Resolving Prompt Conflicts in Text-to-X Models with Abstract Argumentation

Jonas Klein, Matthias Thimm

FernUniversität in Hagen, Hagen, Germany

Abstract

This work addresses conflicts in natural-language input prompts for generative models. To resolve such conflicts in a reliable and explainable way, this work combines modern Large Language Models (LLMs) with abstract argumentation. The proposed multi-step system uses LLM agents to extract descriptive elements, detect conflicts, resolve them using abstract argumentation algorithms, and recompose a coherent, conflict-free prompt. To evaluate the proposed system, five datasets with conflicting prompts for a text-to-image model are generated, and a comprehensive experimental evaluation is conducted. The results demonstrate reliable conflict identification and resolution, achieving a resolution rate of up to 93.4%, and producing prompts highly similar to conflict-free ground truth (F1 BERTScore of up to 0.964).

Keywords

Large Language Models, Abstract Argumentation, Conflict Resolution

1. Introduction

Recent advances in generative artificial intelligence have launched a new era of digital content creation. Bridging the gap between natural language and other modalities has contributed significantly to the success of modern generative models. Modern generative models, often referred to as text-to-X models, where X denotes the target output modality such as text, images, music, or video, enable users to generate novel content through simple text-based instructions known as prompts [1]. An important factor in the performance of such systems is the design of these prompts, i. e., the way in which input is presented to the model. It has been shown that prompt design influences the quality of generation, as tailored prompts help models better capture user intentions and yield higher-quality outputs [2, 3]. However, handling these prompts in practice is challenging. Arguably, one of the biggest problems is the semantic alignment between user input and model output [4, 5]. While a model's ability to follow prompts is crucial for practical usability, at the same time, prompt following should not be thought of as unconditional compliance. Mechanisms are needed to distinguish benign instructions from harmful ones that should be rejected, constrained, or transformed. Several works address this problem at the model level, for example, with alignment techniques such as reinforcement learning from human feedback [6]. However, these methods are not always feasible, as they may require access to the model or be very resource-intensive. For this reason, much simpler mechanisms are often employed in practice. Such mechanisms include prompt filtering, prompt sanitization or rewriting, and output classification [7]. In particular, prompt rewriting provides a simple alternative in black-box settings, where unsafe prompts are transformed into safer variants while preserving as much benign semantic content as possible [7].

In this paper, we consider the examples mentioned above, in a more general sense, as *prompt conflicts* and propose a formal and logically grounded approach to modelling and resolving conflicts in prompts, by using approaches to formal argumentation [8]. One of the most influential theories in this area is the pioneering work on abstract argumentation by Dung [9], which introduced *abstract argumentation frameworks* to model the interplay among arguments and various semantics to determine

SAFA'26: Sixth International Workshop on Systems and Algorithms for Formal Argumentation, September, 2026, Barcelona, Spain

*Corresponding author.

jonas.klein@fernuni-hagen.de (J. Klein); matthias.thimm@fernuni-hagen.de (M. Thimm)

0009-0000-6521-5991 (J. Klein); 0000-0002-8157-1053 (M. Thimm)

their acceptability. Argumentation scenarios are represented as directed graphs in which vertices represent arguments and attacks between arguments are modeled as directed edges. Following the recent line of research on combining LLMs and formal argumentation [10], we propose the combination of modern LLM agents with abstract argumentation in a multi-step system to resolve prompt conflicts for text-based generative models. To be precise, different LLM agents are used to (1) extract descriptive elements from natural-language inputs, (2) identify conflicts among these elements, and (3) recompose the conflict-free elements into a coherent prompt after resolution. Framing conflict resolution in text elements as an argumentation scenario enables the use of sound, complete algorithms to determine sets of mutually accepted elements. By combining LLMs and formal argumentation, we obtain a flexible, reliable, and explainable system for resolving conflicts in user prompts for text-based generative models.

To summarize, the contributions of this work are:

- Development of a multi-step approach that integrates LLMs and abstract argumentation to resolve conflicts in prompts for text-based generative models.
- Comprehensive experimental evaluation of the system using five different datasets.
- The experiments show that the proposed approach successfully identifies and resolves conflicts in prompts on the examined datasets, achieving a resolution rate of up to 93.4%, with high similarity to the original prompts (F1 BERTScore of up to 0.964).

The remainder of this paper is structured as follows. In Section 2, the relevant preliminaries with regard to abstract argumentation and large language models are discussed. In Section 3, related work is outlined. Section 4 comprises an overview of the developed approach and methodology. Section 5 presents an experimental analysis, and the conclusion is presented in Section 6.

2. Preliminaries

In the following, an overview of the fundamentals of abstract argumentation on the one hand, and of text-to-image models as well as large language models on the other hand, is provided.

2.1. Abstract Argumentation

An *abstract argumentation framework* is a tuple $\mathsf{AF} = (\mathsf{A}, \mathsf{R})$ where A is a set of arguments and R is a relation $\mathsf{R} \subseteq \mathsf{A} \times \mathsf{A}$. For two arguments $a, b \in \mathsf{A}$ the relation $a\mathsf{R}b$ means that argument a attacks argument b. For $a \in \mathsf{A}$ define $a^- = \{b \mid b\mathsf{R}a\}$ and $a^+ = \{b \mid a\mathsf{R}b\}$. We say that a set $S \subseteq \mathsf{A}$ *defends* an argument $b \in \mathsf{A}$ if for all a with $a\mathsf{R}b$ then there is $c \in S$ with $c\mathsf{R}a$.

Semantics are given to abstract argumentation frameworks by means of extensions [9]. An extension E is a set of arguments $E \subseteq \mathsf{A}$ that is intended to represent a coherent point of view on the argumentation modelled by AF. Arguably, the most important property of a semantics is its admissibility. An extension E is called *admissible* if and only if

1. E is *conflict-free*, i. e., there are no arguments $a, b \in E$ with $a\mathsf{R}b$ and
2. E *defends* every $a \in E$,

and it is called *complete* (CO) if, additionally, it satisfies

3. if E defends a then $a \in E$.

Different types of classical semantics can be phrased by imposing further constraints. In particular, a complete extension E

- is *grounded* (GR) if and only if E is minimal,
- is *preferred* (PR) if and only if E is maximal, and
- is *stable* (ST) if and only if $\mathsf{A} = E \cup \{b \mid \exists a \in E : a\mathsf{R}b\}$.

All statements on minimality/maximality are meant to be with respect to set inclusion. Note that the grounded extension is uniquely determined and that stable extensions may not exist [9].

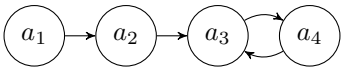

Figure 1: Abstract argumentation framework AF_1 from Example 1.

Example 1. Consider the abstract argumentation framework AF_1 depicted as a directed graph in Figure 1. In AF_1 there are three complete extensions E_1, E_2, E_3 defined via

$$E_1 = \{a_1\}$$
$$E_2 = \{a_1, a_3\}$$
$$E_3 = \{a_1, a_4\}$$

E_1 is also grounded and E_2 and E_3 are both stable and preferred.

2.2. Large Language Models

Large language models (LLMs) are neural networks trained on self-supervised objectives (for example, next-token prediction) over large, web-scale text corpora. After pretraining, the models are typically further trained or fine-tuned with task-specific data or reinforcement learning from human feedback to better align model behaviour with user intent [6]. Modern LLMs use the *Transformer* architecture as a backbone [11]. The *attention mechanism* enables efficient modeling of long-range dependencies and allows for parallelization of the training. Empirically, performance for various natural language processing tasks improves with increases in model size, data, and compute [12]. As a result, model parameters were scaled up massively, to hundreds of billions, showing strong few-shot and zero-shot capabilities [13].

LLMs have achieved strong performance across many benchmarks. For example, recent models obtain high scores on broad-coverage evaluations such as *MMLU* [14] and *BIG-bench* [15], competitive results on mathematical reasoning benchmarks such as *GSM8K* [16], and strong performance on programming evaluations such as *HumanEval* [17]. However, benchmark performance does not necessarily imply reliable real-world behaviour. LLM outputs can be factually incorrect or hallucinated [18]. Models may also inherit or amplify social biases and toxic content present in their training data [19]. Moreover, they can memorize and expose sensitive training examples, raising privacy concerns [20]. Furthermore, their training and deployment involve high computational and environmental costs [21].

For this work, a particularly relevant development is the deployment of LLMs as agents. In general, an agent is commonly understood as a system that perceives an environment and acts within it [22]. In LLM-based agents, the language model often serves as a central controller that coordinates actions such as tool calls, API requests, database queries, code execution, or communication with other agents [23]. This extends LLMs from passive text-generation systems to interactive systems that can iteratively plan, act, observe the consequences of their actions, and revise subsequent behaviour.

3. Related Work

In text-to-X generation, prompt engineering and prompt optimization have been studied as important factors for improving the alignment between user intent and generated output [2, 3, 4]. Safety-oriented approaches such as the Universal Prompt Optimizer [7] for safe text-to-image generation transform unsafe prompts into safer variants while aiming to preserve their semantic content. To be precise, first the authors use an LLM to preprocess toxic prompts to produce a dataset composed of toxic-clean dataset pairs. In a second step, another LLM is fine-tuned on the dataset produced in the first step to learn to rewrite toxic prompts. The instruction hierarchy proposed by Wallace et al. [24] explicitly assigns priorities to instructions from different sources and trains models to ignore lower-priority instructions

in case of conflict. More precisely, the authors classify prompts as aligned or misaligned with each other. When multiple instructions are presented to the model, the lower-privileged instructions that are misaligned with higher-privileged ones are ignored. In a related direction, StruQ [25] separates trusted instructions from untrusted data to make attacks like prompt injection less effective. This system is made of (1) a front-end that formats a prompt and user data into a special format, and (2) a fine-tuned LLM that can produce high-quality outputs from these formatted inputs. The core idea is that the LLM will only follow instructions given in the prompt part of the query, in contrast, for example, to prompts in the data part. ReasAlign [26] addresses malicious prompts by reasoning over conflicting instructions, while trying to preserve the intended user task. The authors instruct the LLM to follow guidelines for a so-called structured reasoning process. First, the problem is analyzed and decomposed into multiple subtasks. Following, the LLM tries to extract useful information from external data and identify conflicting injection instructions while fulfilling the original user task.

Unlike the approaches described above, the approach presented in this work does not require resource-intensive training or fine-tuning of the LLM models used. Furthermore, the models are not utilized to resolve conflicts. Rather, LLM agents are solely used for the natural-language tasks, namely extracting useful information, identifying potential conflicts, and generating revised prompts.

Recent work combines this formal argumentation with LLMs. Argumentative LLMs [27] construct argumentation frameworks from natural-language decision problems and use formal reasoning to support explainable and contestable decisions. ArgEval [10] extends this idea towards globally contestable decision support by constructing reusable argumentation structures for decision options. In [28], the authors propose a simple pipeline (MQArgEng) to test the effectiveness of computational argumentation semantics in enhancing LLM performance. Given a user prompt, an LLM agent is instructed to generate short replies to the user prompt and list three supporting arguments for each generated answer. Afterwards, another LLM agent is used to generate the attack relation. The resulting AF is given to an external solver to determine the accepted arguments. Then, the original input is augmented with accepted arguments to produce the final reply. Another related direction investigates whether LLMs themselves can perform argumentation computation. Li et al. [29] study LLMs on the task of computing extensions under different abstract argumentation semantics.

4. Conflict Detection and Resolution using Abstract Argumentation

In this work, *prompt conflict* denotes two forms of conflict: *internal* and *external*. Internal conflicts occur within the prompt itself, as in "an empty room full of furniture." External conflicts arise when a prompt violates constraints on the model's permitted output. For example, "Create an image of a desk with a MacBook on it" conflicts with a constraint prohibiting brands or branded products. To resolve such cases, we propose a multi-step pipeline that separates conflict detection from conflict resolution: large language models handle natural-language analysis, while abstract argumentation solving determines conflict-free prompts for downstream generation tasks.

Figure 4 provides an overview of the system, consisting of multiple stages: (1) Descriptive Element Extraction, (2) Conflict Identification, (3) Argumentation Framework Construction, (4) Conflict Resolution via an External Argumentation Solver, and (5) Conflict-Free Prompt Generation.

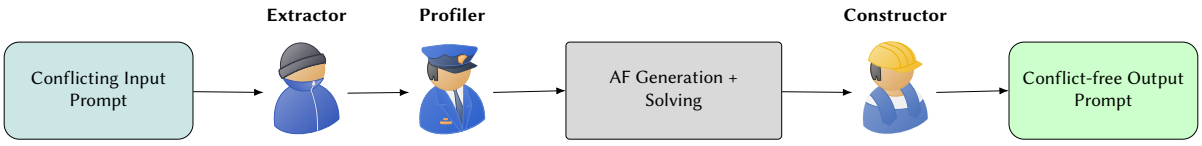

Figure 2: Overview of the different stages of the proposed pipeline. The LLM agents are represented as icons. First, the *Extractor* identifies descriptive elements in the input prompt. The *Profiler* detects conflicts among them. Next, an AF is constructed based on the conflicts, and a solver determines the accepted elements. Finally, the *Constructor* generates a conflict-free prompt for the downstream model.

Algorithm 1: Overview of Argumentation-based Prompt Conflict Resolution

Input: input prompt P; constraint framework $\mathsf{AF_c}$; constraint explanations EX; type information $(T, \tau, \rightarrow_T)$; semantics σ; computational task p; maximum number of retries $r_{\max}$

Output: conflict-free prompt P^*

```
// Stage 1: Descriptive Element Extraction (Algorithm 2)
```
$A \leftarrow \text{DescriptiveElementExtraction}(P, r_{\max})$;
```
// Stage 2: Conflict Identification (Algorithm 3)
```
$\widehat{M} \leftarrow \text{ConflictIdentification}(A, \mathsf{AF_c}, \text{EX}, r_{\max})$;
```
// Stage 3: Argumentation Framework Construction (Algorithm 4)
```
$\mathsf{AF_f} \leftarrow \text{ArgumentationFrameworkConstruction}(A, \mathsf{AF_c}, \widehat{M}, T, \tau, \rightarrow_T)$;
```
// Stage 4: Conflict Resolution (Algorithm 5)
```
$A^* \leftarrow \text{ConflictResolutionViaSolver}(\mathsf{AF_f}, A, \sigma, p)$;
```
// Stage 5: Conflict-Free Prompt Generation (Algorithm 6)
```
$P^* \leftarrow \text{ConflictFreePromptGeneration}(P, A^*)$;

return P^*;

In the first stage, all descriptive elements are extracted from the input prompt using an LLM agent (*Extractor*). The extracted elements are then analyzed by a second LLM agent (*Profiler*) to identify conflicts. Based on the identified conflicts, an AF is constructed. This AF is subsequently passed to an abstract argumentation solver, which computes the set of descriptive elements that can be accepted under a given semantics. Finally, using these accepted elements, another LLM agent (*Constructor*) generates a conflict-free prompt that preserves the style and wording of the original. This conflict-free prompt can then be passed to a downstream model. The complete processing pipeline is summarized in Algorithm 1. The individual steps are explained in detail below.

Descriptive Element Extraction The goal of this stage is to extract meaningful and descriptive elements from the input prompt. What is considered meaningful or descriptive is highly dependent on the use case. Generally speaking, meaningful and descriptive elements are prompt components that convey semantic content, which is relevant for the specific use case, such as entities, attributes, or relationships between entities. Formally, we define the problem of the descriptive element extraction as follows. Let Σ denote the vocabulary of all possible words, and let Σ^* be the set of all finite sequences of words over Σ. A prompt is then represented as an element

$$P = (w_1, w_2, \ldots, w_n) \in \Sigma^*,$$

that is, a finite ordered sequence of words $w_i \in \Sigma$. From a given prompt P, we define the set of all contiguous subsequences as

$$\mathcal{T}(P) = \{(w_i, w_{i+1}, \ldots, w_j) \mid 1 \leq i \leq j \leq n\}.$$

This set contains all possible substrings of the prompt that preserve the original word order. To select those subsequences that possess a descriptive character, an oracle

$$O_P : \mathcal{T}(P) \rightarrow \{0, 1\},$$

is defined, which evaluates whether a given subsequence is considered descriptive. Based on this oracle, we want to identify:

$$D(P) = \{T \in \mathcal{T}(P) \mid O_P(T) = 1\}.$$

The function D maps each prompt P to a subset of its contiguous subsequences, namely those which satisfy the descriptive criterion of O_P. The set $D(P) = \{d_1, ..., d_m\}$ is the set of *descriptors* in P. In

Algorithm 2: Descriptive Element Extraction with Repair and Retry

Input: input prompt P; maximum number of retries $r_{\max}$
Output: descriptor set A

```
r ← 0;
Â ← Extractor(P);
while ¬ ValidDescriptors(Â) do
    Â ← RepairDescriptors(P, Â);
    if ¬ ValidDescriptors(Â) then
        if r ≥ r_max then
            return ⊥;
        r ← r + 1;
        Â ← Extractor(P);
A ← ParseDescriptors(Â);
return A;
```

the presented approach, an LLM agent, referred to as the *Extractor*, is utilized to construct the set $D(P)$. An instruction prompt specifies which elements are to be extracted and how the output of the *Extractor* should be structured, by providing multiple paired examples of user prompts and the desired extracted descriptors. As input, the agent receives the prompt P and is instructed to return a list of descriptors A contained in P. The extracted descriptors then serve as input for the subsequent stage, in which conflicts are identified. Since the output of the *Extractor* is generated by an LLM, it may not conform to the expected structured format. Therefore, the output is validated and, if necessary, repaired. Here, $\mathrm{ValidDescriptors}$ checks whether the output conforms to the expected descriptor-list format and can be parsed successfully. If the repaired output is still invalid, the extraction is retried up to $r_{\max}$ times. The complete procedure is shown in Algorithm 2.

In the following, we give a simple example to demonstrate the *Extractor*.

Example 2. Let's consider the input prompt $P =$ An empty room full of furniture. The agent may extract the following descriptors: $A = \{\text{empty room},\ \text{full of furniture}\}$.

Conflict Identification In this phase, conflicts among the previously extracted descriptors are identified and returned in a structured form for further processing in the subsequent stage. As introduced above, we distinguish between two types of conflicts: (1) *internal conflicts* and (2) *external conflicts*. Internal conflicts are conflicts *within* the set of extracted descriptors. Given the descriptor set $A = \{a_1, \ldots, a_k\}$, the goal is to identify pairs of distinct descriptors $(a_i, a_j) \in A \times A$ with $i \neq j$ such that a_i and a_j are conflicting. External conflicts are conflicts between extracted descriptors and predefined constraints. We model these constraints as an abstract argumentation framework $\mathsf{AF_c} = (\mathsf{E}, \mathsf{R}')$, where $\mathsf{E} = \{e_1, \ldots, e_\ell\}$ is the set of constraint arguments and $\mathsf{R}' \subseteq \mathsf{E} \times \mathsf{E}$ is the attack relation between constraint arguments. Analogously to internal conflicts, the goal is to identify pairs $(a_i, e_j) \in A \times \mathsf{E}$ such that descriptor a_i conflicts with constraint argument e_j. Analogously, for symmetrical case $(e_j, a_i) \in \mathsf{E} \times A$.

Let $U = A \cup \mathsf{E}$ denote the set of all elements considered in this stage. We define a conflict relation $C \subseteq U \times U$ such that, for $x, y \in U$,

$$(x, y) \in C \quad \Longleftrightarrow \quad x \text{ is in conflict with } y.$$

This relation is analogous to the attack relation in abstract argumentation frameworks, as introduced in Section 2.1. For each element $x \in U$, we define the mapping

$$M(x) = \{\, y \in U \mid (x, y) \in C \,\},$$

Algorithm 3: Conflict Identification with Repair and Retry

Input: descriptor set A; constraint framework $\mathsf{AF_c} = (\mathsf{E}, \mathsf{R}')$; constraint explanations EX; maximum number of retries $r_{\max}$
Output: conflict mapping $\widehat{M}$

$r \leftarrow 0$;
$\widehat{M}_{\mathrm{raw}} \leftarrow$ *Profiler*$(A, \mathsf{AF_c}, \mathrm{EX})$;
while $\neg\, \mathrm{ValidMapping}(\widehat{M}_{\mathrm{raw}}, A, \mathsf{E})$ **do**
  $\widehat{M}_{\mathrm{raw}} \leftarrow \mathrm{RepairMapping}(A, \mathsf{AF_c}, \mathrm{EX}, \widehat{M}_{\mathrm{raw}})$;
  if $\neg\, \mathrm{ValidMapping}(\widehat{M}_{\mathrm{raw}}, A, \mathsf{E})$ **then**
    if $r \geq r_{\max}$ **then**
      return $\bot$;
    $r \leftarrow r + 1$;
    $\widehat{M}_{\mathrm{raw}} \leftarrow$ *Profiler*$(A, \mathsf{AF_c}, \mathrm{EX})$;
$\widehat{M} \leftarrow \mathrm{ParseMapping}(\widehat{M}_{\mathrm{raw}})$;
return $\widehat{M}$;

which assigns to each descriptor or constraint argument the set of elements with which it conflicts. The mapping M is constructed using another LLM agent, referred to as the *Profiler*. To improve the reliability of the *Profiler*, it is provided not only with the extracted descriptors and predefined constraints, but also with explanations of the constraints. Each explanation contains a brief description of the corresponding constraint, including examples of descriptors that would violate it.

The input to the *Profiler* is given by the triple $(A, \mathsf{AF_c}, \mathrm{EX})$, where $\mathrm{EX}\colon \mathsf{E} \to \Sigma^*$ maps each constraint argument in E to its natural-language explanation. The resulting conflict mapping is denoted by $\widehat{M} = $ *Profiler*$(A, \mathsf{AF_c}, \mathrm{EX})$. Similar to the *Extractor*, the *Profiler* is instructed by means of a prompt that defines what constitutes a conflict for the task at hand. As the conflict mapping is also produced by an LLM, its structured output is validated before further processing. Here, $\mathrm{ValidMapping}$ checks whether the output conforms to the expected mapping format and only refers to elements in $A \cup \mathsf{E}$. Invalid outputs are first repaired; if repair does not yield a valid mapping, the *Profiler* is retried up to $r_{\max}$ times. This procedure is detailed in Algorithm 3.

The following example illustrates the behavior of the *Profiler*.

Example 3. Consider the descriptor set $A = \{\text{empty room},\ \text{full of furniture}\}$
and the constraint framework $\mathsf{AF_c} = (\mathsf{E}, \mathsf{R}')$, where $\mathsf{E} = \{\texttt{no\_empty\_room}\}$. Let the explanation mapping be given by

$$\mathrm{EX}(\texttt{no\_empty\_room}) = \text{“the room must contain at least one object”}.$$

The *Profiler* may then return the following conflict mapping:

$$\textit{Profiler}(A, \mathsf{AF_c}, \mathrm{EX}) = \left\{ \begin{array}{r} \text{empty room} \mapsto \{\text{full of furniture},\ \texttt{no\_empty\_room}\}, \\ \text{full of furniture} \mapsto \{\text{empty room}\}, \\ \texttt{no\_empty\_room} \mapsto \{\text{empty room}\} \end{array} \right\}.$$

Argumentation Framework Construction The goal of this stage is to construct a single AF that represents the conflict relation among the extracted descriptors and the predefined constraints. In a simple setting, for instance when no external constraints are specified, the construction of the final argumentation framework is straightforward: one argument is generated for each descriptor, and attacks between arguments are added according to the conflict mapping $\widehat{M}$ produced by the *Profiler*. To model more complex scenarios, such as cases where predefined constraints must not be violated by

Algorithm 4: Argumentation Framework Construction

Input: descriptor set A; constraint framework $\mathsf{AF_c} = (\mathsf{E}, \mathsf{R}')$; conflict mapping $\widehat{M}$; type set T; type assignment τ; permitted type-level attack relation $\rightarrow_T$
Output: final argumentation framework $\mathsf{AF_f}$
$\mathsf{E}' \leftarrow \mathsf{E} \cup A$;
$\widehat{C} \leftarrow \{(x, y) \in \mathsf{E}' \times \mathsf{E}' \mid y \in \widehat{M}(x)\}$;
$R' \leftarrow \mathsf{R}' \cup \{(x, y) \in \widehat{C} \mid (\tau(x), \tau(y)) \in \rightarrow_T\}$;
$\mathsf{AF_f} \leftarrow (\mathsf{E}', R')$;
return $\mathsf{AF_f}$;

user-provided descriptors, we additionally model the permitted directions of attacks. To this end, we introduce a set of types T, a type assignment function $\tau\colon A \cup \mathsf{E} \to T$, and a type-level attack relation $\rightarrow_T \subseteq T \times T$.

The set T contains the possible types that can be assigned to descriptors and constraint arguments. For example, the type *hard rule* can be used to represent constraints that must hold and cannot be challenged, whereas the type *soft rule* can be used for constraints that may be attacked. The function τ assigns exactly one type to each element in $A \cup \mathsf{E}$. The relation $\rightarrow_T$ determines which types of arguments are allowed to attack which other types. Thus, the input for the final argumentation-framework construction is given by $B = (A, \mathsf{AF_c}, \widehat{M}, T, \tau, \rightarrow_T)$, where A is the set of descriptors, $\mathsf{AF_c} = (\mathsf{E}, \mathsf{R}')$ is the argumentation framework modelling the constraints, $\widehat{M}$ is the conflict mapping produced by the *Profiler*, T is the set of types, τ is the type assignment function, and $\rightarrow_T$ is the permitted attack relation for the types T.

The type assignment for constraint arguments is defined in advance. For the descriptors, different options are possible. The simplest variant assigns a fixed type to every descriptor. A more sophisticated variant uses an additional LLM agent to assign each descriptor a suitable type $t \in T$, based on the available types and the explanations of the constraint arguments. After each descriptor has been assigned a type, the final argumentation framework $\mathsf{AF_f} = (\mathsf{E}', R')$ can be constructed.

We first extend the set of constraint arguments E to $\mathsf{E}' = \mathsf{E} \cup A$, with A and E disjoint. The final attack relation R' contains the predefined attacks between constraint arguments and the additional attacks induced by the conflict mapping, provided that they respect the permitted attack relation $\rightarrow_T$. Formally, let $\widehat{C} = \{(x, y) \in \mathsf{E}' \times \mathsf{E}' \mid y \in \widehat{M}(x)\}$ be the conflict relation induced by the *Profiler*. Then $R' = \mathsf{R}' \cup \{(x, y) \in \widehat{C} \mid (\tau(x), \tau(y)) \in \rightarrow_T\}$. We denote the resulting framework by $\mathsf{AF_f}$. The construction of $\mathsf{AF_f}$ is summarized in Algorithm 4. In contrast to the other stages, this stage does not involve any use of an LLM: the set of arguments is obtained from the extracted descriptors and constraint arguments, while the attack relation is constructed deterministically from the conflict mapping and the permitted attack relation according to the types as defined above. The constraint framework itself, as well as the types and permitted type-level attacks, are specified in advance for the respective application scenario.

Note that this construction shares similarities with *value-based argumentation frameworks* [30], in which attacks are evaluated not only with respect to conflicts between arguments, but also with respect to the values associated with the arguments and the preference ordering over these values.

Example 4. Let $d_1 =$ empty room, $d_2 =$ full of furniture, and $e_1 =$ `no_empty_room`. Thus, $A = \{d_1, d_2\}$ and $\mathsf{E} = \{e_1\}$. Suppose that predefined constraints are treated as hard constraints that cannot be attacked by user-provided descriptors. We introduce the type set $T = \{t_D, t_C\}$, where t_D denotes the type of user-provided descriptors and t_C denotes the type of predefined constraints. The type assignment is given by $\tau(d_1) = \tau(d_2) = t_D$ and $\tau(e_1) = t_C$. To ensure that descriptors cannot attack predefined constraints, we define the permitted type-level attack relation as $\rightarrow_T = \{(t_D, t_D), (t_C, t_D), (t_C, t_C)\}$. In particular, $(t_D, t_C) \notin \rightarrow_T$, so attacks from user-provided descriptors to predefined constraints are not permitted.

Algorithm 5: Conflict Resolution via an External Argumentation Solver

Input: final argumentation framework $\mathsf{AF_f}$; descriptor set A; semantics σ; computational task $p \in \{\mathrm{SE}, \mathrm{EE}\}$

Output: accepted descriptor set A^*

if $p = \mathrm{SE}$ **then**

  $E^* \leftarrow \mathrm{Solver}(\mathsf{AF_f}, \sigma, \mathrm{SE})$;

else

  $\mathcal{E} \leftarrow \mathrm{Solver}(\mathsf{AF_f}, \sigma, \mathrm{EE})$;

  $E^* \leftarrow S(\mathcal{E})$;

$A^* \leftarrow E^* \cap A$;

return A^*;

The final set of arguments is $\mathsf{E}' = A \cup \mathsf{E} = \{d_1, d_2, e_1\}$. The conflict mapping returned by the *Profiler* induces, among others, the potential attacks (d_1, e_1) and (e_1, d_1). However, (d_1, e_1) is discarded because it would be an attack from a descriptor to a constraint, whereas (e_1, d_1) is retained because attacks from constraints to descriptors are allowed. Assuming $\mathsf{R}' = \emptyset$, the resulting attack relation is therefore $R' = \{(d_1, d_2), (d_2, d_1), (e_1, d_1)\}$. Thus, the resulting argumentation framework is $\mathsf{AF_f} = (\mathsf{E}', R')$.

Conflict Resolution via an External Argumentation Solver After constructing the final argumentation framework $\mathsf{AF_f} = (\mathsf{E}', R')$, an external solver is called to compute extensions under a given semantics σ and computational task $p \in \{\mathrm{SE}, \mathrm{EE}\}$. Let $E_\sigma(\mathsf{AF_f}) \subseteq 2^{\mathsf{E}'}$ denote the set of all extensions of $\mathsf{AF_f}$ under semantics σ. For the task SE, the solver computes a single extension $E \in E_\sigma(\mathsf{AF_f})$. For the task EE, the solver enumerates all extensions, that is, it returns the set $E_\sigma(\mathsf{AF_f})$. An additional selection function $S\colon 2^{2^{\mathsf{E}'}} \to 2^{\mathsf{E}'}$ is used to choose one extension from the set of computed candidates. A simple selection rule is to choose an extension that contains the largest number of accepted user descriptors, with random selection as a tie-breaker. The descriptor set passed to the final stage is then given by $A^* = E^* \cap A$. This set contains exactly the descriptors accepted in the selected extension and therefore serves as the basis for generating a conflict-free prompt. The complete resolution step is shown in Algorithm 5.

Importantly, the framework does not assume that every conflict admits a unique resolution. Depending on the structure of the argumentation framework and the chosen semantics, several extensions may exist, representing different but internally consistent ways of resolving the conflicts. In such cases, there is not necessarily a single extension that can be regarded as the semantically "correct" one without introducing additional preferences. The treatment of such ambiguity depends on the intended application. If a single conflict-free prompt is required, a selection function S can be used to choose one of the extensions according to an application-specific criterion. For example, the simple selection rule described above favors extensions that preserve the largest number of user-provided descriptors. Alternatively, if preserving the ambiguity is desirable, the task EE can be used to enumerate the extensions. Each extension then represents a possible resolution and could, in principle, be used to generate multiple alternative conflict-free prompts. The latter variant is not evaluated in the experiments presented in this work, which focus on producing a single output prompt.

Example 5. We choose the computational task as $SE - ST$, i. e., the computation of a single stable extension. Recall that the final argumentation framework is given by $\mathsf{AF_f} = (\mathsf{E}', R')$, where $\mathsf{E}' = \{d_1, d_2, e_1\}$ and $R' = \{(d_1, d_2), (d_2, d_1), (e_1, d_1)\}$. Here, $d_1 =$ empty room, $d_2 =$ full of furniture, and $e_1 =$ `no_empty_room`. The external solver returns $E^* = \{e_1, d_2\}$. The accepted descriptor set is therefore $A^* = E^* \cap A = \{d_2\}$. Thus, only the descriptor $d_2 =$ full of furniture is passed to the final stage.

Algorithm 6: Conflict-Free Prompt Generation

Input: original prompt P; accepted descriptor set A^*
Output: conflict-free prompt P^*
$P^* \leftarrow$ *Constructor*(P, A^*);
return P^*;

Conflict-Free Prompt Generation In the final stage, a conflict-free prompt P^* is generated from the accepted descriptor set A^*. The goal is to construct a prompt that preserves the style and structure of the original user prompt P while expressing only descriptors that have been accepted in the selected extension.

To this end, we use another LLM agent, referred to as the *Constructor*. The *Constructor* receives the original prompt P and the accepted descriptor set A^* as input and returns a revised prompt $P^* = \textit{Constructor}(P, A^*)$. The agent is instructed to preserve the language, style, and tone of P as far as possible, while ensuring that the generated prompt does not reintroduce rejected descriptors. Thus, P^* represents a conflict-free reformulation of the original prompt with respect to the accepted descriptor set A^*. The final generation step is summarized in Algorithm 6.

Example 6. Continuing the running example, recall that the original prompt is $P =$ "An empty room full of furniture." and that, under stable semantics, the accepted descriptor set is $A^* = \{d_2\}$, where $d_2 =$ full of furniture. The descriptor $d_1 =$ empty room was rejected because it conflicts with the predefined constraint $e_1 =$ `no_empty_room`.

The *Constructor* therefore removes the rejected descriptor while preserving the style of the original prompt. A possible conflict-free prompt is $P^* =$ "A room full of furniture." This prompt retains the accepted descriptor d_2 and avoids reintroducing the rejected descriptor d_1. It can therefore be passed to the downstream generative model as the final conflict-free prompt.

5. Experiments

This section describes the experiments conducted and presents the results of the analysis. The objective of the experiments is to evaluate the performance of the proposed approach on real-world data. In particular, the evaluation aims to examine both the system's ability to identify and resolve conflicts in natural-language prompts and its effectiveness in generating conflict-free prompts that remain faithful to the original style and semantics. To this end, the approach is evaluated on four new datasets containing conflicting prompts for a text-to-image model.

The remainder of this section is structured as follows: In Section 5.1, the experimental setup is described. A brief overview of the base datasets, the process of generating the experiment datasets, and the evaluation pipeline, including the used metrics is given. In Section 5.2, the results of the experimental analysis are presented.

5.1. Setup

To examine the performance of the proposed approach in conflict identification and resolution, an experimental analysis was conducted. In total, we examined five different datasets across two distinct scenarios. In the first scenario, we focused on internal conflicts in the form of contradicting information within the prompt. In particular, we generated four different benchmarks based on the *Pixelrose* and *DiffusionDB* datasets. Subsequently, the original datasets and the dataset generation process are described in more detail.

DiffusionDB The *DiffusionDB* [31] dataset is a large-scale text-to-image prompt dataset containing 14 million images generated by *Stable Diffusion* [32], 1.8 million unique prompts, and hyperparameters specified by real users. The authors downloaded chat logs from *Stable Diffusion* Discord channels using

DiscordChatExporter[1] and saved them as HTML files. The corpus was restricted to channels where users invoked a bot to run *Stable Diffusion* (Version 1) by entering a prompt.

PixelProse *PixelProse* [33] comprises over 16 million diverse images sourced from three different web-scraped databases: (1) *CommonPool* [34], (2) *CC12M* [35] and (3) *RedCaps* [36]. *CommonPool* contains a large pool of image-text pairs, that were filtered using cld3[2] to detect English-only text and select image-text pairs similarity. The *CC12M* dataset comprises 12.4 million web-crawled images and alt-text pairs. *RedCaps* is curated from *Reddit*[3]. It consists of 12 million image-text pairs from 350 different subreddits. In this work, the image captions are interpreted as detailed user prompts to a text-to-image model.

The dataset generation process used in the experiments is as follows. First, we randomly sampled 3000 prompts from *DiffusionDB* and 1000 prompts from each of the sources of *PixelProse*. Following the procedure outlined in Section 4, an LLM agent (*Extractor*) was instructed to extract the descriptive elements from each prompt. From these extracted elements, a random subset was selected (5 to 10), and conflicts in the form of direct contradictions were generated, using another LLM agent. Finally, a third LLM agent, similar to the *Constructor*, composed a single, contradictory prompt that combined the original content with the generated contradictions.

The second scenario is more complex and addresses both internal and external conflicts. Here, we use the *SceneTeller* [37] dataset, which is explained in more detail below.

SceneTeller This dataset is based on the *3D-FRONT* dataset [38], a large-scale indoor-scene dataset comprising $18,797$ rooms furnished with textured 3D objects from *3D-FUTURE* [39]. In addition, each scene comes with a textual description of the layout. In total, the dataset contains $3,397$ training, 453 validation, and 423 test bedroom scenes.

For our experiments, we randomly sample 500 prompts from the bedroom scenes of the dataset. In addition, we construct a constraint argumentation framework tailored to the layout-generation setting. This framework encodes different layout-specific rules, such as avoiding the placement of furniture in the center of the room or ensuring that at least one corner of the room remains empty. Overall, the constraint framework consists of 36 constraint arguments and 18 attacks between constraint arguments. Furthermore, for this more complex scenario, we evaluate an agent-only baseline in which a single LLM agent is given the constraints and instructed to resolve the conflicts directly.

To quantify the semantic similarity between the original and the generated prompts, the *BERTScore* [40] is used, which correlates well with human judgments of semantic similarity. While *BERTScore* indicates how well the generated text preserves meaning, it does not allow any conclusions to be drawn about the ability to resolve conflicts. To directly capture the performance regarding conflict resolution, metrics that quantify the presence of conflicts *pre* and *post* processing are used. To detect conflicts, we use the *Extractor* and *Profiler* agents.

Let a denote the number of instances rated as having at least one conflict both pre- and post-processing steps. Let b be the number of instances where at least one conflict was present pre and no conflicts post, c be the number of instances where new conflicts were introduced, and d be the number of instances that remained conflict-free. The total number of paired observations is $N = a + b + c + d$. The *conflict rate pre-processing* is defined as the proportion of instances with a conflict before processing $p_{\text{pre}} = \frac{a+b}{N}$. Analogously, the *conflict rate post-processing* is given by $p_{\text{post}} = \frac{a+c}{N}$. To quantify improvement, the *absolute reduction rate* (ARR), i. e., the absolute decrease in conflict rate from pre to post processing, is reported. This metric can be expressed either as the difference of prevalences or as the net balance of resolved versus newly introduced conflicts.

$$ARR = p_{\text{pre}} - p_{\text{post}} = \frac{(a+b)-(a+c)}{N} = \frac{b-c}{N}.$$

[1] https://github.com/Tyrrrz/DiscordChatExporter
[2] https://github.com/google/cld3
[3] https://www.reddit.com/

In addition, we employed the *relative reduction rate* (RRR), which quantifies the proportional decrease in conflicts before and after conflict resolution. The relative reduction rate is then defined as

$$RRR = \frac{p_{\text{pre}} - p_{\text{post}}}{p_{\text{pre}}}.$$

As LLM agents, the *gpt-oss:20b*[4] model was used. The *μ-toksia* [41] solver was used to solve the task SE-ST for all instances. The ST semantics was chosen because it yields a keep-or-discard decision rather than leaving conflicting descriptors undecided. The source code, all agent prompts, and the set of constraints used in our experiments are publicly available [5].

5.2. Results

Table 1
Overview of results on all five datasets, in terms of the conflict rate pre (p_{pre}) and post (p_{post}), absolute reduction rate (ARR), relative reduction rate (RRR), and the BERTScore (Precision, Recall, and F1).

Dataset	p_{pre}	p_{post}	ARR	RRR	BERTScore		
					Precision	Recall	F1
DiffusionDB	0.912	0.121	0.791	0.867	0.944	0.935	0.940
Pixelprose$_{cc12}$	0.976	0.064	0.912	0.934	0.948	0.930	0.939
Pixelprose$_{common}$	0.97	0.077	0.893	0.921	0.945	0.926	0.935
Pixelprose$_{redcaps}$	0.985	0.074	0.911	0.924	0.947	0.928	0.937
SceneTeller	0.904	0.319	0.585	0.647	0.971	0.959	0.964

Table 1 summarizes the results across the datasets. Overall, the system substantially reduces conflict rates in both scenarios. Unsurprisingly, the simpler scenario with only internal conflicts shows better results, with p_{post} remaining below 0.121, ARR ranging from 0.791 to 0.912, and RRR ranging from 0.867 to 0.934. The highest reduction is achieved on Pixelprose$_{cc12}$ with an ARR of 0.912 and an RRR of 0.934, closely followed by PixelProse$_{redcaps}$ with an ARR of 0.911 and an RRR of 0.924, while *DiffusionDB* shows the lowest ARR of 0.791 and the lowest RRR of 0.867. For the harder scenario on the *SceneTeller* dataset, the results still show a high reduction in conflicts with an ARR of 0.585 and an RRR of 0.647. In comparison, the agent-only baseline gets a substantially lower ARR of 0.228 and RRR of 0.254. However, the baseline achieves a slightly higher *BERTScore* of 0.970 compared to 0.964. This is most likely because the baseline agent tends to copy the original prompt when it is unable to resolve the conflicts.

The *BERTScore* values remain consistently high across the remaining datasets, with F1 scores between 0.935 and 0.940. Overall, the results indicate that the system reduces conflicts effectively while largely preserving the semantic content of the original prompts.

6. Conclusion

In this work, an argumentation-based approach is presented for resolving conflicts in user prompts for text-based generative models. By combining flexible LLM agents, which identify and extract conflicts, with abstract argumentation methods, which resolve the detected conflicts, the approach generates conflict-free prompts while preserving the style of the original prompt. This was demonstrated in an experimental analysis using five different datasets. The results show that the proposed approach can successfully identify and resolve conflicts with resolution rates of up to 93.4% while maintaining textual similarity to the input, as reflected in high BERTScores. In future work, additional datasets could be examined, as well as various combinations of LLM agents. Since this work has focused

[4] https://ollama.com/library/gpt-oss
[5] https://github.com/aig-hagen/Argumentation-based-Prompt-Conflict-Resolution.git

primarily on resource-efficient LLMs, future work could also use larger models. In addition to the LLM-based evaluation, a human evaluation of the generated prompts would also be desirable. Furthermore, additional experiments could be conducted to compare the prompts not only at the textual level but also in the generative model's output domain. Various other text-conditioned generation tasks, such as text-to-video or text-to-music, could also be investigated, as prompt formulation varies greatly depending on context.

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT and Grammarly in order to check grammar and spelling, paraphrase, and reword. After using these tools/services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] S. Feuerriegel, J. Hartmann, C. Janiesch, P. Zschech, Generative ai: S. feuerriegel et al., Business & information systems engineering 66 (2024) 111–126.

[2] V. Liu, L. Chilton, Design guidelines for prompt engineering text-to-image generative models, in: Proceedings of the 2022 CHI conference on human factors in computing systems, 2022, pp. 1–23.

[3] J. Oppenlaender, A taxonomy of prompt modifiers for text-to-image generation, Behaviour & Information Technology 43 (2024) 3763–3776.

[4] Y. Hao, Z. Chi, L. Dong, F. Wei, Optimizing prompts for text-to-image generation, Advances in Neural Information Processing Systems 36 (2023) 66923–66939.

[5] K. Lee, H. Liu, M. Ryu, O. Watkins, Y. Du, C. Boutilier, P. Abbeel, M. Ghavamzadeh, S. Gu, Aligning text-to-image models using human feedback, arXiv preprint arXiv:2302.12192 (2023).

[6] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, R. Lowe, Training language models to follow instructions with human feedback, in: Advances in Neural Information Processing Systems, volume 35, 2022, pp. 27730–27744.

[7] Z. Wu, H. Gao, Y. Wang, X. Zhang, S. Wang, Universal prompt optimizer for safe text-to-image generation, in: Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics, 2024, pp. 6340–6354. doi:10.18653/v1/2024.naacl-long.351.

[8] K. Atkinson, P. Baroni, M. Giacomin, A. Hunter, H. Prakken, C. Reed, G. Simari, M. Thimm, S. Villata, Towards artificial argumentation, AI magazine 38 (2017) 25–36.

[9] P. Dung, On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games, Artificial intelligence 77 (1995) 321–357.

[10] A. Dejl, M. Williams, F. Toni, Argumentation for explainable and globally contestable decision support with llms, arXiv preprint arXiv:2603.14643 (2026).

[11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Advances in neural information processing systems 30 (2017).

[12] J. Kaplan, S. McCandlish, T. Henighan, et al., Scaling laws for neural language models, arXiv preprint arXiv:2001.08361 (2020).

[13] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., Language models are few-shot learners, Advances in neural information processing systems 33 (2020) 1877–1901.

[14] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, J. Steinhardt, Measuring massive multitask language understanding, arXiv preprint arXiv:2009.03300 (2021).

[15] A. Srivastava, A. Rastogi, A. Rao, A. A. M. Shoeb, A. Abid, A. Fisch, A. R. Brown, A. Santoro, A. Gupta, A. Garriga-Alonso, et al., Beyond the imitation game: Quantifying and extrapolating the capabilities of language models, Transactions on machine learning research (2023).

[16] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, et al., Training verifiers to solve math word problems, arXiv preprint arXiv:2110.14168 (2021).
[17] M. Chen, J. Tworek, H. Jun, Q. Y., et al., Evaluating large language models trained on code, arXiv preprint arXiv:2107.03374 (2021).
[18] P. Liang, R. Bommasani, P. Burr, et al., Holistic evaluation of language models, arXiv preprint arXiv:2211.09110 (2022).
[19] L. Weidinger, J. Uesato, M. Rauh, et al., Ethical and social risks of harm from language models, arXiv preprint arXiv:2112.04359 (2021).
[20] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson, et al., Extracting training data from large language models, in: 30th USENIX security symposium (USENIX Security 21), 2021, pp. 2633–2650.
[21] E. Strubell, A. Ganesh, A. McCallum, Energy and policy considerations for deep learning in nlp, in: Proceedings of the 57th annual meeting of the association for computational linguistics, 2019, pp. 3645–3650.
[22] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 4 ed., Pearson, 2021.
[23] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. Zhao, Z. Wei, J. Wen, A survey on large language model based autonomous agents, Frontiers of Computer Science 18 (2024) 186345. doi:10.1007/s11704-024-40231-1.
[24] E. Wallace, K. Xiao, R. Leike, L. Weng, J. Heidecke, A. Beutel, The instruction hierarchy: Training llms to prioritize privileged instructions, 2024. URL: https://arxiv.org/abs/2404.13208. arXiv:2404.13208.
[25] S. Chen, J. Piet, C. Sitawarin, D. Wagner, {StruQ}: Defending against prompt injection with structured queries, in: 34th USENIX Security Symposium (USENIX Security 25), 2025, pp. 2383–2400.
[26] H. Li, Y. Yang, G. E. Suh, N. Zhang, C. Xiao, Reasalign: Reasoning enhanced safety alignment against prompt injection attack, arXiv preprint arXiv:2601.10173 (2026).
[27] G. Freedman, A. Dejl, D. Gorur, X. Yin, A. Rago, F. Toni, Argumentative large language models for explainable and contestable decision-making, 2024, URL https://arxiv. org/abs/2405.02079 (2024).
[28] F. Castagna, I. Sassoon, S. Parsons, Can formal argumentative reasoning enhance llms performances?, arXiv preprint arXiv:2405.13036 (2024).
[29] Z. Li, X. Fang, C. Chen, M. Li, B. Liao, Argumentation computation with large language models: A benchmark study, arXiv e-prints (2024) arXiv–2412.
[30] T. Bench-Capon, Persuasion in practical argument using value based argumentation frameworks, Journal of Logic and Computation 13 (2003) 429–448.
[31] Z. Wang, E. Montoya, D. Munechika, H. Yang, B. Hoover, D. Chau, Large-scale prompt gallery dataset for text-to-image generative models, arXiv:2210.14896 [cs] (2022). URL: https://arxiv.org/abs/2210.14896.
[32] D. Podell, Z. English, K. Lacey, A. Blattmann, T. Dockhorn, J. Müller, J. Penna, R. Rombach, Sdxl: Improving latent diffusion models for high-resolution image synthesis, arXiv preprint arXiv:2307.01952 (2023).
[33] V. Singla, K. Yue, S. Paul, R. Shirkavand, M. Jayawardhana, A. Ganjdanesh, H. Huang, A. Bhatele, G. Somepalli, T. Goldstein, From pixels to prose: A large dataset of dense image captions, arXiv preprint arXiv:2406.10328 (2024).
[34] S. Gadre, G. Ilharco, A. Fang, J. Hayase, G. Smyrnis, T. Nguyen, R. Marten, M. Wortsman, D. Ghosh, J. Zhang, E. Orgad, R. Entezari, G. Daras, S. Pratt, V. Ramanujan, Y. Bitton, K. Marathe, S. Mussmann, R. Vencu, M. Cherti, R. Krishna, P. Koh, O. Saukh, A. Ratner, S. Song, H. Hajishirzi, A. Farhadi, R. Beaumont, S. Oh, A. Dimakis, J. Jitsev, Y. Carmon, V. Shankar, L. Schmidt, Datacomp: In search of the next generation of mult.dal datasets, Advances in Neural Information Processing Systems 36 (2023) 27092–27112.
[35] S. Changpinyo, P. Sharma, N. Ding, R. Soricut, Conceptual 12m: Pushing web-scale image-text pre-training to recognize long-tail visual concepts, in: Proceedings of ICCV21, 2021, pp. 3558–3568.

[36] K. Desai, G. Kaul, Z. Aysola, J. Johnson, Redcaps: Web-curated image-text data created by the people, for the people, arXiv preprint arXiv:2111.11431 (2021).
[37] B. M. Öcal, M. Tatarchenko, S. Karaoğlu, T. Gevers, Sceneteller: Language-to-3d scene generation, in: European Conference on Computer Vision, Springer, 2024, pp. 362–378.
[38] H. Fu, B. Cai, L. Gao, L. Zhang, J. Li, Z. Xun, C. Sun, R. Jia, B. Zhao, H. Zhang, 3d-front: 3d furnished rooms with layouts and semantics, in: Proceedings of ICCV 2021, 2021, pp. 10933–10942.
[39] H. Fu, R. Jia, L. Gao, M. Gong, B. Zhao, S. Maybank, D. Tao, 3d-future: 3d furniture shape with texture, International Journal of Computer Vision 129 (2021) 3313–3337.
[40] T. Zhang, V. Kishore, F. Wu, K. Weinberger, Y. Artzi, Bertscore: Evaluating text generation with bert, arXiv preprint arXiv:1904.09675 (2019).
[41] A. Niskanen, M. Järvisalo, μ-toksia: An efficient abstract argumentation reasoner, in: Proc. of the Int. Conf. on Principles of Knowledge Representation and Reasoning, volume 17, 2020, pp. 800–804.