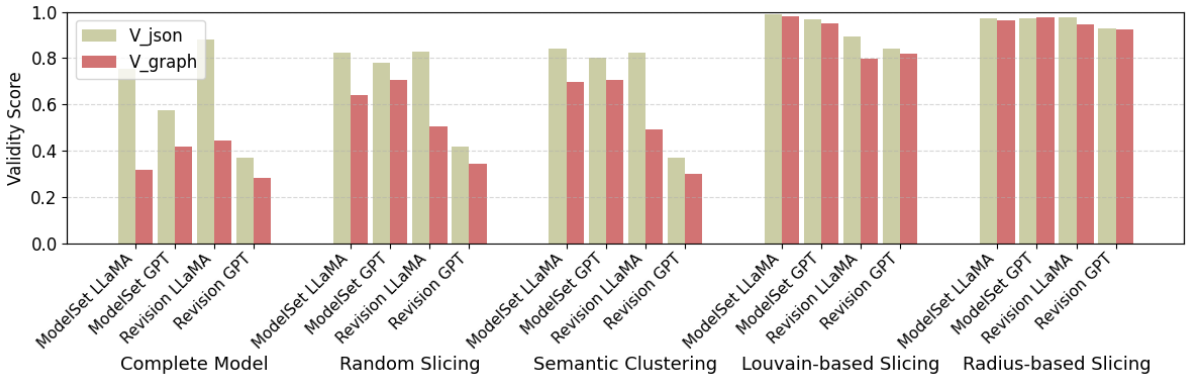


Figure 4: Validity of LLM-generated prediction. Values indicate the fraction of predictions that satisfy each validity criterion (V_{json}: generated LLM output y is a valid JSON, V_{graph}: generated LLM output is a valid graph).

Table 3: Average values for the slicing approaches with respect to different metrics, datasets, and LLMs.

		Identity	Random	LLM-based	Louvain-based	Radius-based
LLaMA-3.1-8B ModelSet	bleu	0.05	0.11	0.14	**0.25**	0.24
	$\cos_{\text{emb}}$	0.51	0.60	0.59	**0.79**	0.76
	$\cos_{\text{tfidf}}$	0.17	0.20	0.22	**0.32**	**0.32**
	Struct.	0.01	0.01	0.04	0.09	**0.10**
	Type	0.00	0.02	0.01	**0.09**	0.08
gpt-oss-20b ModelSet	bleu	0.08	0.17	0.19	**0.26**	**0.26**
	$\cos_{\text{emb}}$	0.34	0.59	0.58	**0.78**	**0.78**
	$\cos_{\text{tfidf}}$	0.15	0.25	0.25	0.34	**0.35**
	Struct.	0.02	0.01	0.03	0.06	**0.09**
	Type	0.02	0.02	0.03	0.05	**0.08**
LLaMA-3.1-8B RepairVision	bleu	0.08	0.10	0.08	0.25	**0.39**
	$\cos_{\text{emb}}$	0.56	0.55	0.41	0.68	**0.80**
	$\cos_{\text{tfidf}}$	0.16	0.20	0.14	0.38	**0.51**
	Struct.	0.00	0.00	0.02	0.16	**0.22**
	Type	0.00	0.02	0.02	0.20	**0.37**
gpt-oss-20b RepairVision	bleu	0.04	0.08	0.05	0.30	**0.40**
	$\cos_{\text{emb}}$	0.24	0.29	0.20	0.69	**0.79**
	$\cos_{\text{tfidf}}$	0.08	0.12	0.08	0.40	**0.52**
	Struct.	0.01	0.00	0.01	0.17	**0.24**
	Type	0.01	0.02	0.02	0.26	**0.42**

the Spearman rank correlations between input token count and performance metrics for each slicing approach and LLM.

For the identity slicing, performance drops with increasing token count across all metrics and LLMs. In particular, performance decreases until around 10^4 tokens, after which it stabilizes at a lower level. With regard to Spearman rank correlations, we observe no consistent overall trend for LLaMA across the evaluated metrics. In contrast, GPT shows a weak negative correlation [70], with ρ values lower between -0.32 (BLEU score) and -0.49 ($\cos_{\text{tfidf}}$). For random slicing, performance decreases steadily, showing a weak negative correlation for both models, with ρ ranging from -0.32 ($\cos_{\phi}$, GPT) to -0.47 ($\cos_{\text{tfidf}}$, LLaMA). For LLM-based slicing, performance decreases up to around 10^4 tokens. Beyond this point, performance shows a slight increase for some metrics. LLM-based slicing shows a moderate negative correlation (-0.48 to -0.52) for LLaMA and a weak negative correlation for GPT. For Louvain-based slicing, performance shows a slight increase up to around $10^{3.5}$ tokens, followed by a decrease for larger token counts. Overall, we observe only a weak negative correlation for LLaMA (ρ: -0.16 to -0.30). In contrast, radius-based slicing exhibits a different pattern, with performance increasing and reaching a peak between 10^3 and $10^{3.4}$ tokens. Accordingly, radius-based slicing shows a weak positive correlation (ρ: 0.06 to 0.23).

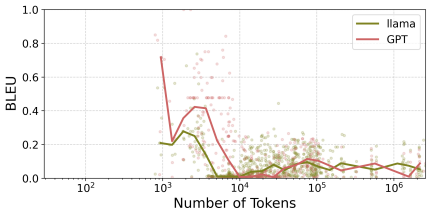
5 Discussion

To better support modelers with more accurate model completion during modeling and maintenance tasks, we study the influence of model slicing on LLM-based model completion performance.

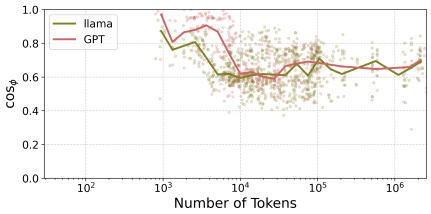
5.1 Findings RQ1

For RQ1, we analyze whether and to what extent slicing improves LLM-based model completion. The results show that providing the complete model as context consistently yields the lowest performance metric scores across datasets and LLMs. Graph-aware slicing methods (louvain-based and radius-based slicing) show the most consistent improvements across both datasets (RepairVision and ModelSet) and LLMs, outperforming LLM-based slicing, random slicing and the identity slicing baseline. This indicates that the choice of slicing strategy has, in fact, a significant impact on model completion performance.

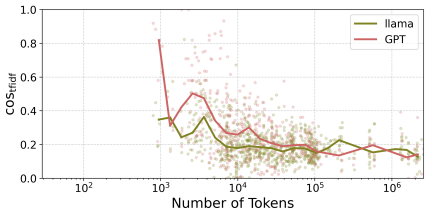
It is important to note that these results not only apply to large models that exceed the context size of the respective LLM, but also to smaller models that fit within the LLM context, as paired comparisons across the same model instances and completion tasks show that slicing significantly improves model completion performance. These findings align with prior work suggesting that large inputs



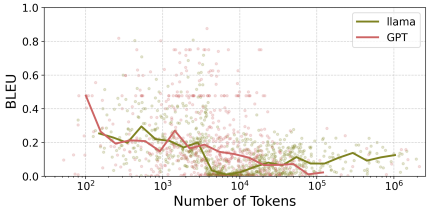
(a) Identity slicing baseline: BLEU score performance depending on the number of tokens.



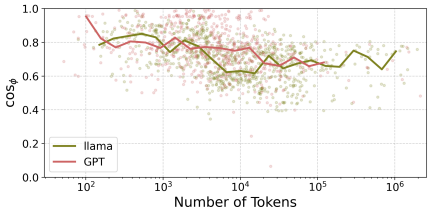
(b) Identity slicing baseline: embedding-based cosine similarity depending on the number of tokens.



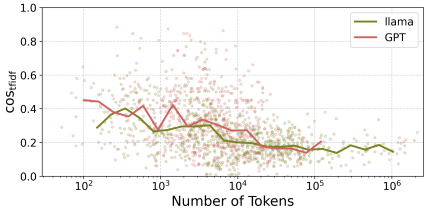
(c) Identity slicing baseline: TF-IDF cosine similarity depending on the number of tokens.



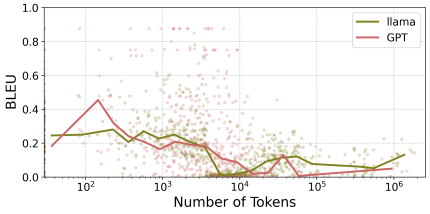
(d) Random slicing baseline: BLEU score performance depending on the number of tokens.



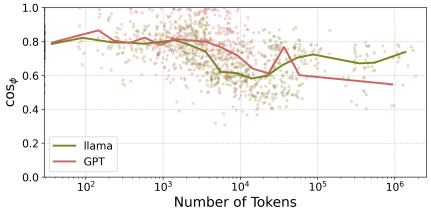
(e) Random slicing baseline: embedding-based cosine similarity depending on the number of tokens.



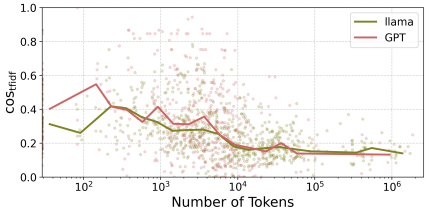
(f) Random slicing baseline: TF-IDF cosine similarity depending on the number of tokens.



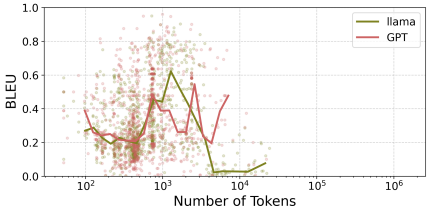
(g) LLM-based slicing: BLEU score performance depending on the number of tokens.



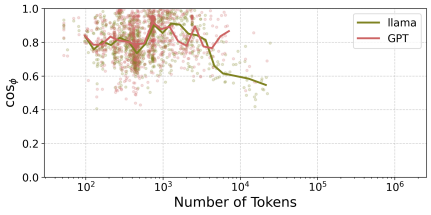
(h) LLM-based slicing: embedding-based cosine similarity depending on number of tokens.



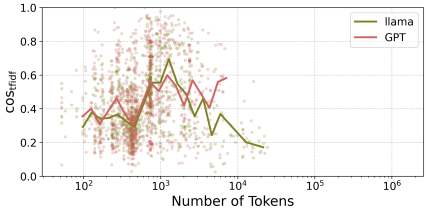
(i) LLM-based slicing: TF-IDF cosine similarity depending on number of tokens.



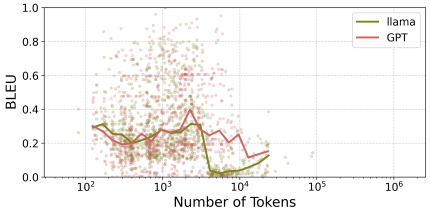
(j) Radius-based slicing: BLEU score performance depending on number of tokens.



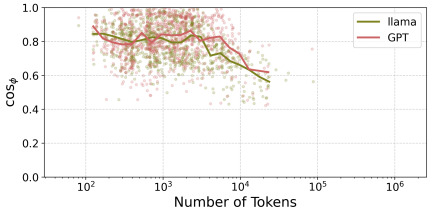
(k) Radius-based slicing: embedding-based cosine similarity depending on number of tokens.



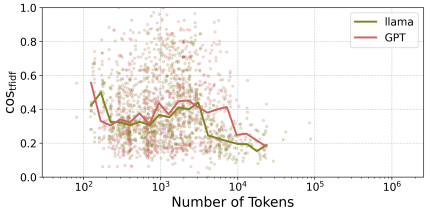
(l) Radius-based slicing: TF-IDF cosine similarity depending on number of tokens.



(m) Louvain-based slicing: BLEU score performance depending on number of tokens.



(n) Louvain-based slicing: embedding-based cosine similarity depending on number of tokens



(o) Louvain-based slicing: TF-IDF cosine similarity depending on number of tokens.

Figure 5: Performance metrics depending on the number of tokens of the respective slice. Instances are grouped into 24 logarithmically spaced token-count bins, and the median performance within each bin is indicated by the line.

can hinder the effective use of task-relevant information even when the relevant information is present in the context [29, 44, 51]. Consistent with this observation, random slicing performs only slightly better and does not always significantly outperform identity slicing, indicating that merely reducing context size is insufficient and that selecting task-relevant information is crucial for improving model completion performance. While these findings cannot be generalized to all LLMs and datasets, we observe similar patterns across 1000 diverse models and both evaluated publicly available LLMs, indicating that the observed effects are not limited to isolated cases.

Beyond the overall trends, a more detailed analysis reveals interesting patterns. On the RepairVision dataset, radius-based slicing further achieves better performance than louvain-based slicing. This could either be due to slice size alone, due to the fact that the RepairVision dataset exhibits a less modular structure leading to larger slices, or because the RepairVision dataset is inherently more difficult to complete. The latter explanation is unlikely, though, as in some cases higher values are observed and no consistent differences between datasets are apparent. To further investigate this, we report additional statistics on our datasets. On average, louvain-based slicing produces slices of around 900 tokens for ModelSet and approximately 4700 tokens for RepairVision. The RepairVision dataset exhibits higher modularity, indicating stronger separation into communities (0.82 for RepairVision, 0.59 for ModelSet). At the same time, RepairVision shows no local clustering, reflecting weak local connectivity (0.00 for RepairVision, 0.20 for ModelSet), and substantially more detected communities (46 for RepairVision, 10 for ModelSet). On RepairVision, we observe an average of 196.54 tokens per node, while ModelSet shows 101.47 tokens per node. RepairVision additionally contains edge labels, suggesting that token size alone may be the main reason.

LLM-based slicing, as a context-aware approach, is the only method that does not consistently achieve better performance across all metrics and datasets. One reason could be that LLM-based slicing requires an additional LLM call to determine relevant model elements. If this step fails to produce parsable output, metric values drop to 0. However, this is not the only reason for the weaker performance. Further analysis (see supplementary Website) shows that LLM-based slicing tends to select large parts of the model graph. Approximately half of the elements in the RepairVision dataset and one third in ModelSet are marked as relevant, resulting in large slices. Even when considered in isolation, i.e., only format-correct results, performance is still not significantly better than the identity baseline across all metrics and datasets.

Answer RQ 1 *Context-aware slicing has a significant impact on model completion performance and should be considered when designing LLM-based model completion approaches. In particular, incorporating graph-aware slicing strategies can improve completion correctness while simultaneously reducing input size.*

5.2 Findings RQ2

In RQ2, we investigated whether the slice size influences model completion performance. In general, the LLMs tend to struggle more with larger software models as shown in Figure 5 (a)-(c). Approaches that produce larger slices, such as identity, random, and LLM-based, appear more sensitive to slice size, with performance decreasing as slice size increases. In contrast, more structured slicing approaches, particularly louvain-based and radius-based, appear less sensitive. Radius-based slicing even shows performance improvements for larger slices. This suggests that incorporating additional nearby context can still be beneficial, indicating that radius-based slicing provides a good conservative starting point from which incrementally additional information can be included.

Interestingly, and not necessarily expected, fewer context tokens tend to correlate with better completion performance. This observation is supported by the findings in RQ1, where LLM-based slicing often produces large slices but shows lower performance. Note that the instruction explicitly constrains completion to the focus area C_m (see Section 4.4). Since we consider a single ground-truth element, this is necessary to ensure a fair evaluation. Otherwise, larger slices would introduce more plausible completion locations, potentially penalizing larger contexts due to multiple valid alternatives being compared against a single ground truth.

Answer RQ 2 *Overall, the results indicate that smaller and more focused contexts lead to better completion performance. This suggests that LLM-based model completion approaches should start with minimal, task-relevant context and incrementally include additional information only when necessary.*

5.3 Threats to Validity

With regard to internal validity, the LLM-based generation procedure may introduce some variability due to stochastic behavior. To mitigate this effect, we evaluated a large number of data points and repeated experiments partially, observing only minor deviations. As LLM performance depends on prompting, we applied structured instructions and used few-shot examples.

Evaluating model completion quality is inherently challenging. While the chosen metrics may not capture all aspects of completion quality, we combine multiple complementary measures to provide a comprehensive yet feasible evaluation. Since we assume a single ground truth despite multiple valid completions potentially existing, the reported results represent a lower bound on performance.

With regard to external validity, the influence of slicing may vary across datasets and LLMs. To mitigate this threat, we evaluate on multiple datasets from different domains, including the real-world RepairVision dataset, and we use a large number of software models. Our goal is to demonstrate the relevance of significant differences, which serve as a basis for further investigations.

6 Conclusion

Software model completion with LLMs aims to support modeling tasks by automatically suggesting relevant model changes. Despite first successful approaches [18, 74, 77], the impact of software model slicing on model completion correctness has remained largely unexplored. Improving correctness reduces manual corrections, further prompting, and enables support for large industrial models.

In this work, we extended a graph-based formalism of software models with slicing criteria from prior literature [55, 67, 74, 77], resulting in a task-independent and metamodel-agnostic formulation. Based on this framework, we compared five slicing approaches, including radius-based, LLM-based, and louvain-based slicing, across multiple metrics, LLMs and datasets.

Our results indicate that smaller, task-relevant contexts consistently improve model completion performance compared to larger contexts or the complete model. In particular, graph-aware slicing approaches outperform less targeted approaches, showing that providing more information does not necessarily lead to better predictions, despite this being an intuitive expectation. These findings suggest that LLM-based software model completion approaches should start with minimal, task-relevant context and incrementally include additional information only when necessary.

Although evaluated in the context of software model completion, these findings can provide guidance for applying slicing in other LLM-based modeling tasks.

References

[1] Henning Agt-Rickauer, Ralf-Detlef Kutsche, and Harald Sack. 2018. DoMoRe–a recommender system for domain modeling. In *Proceedings of the International Conference on Model-Driven Engineering and Software Development*, Vol. 1. Setúbal: SciTePress, 71–82.

[2] Henning Agt-Rickauer, Ralf-Detlef Kutsche, and Harald Sack. 2019. Automated recommendation of related model elements for domain models. In *Model-Driven Engineering and Software Development: 6th International Conference, MODELSWARD 2018, Funchal, Madeira, Portugal, January 22-24, 2018, Revised Selected Papers 6*. Springer, 134–158.

[3] Lissette Almonte, Esther Guerra, Iván Cantador, and Juan de Lara. 2024. Engineering recommender systems for modelling languages: concept, tool and evaluation. *Empirical Software Engineering* 29, 4 (2024), 102.

[4] Kelly Androutsopoulos, David Clark, Mark Harman, Jens Krinke, and Laurence Tratt. 2013. State-based model slicing: A survey. *ACM Computing Surveys (CSUR)* 45, 4 (2013), 1–36.

[5] Sajid Anwer, Lian Wen, Shaoyang Zhang, Zhe Wang, and Yong Sun. 2024. BECIA: a behaviour engineering-based approach for change impact analysis. *International Journal of Information Technology* 16, 1 (2024), 159–168.

[6] Ludovic Apvrille and Bastien Sultan. 2024. System architects are not alone anymore: Automatic system modeling with ai. In *MODELSWARD 2024: 12th International Conference on Model-Based Software and Systems Engineering*, Vol. 1. SCITEPRESS-Science and Technology Publications, 27–38.

[7] Sathurshan Arulmohan, Marie-Jean Meurs, and Sébastien Mosser. 2023. Extracting domain models from textual requirements in the era of large language models. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 580–587.

[8] Afef Awadid and Rémi Boyer. 2023. Supporting Change Impact Analysis in System Architecture Design: Towards a Domain-Specific Modeling Method. In *11th International Conference on Model-Based Software and Systems Engineering (MODELSWARD)*.

[9] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 675–698.

[10] Angela Barriga, Rogardt Heldal, Adrian Rutle, and Ludovico Iovino. 2022. PARMOREL: a framework for customizable model repair. *Software and Systems Modeling* 21, 5 (2022), 1739–1762.

[11] Angela Barriga, Adrian Rutle, and Rogardt Heldal. 2019. Personalized and automatic model repairing using reinforcement learning. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 175–181.

[12] Angela Barriga, Adrian Rutle, and Rogardt Heldal. 2022. AI-powered model repair: an experience report—lessons learned, challenges, and opportunities. *Software and Systems Modeling* 21, 3 (2022), 1135–1157.

[13] Christopher M Bishop. [n. d.]. *Pattern recognition and machine learning*. Vol. 4. Springer.

[14] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.

[15] Arnaud Blouin, Benoît Combemale, Benoit Baudry, and Olivier Beaudoux. 2015. Kompren: modeling and generating model slicers. *Software & Systems Modeling* 14, 1 (2015), 321–337.

[16] Loli Burgueño, Robert Clarisó, Sébastien Gérard, Shuai Li, and Jordi Cabot. 2021. An NLP-based architecture for the autocompletion of partial domain models. In *Proceedings of the International Conference on Advanced Information Systems Engineering*. Springer, 91–106. doi:10.1007/978-3-030-79382-1_6

[17] Javier Cámara, Javier Troya, Lola Burgueño, and Antonio Vallecillo. 2023. On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML. *Software and Systems Modeling* 22, 3 (2023), 781–793.

[18] Meriem Ben Chaaben, Lola Burgueño, Istvan David, and Houari Sahraoui. 2024. On the Utility of Domain Modeling Assistance with Large Language Models. *arXiv preprint arXiv:2410.12577* (2024).

[19] Meriem Ben Chaaben, Lola Burgueño, and Houari Sahraoui. 2023. Towards using few-shot prompt learning for automating model completion. In *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 7–12.

[20] Boqi Chen, Ou Wei, Bingzhou Zheng, and Gunter Mussbacher. 2025. Accurate and Consistent Graph Model Generation from Text with Large Language Models. In *2025 ACM/IEEE 28th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 130–141.

[21] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2008. Meta-model differences for supporting model co-evolution. In *Proceedings of the 2nd Workshop on Model-Driven Software Evolution-MODSE*, Vol. 1.

[22] Aaron Conrardy and Jordi Cabot. 2024. From image to uml: first results of image based uml diagram generation using llms. *arXiv preprint arXiv:2404.11376* (2024).

[23] Carlos Durá Costa, José Antonio Hernández López, and Jesús Sánchez Cuadrado. 2024. ModelMate: A recommender for textual modeling languages based on pre-trained language models. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*. 183–194.

[24] Shuiguang Deng, Dongjing Wang, Ying Li, Bin Cao, Jianwei Yin, Zhaohui Wu, and Mengchu Zhou. 2016. A recommendation system to facilitate business process modeling. *IEEE transactions on cybernetics* 47, 6 (2016), 1380–1394.

[25] Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong T Nguyen, and Alfonso Pierantonio. 2023. MemoRec: a recommender system for assisting modelers in specifying metamodels. *Software and Systems Modeling* 22, 1 (2023), 203–223.

[26] Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong T Nguyen, and Riccardo Rubei. 2025. On the use of large language models in model-driven engineering: J. Di Rocco et al. *Software and Systems Modeling* 24, 3 (2025), 923–948.

[27] Juri Di Rocco, Claudio Di Sipio, Phuong T Nguyen, Davide Di Ruscio, and Alfonso Pierantonio. 2022. Finding with nemo: a recommender system to forecast the next modeling operations. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*. 154–164.

[28] Claudio Di Sipio, Juri Di Rocco, Davide Di Ruscio, and Phuong T Nguyen. 2023. MORGAN: a modeling recommender system based on graph kernel. *Software and Systems Modeling* (2023), 1–23.

[29] Yufeng Du, Minyang Tian, Srikanth Ronanki, Subendhu Rongali, Sravan Bodapati, Aram Galstyan, Azton Wells, Roy Schwartz, Eliu A Huerta, and Hao Peng. 2025. Context length alone hurts LLM performance despite perfect retrieval. *arXiv preprint arXiv:2510.05381* (2025).

[30] Tobias Eisenreich, Nicholas Friedlaender, and Stefan Wagner. 2025. Leveraging Large Language Models for Use Case Model Generation from Software Requirements. *arXiv preprint arXiv:2511.09231* (2025).

[31] Tobias Eisenreich, Sandro Speth, and Stefan Wagner. 2024. From requirements to architecture: An ai-based journey to semi-automatically generate software architectures. In *Proceedings of the 1st International Workshop on Designing Software*. 52–55.

[32] Akil Elkamel, Mariem Gzara, and Hanêne Ben-Abdallah. 2016. An UML class recommender system for software design. In *Proceedings of the International Conference of Computer Systems and Applications (AICCSA)*. IEEE, 1–8.

[33] Alessio Ferrari, Sallam Abualhaija, and Chetan Arora. 2024. Model generation with LLMs: From requirements to UML sequence diagrams. In *2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW)*. IEEE, 291–300.

[34] Dominik Fuchß, Tobias Hey, Jan Keim, Haoyu Liu, Niklas Ewald, Tobias Thirolf, and Anne Koziolek. 2025. LiSSA: toward generic traceability link recovery through retrieval-augmented generation. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering. ICSE*, Vol. 25.

[35] Polydoros Giannouris and Sophia Ananiadou. 2025. NOMAD: A Multi-Agent LLM System for UML Class Diagram Generation from Natural Language Requirements. *arXiv preprint arXiv:2511.22409* (2025).

[36] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[37] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. 2008. Automatability of coupled evolution of metamodels and models in practice. In *International conference on model driven engineering languages and systems*. Springer, 645–659.

[38] Ludovico Iovino, Angela Barriga, Adrian Rutle, Rogardt Heldal, et al. 2020. Model repair with quality-based reinforcement learning. *Journal of Object Technology* 19, 2 (2020).

[39] Huzefa Kagdi, Jonathan I Maletic, and Andrew Sutton. 2005. Context-free slicing of UML class models. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 635–638.

[40] Dhikra Kchaou, Nadia Bouassida, and Hanêne Ben-Abdallah. 2017. UML models change impact analysis using a text similarity technique. *IET Software* 11, 1 (2017), 27–37.

[41] Stefan Kögel. 2017. Recommender system for model driven software development. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 1026–1029.

[42] Stefan Kögel, Raffaela Groner, and Matthias Tichy. 2016. Automatic Change Recommendation of Models and Meta Models Based on Change Histories.. In *ME@ MoDELS*. 14–19.

[43] Vinay Kulkarni, Sreedhar Reddy, Souvik Barat, and Jaya Dutta. 2023. Toward a Symbiotic Approach Leveraging Generative AI for Model Driven Engineering. In *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 184–193. doi:10.1109/MODELS58315.2023.00039

[44] Yury Kuratov, Aydar Bulatov, Petr Anokhin, Ivan Rodkin, Dmitry Sorokin, Artyom Sorokin, and Mikhail Burtsev. 2024. Babilong: Testing the limits of llms with long context reasoning-in-a-haystack. *Advances in Neural Information Processing Systems* 37 (2024), 106519–106554.

[45] Tobias Kuschke and Patrick Mäder. 2017. RapMOD—In Situ Auto-Completion for Graphical Models. In *Proceedings of the International Conference on Software Engineering (ICSE): Companion Proceedings*. IEEE, 303–304.

[46] Tobias Kuschke, Patrick Mäder, and Patrick Rempel. 2013. Recommending auto-completions for software modeling activities. In *International conference on model driven engineering languages and systems*. Springer, 170–186.

[47] Jaiprakash T Lallchandani and Rajib Mall. 2008. Slicing UML architectural models. *ACM SIGSOFT Software Engineering Notes* 33, 3 (2008), 1–9.

[48] Jaiprakash T Lallchandani and Rajib Mall. 2011. A dynamic slicing technique for UML architectural models. *IEEE Transactions on Software Engineering* 37, 6 (2011), 737–771.

[49] Alexander Lauer, Jens Kosiol, and Gabriele Taentzer. 2023. Empowering model repair: a rule-based approach to graph repair without side effects. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 831–840.

[50] Ying Li, Bin Cao, Lida Xu, Jianwei Yin, Shuiguang Deng, Yuyu Yin, and Zhaohui Wu. 2013. An efficient recommendation method for improving business process modeling. *IEEE Transactions on Industrial Informatics* 10, 1 (2013), 502–513.

[51] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.

[52] José Antonio Hernández López, Javier Luis Cánovas Izquierdo, and Jesús Sánchez Cuadrado. 2022. Modelset: a dataset for machine learning in model-driven engineering. *Software and Systems Modeling* (2022), 1–20.

[53] José Antonio Hernández López, Carlos Durá, and Jesús Sánchez Cuadrado. 2023. Word embeddings for model-driven engineering. In *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 151–161.

[54] José Antonio Hernández López, Riccardo Rubei, Jesús Sánchez Cuadrado, and Davide Di Ruscio. 2022. Machine learning methods for model classification: a comparative study. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*. 165–175.

[55] Dipeeka Luitel, Shiva Nejati, and Mehrdad Sabetzadeh. 2024. Requirements-driven slicing of simulink models using LLMs. In *2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW)*. IEEE, 72–82.

[56] Nuno Macedo, Tiago Jorge, and Alcino Cunha. 2016. A feature-based classification of model repair approaches. *IEEE Transactions on Software Engineering* 43, 7 (2016), 615–640.

[57] Bennett Mackenzie, Vera Pantelic, Gordon Marks, Stephen Wynn-Williams, Gehan Selim, Mark Lawford, Alan Wassyng, Moustapha Diab, and Feisel Weslati. 2020. Change impact analysis in simulink designs of embedded systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1274–1284.

[58] Luciano Marchezan, Roland Kretschmer, Wesley KG Assunção, Alexander Reder, and Alexander Egyed. 2023. Generating repairs for inconsistent models. *Software and Systems Modeling* 22, 1 (2023), 297–329.

[59] Tom Mens, Ragnhild Van Der Straeten, and Maja D'Hondt. 2006. Detecting and resolving model inconsistencies using transformation dependency analysis. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 200–214.

[60] Patrick Mäder, Tobias Kuschke, and Mario Janke. 2021. Reactive Auto-Completion of Modeling Activities. *Transactions on Software Engineering* 47, 7 (2021), 1431–1451. doi:10.1109/TSE.2019.2924886

[61] Nebras Nassar, Hendrik Radke, and Thorsten Arendt. 2017. Rule-based repair of EMF models: An automated interactive approach. In *International conference on theory and practice of model transformations*. Springer, 171–181.

[62] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, Lars Grunske, and Timo Kehrer. 2021. History-Based Model Repair Recommendations. *Transactions of Software Engineering Methodology* 30, 2, Article 15 (2021). doi:10.1145/3419017

[63] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, and Timo Kehrer. 2018. ReVision: A tool for history-based model repair recommendations. In *Proceedings of the International Conference on Software Engineering (ICSE): Companion Proceedings*. ACM, 105–108. doi:10.1145/3419017

[64] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[65] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Alejandro Padilla. 2021. A Study of Feature Scattering in the Linux Kernel. *IEEE Transactions on Software Engineering* 47, 1 (2021), 146–164.

[66] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wasowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of Variability Models and Related Software Artifacts: A Fresh Look at Evolution Patterns in the Linux Kernel. *Empirical Software Engineering* 21, 4 (2016), 1744–1793.

[67] Christopher Pietsch, Manuel Ohrndorf, Udo Kelter, and Timo Kehrer. 2017. Incrementally slicing editable submodels. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 913–918.

[68] Alexander Reder and Alexander Egyed. 2012. Computing repair trees for resolving inconsistencies in design models. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 220–229.

[69] Gerard Salton and Christopher Buckley. 1988. Term-weighting approaches in automatic text retrieval. *Information processing & management* 24, 5 (1988), 513–523.

[70] Patrick Schober, Christa Boer, and Lothar A Schwarte. 2018. Correlation coefficients: appropriate use and interpretation. *Anesthesia & analgesia* 126, 5 (2018), 1763–1768.

[71] Louis Richard Timperley, Lucy Berthoud, Chris Snider, and Theo Tryfonas. 2025. Assessment of large language models for use in generative design of model based spacecraft system architectures. *Journal of Engineering Design* 36, 4 (2025), 550–570.

[72] Christof Tinnes, Timo Kehrer, Mitchell Joblin, Uwe Hohenstein, Andreas Biesdorf, and Sven Apel. 2023. Mining domain-specific edit operations from model repositories with applications to semantic lifting of model differences and change profiling. *Automated Software Engineering* 30, 2 (2023), 17.

[73] Christof Tinnes, Wolfgang Rössler, Uwe Hohenstein, Torsten Kühn, Andreas Biesdorf, and Sven Apel. 2022. Sometimes you have to treat the symptoms: tackling model drift in an industrial clone-and-own software product line. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1355–1366.

[74] Christof Tinnes, Alisa Welter, and Sven Apel. 2025. Software Model Evolution with Large Language Models: Experiments on Simulated, Public, and Industrial Datasets. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 950–962. doi:10.1109/ICSE55347.2025.00112

[75] Yuan Wang, Ning Ge, Jiangxi Liu, Zhilong Cao, Zheping Chen, and Chunming Hu. 2025. Generating SysML Behavior Models via Large Language Models: an Empirical Study. In *Proceedings of the 16th International Conference on Internetware*. 366–377.

[76] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 172–184.

[77] Alisa Welter, Christof Tinnes, and Sven Apel. 2026. Multi-Location Software Model Completion. (2026).

[78] Martin Weyssow, Houari Sahraoui, and Eugene Syriani. 2022. Recommending metamodel concepts during modeling activities with pre-trained language models. *Software and Systems Modeling* 21, 3 (2022), 1071–1089. doi:10.1007/s10270-022-00975-5

[79] Chi Xiao, Daniel Ståhl, and Jan Bosch. 2025. UML Sequence Diagram Generation: A Multi-Model, Multi-Domain Evaluation. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 272–283. doi:10.1109/ICSE-SEIP66354.2025.00030

[80] Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tegawendé F Bissyandé, Claire Le Goues, and Shunfu Jin. 2024. Multi-objective fine-tuning for enhanced program repair with llms. *arXiv preprint arXiv:2404.12636* (2024).

[81] He Ye and Martin Monperrus. 2024. Iter: Iterative neural repair for multi-location patches. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.