Generative Timelines for Instructed Visual Assembly

Alejandro Pardo[1,2], **Jui-Hsien Wang**[2], **Bernard Ghanem**[1], **Josef Sivic**[2,3], **Bryan Russell**[2]
Fabian Caba Heilbron[2]

[1]AI Initiative, KAUST
[2]Adobe Research
[3]CIIRC CTU*
https://sites.google.com/kaust.edu.sa/timeline-assembler

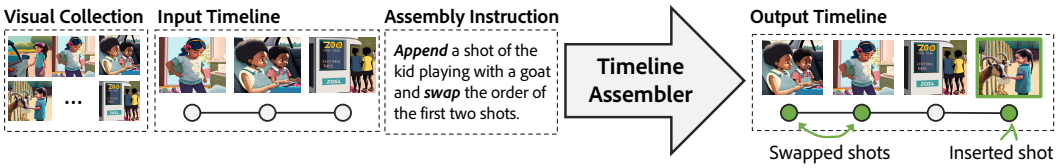

Figure 1: **Instructed Visual Assembly.** Given a visual collection, an input timeline, and an assembly instruction, our model (called the Timeline Assembler) performs the instructed assembly task and generates an output timeline with the desired edits. The collection comprises various media elements, such as video clips or images. The timeline is a sequential arrangement of these elements.

Abstract

The objective of this work is to manipulate visual timelines (*e.g.,* a video) through natural language instructions, making complex timeline editing tasks accessible to non-expert or potentially even disabled users. We call this task *Instructed visual assembly*. This task is challenging as it requires (i) identifying relevant visual content in the input timeline as well as retrieving relevant visual content in a given input (video) collection, (ii) understanding the input natural language instruction, and (iii) performing the desired edits of the input visual timeline to produce an output timeline. To address these challenges, we propose the Timeline Assembler, a generative model trained to perform instructed visual assembly tasks. The contributions of this work are three-fold. First, we develop a large multimodal language model, which is designed to process visual content, compactly represent timelines and accurately interpret timeline editing instructions. Second, we introduce a novel method for automatically generating datasets for visual assembly tasks, enabling efficient training of our model without the need for human-labeled data. Third, we validate our approach by creating two novel datasets for image and video assembly, demonstrating that the Timeline Assembler substantially outperforms established baseline models, including the recent GPT-4o, in accurately executing complex assembly instructions across various real-world inspired scenarios.

*Czech Institute of Informatics, Robotics and Cybernetics at the Czech Technical University in Prague.
[1]Work partially done during an internship at Adobe.

Workshop on Video-Language Models at 38th Conference on Neural Information Processing Systems (NeurIPS 2024).

1 Introduction

Imagine returning from a zoo trip and finding a visual timeline, *e.g.,* in the form of a short video, of highlights automatically generated by your device [33, 9]. While reviewing the timeline, you notice some shots are misplaced and a memorable moment with that kid petting a goat is missing. Traditionally, making such modifications would require finding the additional shot in your video collection as well as potential cumbersome interactions with video editing tools. This may be even harder for users with small-screen devices or users with disabilities for whom it is hard to interact with traditional interfaces. A more intuitive approach, illustrated in Figure 1, involves using natural language to direct edits. The users simply state their desired changes, much like setting a calendar event by voice. Such a system uses the collection of visual assets, the current timeline, and the provided instruction to return the refined timeline with the requested edits.

We define this capability as instructed visual assembly, a process that involves the automated editing of a visual timeline in response to user-provided natural language instructions. To effectively automate this task, the system requires a comprehensive understanding of three key elements: the instruction itself, the existing visual timeline, and the collection of visual assets. For instance in Figure 1, if a user instructs the system to *"swap the first two clips and add the shot of the kid with the goat,"* the system must first understand the multimodal context by interpreting the wording of the instruction and relating it to the visual data in the timeline and the collection. The system must then identify the specific elements to be edited, such as the initial two shots, and retrieve the additional clip of the child with the goat from the collection. This process involves deep multimodal understanding to determine not only what the user intends but also how these intentions translate into direct manipulation of visual content. Despite these challenges, instructed visual assembly can enable intuitive video creation interfaces for novices, and users with disabilities, mitigating the complexities associated with mastering traditional video editing tools and interfaces, as well as easing the management of visual content. Moreover, it can ease the editing of videos on devices with small screens where simple operations like drag and drop become cumbersome.

This is the first work that explicitly addresses the task of instructed visual assembly. While this task shares similarities with language-based video editing [12, 39], which consists of editing the pixels of a source video based on language instructions, instructed visual assembly distinctively focuses on executing instructions to arrange visual elements in a timeline. There are other related tasks in the video assembly space, such as automatic shot transitioning [38, 37] and B-roll recommendation [18, 52, 47]. However, these tasks primarily provide creative guidance, diverging from the instructed visual assembly goal of composing timelines through natural language instructions.

In this paper, we present a method for teaching a generative model, named Timeline Assembler, to follow assembly instructions and generate timelines with the appropriate edits. The Timeline Assembler builds on Large Language Models (LLMs) to leverage their remarkable skills in following instructions [51, 49] and interpreting multimodal content [29, 56]. Specifically, we adapt a multimodal LLM [56] to handle the nuances of instructed visual assembly. This adaptation presents two major challenges. Firstly, existing multimodal LLMs are designed to process a single image or video. Thus, devising a representation for handling visual collections and timelines using LLMs remains an open challenge. Secondly, as shown by our experiments, existing multimodal LLMs struggle to understand and execute visual assembly instructions, hinting that further tuning for the assembly task is required. However, we face the challenge of doing so without human-labeled data, which is hard to acquire at scale for such specialized tasks.

Our work makes the following contributions to address the challenges above:

(1) We design a multimodal LLM architecture to process visual collections, encode timelines, and ingest natural language instructions to generate edited output timelines. Our architecture represents each image or video clip in the collection with a unique identifier and a visual representation compatible with the LLM's input space [29]. Secondly, we encode the timeline using the unique identifiers of each arranged visual element. Finally, we map the LLM's output tokens directly back to a timeline, associating each token to its relevant visual element. This design not only ensures a compact representation of the visual collection and the timeline, but also facilitates a straightforward reconstruction of timelines from the model's output tokens.

(2) We propose a new approach to automatically generate a paired dataset (input/output) for instructed visual assembly. Our method programmatically creates a collection, input/output timelines, and

assembly instructions from an input visual sequence and task candidates. Using this approach, we generate data to learn the projection layer and Low-Rank Adapters (LoRA) [15] for effective LLM performance on assembly tasks. Our training approach is not only human-labels-free but also efficient and enables a wide range of timeline editing tasks.

(3) We construct two datasets, one for image sequence assembly (of still images) and one for video assembly, to evaluate instructed visual assembly. On both datasets, our method substantially outperforms baseline approaches, including powerful LLMs such as GPT3.5, highlighting the benefits of our approach. Moreover, we show that our model can match or even exceed the performance of specialized (single-task) assembly models, perform equally well regardless of the length of the timelines, and execute multiple complex instructions at a time.

2 Related Work

Multimodal LLMs as Multi-task Interfaces. Our work shares a similar spirit with recent trends that use LLM capabilities to perform multiple tasks for different applications [14]. Several methods have been proposed for image and language tasks [56, 2, 29, 23, 8, 21], video understanding [32, 25, 7, 45, 30], multimodal understanding [53, 54], recommendation systems [13], and robotics and motion planning [19, 27, 4, 10, 16]. Our Timeline Assembler distinguishes itself by specializing in the domain of visual timelines, interpreting natural language to modify sequences of visual data. This unique application extends the use of LLMs beyond physical tasks to the manipulation of multimedia content.

Video Assembly. Several computer vision techniques have been proposed for automated video editing tasks [41, 38, 44, 3, 22, 37, 6, 42]. Likewise, multiple works have proposed to tackle video assembly. Li et al. [26] propose multi-shot vlog assembly, drawing parallels to our approach where an initial sequence guides subsequent shot selections from a candidate pool. Furthermore, other works [48, 50, 52, 31] have developed tools that assemble videos from input queries, leveraging multiple components for an intuitive video creation experience. While these works emphasize on providing creative guidance, our model introduces a user-guided approach to video assembly, enabling a novel interface to manipulate and arrange timelines with language instructions.

Instruction Fine-tuning and Aligning with User Intent. Our paper builds on methods that adapt language models to closely following human instructions [40, 5]. Wei *et al.*[51] first introduced instruction-tuning, enhancing the usability and multi-tasking of LLMs. [36] expanded this by integrating Reinforcement Learning from Human preferences (RLHF). [49] demonstrated that GPT3-generated instructions could achieve results similar to InstructGPT through self-instruct fine-tuning. Our work uses similar ideas and tailors instruction fine-tuning for visual assembly tasks. We do so by automatically gathering novel visual assembly instruction data that serves as training data to our LoRA [15] fine-tuned multimodal LLM.

3 Timeline Assembler

3.1 Timeline Assembler Architecture

Our goal is to design the Timeline Assembler to integrate visual and textual data to generate output timelines that precisely align with the given assembly instructions. The Timeline Assembler generates an edited timeline $\tilde{S}$ from an assembly instruction q, an initial timeline S, and a collection of video clips C. With instructions often phrased in multiple ways, ensuring consistent output results poses a considerable challenge. To tackle these challenge, the Timeline Assembler leverages an instruction-following Large Language Model (LLM) that reasons over text and visual tokens. Our overall architecture, illustrated in Figure 2, comprises a set of tokenizers that convert multimodal inputs into a set of tokens $\mathbf{X}$. These tokens are then passed to an LLM, which returns an output sequence of tokens $\mathbf{X}_{\tilde{\mathbf{S}}}$ that represent a timeline in the LLM space. Finally, we reconstruct the output timeline $\tilde{S}$ from $\mathbf{X}_{\tilde{\mathbf{S}}}$. We describe each of these components in detail next.

Collection Tokenizer. Our goal is to represent a collection of visual elements as an array of unique identifier tokens and visual tokens to enable an LLM to process the visual content in the collection. Each identifier token serves to distinctly identify the visual elements within the collection, while the

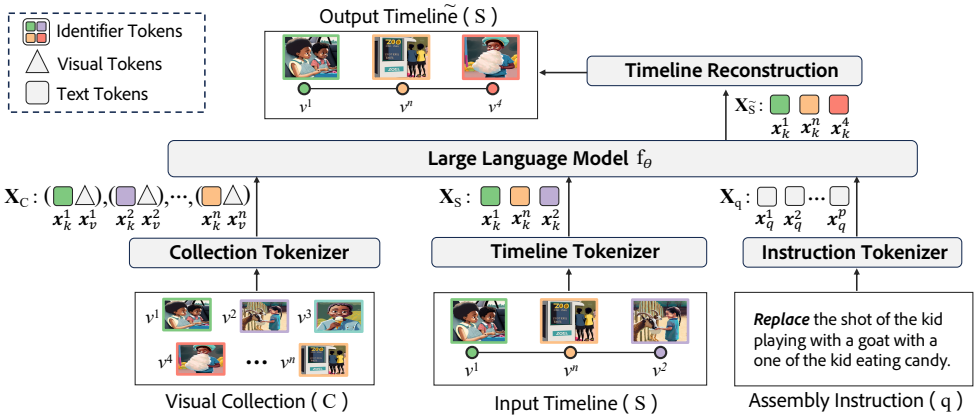

Figure 2: **Timeline Assembler Architecture.** We design a multimodal architecture to execute visual assembly instructions to generate visual timelines. Our model takes as inputs: a collection of images/videos C, a timeline S, and an assembly instruction q. Each image/video in the collection is represented with a unique identifier token $\mathbf{x}_k^i$ (color-coded) and a visual token $\mathbf{x}_v^i$, forming the sequence $\mathbf{X}_C$. The input timeline is represented with the sequence of tokens $\mathbf{X}_S$, which comprises the list of identifier tokens of the images/videos in the timeline. The assembly instruction is tokenized into $\mathbf{X}_q$. Given the input tokens, the task of the Large Language Model is to generate output timeline tokens $\mathbf{X}_{\tilde{S}}$, which are reconstructed into the output timeline $\tilde{S}$.

visual tokens encapsulate the visual information. With this design, the LLM can reference the visual representations associated with each element in the collection.

Let $C = \{v^1, v^2, \ldots, v^n\}$ denote the visual collection, where v^i represents a visual element composed of an image or video clip. The collection tokenizer has two key components: a mapping function $\mathcal{H}$ that generates a unique identifier token $\mathbf{x}_k^i$ for each visual element v^i; and a visual encoder that embeds every visual input v^i to produce a visual token $\mathbf{x}_v^i$. The mapping function $\mathcal{H}$ obtains a unique text token $\mathbf{x}_k^i$ by assigning a token from a set of previously tokenized integers. In practice, this mapping function operates as a look-up table that helps to assign (and find) the unique identifier token of a given visual element. The visual encoder ingests a visual element v^i and outputs a visual token $\mathbf{x}_v^i$. In practice, a pretrained visual encoder g ingests and extracts visual representations from the input visual elements, and a projection layer $h_\gamma(\cdot)$ maps these visual representations into a visual token that is aligned with the input space of the LLM [34]. Each visual element v^i is then represented as a tuple $(\mathbf{x}_k^i, \mathbf{x}_v^i)$ consisting of a unique identifier token and a visual token, respectively, such that the entire array of collection tokens $\mathbf{X}_C = [(\mathbf{x}_k^1, \mathbf{x}_v^1), (\mathbf{x}_k^2, \mathbf{x}_v^2), \ldots, (\mathbf{x}_k^n, \mathbf{x}_v^n)]$.

Timeline Tokenizer. The goal of the timeline tokenizer is to map a sequence of visual elements within a timeline to their corresponding identifier tokens, allocated during the collection's initial tokenization. By utilizing these pre-existing tokens, the tokenizer avoids the need for re-tokenizing visual elements as they appear in the timeline. Doing so prevents redundant visual tokenization, enabling the LLM to handle in practice more extensive collections and longer timelines. As a result, the visual elements (images or videos) are represented only once to save space. Their corresponding identifier token acts as a reference (or pointer) to the (typically high-dimensional) visual token.

In detail, let $S = \{v^{S1}, v^{S2}, \ldots, v^{Sl}\}$ denote the input timeline, where v^{Si} denotes the i-th visual element in the timeline, and l is the length of the timeline. Using the mapping function $\mathcal{H}$, we can retrieve the identifier token for each visual element v^{Si} and construct the tokenized timeline $\mathbf{X}_S = [\mathbf{x}_k^{S1}, \mathbf{x}_k^{S2}, \ldots, \mathbf{x}_k^{Sl}]$.

Instruction Tokenizer. Given an input instruction text prompt q, the goal of the tokenizer is to map the sequence of strings/words into a discrete set of p tokens such that: $\mathbf{X}_q = [\mathbf{x}_q^1, \mathbf{x}_q^2, \ldots, \mathbf{x}_q^p]$, where $\mathbf{X}_q$ is a sequence of tokens that represents the full text prompt q. In practice, we leverage the byte-level BPE text tokenizer [43] of our large language model.

Finally, the input to the LLM is the union of token sequences $\mathbf{X}$, defined as $\mathbf{X} = [\mathbf{X}_C, \mathbf{X}_S, \mathbf{X}_q]$.

Large Language Model. Our goal is to develop a model that can assemble and edit timelines of visual sequences. The model must be able to understand natural language instructions and a

multimodal context representing a visual collection and the input timeline. To do so, we leverage multimodal LLMs given their capabilities at managing multiple multimodal tasks [29, 8], reasoning over long sequences [7, 28], and encapsulating knowledge from a plethora of sources [36, 46].

Therefore, we employ a Large Language Model $f_\theta(\cdot)$, parameterized by θ, to generate an updated timeline sequence $\mathbf{X}_{\tilde{S}}$ in response to an input instruction. The model takes a sequence of previously defined multi-modal tokens $\mathbf{X}$ as input and, at test time, outputs a sequence of identifier tokens representing the updated timeline's visual elements: $f_\theta(\mathbf{X}) \rightarrow \mathbf{X}_{\tilde{S}} = [\mathbf{x}_k^{\tilde{S}1}, \mathbf{x}_k^{\tilde{S}2}, \ldots, \mathbf{x}_k^{\tilde{S}Q}]$, where Q is the output timeline length, and $\mathbf{x}_k^{\tilde{S}i}$ is the identifier token for the i-th element in the updated timeline.

Timeline Reconstruction. Given the output tokens $\mathbf{X}_{\tilde{S}}$, produced by the LLM $f_\theta(.)$, the goal of this step is to reconstruct the output timeline $\tilde{S}$. To do so, we map each output token, which in practice are identifier tokens, to their corresponding visual elements using a reverse mapping operation from $\mathcal{H}$.

3.2 Constructing Visual Assembly Tasks

Our goal is to train the Timeline Assembler to effectively perform various visual assembly tasks. To facilitate this, we define a suite of atomic operations that act as the foundational elements for constructing and manipulating visual timelines. These operations are:

- *Insert* (in): Add an element from the collection to the timeline.
- *Remove* (rm): Delete an entry from the timeline.
- *Replace* (rp): Substitute one element in the timeline with another from the collection.
- *Swap* (sw): Exchange positions of two elements within the timeline.

To reference elements, either in the timeline or the collection, we use two types of cues:

- *Positional Cue* (p): Refers to an element by its identifier or position in the timeline.
- *Semantic Cue* (r): Refers to an element through a language-based description of its visual content.

These operations and cues combine to form eight distinct assembly tasks $T = t_c$, where t represents the operations in, rm, rp, sw and c denotes the types of cues p, r . An illustration of each one of these tasks can be found in Figure A5. To generate training data for these tasks, we apply a transformation function ϕ_{t_c} to an initial timeline S^i. This function manipulates the timeline to produce a modified version $\tilde{S}^i$. For example, to train the model for the "remove" operation, we artificially introduce an additional shot into the timeline. An automatically generated caption explicitly instructs the model to identify and remove this newly added shot, thus reverting the timeline to its original form. This direct instruction ensures the model learns the specific task through controlled adjustments. Each transformation is guided by instruction templates q_t, filled with the applicable cues c^i, to ensure the instructions are clear and relevant to the task at hand. We use this procedure to generate data with multiple lengths, and multiple instructions at a time that combine multiple atomic operations in one instruction, as we will show later in the experiment Section 4. We explain the data generation procedure in more detail in Section 3.2 of the supplementary material.

3.3 Training the Timeline Assembler

We illustrate our training procedure in Figure 3. The Timeline Assembler's model $\mathcal{M}_{\gamma,\theta}$ has two learnable modules: the projection layer $h_\gamma(\cdot)$ and the language model $f_\theta(\cdot)$. Since we want to keep the multitask, instruction-following capabilities of the LLM, $f_\theta(\cdot)$, we keep its weights mostly frozen except for a lightweight set of learnable Low-Rank Adapters (LoRA) [15].

Our goal is to fine-tune the weights of the projection layer and large language model using our proxy assembly tasks gathered in the training dataset $\mathcal{D}$. In practice, we optimize the negative log-likelihood loss (NLL) as cited in Brown et al. (2020) by using the following: $\text{Loss}(\theta, \gamma) = -\sum_{i=1}^{N} \log P(\hat{\mathbf{X}}^i | \mathbf{X}^i; \theta, \gamma)$. where N is the number of samples and $P(\hat{\mathbf{X}}^i | \mathbf{X}^i; \theta, \gamma)$ is the probability assigned by the model $\mathcal{M}_{\gamma,\theta}$ to the correct output sequence $\hat{\mathbf{X}}^i$ given the input $\mathbf{X}^i$ formed by the collection tokens $\mathbf{X}_C^i$, timeline tokens $\mathbf{X}_S^i$, and assembly instruction tokens $\mathbf{X}_q^i$ for sample i.

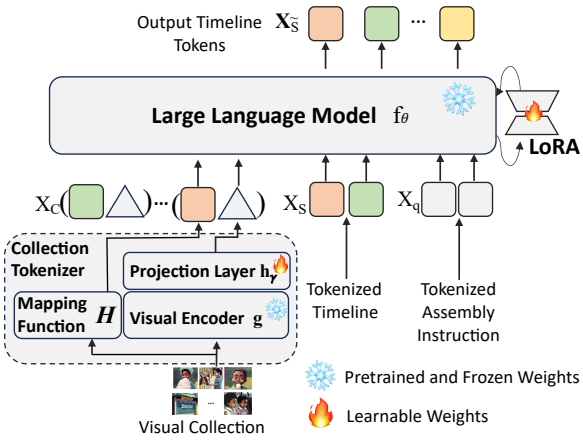

Figure 3: **Training the Timeline Assembler.** The Collection Tokenizer is composed of a frozen visual encoder $g(\cdot)$, a mapping function $\mathcal{H}$ that generates an identifier token for each input visual asset (image/video) in the collection, and a learnable projection layer $h_\gamma(\cdot)$ that maps visual embeddings into visual tokens aligned with the LLM. We keep the LLM $f_\theta(\cdot)$ mostly frozen except for a lightweight set of learnable LoRA Adapters [15].

4 Experiments

Implementation Details. Our model has four main components as shown in Figure 3: the visual backbone, the projection layer, the LLM, and the low-rank matrices from LoRA [15]. As shown in Figure 3, the visual backbone and the LLM are kept frozen throughout the training. We only train the projection layer and the LoRA parameters. For the visual encoder, we adopt the same architecture as BLIP-2 [24] with ViT-g/14 from EVA-CLIP [11] and a Q-former that outputs 32 tokens per image [24]. We initialize the projection layer with that from MiniGPT-4 [56]. Finally, for the instruction-following LLM, we use Vicuna [55]. For the visual tasks, we use BLIP-2 Flan T5 XL [24] captions as cues c. We provide additional details in Section E in the supplementary material.

Evaluation Metrics. Our system's performance is assessed using *assembly accuracy*, which determines whether a generated timeline exactly matches the corresponding ground-truth timeline. A true positive is a timeline with all its elements identical to the ground-truth, while a false positive occurs if any element in the predicted timeline differs from the ground-truth. We calculate the *Overall* assembly accuracy by evaluating each instance in the testing datasets and computing the percentage of correctly predicted samples. Additionally, we report the assembly accuracy for each cue type.

4.1 New Datasets for Instructed Visual Assembly

We present the details of our Instructed Assembly Datasets. For both cases, the size of the collection is 20 visual assets (images or videos). However, the design is not limited to this number (we show results with different collection sizes in the supplementary material). For the timeline length, we use $l = 5$ for Table 1, and $2 <= l <= 19$ for Section 4.4. Every task implies a single modification to the timeline. We use two data sources $\tilde{D}$ to create Instruction Visual Assembly Datasets. Both are described below. To ensure instruction-phrasing generalization, we ensure that the training templates do not overlap with the validation and testing ones.

Visual Storytelling Assembly Dataset (VIST-A). We use images from [17] as source of image sequences to form our new Visual Storytelling Assembly dataset. The original dataset in [17], contains Flickr images linked to each other by annotators, to create visual stories each consisting of 5 images and their captions. We use the procedure explained in Section 1 to generate offline samples for each one of the assembly tasks. For testing, we create 80 samples per task for a total of 640 visual storytelling assembly tasks. For training, we create data online.

Video Sequence Assembly Dataset (VID-A). We collect a total of $12,088$ YouTube Shorts [1]. We use these videos to show the capabilities of the Timeline Assembler for video assembly. We divide each video into shots and take the center frame as the shot representation. We create all possible timelines of a video by sampling every possible sequence of n consecutive shots. *The visual collection is formed by the shots of the video and when needed.* We construct the *Video Sequence Assembly Dataset (VID-A)* by creating 80 samples per task for a total of 640 video assembly tasks for testing. We create three validations sets using this procedure, one with timeline length of 5 for Section 4.2,

Table 1: **Instructed Visual Assembly Results.** We compare the Timeline Assembler against multiple baseline approaches. We include zero-shot results for powerful open-source and private VLMs. We compare the performance of our Assembler across various model capacities on two novel datasets, VST and VISTA, reporting assembly accuracy for Overall, Positional, and Semantic (cues). * denotes adjustment on the original implementation of the models.

	Assembly Accuracy(%)					
	VIST-A (Image-based)			**VID-A** (Video-based)		
	Positional	**Semantic**	**Overall**	**Positional**	**Semantic**	**Overall**
Zero-shot						
MiniGPT-4[56] *	0.0	0.0	0.0	0.0	0.0	0.0
LLaVA-1.5[8] *	5.6	0.0	2.8	4.7	0.3	2.5
GPT-4o	72.8	25.0	48.9	72.8	18.8	45.8
Ours						
Timeline Assembler-7B	90.1	58.1	74.1	91.8	41.8	66.8
Timeline Assembler-13B	**96.9**	**66.4**	**81.6**	**93.7**	**47.5**	**70.6**

one with variable timeline lengths for multi-len experiments in Section 4.4, and one with several instructions and variable timeline lengths for compositional experiments in Section 4.4. For training, we generate data samples using Algorithm 1 online.

4.2 Instructed Visual Assembly Results

In Table 1, we compare our model with state-of-the-art multimodal models. We observed that both MiniGPT-4 and LLaVA-1.5 lacked the capacity to manage a vast amount of visual data simultaneously. Accordingly, we adjusted these models to handle the entire collection at once; for more details on these adjustments, please refer to the supplementary material C. After modifying both models for our task, neither showed satisfactory performance in these challenging tasks. We discuss the failure cases of these two models further in Section C of the supplementary material. We attribute their inadequate performance primarily to the data and instructions on which they were trained, which significantly differ from the task of instructed visual assembly.

To establish a stronger baseline, we evaluated the recent GPT-4o and found that this model achieved very impressive zero-shot performance on both datasets, recording 48.9% on VIST-A and 45.8% on VID-A. However, when trained specifically for the task, our Timeline Assembler clearly emerged as the best alternative for instructed visual assembly, outperforming all models in both positional and semantic tasks. The smaller Timeline Assembler-7B achieves 66.8% on the VID-A dataset. When scaled up to 13B parameters, the Timeline Assembler improves to 70.6%. Thus, our strategy proves effective and further enhances the capabilities of LLMs and multimodal LLMs for assembly tasks. For a detailed breakdown of the results for each task and additional analysis on the low performance of the baselines, please refer to Table A6 in the supplementary material. Additionally, we implemented additional baselines to perform instructed visual assembly using text-only language models, leveraging BLIP-2 [24] and replacing every image with its corresponding caption. Detailed explanations and results of these additional experiments are presented in Table A5 in the supplementary material.

4.3 Analysis of the Timeline Assembler

Impact of Multi-Task Training (Table 2). We compare the Timeline Assembler against specialized models trained for each one of the assembly tasks. We use a similar training strategy but train 8 individual models, one for each assembly task on the VIST-A dataset. To determine which of the 8 models to use for a given instruction, we propose three approaches:

- *Random Selection*: selects one of the eight models for a given instruction.
- *Oracle Task Classifier*: employs an Oracle Task classifier to choose the model.
- *GPT-4o Classification*: uses GPT-4o to classify the instruction and select the appropriate model. This classifier achieves an accuracy of 81.6%. Details about the performance of the GPT-4o Task classifier are provided in the supplementary material.

Table 2: **Impact of Multi-Task Training.** We present a comparative analysis of our proposed model's performance under single-task versus multi-task training paradigms. For the single-task case, we use one model per assembly task and a Task Classifier to decide which model to use. The Random and Oracle classifiers serve as lower-bound and upper-bound references for the single-task cases, respectively. We use GPT-4o to classify every instruction into one of the tasks. Our model (featured in the last row) is a single model that understands and performs every task and does not need any task classifier. For each task we report the assembly accuracy (%) on the VIST-A dataset.

	Multi Task	**Task Classifier**	**Positional Cues**				**Semantic Cues**				**Avg.**
			Insert	Remove	Replace	Swap	Insert	Remove	Replace	Swap	
Lower-Bound	✗	Random	10.0	15.0	17.5	11.3	20.0	17.5	15.0	16.3	15.3
Upper-Bound	✗	Oracle	99.2	98.8	100.0	98.8	71.3	70.0	69.6	26.3	79.2
Single-Task Models	✗	GPT-4o	88.1	**97.6**	80.0	76.5	**67.7**	**70.0**	46.1	12.1	67.3
Timeline Assembler	✓	N/A	**98.8**	90.4	**85.4**	**85.8**	65.8	66.7	**54.2**	**45.8**	**74.1**

Table 2 shows the remarkable performance of the Timeline Assembler. It is surpassed by only 5% by the single-task models in conjunction with the Oracle classifier. This finding highlights the Timeline Assembler's strong multi-tasking capabilities. The Timeline Assembler, being a single model that effectively handles multiple tasks without the need for a task classifier, proves to be highly practical. These results suggest that incorporating more assembly tasks could further enhance the Timeline Assembler model's efficacy.

Impact of Training Scale. We analyze how data size impacts the Timeline Assembler's learning. We limit the percentage of VIST-A and VID-A data samples available during training. Since we create training data on-the-fly, cutting down the number of sequences available during training means less variability in the data. Therefore, we scale the number of epochs accordingly. Table 3a shows three sizes for VIST-A and VID-A datasets. We observe that the availability of more data for creating assembly tasks on-the-fly leads to better performance.

Cross Dataset Analysis. Table 3b presents the results when our model is trained on one dataset and tested on the other. Our model demonstrates generalizability across datasets. When the Timeline Assembler is trained on visual stories, it can still learn how to perform video sequence assembly and vice versa. Complementary to the observations in Table 3a, when merging the two datasets during training, the Timeline Assembler performs the best across both datasets by gaining a notable 6% on VIST-A and 8% on VID-A. Thus, the favored approach is to continuously incorporate more data.

Ablation of Learnable Modules. Table 3c contrasts our model's performance with and without its key components. First, when deactivating LoRA (first row), the Timeline Assembler's performance drops by a significant 31%. Second, initializing the projection layer from [56] but freezing it (second row) instead of finetuning it, substantially lowers the assembly accuracy by 12.5%. Finally, when the projection layer is trained from scratch (third row), the performance is still competitive, as it only drops by 4%. These results validate that training both, the projection layer and LoRA adapters, is crucial in training Timeline Assembler to perform instructed visual assembly tasks.

4.4 Timeline Assembler Capabilities

In this section, we evaluate the Timeline Assembler on more complex tasks inspired by real video editing applications. We aim at evaluating the Timeline Assembler under two difficult scenarios: **(i)** The first scenario consists of handling variable input timeline lengths. This scenario challenges the Timeline Assembler with varying-length sequences and increasingly harder tasks as the length increases. **(ii)** The second scenario challenges the Timeline Assembler to deal with the composition of instructions specified in a single query. In this task, the model must understand multiple instructions at once and perform several timeline modifications to successfully resolve the user's input.

Multi-Length Timeline Assembler. In the previous sections, we evaluated the Timeline Assembler's capabilities with fixed input timeline lengths of 5 assets. However, in real-life applications, dealing with multiple timeline lengths is essential. Therefore, using Algorithm 1 with the same data source as VID-A, we construct a test set for multiple-length assembly tasks, which we call multilen-VID-A.

Table 3: **Analysis of the Timeline Assembler**. (a) studies the impact of varying the training data size. (b) delves into the model's generalization by testing its performance when trained on one dataset and tested on another; 'mean' is the average accuracy across both tests. (c) presents an ablation study, indicating the model's performance with different variations of the model on VIST-A.

(a) **Training scale**

Training scale	VIST-A	VID-A
10%	17.2	15.9
50%	71.9	61.7
100%	**74.1**	**62.2**

(b) **Cross dataset analysis**

	Testing		
Training	VIST-A	VID-A	mean
VIST-A	74.1	62.2	68.2
VID-A	71.7	66.8	69.2
all	**80.0**	**68.8**	**74.4**

(c) **Learnable module ablation**

	VIST-A
w/o LoRA	43.0
frozen $h_\gamma(\cdot)$ init. from [56]	61.6
$h_\gamma(\cdot)$ from scratch	70.5
Ours	**74.1**

At training time, we also create data on-the-fly using Algorithm 1. Figure 4 reports the assembly accuracy of the assembler trained on multiple lengths against the strongest baseline GPT-4o from Table 1. We report the average assembly accuracy across all tasks within different ranges of timeline lengths (detailed per-task results can be found in the supplementary material). We observe a decrease in GPT-4o's performance as the input timeline length increases to 5 and above, indicating that the tasks become more difficult. Our Timeline Assembler performs consistently across different lengths.

Compositional Timeline Assembler. Another desirable feature for the assembler is to handle compositions of assembly instructions. This compositional capability would allow greater flexibility in the assembler's functionality. Therefore, we create a test set of compositional semantic assembly tasks, named Compositional-VID-A. Since semantic tasks are more challenging (as shown in Table 1), we combine them for our test set, presenting a highly challenging yet practical scenario for the assembler. Our test set includes combinations of two semantic tasks in a single instruction. Examples of such instructions can be found in Figures A9 to A12. In Table A8, we present different training strategies and their impact on assembly accuracy for multilen-VID-A and Compositional-VID-A. We note that GPT-4o struggles with multiple tasks simultaneously. Notably, the compositional Timeline Assembler outperforms GPT-4o and manages to perform compositional operations 36.3% of the time. Interestingly, the best way to train the Timeline Assembler for compositional tasks is to also incorporate single tasks ("Atomic-Task Training") during training, as shown in rows 2 (compositional only) vs. 3 (compositional and atomic) of Table A8. The Compositional Timeline Assembler also performs well on Multi-len VID-A and VID-A, we report these results in the supplementary material.

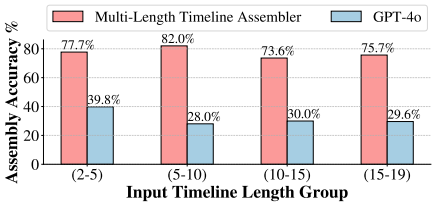

Figure 4: **Multi-Length Timeline Assembler**. Performance across various timeline lengths. Unlike GPT-4o, which has decreasing performance after timeline lengths larger than 5, our Timeline Assembler is consistent across different lengths.

Table 4: **Compositional Timeline Assembler.** We show results on compositional tasks. Our Compositional Timeline Assembler outperforms GPT-3.5. Additionally, we show the benefit of including single-tasks (atomic) during training.

Model	Training Atomic	Training Comp.	Semantic Comp. VID-A (%)
GPT-4o	✗	✗	3.8
Timeline Assembler	✗	✓	26.3
Timeline Assembler	✓	✓	**36.3**

5 Conclusion

We introduce the Timeline Assembler, the first generative model for instructed visual assembly. We train our model using automatically generated assembly tasks. We validate our approach on two newly built instruction visual assembly datasets and show that the Timeline Assembler follows assembly instructions more accurately than competing strong multi-modal large language models. Looking ahead, we believe the Timeline Assembler is a step towards generative models capable of complex reasoning (such as timeline editing or long-form story telling) over large collections of multi-modal assets. We include the limitations of our paper in A.

Acknowledgments The research reported in this publication was partially supported by funding from King Abdullah University of Science and Technology (KAUST) - Center of Excellence for Generative AI, under award number 5940.

References

[1] YouTube Shorts - Creators. https://www.youtube.com/intl/en_in/creators/shorts/. Accessed: August 15, 2023. 6

[2] J.-B. Alayrac, J. Donahue, P. Luc, A. Miech, I. Barr, Y. Hasson, K. Lenc, A. Mensch, K. Millican, M. Reynolds, et al. Flamingo: a visual language model for few-shot learning. *Advances in Neural Information Processing Systems*, 35:23716–23736, 2022. 3

[3] I. Arev, H. S. Park, Y. Sheikh, J. Hodgins, and A. Shamir. Automatic editing of footage from multiple social cameras. *ACM Transactions on Graphics (TOG)*, 33(4):1–11, 2014. 3

[4] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, X. Chen, K. Choromanski, T. Ding, D. Driess, A. Dubey, C. Finn, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. *arXiv preprint arXiv:2307.15818*, 2023. 3

[5] 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*, 33:1877–1901, 2020. 3

[6] B. Chen, A. Ziai, R. S. Tucker, and Y. Xie. Match cutting: Finding cuts with smooth visual transitions. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 2115–2125, 2023. 3

[7] G. Chen, Y.-D. Zheng, J. Wang, J. Xu, Y. Huang, J. Pan, Y. Wang, Y. Wang, Y. Qiao, T. Lu, et al. Videollm: Modeling video sequence with large language models. *arXiv preprint arXiv:2305.13292*, 2023. 3, 5

[8] W.-G. Chen, I. Spiridonova, J. Yang, J. Gao, and C. Li. Llava-interactive: An all-in-one demo for image chat, segmentation, generation and editing. 2023. 3, 5, 7, 15

[9] F. Company. Google photos just got much better. here's how [exclusive]. https://www.fastcompany.com/90938324/google-photos-just-got-much-better-heres-how-exclusive, 2024. Accessed: May 17, 2024. 2

[10] D. Driess, F. Xia, M. S. M. Sajjadi, C. Lynch, A. Chowdhery, B. Ichter, A. Wahid, J. Tompson, Q. Vuong, T. Yu, W. Huang, Y. Chebotar, P. Sermanet, D. Duckworth, S. Levine, V. Vanhoucke, K. Hausman, M. Toussaint, K. Greff, A. Zeng, I. Mordatch, and P. Florence. Palm-e: An embodied multimodal language model. In *arXiv preprint arXiv:2303.03378*, 2023. 3

[11] Y. Fang, W. Wang, B. Xie, Q. Sun, L. Wu, X. Wang, T. Huang, X. Wang, and Y. Cao. Eva: Exploring the limits of masked visual representation learning at scale. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 19358–19369, 2023. 6

[12] T.-J. Fu, X. E. Wang, S. T. Grafton, M. P. Eckstein, and W. Y. Wang. M3l: Language-based video editing via multi-modal multi-level transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10513–10522, 2022. 2

[13] S. Geng, J. Tan, S. Liu, Z. Fu, and Y. Zhang. Vip5: Towards multimodal foundation models for recommendation. *arXiv preprint arXiv:2305.14302*, 2023. 3

[14] Y. Hao, H. Song, L. Dong, S. Huang, Z. Chi, W. Wang, S. Ma, and F. Wei. Language models are general-purpose interfaces. *arXiv preprint arXiv:2206.06336*, 2022. 3

[15] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021. 3, 5, 6, 19

[16] S. Huang, Z. Jiang, H. Dong, Y. Qiao, P. Gao, and H. Li. Instruct2act: Mapping multi-modality instructions to robotic actions with large language model. *arXiv preprint arXiv:2305.11176*, 2023. 3

[17] T.-H. Huang, F. Ferraro, N. Mostafazadeh, I. Misra, A. Agrawal, J. Devlin, R. Girshick, X. He, P. Kohli, D. Batra, et al. Visual storytelling. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: Human language technologies*, pages 1233–1239, 2016. 6

[18] B. Huber, H. V. Shin, B. Russell, O. Wang, and G. J. Mysore. B-Script: Transcript-based B-roll video editing with recommendations. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2019. 2

[19] Y. Jiang, A. Gupta, Z. Zhang, G. Wang, Y. Dou, Y. Chen, L. Fei-Fei, A. Anandkumar, Y. Zhu, and L. Fan. Vima: General robot manipulation with multimodal prompts. In *Fortieth International Conference on Machine Learning*, 2023. 3

[20] kaiokendev. Extending context to 8k. https://kaiokendev.github.io/til#extending-context-to-8k, 2023. Accessed: 2023-11-26. 17

[21] S. Kwon, J. Park, M. Kim, J. Cho, E. K. Ryu, and K. Lee. Image clustering conditioned on text criteria. *arXiv preprint arXiv:2310.18297*, 2023. 3

[22] M. Leake, A. Davis, A. Truong, and M. Agrawala. Computational video editing for dialogue-driven scenes. *ACM Trans. Graph.*, 36(4):130–1, 2017. 3

[23] B. Li, Y. Zhang, L. Chen, J. Wang, J. Yang, and Z. Liu. Otter: A multi-modal model with in-context instruction tuning. *arXiv preprint arXiv:2305.03726*, 2023. 3

[24] J. Li, D. Li, S. Savarese, and S. Hoi. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. *arXiv preprint arXiv:2301.12597*, 2023. 6, 7, 13, 14, 19

[25] K. Li, Y. He, Y. Wang, Y. Li, W. Wang, P. Luo, Y. Wang, L. Wang, and Y. Qiao. Videochat: Chat-centric video understanding. *arXiv preprint arXiv:2305.06355*, 2023. 3

[26] Y. Z. B. G. N. Li and Y. Z. Q. W. Z. Yu. Representation learning of next shot selection for vlog editing. *CVEU Workshop at ICCV*, 2023. 3

[27] K. Lin, C. Agia, T. Migimatsu, M. Pavone, and J. Bohg. Text2motion: From natural language instructions to feasible plans. *arXiv preprint arXiv:2303.12153*, 2023. 3

[28] K. Lin, F. Ahmed, L. Li, C.-C. Lin, E. Azarnasab, Z. Yang, J. Wang, L. Liang, Z. Liu, Y. Lu, et al. Mm-vid: Advancing video understanding with gpt-4v (ision). *arXiv preprint arXiv:2310.19773*, 2023. 5

[29] H. Liu, C. Li, Q. Wu, and Y. J. Lee. Visual instruction tuning. *arXiv preprint arXiv:2304.08485*, 2023. 2, 3, 5

[30] R. Liu, C. Li, Y. Ge, Y. Shan, T. H. Li, and G. Li. One for all: Video conversation is feasible without video instruction tuning. *arXiv preprint arXiv:2309.15785*, 2023. 3

[31] Y. Lu, F. Ni, H. Wang, X. Guo, L. Zhu, Z. Yang, R. Song, L. Cheng, and Y. Yang. Show me a video: A large-scale narrated video dataset for coherent story illustration. *IEEE Transactions on Multimedia*, 2023. 3

[32] M. Maaz, H. Rasheed, S. Khan, and F. S. Khan. Video-chatgpt: Towards detailed video understanding via large vision and language models. *arXiv preprint arXiv:2306.05424*, 2023. 3

[33] F. Manjoo. Google photos is your perfect jukebox for all your memories. *The New York Times*, 11 2018. Accessed: May 17, 2024. 2

[34] J. Merullo, L. Castricato, C. Eickhoff, and E. Pavlick. Linearly mapping from image to text space. *arXiv preprint arXiv:2209.15162*, 2022. 4

[35] R. OpenAI. Gpt-4 technical report. arxiv 2303.08774. *View in Article*, 2(5), 2023. 15

[36] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022. 3, 5

[37] A. Pardo, F. Caba, J. L. Alcázar, A. K. Thabet, and B. Ghanem. Learning to cut by watching movies. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6858–6868, 2021. 2, 3

[38] S. Pei, J. Yu, Q. Chen, and W. He. Automatch: A large-scale audio beat matching benchmark for boosting deep learning assistant video editing. *arXiv preprint arXiv:2303.01884*, 2023. 2, 3

[39] B. Qin, J. Li, S. Tang, T.-S. Chua, and Y. Zhuang. Instructvid2vid: Controllable video editing with natural language instructions. *arXiv preprint arXiv:2305.12328*, 2023. 2

[40] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019. 3

[41] A. Rao, X. Jiang, S. Wang, Y. Guo, Z. Liu, B. Dai, L. Pang, X. Wu, D. Lin, and L. Jin. Temporal and contextual transformer for multi-camera editing of tv shows. *arXiv preprint arXiv:2210.08737*, 2022. 3

[42] A. Rao, J. Wang, L. Xu, X. Jiang, Q. Huang, B. Zhou, and D. Lin. A unified framework for shot type classification based on subject centric lens. In *The European Conference on Computer Vision (ECCV)*, 2020. 3

[43] R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015. 4

[44] Y. Shen, L. Zhang, K. Xu, and X. Jin. Autotransition: Learning to recommend video transition effects. In *European Conference on Computer Vision*, pages 285–300. Springer, 2022. 3

[45] E. Song, W. Chai, G. Wang, Y. Zhang, H. Zhou, F. Wu, X. Guo, T. Ye, Y. Lu, J.-N. Hwang, et al. Moviechat: From dense token to sparse memory for long video understanding. *arXiv preprint arXiv:2307.16449*, 2023. 3

[46] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023. 5, 17

[47] A. Truong, F. Berthouzoz, W. Li, and M. Agrawala. Quickcut: An interactive tool for editing narrated video. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 497–507, 2016. 2

[48] M. Wang, G.-W. Yang, S.-M. Hu, S.-T. Yau, A. Shamir, et al. Write-a-video: computational video montage from themed text. *ACM Trans. Graph.*, 38(6):177–1, 2019. 3

[49] Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, and H. Hajishirzi. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*, 2022. 2, 3

[50] Z. Wang, J. Li, and Y.-G. Jiang. Story-driven video editing. *IEEE Transactions on Multimedia*, 23:4027–4036, 2020. 3

[51] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021. 2, 3

[52] Y. Xiong, F. C. Heilbron, and D. Lin. Transcript to video: Efficient clip sequencing from texts. In *Proceedings of the 30th ACM International Conference on Multimedia*, pages 5407–5416, 2022. 2, 3, 17

[53] Z. Yin, J. Wang, J. Cao, Z. Shi, D. Liu, M. Li, L. Sheng, L. Bai, X. Huang, Z. Wang, et al. Lamm: Language-assisted multi-modal instruction-tuning dataset, framework, and benchmark. *arXiv preprint arXiv:2306.06687*, 2023. 3

[54] H. Zhang, X. Li, and L. Bing. Video-llama: An instruction-tuned audio-visual language model for video understanding. *arXiv preprint arXiv:2306.02858*, 2023. 3

[55] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023. 6, 14

[56] D. Zhu, J. Chen, X. Shen, X. Li, and M. Elhoseiny. Minigpt-4: Enhancing vision-language understanding with advanced large language models. *arXiv preprint arXiv:2304.10592*, 2023. 2, 3, 6, 7, 8, 9, 15

Supplementary Material

In the following document, we present the limitations of our work in Section A. Additional details on how to construct visual assembly tasks in Section 3.2. In Section C, we present additional results as well as detail explanations of the results presented in the main paper. In Sections D and E present details on the tokenizers, and implementation details of the models, respectively. Finally, Sections F, and G present qualitative failure cases and qualitative success cases of the Timeline Assembler.

A Limitations

Our paper's main limitation is efficiency. Generating timelines autoregressive is costly in high-capacity GPUs, let alone low-power devices. However, our work can directly benefit from the efforts of the community to make Language models run faster and more efficiently. Additionally, our model is only limited to the set of operations we have described. Further directions could work on incorporating another type of operations of the timeline like pixel editing or even trimming shots.

B Constructing Visual Assembly Tasks

We illustrate our assembly tasks in Figure A5. We consider 8 types of basic operations as defined in Section 3.2 of the main manuscript. Each one of the tasks is associated with a corresponding transformation function ϕ_{t_c} and a set of instruction templates q_t. Such transformation function ϕ_{t_c} is a timeline operation that we use to get a pair of input and target timelines. For instance, the remove function ϕ_{rm_c} removes one element from the input timeline S^i to get the target timeline $\tilde{S}^i$. On the other hand, the set of instruction templates q_t are templates that are filled up with the given cues c^i. For instance in Figure A5, one of the removing instructions is "`Remove the last shot`". It was created using the instruction template q_{rm} "`Remove the { } shot`" and the cue "`last`". Using all the elements previously introduced, we explain how to create an instructed assembly dataset $\mathcal{D}$ next.

Algorithm 1 Creating Visual Assembly Task Datasets

Require: Source Data $\tilde{\mathcal{D}}$
Require: Set of tasks T
Require: Dataset Size N
Ensure: Dataset $\mathcal{D}$
Initialize $\mathcal{D} = \{\}$
for i in range(N) **do**
  Initialize Collection $C^i = \{\}$
  choose sequence length l
  Sample sequence S^i of size l from $\tilde{\mathcal{D}}$
  Sample task T^i, with operation t^i, cue c^i, instruction template q_t^i and transformation $\phi_{t_c}^i$
  $\tilde{S}^i = \phi_{t_c}^i(S^i)$
  Fill up instruction template q_t^i using cue c^i
  Fill up Collection C^i using $\tilde{\mathcal{D}}$ and S
  $\mathcal{D} = \mathcal{D} \cup \{(S^i, \tilde{S}^i, q^i, C^i)\}$
end for
return $\mathcal{D}$

Creating Visual Assembly Task Datasets (Algorithm 1). Starting from an existing Data Source $\tilde{\mathcal{D}}$ comprised of visual sequences, a predefined set of assembly tasks T, with corresponding transformations ϕ_{t_c}, and a predefined template instruction q_t, we sample a timeline S^i of length l^i from $\tilde{\mathcal{D}}$, an assembly operation t^i, a cue c^i, and an instruction template q_t^i corresponding to the assembly task. We then obtain the output timeline S^i using transformation $\phi_{t_c}^i$. Then, we populate the template q_t^i using the cue c^i. Finally, we fill in the collection C with the l^i visual elements in $\tilde{S}^i$ and $|C| - l^i$ elements from $\tilde{\mathcal{D}}$. We repeat this process N times to obtain a visual assembly dataset $\mathcal{D} = \{(S^1, \tilde{S}^1, q^1, C^1), \ldots, (S^N, \ldots\}$.

C Additional Results

Performance Comparison Using BLIP-2 and Text-only Models. As mentioned before, we perform instructed visual assembly using text-only language models, by replacing each image in the collection with their corresponding BLIP2 [24] caption. By doing this, the text-only language models can now perform visual assembly tasks, seamlessly. Table A5 presents the results of the text-only models.

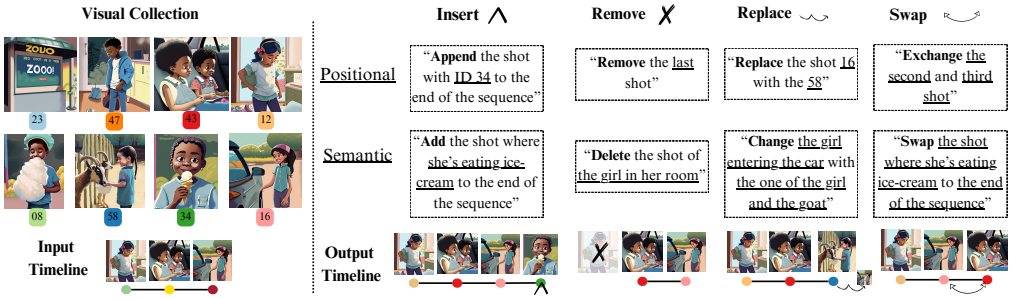

Figure A5: **Visual Assembly Tasks.** The Visual Collection (left) showcases individual visual assets labeled with their corresponding identifiers. The Input Timeline (bottom left) indicates the initial state of the timeline, while the Output Timeline (right) displays the edited timeline after applying the respective editing instructions. Each instructions is described both positionally and semantically, highlighting the system's ability to interpret and execute edits based on shot identification and semantic information. The bold font indicates the assembly operation, while the underlined font indicates the cue (shot ID or a shot description).

	Assembly Accuracy (%)								
	Positional Cues				Semantic Cues				
	Ins.	Rem.	Repl.	Swap	Ins.	Rem.	Repl.	Swap	**Avg.**
Zero-shot									
Vicuna-7B [55]	18.8	7.5	48.8	7.5	2.5	6.3	13.8	5.0	13.8
Vicuna-13B [55]	25.0	12.5	38.8	7.5	8.8	16.3	17.5	3.8	16.3
GPT-4o	60.0	96.3	63.8	71.3	**50.0**	**97.5**	**72.5**	**52.5**	**70.5**
Fine-tuned Text-Only									
Vicuna-7B [55]	17.9	99.6	**94.2**	**98.3**	0.0	20.8	4.6	11.7	43.4
Vicuna-13B [55]	**82.1**	**100.0**	80.0	97.5	4.6	20.4	6.3	5.8	49.6

(a) VIST-A

	Assembly Accuracy (%)								
	Positional Cues				Semantic Cues				
	Ins.	Rem.	Repl.	Swap	Ins.	Rem.	Repl.	Swap	**Avg.**
Zero-shot									
Vicuna-7B [55]	23.8	10.0	7.5	11.3	2.5	10.0	1.3	3.8	8.8
Vicuna-13B [55]	21.3	11.3	7.5	10.0	10.0	12.5	7.5	6.3	10.8
GPT-4o	72.5	93.8	60.0	65.0	**47.5**	**91.3**	**71.3**	**58.8**	**70.0**
Fine-tuned Text-Only									
Vicuna-7B [55]	56.3	**100.0**	**97.1**	91.3	2.5	24.6	6.7	8.8	48.4
Vicuna-13B [55]	**85.0**	**100.0**	91.3	**98.3**	7.5	22.5	5.0	10.4	52.5

(b) VID-A

Table A5: **Performance Comparison Using BLIP-2 [24] and Text-only Models.** Similar to what we saw in the main manuscript, GPT-4o surpasses the other baselines by a good margin. However, when training with out method, the vicuna models become our text-only Timeline Assemblers surpassing all the other alternatives.

Assembly accuracy per task for Table 1. Additional details on MiniGPT4, LLava, and GPT-4o. In Table A6 we showcase an expanded version of Table 1 of the main paper where we show the performance breakdown per task.

Additional details on MiniGPT-4, LLava, and GPT-4o As indicated in Tables 1 and A6, MiniGPT-4 and Llava do not perform adequately on these tasks. These models are inherently at a disadvantage because they were designed for visual instruction tuning, which typically involves handling one visual

	Assembly Accuracy (%)								
	Positional Cues				Semantic Cues				
	Ins.	Rem.	Repl.	Swap	Ins.	Rem.	Repl.	Swap	**Avg.**
Zero-shot									
MiniGPT-4 [56]	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Llava-1.5 [8]	12.5	7.5	2.5	0.0	0.0	0.0	0.9	0.0	2.9
GPT-4o	60.0	96.3	63.8	71.3	11.3	30.0	45.0	13.8	48.9
Ours									
Timeline Assembler-7B	**98.8**	90.4	85.4	85.8	65.8	66.7	54.2	45.8	74.1
Timeline Assembler-13B	89.2	99.2	**99.2**	**100.0**	54.6	**76.7**	**73.8**	**60.4**	**81.6**

(a) VIST-A

	Assembly Accuracy (%)								
	Positional Cues				Semantic Cues				
	Ins.	Rem.	Repl.	Swap	Ins.	Rem.	Repl.	Swap	**Avg.**
Zero-shot									
MiniGPT-4 [56]	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Llava-1.5 [8]	8.8	7.5	2.5	0.0	0.0	0.0	1.3	0.0	2.5
GPT-4o	72.5	93.8	60.0	65.0	10.0	21.3	27.5	16.3	45.8
Ours									
Timeline Assembler-7B	75.4	98.8	95.8	97.1	42.5	55.0	41.7	27.9	66.8
Timeline Assembler-13B	**89.6**	**100.0**	96.3	**99.6**	**58.3**	**67.9**	**64.6**	**58.3**	**79.3**

(b) VID-A

Table A6: **Performance Breakdown Per Task (Table 1 Main Paper Supplemental)**

instance at a time. Instructed visual assembly, in contrast, requires managing a large collection of at least 20 images per instruction, making these models unsuitable for the task. However, they were the closest available open-source models to serve as our baselines.

In the case of MiniGPT-4, when prompted with several images (our collection) and an instructed assembly task, it tends to produce generic captions for some images in the collection. We will showcase examples of these failure cases later in the manuscript. One notable advantage of MiniGPT-4 is the use of Q-former, which reduces the number of tokens needed to represent a single image. This allowed us to utilize this model without major modifications since our entire task fit within the context limit of MiniGPT-4. Conversely, for Llava-1.5 [8], a single image is represented by 576 tokens; with a collection of 20 images, the input sequence increases to approximately 11,520 tokens, far exceeding the original design limitations of LLMs [35]. We attempted to use vanilla Llava-1.5 but encountered significant context limitations that prevented the model from producing any meaningful output. We then modified Llava to reduce the token count per image by trying methods such as averaging all tokens, max pooling, or using the [CLS] token. We determined that using the [CLS] token was most effective for our task. Despite these efforts, Llava-1.5 was still unable to perform assembly tasks adequately. Unlike MiniGPT-4, Llava-1.5 attempted to follow instructions and often produced valid output sequences, yet it struggled to generate correct results.

For GPT-4o, we developed a prompting strategy that involved overlaying the identifier ID on each image in the collection and displaying the images as a high-resolution grid of 5x4. This approach eliminated the need for overlaying images and text from Algorithm 2 when prompting GPT-4o. This method was only feasible because GPT-4o can process high-resolution images; attempts with MiniGPT-4 and Llava-1.5 failed as the resolution was insufficient to display the entire collection with visible IDs in a mosaic format.

Expanded results of Multilen Assembler. In Table A7, we showcase per-task results of GPT-4o and the Mutilen-Timeline Assembler from Figure 5 in the main manuscript. We report the average

Table A7: **Multilen-VID-A**. Extended results of Figure 5 in main paper. We show the results per task of GPT-4o and the Multilen-Timeline Assembler on the Multilen-VID-A dataset.

	Assembly Accuracy (%)								
	Positional Cues				Semantic Cues				
	Ins.	Rem.	Repl.	Swap	Ins.	Rem.	Repl.	Swap	**Avg.**
GPT-4o	35.0	77.5	31.3	62.5	8.8	13.8	12.5	8.8	31.3
Timeline Assembler-7B	**98.8**	**96.3**	**98.8**	**98.8**	**51.3**	**53.8**	**48.8**	**75.0**	**77.7**

Table A8: **Compositional Timeline Assembler results**. We showcase the results of the Compositional Timeline Assembler on semantic tasks on VID-A and Multi-len-VID-A.

	Assembly Accuracy (%)				
	Semantic Cues				
	Insertion	Removal	Replacement	Swap	**Average**
VID-A					
Timeline Assembler	**42.5**	55.0	41.7	27.9	41.8
Compositional Timeline Assembler	40.00	**58.75**	**42.50**	**35.00**	**44.06**
Mutlilen-VID-A					
Multilen Timeline Assembler	**51.3**	**53.8**	**48.8**	**75.0**	**57.2**
Compositional Timeline Assembler	48.8	51.2	45.0	40.0	46.2

across all lengths. As observed in Figure 5 in the main manuscript, the Multilen-Timeline Assembler outperforms GPT-4o on all assembly tasks.

Compositional Timeline Assembler performance at single tasks. In Table A8, we showcase the performance of the Compositional Assembler on single tasks. We show results only on semantic cues since the model was trained on compositions of semantic tasks. Interestingly, the results show that adding compositional tasks during training improves the average performance on VIST-A. Note that the Compositional Timeline Assembler was also trained on variable input timeline lengths, so it also performance relatilevely well on Multilen-VID-A.

Table A9: **Impact of Collection Size on the Timeline Assembler trained with Collection Size 20.** We show the impact of increasing the Collection Size of a model trained with a Collection Size equal to 20 (the original model is shown as gray in the row). We evaluate the model by increasing the Collection Size without training it. We observe the performance degrades when the Collection is more than double what the model was trained for.

	Positional Cues				Semantic Cues				
Collection Size	Ins.	Rem.	Repl.	Swap	Ins.	Rem.	Repl.	Swap	**Avg.**
20	98.8	90.4	85.4	85.8	65.8	66.7	54.2	45.8	74.1
30	97.5	89.6	84.6	86.3	56.3	62.1	43.8	39.2	69.9
40	86.3	86.7	85.0	85.4	36.7	52.9	34.2	25.4	61.6
50	50.0	66.3	77.9	59.6	19.2	38.3	25.8	14.2	43.9
60	4.2	29.6	0.0	17.5	0.0	6.7	0.0	0.0	7.2

(a) Impact of Collection Size on VIST-A

	Positional Cues				Semantic Cues				
Collection Size	Ins.	Rem.	Repl.	Swap	Ins.	Rem.	Repl.	Swap	**Avg.**
20	75.4	98.8	95.8	97.1	42.5	55.0	41.7	27.9	66.8
30	80.4	97.1	93.3	95.8	28.3	49.2	45.4	21.3	63.9
40	72.9	90.8	93.8	88.8	27.5	40.0	31.7	20.0	58.2
50	40.4	50.0	64.2	53.3	13.8	26.7	17.9	14.2	35.1
60	12.9	33.3	32.5	39.6	0.0	9.6	0.0	1.3	16.2

(b) Impact of Collection Size on VID-A

Performance at different collection sizes. In Table A9, we demonstrate the Timeline Assembler-7B model's performance, which was trained with a collection size ($|C|$) of 20, when tested with varying $|C|$. As expected, a gradual decrease in performance was observed as $|C|$ increases. Notably, up to $|C| = 40$, Timeline Assembler maintains reasonable performance despite not being trained for larger $|C|$. However, at $|C| = 50$ and more significantly at $|C| = 60$, we see a substantial drop in performance, making the model less effective for the task. This finding could be due to the increased difficulty in locating a specific item as $|C|$ grows, since the chance of randomly finding an item is $1/|C|$. Additionally, the original Llama model's [46] context size limit may contribute to this effect. When $|C|$ exceeds 40, the used context approaches 2048 tokens, the Llama's limit. To mitigate this issue, we employ RoPE scaling [20] to double the context to 4096 tokens. Despite RoPE scaling's relative effectiveness, it does incur some performance degradation as $|C|$ increases, as explained in [20]. Therefore, we suggest the dramatic performance drop at $|C| = 60$ is due to both the increased task complexity and the impact of context scaling. We believe future research in enabling longer context in LLMs will empower our Timeline Assembler to handle much larger collection sizes.

Comparison with Transcript to Video[52]. We classify assembly instructions using a GPT-4o "task classifier" as described in our Section 4.3 (Table 2). Second, we extract BLIP captions for each visual asset in the timeline. We then use CLIP text embeddings to find and edit the prompt according to the assembly instruction. Our re-implementation of TtV, builds on the stronger CLIP backbone (instead of S3D). When compared ins VIST-A, we observe that TtV underperforms compared to our model (39.7% vs. 81.6%), likely due to compounding errors introduced by each step of the system.

MiniGPT-4 Failure Mode. We extended MiniGPT-4 to our task by enabling it to accommodate multiple visual elements as input, as described in Section 3. Despite this adaptation, MiniGPT-4 was unable to comprehend assembly instructions, resulting in 0% assembly accuracy and often producing text related to isolated assets in the collection without forming any interpretable output timeline (please see Figure A6). These results emphasize the novelty/contribution of our work, which lies in: (i) adapting multimodal LLMs to handle sets (the input collection) and sequences (the input and output timelines) of visual assets (video clips or images), and in (ii) developing a unique instruction fine-tuning pipeline that uses automatically generated assembly instructions for training the visual assembly task.

Figure A12 example output.

Instruction: Remove the “second"’ clip and replace it with the clip “showing a woman pouring water into a bowl of berries"".

```
{ 1: {"clip_id": "4864"},2: {"clip_id": "6085"},
3: {"clip_id": "3651"}, 4: {"clip_id": "7257"},
5: {"clip_id": "7580"}, 6: {"clip_id": "3515"} }
```

MiniGPT4 Output: ‘“a mushroom”’

(a) **Output for sample in Fig A12.** MiniGPT4 simply generates the phrase ‘a mushroom’, showing its inability to perform assembly tasks out of the shelf.

Figure A9 example output.

Instruction: Add the clip of the "bird on a tree" in the the "first" position of the timeline. Then, remove the clip of the "dog rolling in the grass".

```
{1: {'clip_id': '0975'},2: {'clip_id': '7182'},
 3: {'clip_id': '6278'},4: {'clip_id': '1817'},
 5: {'clip_id': '9096'}}
```

MiniGPT4 Output: ‘The dog’s on a tree is a "92" in the position of the timeline. Add the clip of the "dog crawling on grass" in the "first" position of the timeline, and remove the clip of the "dog crawling on grass" in the "second" position.’

(b) **Output for sample in Fig A9.** In this case, MiniGPT4 gives some descriptions related to the iamge and the instruction, but does not output the updated timeline.

Figure A6: **Examples of MiniGPT-4 Outputs.** We showcase three examples of MiniGPT4 outputs for the assembly samples in our dataset. MiniGPT4 yields close to 0% assembly accuracy due to its inability to understand the assembly tasks. To observe how the Timeline Assembler succesfully performs the assembly tasks please refer to figures in Section G.

D Input Tokenizer

Visual-Text Inputs. In Section 3 of the main paper, we explained how we manage the visual elements through a collection that contains all the visual information. This modeling allows us to manage the timeline as pointers to the visual elements. To clarify how exactly we do this process, please refer to Figure A7 that shows an example before being tokenized and fed to the LLM. Figure A7 shows the basic structure of the collection containing the fields `clip_id` and `clip`. We use these two fields to store the information of each clip, such that `clip_id` contains the identifier token $\mathbf{x}_k^i$ for every clip, and `clip` contains the visual information $\mathbf{x}_v^i$) which stores the visual tokens coming from the output of the projection layer h_γ. Algorithm 2 shows how we parse each input to place the visual token in their corresponding fields in the collection. In a nutshell, Algorithm 2 splits the inputs into several pieces based on the string "`<VisualHere>`". For each piece we know that there is a visual token separating them (given the structure of the collection shown in Figure A7). Thus, we simply find all the "`<VisualHere>`" placeholders and replace them with the visual tokens coming the visual backbone g and the projection layer h_γ. The rest of the input is simply processed as standard text.

Algorithm 2 Visual-Text Tokenizer Function

Require: $collection, prompt$
Ensure: $prompt$ is a string
Ensure: $collection$ is a list
Ensure: $collection[i]$ is an Image
  $prompt_splits \leftarrow$ split($prompt$, "<VisualHere>")
  $sequence \leftarrow$ empty list
  for $i \leftarrow 0$ **to** length($prompt_splits$) $- 1$ **do**
    $prompt_piece = prompt_splits[i]$
    $prompt_piece_emb =$ llm.embed($prompt_piece$)
    append($sequence, prompt_piece_emb$)
    if $i <$ length($prompt_splits$) $- 1$ **then**
      $visual_embed =$ vit.embed($collection[i]$)
      append($sequence, visual_embed$)
    end if
  end for
  return $sequence$

Text-Only Inputs. For text-only baselines, we replace the string "`<VisualHere>`" with the BLIP-2 [24] generated caption of the corresponding visual asset, and tokenize the entire input as text. In this case, Algorithm 2 is not needed.

E Implementation Details

Additional Training Details. In Table A10 we show more training details, including the LoRA [15] hyper-parameters. Our model trains on 1 GPU A100 for 100k iterations, which is around 16hours of compute per training. At test time, the model takes around 5 seconds per instruction on the same hardware.

Assembly Task Classifier with GPT4o. In the multi-task analysis (Table 2) of the main paper we used a task-classifier based on GPT-4o to decide which model to use depending on the input query. We use the input prompt shown in Figure A8 to query GPT-4o and classify into one of eight tasks, each of the assembly instructions in our testing set. The results of this classifier are shown in Table A11. We observed that the Semantic Swap task can be challenging to identify,

Input Example

Human: You are presented with a collection of images, you have to modify the timeline accordingly, to complete the instruction you are given.
Collection:

```
[
{"clip_id":0935,"clip":"<VisualHere>"},
{"clip_id":0721,"clip":"<VisualHere>"},
...
{"clip_id":0612,"clip":"<VisualHere>"},
{"clip_id":0899,"clip":"<VisualHere>"},
...
{"clip_id":0327,"clip":"<VisualHere>"}
]
```

The current timeline is:

```
{
    1: {"clip_id": 0894},
    2: {"clip_id": 0374},
    3: {"clip_id": 0899},
    4: {"clip_id": 0464},
    5: {"clip_id": 0665}
}
```

Instruction: Remove the "second" clip and replace it with the clip "showing a woman pouring water into a bowl of berries".

```
{
    1: {"clip_id": 0894},
    2: {"clip_id": 0374},
    3: {"clip_id": 0899},
    4: {"clip_id": 0464},
    5: {"clip_id": 0665}
}
```

Figure A7: **Input Example.** We show an example of the whole input required to perform assembly tasks. This input includes the Collection, the Input Timeline, and the Instruction that tells the model what kind of operation to perform on the Timeline.

which might be one of the reasons why this task is the lowest-performing one across all models.

Table A10: **Training Parameters**

Configuration	Value
LoRA - r	16
LoRA - α	32
LoRA - dropout	0.1
Optimizer	AdamW
Optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.999$
Weight decay	0.05
Base learning rate	6×10^{-5}
Warmup learning rate	5×10^{-7}
Minimum learning rate	2×10^{-6}
Learning rate schedule	cosine decay
Warm-up iterations	*VST*,*VISTA*=$7K$,$10K$
Total iterations	*VST*,*VISTA*=$70K$,$100K$
Batch size	1

Table A11: **GPT-4o Task-Classifier.**

Task	**Accuracy**
Positional Insertion	88.8%
Positional Removal	98.8%
Positional Replacement	80.0%
Positional Swap	77.5%
Semantic Insertion	95.0%
Semantic Replacement	66.2%
Semantic Removal	100.0%
Semantic Swap	46.2%

GPT-4o Classifier: Input Prompt

TEMPLATE = You task is to classify an assembly instruction into 8 different categories. There are four different types of operations:
Insert: Inserts an element from the visual collection into the timeline.
Remove: Deletes an element from the timeline.
Replace: Changes an element from the timeline with one from the collection.
Swap: Interchanges two elements from the timeline.
We can refer to the visual elements by:
(a) a semantic reference, which describes the visual appearance of the elements with text, and
(b) ID reference, which specifies the elements to manipulate by their unique ID and position in the timeline.
That said, in total there could be 8 categories, which we provide an example for each below:
positional_insertion: "Append the shot with ID 34 to the end of the sequence"
positional_removal: "Remove the last shot"
positional_replacement: "Replace the shot 16 with shot 58"
positional_swap: "Exchange the second and third shot"
semantic_insertion: "Add the shot where the girl is eating ice-cream to the end of the sequence"
semantic_removal: "Delete the shot of the girl in her room"
semantic_replacement: "Change the shot of the girl entering the car with the one of the girl and the goat"
semantic_swap: "Swap the shot of the girl eating ice-cream with the last shot of the sequence"
IMPORTANT: Just output the query class. DO NOT explain.
Query: "{ }"

Figure A8: **Input query for GPT-4o Task Classifier.**

F Failure Modes

We include failure modes of the Timeline Assembler in Figures A9 to A12. We explain each failure mode in details inside the red boxes. In general, the Timeline Assembler's failures often occur due to misinterpretation of instructions (Figures A10,A11,A12), not finding elements in the collection that match the visual description (Figures A9,A11,A12), or executing only one of the of the two tasks it was instructed to do (Figures A10, and A11).

G Additional Qualitative Results.

In Figures A9 to A12 we show more qualitative results. We include the collections that we omitted in the main paper due to space reasons. We can see the Collections along with some examples in the same order that they appeared in the main paper. We show 3 additional examples that showcase the Timeline Assembler's ability to manage timelines and modify them given a wide variety of instructions.

Figure A9: **Qualitative Results 1.** We showcase an extended version of the first example in the main paper. We showcase its collection and additional examples.

Figure A10: **Qualitative Results 2.** We showcase an extended version of the second example in the main paper. We showcase its collection and additional examples.

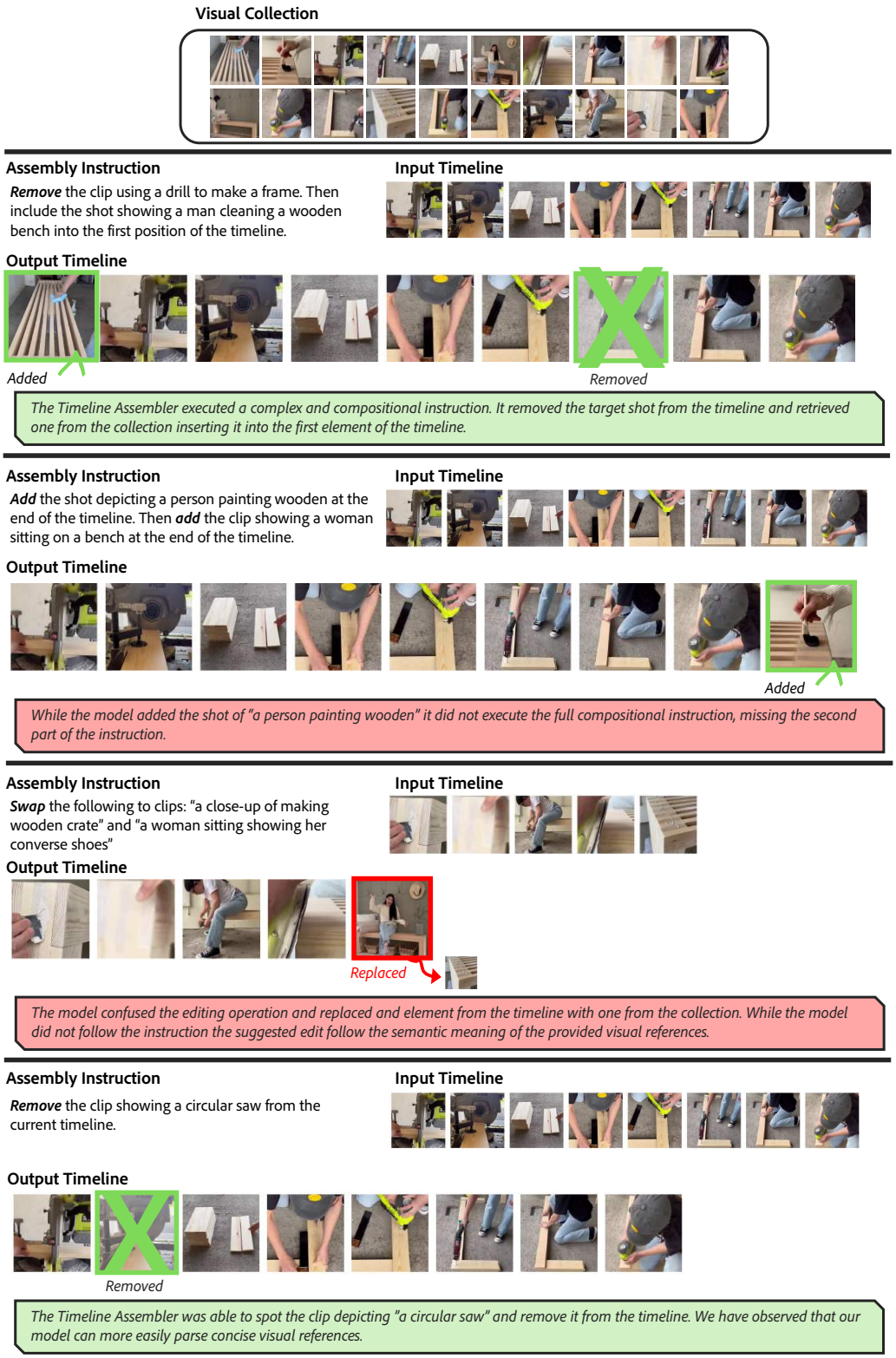

Figure A11: **Qualitative Results 3.** We showcase an extended version of the third example in the main paper. We showcase its collection and additional examples.

Figure A12: **Qualitative Results 4.** We showcase an extended version of the fourth example in the main paper. We showcase its collection and additional examples.