Atom: Efficient On-Device Video-Language Pipelines Through Modular Reuse

Kunjal Panchal
University of Massachusetts Amherst
Amherst, MA, USA

Saayan Mitra
Adobe Research
San Jose, CA, USA

Somdeb Sarkhel
Adobe Research
San Jose, CA, USA

Haoliang Wang
Adobe Research
San Jose, CA, USA

Ishita Dasgupta
Adobe Research
San Jose, CA, USA

Gang Wu
Adobe Research
San Jose, CA, USA

Hui Guan
University of Massachusetts Amherst
Amherst, MA, USA

Abstract

Recent advances in video-language models have enabled applications like video retrieval, captioning, and assembly. However, executing such multi-stage pipelines efficiently on mobile devices remains challenging due to redundant model loads and fragmented execution. We introduce Atom, an on-device system that restructures video-language pipelines for fast and efficient execution. Atom decomposes a billion-parameter model into reusable modules, such as the visual encoder and language decoder, and parallely reuses them across subtasks like captioning, reasoning, and indexing. This reuse-centric design eliminates repeated model loading and enables parallel execution, reducing end-to-end latency without sacrificing performance. On commodity smartphones, Atom achieves 27–33% faster execution compared to non-reuse baselines, with only marginal performance drop (≤2.3 Recall@1 in retrieval, ≤1.5 CIDEr in captioning). These results position Atom as a practical, scalable approach for efficient video-language understanding on edge devices.

CCS Concepts

• **Computer systems organization** → *Real-time systems.*

Keywords

On-Device Machine Learning, Mobile AI Inference, Video Retrieval, Video Assembly

ACM Reference Format:
Kunjal Panchal, Saayan Mitra, Somdeb Sarkhel, Haoliang Wang, Ishita Dasgupta, Gang Wu, and Hui Guan. 2026. Atom: Efficient On-Device Video-Language Pipelines Through Modular Reuse. In *ACM Multimedia System Conference (MMSys '26), April 4–8, 2026, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3793853.3795759

MMSys '26, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-2481-7/26/04
https://doi.org/10.1145/3793853.3795759

1 Introduction

Video-language model (VideoLM) pipelines [8, 12, 29] are rapidly becoming the engine behind emerging mobile applications such as on-device video retrieval [20, 32] and video assembly [30], where users locate or stitch together video clips through natural-language prompts. Such applications rely on multi-stage pipelines that typically involve three core subtasks: video encoding, caption generation, and downstream reasoning or indexing. Running these pipelines locally is highly desirable because it preserves privacy and does not require a cloud server infrastructure.

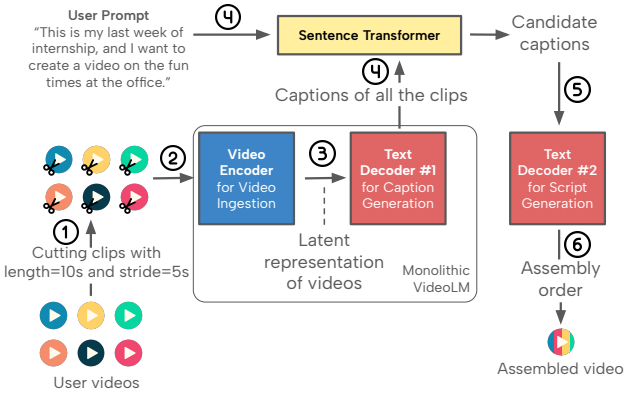
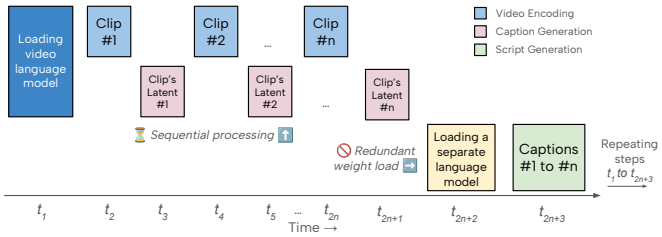
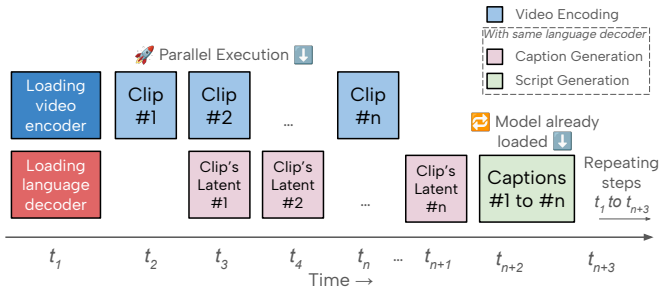


Figure 1: An end-to-end on-device sequential pipeline for video assembly. Given (i) user videos and (ii) a user prompt, the pipeline converts all the video clips into their text representations, generates a script and assembles selected video clips accordingly. Video Clip icons by Good Ware via Flaticon.com. Scissors (Cut) icons by Fajrul Fitrianto via Flaticon.com.

However, deploying VideoLM pipelines on mobile devices introduces fundamental challenges due to both memory and latency constraints. In standard pipelines, subtasks such as video encoding, caption generation, and script generation are handled by separate models [27, 30], each accruing its own memory footprint and processing time. For instance, even after applying 8-bit quantization, mPLUG2 [26], a model popularly used for video-to-caption generation, requires approximately 5.6GB of RAM. Adding a script-generation model such as Llama 3.2 can consume an additional



(a) Sequential execution in de facto video-language pipelines. Subtasks are executed one after another, with separate models loaded at each stage (e.g., at t_1 and t_{2n+2}), resulting in higher latency and memory overhead.



(b) Modular, reuse-centric, and parallel execution in Atom. It reduces pipeline latency of video-language tasks by modularizing the video encoder and language decoder; and parallelizing their execution across different inputs (e.g., starting at t_3).

Figure 2: Sequential (*left*) vs. Atom's modular-parallel (*right*) pipeline execution for video assembly task which requires modules for video encoding, caption generation, and script generation.

1.4GB. Running the full pipeline would therefore require around 7GB of memory, which can strain mobile devices that typically have only 6 to 8GB of RAM. As a result, models used in the pipeline are loaded sequentially at each stage, as shown in Figure 1. This back-to-back loading prohibits pipelined execution across tasks and inflates end-to-end latency. Empirically, we observe up to a 17% increase in latency when reloading quantized versions of these models between subtasks. This inefficiency not only prolongs wait time but also limits scalability across multiple user prompts, where each prompt restarts the model-loading cycle.

Prior works have predominantly optimized individual model performance: accelerating token generation or reducing memory consumption per model [4, 9, 25]. While effective in improving pipeline performance, such efforts overlook *pipeline-level inefficiencies* inherent to chaining multiple subtasks. In a conventional video assembly pipeline (Figure 2a), all video clips are first passed through the video encoder and caption generator before a separate script generator is invoked. This serialized execution means (a) the shared VideoLM cannot process the next batch of video clips while script generation is still running, and (b) the use of a standalone script-generation model imposes a memory tradeoff: either keep both models in memory (which may exceed device limits) or pay the overhead of loading them sequentially. These limitations amplify latency and hinder responsiveness on edge devices. Overcoming them requires a modular, reuse-centric execution model that enables parallelism across inputs and subtasks.

Our Approach. We propose Atom[1], a reuse-centric inference system that accelerates video-language pipelines by addressing the pipeline-level inefficiencies. Unlike the sequential baseline, which uses separate models for each subtask in the pipeline, our approach leverages a single modularized model architecture that supports all subtasks, enabling end-to-end reuse without compromising model quality. Specifically, Atom decomposes the VideoLM into persistent and reusable modules: a video encoder and a generalizable language decoder, both of which remain resident in memory throughout the application's lifetime. By reusing these modules across subtasks and avoiding repeated model loading, Atom eliminates data movement overhead and enables parallel execution without increasing memory usage. This reuse-centric design is compatible with existing model compression techniques, offering additional opportunities to reduce memory consumption.

Figure 2 contrasts this reuse-centric design with the de facto multi-model baseline. In the status quo (Figure 2a), each video clip triggers a strictly serial workflow: the device first loads a monolithic encoder-decoder pair to generate captions sequentially for each video clip and then bring in a separate language model for script generation. Our approach (Figure 2b) decomposes a VideoLM into separately callable encoder and decoder modules that operate concurrently: while the decoder produces captions for clip #1, the encoder can already process clip #2. Then the same decoder is used for processing all the captions for generating a script for video assembly. This modular parallelism reduces cumulative model-loading time and boosts hardware utilization without increasing the memory footprint, delivering substantial end-to-end speed-ups on mobile devices.

Empirical Results. We evaluate Atom on two representative video-language tasks (video retrieval and video assembly) on commercial mobile devices including the Google Pixel 5a, Pixel 8a, and Samsung Galaxy S23. Compared to conventional multi-model pipelines, Atom (a) reduces model loading latency by up to 50%; (b) achieves up to 33% lower end-to-end latency through reuse and parallelism; (c) maintains competitive output quality with only marginal differences in performance metrics (e.g., 1.3–2.3 drop in Recall@k for retrieval, 0.6–1.5 drop in CIDEr for captioning); and (d) keeps peak memory virtually unchanged (only 0.76% ≈ 40MB higher than the baseline) while using a stronger, reusable text decoder that still fits within a 6GB smartphone RAM budget. While our evaluation focuses on video retrieval and video assembly, the reuse-centric pipeline design introduced by Atom is broadly applicable to a range of video-language applications, including summarization, question answering, and temporal event localization.

This work makes the following contributions:

- We propose a reuse-centric pipeline design that highlights a new system-level direction for efficient video-language inference. The design accelerates video-language inference by sharing a single model's internal modules across subtasks. This eliminates the overhead of managing multiple models during execution.

[1]The name Atom reflects its design goal of enabling video-language models to run under low latency and small memory budgets on edge devices through modular decomposition and reuse.

- We develop Atom, a mobile inference system that enables the execution of full video-language pipelines on-device without relying on server infrastructure.
- We evaluate Atom on real-world smartphones and demonstrate significant latency reductions and system-level efficiency improvements while preserving high output quality across tasks.

2 Related Works

We organize related work into three categories: (a) on-device inference for large-language models (LLMs), (b) on-device inference for vision-language models (VLMs) and deployment strategies, and (c) pipelines for video-centric tasks such as assembly and retrieval. Atom bridges these areas with a reuse-centric design that improves end-to-end latency of video-language pipelines on mobile devices.

On-device Large Language Models. Recent efforts have explored running LLMs on mobile and edge devices. `llm.npu` [25] accelerates the prefill stage of LLM inference on mobile NPUs, achieving a 22.4× speedup for text-only models. However, it is limited to unimodal (text) models and does not support multimodal pipelines involving vision inputs or cross-modal reasoning. MobileLLM [11] proposes a lightweight text-based model architecture with sub-billion parameter count, optimized for mobile hardware via embedding sharing and grouped-query attention. In contrast, Atom targets billion-parameter-scale video-language models that require cross-modal understanding and support complex video-based applications. Hybrid solutions like EdgeShard [33] offload partial inference to high-end servers, compromising user asset privacy. Other works, such as LLMCad [24] and Any Precision LLM [17], aim to accelerate a single LLM's inference via smaller model-based text generation and support for multiple quantization precisions, respectively. Model compression techniques such as pruning [15], distillation [19], and quantization [10, 14] are complementary to our work. Atom integrates quantization but focuses primarily on architectural and pipeline-level reuse to reduce overhead and latency during multi-subtask execution.

On-device Vision-Language Models. Recent work has explored adapting VLMs for mobile hardware. MobileVLM [4] and Spotlight [9] demonstrate that billion-parameter VLMs can run efficiently on smartphones by pairing lightweight visual encoders with language decoders. However, these efforts primarily target image-based tasks and assume single-stage inference. They do not address the challenge of executing multi-stage pipelines, where repeated model loading and lack of reuse can introduce substantial latency overhead. Atom complements these efforts by focusing on architectural reuse within the pipeline to accelerate full-stack video-language inference on-device, rather than optimizing individual models in isolation.

Video-Centric Pipelines. Video-language tasks such as assembly and retrieval typically rely on modular pipelines composed of separate models for video encoding, captioning, and reasoning [27, 30]. While effective, this approach incurs high runtime overhead, particularly on mobile devices, due to the need to repeatedly load and execute distinct models for each subtask into limited memory. For video assembly, methods like RATV [30] and TV-MGI [31] align video segments to human-written scripts using powerful server-hosted models. These approaches assume expert-written inputs and high-resource environments. Atom, by contrast, automates both script generation and clip selection directly on-device using a unified model, making the process more accessible to end users. In video retrieval, prior work such as CLIP4Clip [13] and Frozen [1] relies on embedding-based alignment between video and text. More recent methods [7, 27] integrate LLM-based reasoning to enhance retrieval quality, but depend on large proprietary models and cloud infrastructure. These methods are not deployable in resource-constrained or privacy-sensitive settings. Atom shows that retrieval, assembly, and captioning can all be supported within a single reusable model on mobile devices, eliminating the need for cloud-based processing or manual supervision.

3 Design of Atom

Atom accelerates video-language pipeline inferences by eliminating pipeline-level redundancies through three key design principles:

- *Modularization:* A VideoLM is split into a video encoder and a stronger, reusable language decoder, each independently quantized and callable for reuse and concurrent execution.
- *Reuse:* Modules are persistently kept in memory and shared across subtasks, avoiding repeated loading.
- *Parallelism:* Decoupled modules enable thread-safe parallel execution, improving throughput and device utilization.

Here we focus on the system-level design of Atom, which reduces latency of VideoLM pipelines. This design is model-agnostic.

3.1 Design Overview

We discuss Atom's framework using the video assembly task as a primary example (Figure 1). The same principles apply to other applications, such as video retrieval, with minor task flow variations.

Given a user prompt (e.g., "make a highlight reel") and a set of user-provided videos, the pipeline proceeds as follows:

(1) **Video segmentation.** Input videos are divided into overlapping clips of fixed duration ($\ell = 10s$) with a stride of $s = 5s$, enabling finer temporal granularity and reducing memory load per inference call.
(2) **Video captioning.** Each clip is passed through the VideoLM in a two-stage process: (a) The video encoder extracts high-dimensional visual embeddings. (b) The text decoder generates a concise natural language caption from the visual embedding.
(3) **Embedding and ranking.** Captions and the user prompt are embedded using a Sentence Transformer. Clip relevance is computed via cosine similarity between caption and prompt embeddings.
(4) **Script generation.** The top-k relevant captions are composed into a coherent video script using the same VideoLM text decoder, now operating in text-only mode. This script determines the selected clip order and narration.
(5) **Final assembly.** Selected clips are stitched together on-device into the final video output.

Preprocessing for Mobile Constraints. To reduce inference latency and memory overhead, we preprocess video clips using the following strategies:

- *Resolution and frame rate reduction.* We downsample smartphone video input in both spatial and temporal dimensions.
- *Clip formatting.* Each video clip is reshaped to a tensor of size $(1, 3, 6, 224, 224)$, corresponding to batch size, channels, sampled frames, height, and width. This size achieves a good trade-off between accuracy and efficiency for scene-level understanding.

The entire pipeline, from segmentation to script generation, runs fully on-device, benefiting from persistent module reuse to avoid unnecessary reloading. Figure 1 visualizes this process.

3.2 Modular Execution of VideoLM Pipelines

In traditional monolithic VideoLM pipelines [30], tasks such as video captioning and script generation (Figure 1) are handled by separate models or tightly coupled end-to-end architectures. This leads to high latency from serialized execution and repeated model loading, and further prevents concurrent execution across subtasks.

We formalize the issue of high latency as follows: Let a video-language pipeline consist of T subtasks $\tau_1, \tau_2, \ldots, \tau_T$, each implemented by a model M_t (this shows the worst case scenario where every task requires a separate model) with parameter size S_t and execution latency L_t. In a *monolithic pipeline*, total latency and peak memory are:

$$L_{\text{mono}} = \sum_{t=1}^{T} (L_t + L_t^{\text{load}}), \qquad M_{\text{mono}} = \max_t (S_t + A_t), \tag{1}$$

where L_t^{load} denotes model-loading latency and A_t the peak activation memory during inference.

Under modularization, the VideoLM is decomposed into shared modules E, D (encoder, decoder) such that each subtask τ_t uses a subset of them, e.g., $\tau_t = f_t(E, D; \theta_t)$ with optional small task adapters θ_t. The total latency becomes:

$$L_{\text{mod}} = L_E^{\text{load}} + L_D^{\text{load}} + \sum_{t=1}^{T} L_t, \tag{2}$$

where $L_E^{\text{load}}, L_D^{\text{load}}$ occur once at initialization. Thus, modularization removes $(T-1)$ redundant load operations, yielding an expected latency reduction of $\Delta L_{\text{load}} = \sum_{t=2}^{T} L_t^{\text{load}}$.

This formulation highlights that the primary inefficiencies of monolithic VideoLMs arise from repeated model loading and strictly sequential execution. Building on this insight, Atom mitigates these issues through modular decomposition, exposing the video encoder and text decoder as independently quantized, callable modules. This design enables concurrent execution: for example, encoding a new video clip while decoding captions from prior clips.

Modularization further allows reuse of the same high-capacity language decoder across all text-centric stages. Replacing multiple specialized, lightweight decoders with a single shared decoder (a) eliminates repeated model loading, (b) promotes generalization by applying a stronger linguistic prior across tasks such as captioning, retrieval indexing, and script generation, and (c) enables the entire pipeline to operate within tight mobile memory budgets.

In contrast, pipelines that load separate models for captioning and script generation often exceed the 6 GB RAM limit of common smartphones. Atom avoids this constraint while improving quality; empirical comparisons are provided in Section 5.2.

3.3 Persistent Module Reuse

Even with modularization, sequential pipelines using task-specific models repeatedly reload modules across stages, wasting compute and memory bandwidth. Atom eliminates this overhead via a persistent reuse strategy, keeping core components; such as the video encoder and a unified, stronger language decoder-resident in memory across tasks. For instance, the decoder used for captioning is reused for script composition, avoiding the need to load a separate language model.

This reuse strategy reduces latency and prevents memory thrashing under tight resource constraints. As shown in Figure 2a, non-reuse pipelines suffer significant overhead from repeated model loading and unloading, particularly with large language decoders. In contrast, Atom (Figure 2b) keeps modules resident throughout execution, enabling efficient subtask transitions and smoother operation on mobile hardware.

To formalize the effect of this design, let M_{reuse} denote the peak memory under persistent reuse. Since the encoder E and decoder D remain resident throughout execution, the total memory requirement can be expressed as:

$$M_{\text{reuse}} = S_E + S_D + \max_t A_t. \tag{3}$$

Here, S_E and S_D are the parameter sizes of encoder and decoder, and $\max_t A_t$ captures the highest activation memory across subtasks.

In contrast, sequential reloading introduces additional transient allocations during model swaps. The corresponding peak memory for such pipelines is therefore:

$$M_{\text{seq}} = \max_t (S_t + A_t) + B_{swap}, \tag{4}$$

where B_{swap} denotes the transient memory overhead introduced by model swapping, including weight-loading buffers, initialization tensors, and temporary I/O allocations. Although these buffers are short-lived, they contribute to peak memory usage and can trigger out-of-memory failures.

This comparison highlights that while persistent reuse does not reduce the intrinsic model size (S_E, S_D), it effectively removes redundant transient allocations, yielding a more stable memory footprint and smoother runtime behavior, especially under mobile constraints. Sections 5.4 and **??** confirm this observation: although the weight and activation footprints are similar, the reuse-centric design significantly lowers auxiliary memory usage (e.g., temporary allocations during weight loading).

3.4 Parallelism through Modular Design

The goal and a key advantage of modularizing the VideoLM is enabling *parallel execution* of subtasks, something infeasible with monolithic or tightly coupled models. Traditional video-language pipelines enforce serialized execution due to thread-unsafe or memory intensive models: caption generation must wait for video encoding to finish, and script generation must wait for captioning to complete. This leads to under-utilization of compute resources and increased end-to-end latency.

In contrast, Atom's decoupled encoder and decoder are exposed as independently callable, thread-safe modules that persist in memory throughout execution. This persistent modular design allows pipeline stages to overlap: for example, while the decoder generates

captions for clip #1, the encoder can begin processing clip #2. Figure 2b illustrates how overlapping compute windows boost throughput and improve hardware utilization on edge devices.

To quantify the impact of this overlap, consider the following simplified latency model. Let L_E and L_D denote the average per-clip latency of the encoder and decoder. In traditional serialized execution, total latency scales linearly with the number of clips:

$$L_{\text{serial}} = N(L_E + L_D), \tag{5}$$

where N is the total number of clips. When modular execution introduces partial overlap between encoder and decoder, captured by overlap ratio $\alpha \in [0, 1]$, the effective latency becomes:

$$L_{\text{parallel}} = N\big[(1-\alpha)L_E + \alpha L_D\big]. \tag{6}$$

Higher overlap (larger α) yields greater throughput gains, particularly when encoder latency dominates. Under ideal conditions ($\alpha \approx 1$), Atom achieves near-full overlap, with clip #(i+1) encoding proceeding fully in parallel with clip #i decoding. As a result, the effective per-clip latency approaches $\max(L_E, L_D)$ rather than their sum, substantially reducing end-to-end runtime.

While this parallel execution increases peak activation memory, we show in Section 5.4 that Atom offsets this by eliminating temporary memory overhead from repeated model loading, resulting in only 40MB higher overall usage.

Finally, we summarize the latency-memory tradeoff using a conceptual cost formulation:

$$C = \lambda_L \widetilde{L}_{\text{total}} + \lambda_M \widetilde{M}_{\text{peak}}, \tag{7}$$

subject to the hardware constraint $\widetilde{M}_{\text{peak}} \le M_{\text{device}}$. Here, $\widetilde{L}_{\text{total}}$ and $\widetilde{M}_{\text{peak}}$ denote normalized latency and memory terms, respectively, and λ_L and λ_M are platform-dependent weighting coefficients that reflect the relative importance of latency versus memory on a given device. This formulation provides an abstract lens for reasoning about design tradeoffs under heterogeneous units and constraints.

In practice, Atom treats memory as a hard constraint imposed by the target device, and focuses on minimizing end-to-end latency within the feasible memory budget. Persistence module reuse primarily reduces loading overhead ($\downarrow L_{\text{load}}$), while encoder–decoder overlap reduces execution time ($\downarrow L_{\text{total}}$), together achieving improved performance without exceeding M_{device}.

Applicability to Other Tasks. While our analysis focuses on video retrieval and assembly for concreteness, the modular and reuse-centric design in Atom generalizes to other video-language tasks, including summarization, and question answering. We demonstrate this with additional task implementation outlines described in Supplementary Material Section 1.

4 Implementation of Atom

Here we describe implementation-specific modifications to a VideoLM to enable operation within the tight memory constraints of mobile devices. We implement Atom on top of the video-language foundation model mPLUG2 [26]. We modularize its components to support persistent module reuse and parallel execution, enabling efficient on-device inference.

We choose mPLUG2 for three reasons: (a) it has demonstrated strong performance in video-language tasks like captioning and retrieval [3, 22, 23], (b) it is open-source and modular, allowing targeted architectural changes, and (c) it is compatible with quantization and serialization frameworks such as TorchAO [16] for mobile deployment. The original mPLUG2 consists of a BERT-based text encoder and decoder (0.3B parameters each), a CLIP-ViT/L-14 visual encoder (0.4B), and a fusion module (0.5B), totaling 1.5B parameters. However, it is not optimized for task-level reuse: the decoder requires fine-tuning for different language tasks (e.g., captioning vs. script generation), increasing training and runtime overhead.

To enable reuse across subtasks, we make two key modifications. First, we discard the text encoder and fusion module, which are not required for our tasks. Second, we replace the BERT-based decoder with a 1B parameter Llama 3.2-based autoregressive decoder [5]. This decoder generalizes across captioning and reasoning tasks, allowing the same module to serve both roles without retraining.

The resulting model, which we refer to as mPLUG2+, consists of a CLIP-based video encoder and a Llama-based text decoder, totaling 1.4B parameters. We fine-tune the model on video captioning datasets to align the visual and text components; script generation requires no additional training. Compared to the original mPLUG2, this update improves captioning (by 2.8–3.7 CIDEr) and retrieval (by 3.8–4.9 Recall@1) performance, as shown in Section 5.2. To support reuse-centric execution, each module is serialized independently and kept persistently loaded. This design enables both shared reuse and thread-safe parallelism across subtasks, allowing reliable execution on mobile devices with 6GB+ RAM.

Adaptability to Other VideoLMs. While our implementation uses mPLUG2+ as the base model, Atom's reuse-centric design is broadly compatible with many modern VideoLMs that follow a modular architecture, such as BLIP-2, Flamingo, and VideoCoCa. These models decouple video encoders and language decoders, making them suitable for persistent reuse. Although earlier tightly coupled models with joint cross-modal attention layers may resist such decomposition, deployment on mobile devices generally necessitates modularization for latency and memory efficiency, an assumption increasingly reflected in newer model architectures.

Quantization. To support deployment on memory-constrained mobile hardware, we apply model quantization to reduce the memory footprint. We apply dynamic quantization using TorchAO to enable efficient deployment of our video-language models on mobile devices. Focusing on Linear and Embedding layers (responsible for over 91% of memory use), we achieve int8 compatibility and serialization for ARM CPUs while avoiding unreliable Conv3D quantization. Details of quantization, and operator compatibility are included in Supplementary Material Section 2.

Serialization and Deployment. Deploying quantized video-language models on mobile devices requires a fundamentally different approach than standard Python-based execution on high-end servers. Instead of storing weights as dictionaries, the models must be serialized by tracing their full forward passes, including nested module calls, to ensure compatibility with mobile runtimes and enable efficient inference on resource-constrained hardware. We use `torch.jit.trace` to record the computation graph of the model's forward execution, but several refactors are necessary: submodules must have statically typed forward passes, dynamic data flows (especially in the text decoder) are handled via fixed-length sample inputs, and incompatible operations like `torch.rsub` and

`torch.zeros` are replaced with JIT-friendly equivalents. After tracing, additional mobile-specific optimizations are applied, including operator fusion to merge frequently paired kernels and parameter hoisting to reduce metadata overhead and memory footprint. Together, these modifications enable fast, memory-efficient, and fully compatible deployment of quantized models on mobile hardware. Details of serialization, and deployment constraints are included in Supplementary Material Section 3.

5 Results and Discussion

We evaluate Atom on two representative video-language pipelines (video assembly and video retrieval) demonstrating the following key findings: (a) The modified mPLUG2+ architecture supports task reuse without performance degradation; and (b) Atom's reuse-centric design substantially reduces end-to-end latency by avoiding repeated weight loading and enabling parallelism.

We begin by detailing our experimental setup in Section 5.1, followed by results and analysis in Sections 5.2 and 5.3. The qualitative results are provided in Section 6. Additional results are provided in Supplementary Material Section 4.

5.1 Experimental Setup

Tasks and Models. We choose video retrieval and video assembly to represent two ends of the video-language task spectrum: retrieval-based reasoning and generative storytelling. Despite their differences, both benefit from caption generation as a shared intermediate step. This enables us to reuse the same language decoder across tasks, reducing latency and memory spent in the model loading phase. While server-side systems may afford task-specific models, our design favors reuse to meet mobile constraints. Supplementary Material Section 1 discusses how this approach extends to other tasks like summarization.

Both pipelines rely on three core modules: a visual encoder for video encoding, a language decoder for caption generation (conditioned on video features), and a sentence transformer for matching user prompts to generated captions. The language decoder is also reused for script generation, enabling efficiency in Atom's reuse-centric design.

As described in Section 4, we use CLIP-ViT/L-14 (0.4B parameters) as the visual encoder and Llama 3.2 (1B parameters) as the language decoder. The model is fine-tuned for video captioning using four NVIDIA A100 GPUs and the datasets described below. For semantic matching, we use all-MiniLM-L6-v2 [18], a 22.7M parameter sentence transformer.

Datasets. We use three datasets to fine-tune and evaluate the modified mPLUG2+ for video captioning, video retrieval, and qualitative video assembly (Section 6): (a) MSR-VTT [28], (b) MSVD [2], and (c) DiDeMo [6]. The model is trained on the combined training splits of all three datasets and evaluated on their respective test sets. The first two were also used in training the original mPLUG2.

MSR-VTT contains 10K YouTube videos with 200K captions, split into 6.5K training, 0.5K validation, and 3K test samples. MSVD has 1,970 YouTube clips with captions, divided into 1.2K/100/670 training/val/test samples. DiDeMo includes 10K Flickr videos, split into 8K/1K/1K, each paired with four text descriptions.

Counterparts for Comparison. Prior works on video assembly [30, 31] rely on human-written scripts to retrieve relevant clips. In contrast, Atom generates scripts from video clips based on user prompts. These prior systems are also closed-source, preventing reproducible comparisons. For video retrieval, existing methods focus on (a) architectural advances [26, 27] or (b) improving retrieval relevance [21], but those designed for mobile settings [20, 32] do not leverage large language models for reasoning.

Due to these limitations, we evaluate Atom using system-level baselines focused on performance and efficiency: (a) a 96-core x86_64 CPU server, both with and without Atom's reuse-centric design; (b) the same server equipped with an NVIDIA A40 GPU, evaluated in both quantized and full-precision modes, again with and without reuse; and (c) Atom on mobile devices without reuse, where separate models are loaded and unloaded for each subtask. The third baseline reflects typical multi-stage pipelines without coordination. Comparing against it isolates the benefits of Atom's modular reuse in latency and memory efficiency.

Metrics. For evaluating **video captioning subtask** performance, we employ four metrics: (a) BLEU@4, (b) ROUGE-L, (c) METEOR, and (d) CIDEr. BLEU@4 calculates the geometric mean of n-gram precision up to 4-grams between generated and reference captions, applying a brevity penalty. ROUGE-L assesses content overlap and word order by finding the longest common subsequence. METEOR computes a weighted harmonic mean of precision and recall, incorporating various linguistic features. CIDEr measures similarity using TF-IDF weighted cosine similarity of n-grams, emphasizing distinctive phrases. These metrics collectively provide a comprehensive assessment of caption quality, considering factors such as precision, recall, semantic similarity, and distinctiveness.

For evaluating **video retrieval task** performance, we use (a) Recall@1, (b) Recall@5, and (c) Recall@10. Recall@k measures the proportion of relevant videos successfully retrieved within the top k results of the video retrieval pipeline.

Meanwhile, the evaluation of **video assembly** has more human factors involved and there are no established metrics for the task. Hence we have performed qualitative analysis to measure efficacy of Atom, as shown in Section 6. For the task of **semantic matching** through sentence transformer, the metric of Pearson correlation measures the linear relationship between two sentence embeddings, quantifying semantic similarity based on how closely their values co-vary. **Latency** is measured in seconds. We have measured the latency of loading model(s), running each component or a subtask (video encoding, caption generation) of a pipeline, as well as the latency of running the end-to-end pipeline. **Memory consumption** is measured in Gigabytes (GBs).

Hardware. We evaluated Atom on 3 physical mobile devices with diverse computing capabilities: (a) Google Pixel 5a (6GB RAM), (b) Google Pixel 8a (8GB RAM), and (c) Samsung Galaxy S23 (8GB RAM). We have used Pytorch Mobile as the inference engine. Built for PyTorch Mobile, we also utilized TorchAO for optimized execution through operator fusion, quantization, and lightweight runtime adaptations. The experiments focused on executing the end-to-end task pipelines on mobile CPUs, given the limited compatibility of TorchAO (this limitation has been observed in other works as

Table 1: Captioning Performance comparison of the original mPLUG2 (with BERT text decoder) and mPLUG2+ (with LLaMa text decoder) for both quantized and full-precision versions for the task of video captioning. All 4 metrics B (BLEU@4), R (ROUGE-L), M (METEOR), and C (CIDEr) are "higher the better". mPLUG2+ of our method Atom has a lower performance drop even after quantization, compared to the original mPLUG with a weaker text decoder.

Model Variant	MSR-VTT				MSVD			
	B	R	M	C	B	R	M	C
	Without Quantization (32-bit weights)							
mPLUG2	56.9	34.0	68.2	79.4	73.4	46.8	84.7	163.6
mPLUG2+	62.4	36.8	70.8	83.1	75.3	49.1	86.8	166.4
	With Quantization (8-bit weights)							
mPLUG2	52.4	30.8	65.4	75.3	70.8	43.2	80.7	158.1
mPLUG2+	60.7	35.9	68.6	82.5	74.2	46.3	84.5	164.9

well [24, 25]) with alternative processing units such as GPUs and NPUs. However, Atom remains adaptable to other processing units, making it orthogonal to the choice of hardware accelerators.

Hyperparameters. We follow the hyperparameters used for training mPLUG2 [26]. The training lasts for 10 epochs, with the learning rate of 2e-5 for the task of video description. The batch size is set to 128. The max caption generation length is set to 23 tokens, which is needed for fixing the decoding loop iterations of JIT tracing. Beam size of 4 is used for the caption generation. The videos are preprocessed to have sampling rate of 6 frames per second, and each frame's resolution is decreased to 224×224. The hyperparameter selection is discussed in Supplementary Material Section 5.

5.2 Performance

In this section, we demonstrate the advantages of integrating a stronger language decoder in on-device video-language pipelines for retrieval and captioning tasks. Our enhanced mPLUG2+, powered by a LLaMa text decoder, delivers substantial gains in prediction performance compared to the original mPLUG2 with a BERT decoder, achieving 2.8–3.7 and 6.8–7.2 CIDEr point improvements without and with quantization, respectively. Moreover, quantization has a minimal impact on performance, causing only a 0.6–1.3 CIDEr reduction, while enabling efficient deployment on mobile and edge devices. These advancements allow us to build the first fully operable on-device video assembly pipeline, making real-time video understanding feasible. We first discuss the performance of video captioning and video retrieval tasks, as detailed in Tables 1 and 2 respectively. Due to the lack of quantitative metrics for the video assembly task, we have conducted a qualitative analysis on the assembled videos with 10 human participants, those results are included in Section 6. Compared to the original mPLUG2, our modified mPLUG+ improves BLEU@4 by 1.9–5.5 points, indicating better alignment between generated captions and reference captions. It also achieves ROUGE-L gains of 2.3–2.8 points, reflecting improved phrase- matching accuracy, and METEOR gains of 2.1–2.6 points, which capture better word alignment. Furthermore, CIDEr, which measures caption informativeness, improves by 2.8–3.7 points. Even after quantization, mPLUG2+ consistently outperforms the original model by 6.8–7.2 CIDEr score. Due to quantization, we see a minor degradation of 0.6–1.5 CIDEr score compared to full-precision mPLUG+, while maintaining a negligible qualitative difference, as detailed in Section 6.

Table 2: Retrieval Performance comparison of the original mPLUG2 (with BERT text decoder) and our modified mPLUG2+ (with Llama text decoder) for both quantized and full-precision versions for the task of video retrieval. All 3 metrics R@1 (Recall@1), R@5 (Recall@5), and R@10 (Recall@10) are "higher the better". mPLUG2+ of Atom has a lower performance drop even after quantization, compared to the original mPLUG with a weaker text decoder.

Model Variant	MSR-VTT			MSVD			DiDeMo		
	R@1	R@5	R@10	R@1	R@5	R@10	R@1	R@5	R@10
Without Quantization (32-bit floating point weights)									
mPLUG2	52.6	75.8	83.1	65.3	82.0	90.6	55.1	78.3	84.6
mPLUG2+	57.2	80.7	85.1	69.1	84.5	93.8	60.0	86.2	90.1
With Quantization (8-bit integer weights)									
mPLUG2	48.7	72.6	80.5	62.1	77.3	87.9	53.7	75.1	80.8
mPLUG2+	55.8	78.4	82.9	66.8	82.7	90.3	58.7	83.1	88.4

Similarly, our retrieval results in Table 2 show consistent performance improvements across MSR-VTT, MSVD and DiDeMo datasets. The full-precision mPLUG2+ model achieves Recall@1 gains of 3.8–4.9 points over mPLUG, demonstrating its stronger ability to retrieve the most relevant video. We observe similar trends for Recall@5 and Recall@10, reinforcing the robustness of our model across different ranking positions. Even with quantization, mPLUG2+ maintains a retrieval advantage over the original mPLUG2, with a Recall@1 drop of only 1.3–2.3 points, yet still outperforming the quantized mPLUG model by 4.7–7.1 points. Additionally, we show (in Supplementary Material Section 4.2) that the all-MiniLM-L6-v2 sentence transformer for caption-to-prompt matching in both retrieval and assembly pipelines has negligible impact on the end-to-end quality of our system, even after quantization. This ensures efficient and accurate matching in resource-constrained settings.

Key Takeaway: By integrating a stronger LLaMa decoder, Atom sustains its reuse- and parallelism-oriented pipeline design on-device while improving performance: gaining 2.8–3.7 CIDEr (full precision) and 6.8–7.2 CIDEr (quantized), along with up to 4.9 Recall@1 improvement in retrieval, demonstrating efficiency without compromising accuracy.

5.3 Latency

This section presents how the reuse-centric design for efficient pipeline execution of Atom leads to latency reduction of 27–33% for video assembly and video retrieval tasks.

Table 3 shows that Atom achieves substantial latency reductions across resource-constrained devices. On the Google Pixel 5a, our reuse-centric design lowers the end-to-end video retrieval latency by 33.06%, and video assembly latency by 30.78%. Similar trends are observed on the Google Pixel 8a and Samsung Galaxy S23, with latency reductions of approximately 30.43% and 29.14% for retrieval,

Table 3: Latency (in seconds, lower the better) comparison of Atom against a sequential (w/o reuse) implementation. The numbers in parentheses (↓) indicate the percentage savings of Atom over the sequential baseline. The table reports both component-wise latencies of each subtask (video encoding, caption generation, indexing, and script generation) and end-to-end latencies for retrieval and video assembly tasks, on high-end servers and resource-constrained mobile devices. GPU: NVIDIA A40. *Note: Component-wise latencies are measured independently per input. End-to-end latency, however, reflects execution over 5 inputs, showcasing the benefits of Atom's parallel execution and amortized model reuse. This design significantly reduces overall runtime by loading model modules only once and enabling concurrent processing across inputs, leading to end-to-end latencies lower than the naive sum of individual component times.

			Component-wise*				End-to-end	
	Platform Variant	Model Loading	Video Encoding (CLIP-ViT)	Caption Generation (Llama)	Indexing (MiniLM)	Script Generation (ST + Decoder)	Video Retrieval	Video Assembly
High-end Hardware	Without Quantization							
	x86_64 CPU (w/o reuse)	6.19	1.68	1.15	10.35	11.09	36.05	40.67
	x86_64 CPU + GPU (w/o reuse)	8.66	0.15	0.08	6.48	6.84	12.12	14.95
	x86_64 CPU (**Atom**)	3.44 (↓ 44.43%)	1.68	1.15	10.35	11.09	26.58 (↓ 26.27%)	30.28 (↓ 25.55%)
	x86_64 CPU + GPU (**Atom**)	4.38 (↓ 49.42%)	0.15	0.08	6.48	6.84	8.65 (↓ 28.63%)	10.46 (↓ 30.03%)
	With Quantization							
	x86_64 CPU (w/o reuse)	7.32	2.95	2.14	13.66	15.40	44.84	47.59
	x86_64 CPU + GPU (w/o reuse)	8.89	0.46	0.32	10.78	13.02	19.20	22.18
	x86_64 CPU (**Atom**)	3.64 (↓ 50.27%)	2.95	2.14	13.66	15.40	32.12 (↓ 28.37%)	34.69 (↓ 27.11%)
	x86_64 CPU + GPU (**Atom**)	4.56 (↓ 48.71%)	0.46	0.32	10.78	13.02	14.02 (↓ 26.98%)	16.38 (↓ 26.15%)
Resource-constrained Hardware	Google Pixel 5a (w/o reuse)	32.94	8.68	5.57	24.39	26.61	162.33	167.52
	Google Pixel 8a (w/o reuse)	29.63	6.34	4.22	21.20	22.41	147.04	153.59
	Samsung Galaxy S23 (w/o reuse)	28.43	6.83	3.54	21.03	22.68	146.47	152.35
	Google Pixel 5a (**Atom**)	17.23 (↓ 47.69%)	8.68	5.57	24.39	26.61	108.66 (↓ 33.06%)	115.95 (↓ 30.78%)
	Google Pixel 8a (**Atom**)	15.87 (↓ 46.44%)	6.34	4.22	21.20	22.41	102.30 (↓ 30.43%)	111.06 (↓ 27.69%)
	Samsung Galaxy S23 (**Atom**)	14.12 (↓ 50.33%)	6.83	3.54	21.03	22.68	103.78 (↓ 29.14%)	110.39 (↓ 27.54%)

and 27.69% and 27.54% for assembly, respectively. These reductions are driven by the parallel execution and amortized reuse of computational modules, significantly reducing redundant processing.

The key factor behind these improvements is Atom's ability to avoid repeated model loading and execution overhead. Traditional sequential, multi-model implementations reload models for different tasks, introducing substantial latency penalties, especially on mobile devices with constrained computational power. By reusing modules of a large video-language model, specifically our modified mPLUG2+, Atom eliminates redundant computations and allows concurrent execution of different pipeline stages, leveraging hardware parallelism more effectively. This leads to noticeable latency reduction for video assembly and video retrieval, where sequential execution would otherwise introduce significant bottlenecks, as shown in "(w/o reuse)" counterparts of the table.

Another critical insight is the comparison between high-end server configurations and mobile hardware. While GPUs in high-end setups, such as the NVIDIA A40, greatly reduce latency through efficient vectorized computations, mobile devices lack such accelerations. In these environments, optimizing execution flow is even more crucial. The results confirm that Atom's reuse-focused approach is effective in bridging this performance gap, delivering end-to-end latency reductions exceeding 30% without requiring additional specialized hardware.

Furthermore, the impact of quantization on latency underscores the importance of Atom's design. While quantization generally reduces a model's memory consumption, it introduces dynamic conversion overheads that can increase latency in implementations of video-based pipelines without reuse. However, Atom mitigates this by parallelizing the execution on video encoder and language decoder, preventing performance degradation. The improvements are particularly evident in resource-constrained mobile device setups, where our approach prevents excessive delays caused by sequential processing of quantized operations.

Overall, these findings validate the effectiveness of Atom's reuse-centric pipeline execution. By strategically leveraging modular reuse and concurrent execution, Atom significantly accelerates video retrieval and assembly tasks, making real-time video processing feasible even on mobile hardware. The results highlight the importance of optimizing computational reuse to achieve low-latency, high-efficiency AI-driven applications.

Key Takeaway: Atom's reuse-centric pipeline achieves 27–33% lower end-to-end latency for video-based tasks on mobile devices by amortizing model loading and enabling parallel execution.

5.4 Memory Footprint

Here we quantify the memory footprint of Atom and show that its reuse-centric design does not exceed the tight RAM budgets of mobile devices.

Table 4: Memory Consumption (in GBs, lower the better) of the video-language model used for the task pipeline of video assembly. The pipeline with Atom consumes 5.3GB of RAM, which is feasible for recent commodity smartphones with 6GB of RAM. Atom incurs only 0.76% more memory usage compared to the implementation without modularization and reuse, due to the elimination of 40.34% of temporary allocations created during model loading.

	Atom without modularization and reuse	Atom
Model Weights	4.623	4.623
Inputs	0.005	0.005
Activations (highest across all the models or modules)	0.297	0.479
Miscellaneous (temporary variables in model loads)	0.352	0.210
Peak Memory Consumption for One Run of the Pipeline	5.277	5.317

Table 4 reports the peak memory usage of the video-language pipeline with and without modular reuse. The two configurations differ by only 40MB: 5.28GB for the baseline versus 5.32GB for Atom, a negligible 0.76% increase that still fits comfortably within the 6 GB RAM of mainstream smartphones. The extra 40MB are due to the activations being slightly higher under Atom (0.479GB vs. the sequential baseline's 0.297GB) because the encoder and decoder can be active at the same time. This is offset by a sharp drop in the miscellaneous allocations, which are the temporary buffers created during model loading. The max memory consumption related to the temporary allocations shrink by 40.34% (Atom's 0.210GB vs. the sequential baseline's 0.352GB), due to keeping the heavy modules permanently resident. Model weights and input memory remain identical across the two variants.

Latency–memory trade-off. Although Atom sacrifices a 40MB extra RAM to enable parallelism, it delivers 27–33% end-to-end latency reductions (see Section 5.3), making the trade-off highly favorable. Tighter memory savings would require pushing quantization beyond 8-bit, which we found degrades accuracy unacceptably. Instead, Atom prioritizes reuse to maximize speed while remaining within the practical 6GB envelope of commodity phones.

Supplementary Material Section 4.3 provides a layer-wise breakdown of mPLUG2+'s memory usage, showing that the Linear and Embedding layers dominate the footprint. This analysis highlights the modules responsible for most memory consumption and reinforces that Atom's reuse-centric design effectively reduces temporary allocations while keeping peak memory manageable on mobile devices.

Furthermore, Supplementary Material Section 4.4 quantifies Atom's energy efficiency on mobile devices, showing that reuse-centric execution reduces CPU and total energy consumption by nearly 50% per pipeline run while maintaining peak memory usage.

5.5 Ablation Studies

Table 5 demonstrates the impact of varying input configurations and hyperparameters on the latency and performance of Atom's modified mPLUG2+ on a Google Pixel 5a. The default configuration

Table 5: Changing input sizes and hyperparameters to measure their impact on latency and performance. All experiments are measured on Google Pixel 5a for the task of video assembly with the test dataset of MSR-VTT. The default input shape is (1, 3, 6, 224, 224), with traced output length of 23 and generated beam size of 4.

Input Shape (Batch size, Channels, FPS, Width, Height)	Latency	CIDEr Score
(1, 3, 6, 224, 224): default case	5.57	82.50
(1, 3, 2, 224, 224)	4.78	50.89
(1, 3, 6, 112, 112)	3.29	75.64
(4, 3, 6, 224, 224)	22.35	82.50
(1, 3, 6, 224, 224): generator beam size = 1	3.07	40.98
(1, 3, 6, 224, 224): traced output length = 12	4.55	74.19

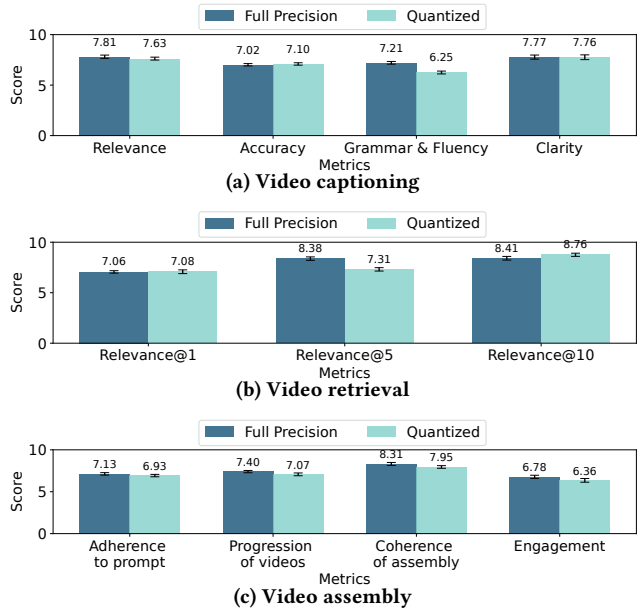
(1, 3, 6, 224, 224) achieves a latency of 5.75s and a CIDEr score of 82.5, providing a baseline for comparison.

The insights are as follows: (a) **Adjusting the traced output length** to 12 reduces latency by 5.40s but leads to a moderate performance drop (CIDEr score: 74.19), since now the text decoder has to generate shorter outputs (leading to lower latency) but the shorter outputs do not always capture all the details of a video clip available in the ground truth (leading to a lower CIDEr score). (b) **Lowering the input frame rate (FPS)** from 6 to 2 marginally decreases latency by 2.77s, though at the cost of a substantial performance degradation (CIDEr score: 50.89) due to not capturing enough details from each frame, to construct a comprehensive text description of the video clips. (c) **Reducing the frame resolution** to `(112, 112)` significantly decreases latency by 10.66s (because smaller inputs can be processed faster due to smaller matrix multiplications), with only a minor performance drop (CIDEr score: 75.64). (d) Conversely, **reducing the beam size** to 1 yields the lowest latency (of 26.07s) and drastically compromises performance (CIDEr score: 40.98) since the quality of the text description degrades as the generator has fewer branches to explore. (e) Finally, **increasing the batch size** to 4 results in a substantial latency increase to 180.35 seconds (almost 4× that of an input with batch size 1) while maintaining the CIDEr score of 82.5, indicating that batch size scaling is inefficient for resource-constrained devices and increases the latency linearly. These findings highlight the critical role of tuning hyperparameters and input configurations in optimizing latency-performance trade-offs, particularly for resource-limited mobile environments.

6 Qualitative Analysis

We conducted a qualitative evaluation with 10 human participants to assess video captioning, retrieval, and assembly using Atom's full-precision and quantized versions of modified mPLUG2+ (Figure 3).

We chose 10 participants because our goal was to evaluate relative, human-perceived differences between models rather than estimate absolute perceptual scores. A within-subject, paired design; where each participant compared outputs from both models on identical prompts and videos, substantially reduces inter-subject variability, making meaningful comparisons possible even with a moderate sample size. We adopted a 1–10 scoring scale to provide finer granularity, allowing participants to capture subtle differences



(a) Video captioning

(b) Video retrieval

(c) Video assembly

Figure 3: Qualitative analysis of the three tasks: (a) video captioning, (b) video retrieval, (c) video assembly. The results are based on 10 participants. The scale of different metric scores is [0, 10].

and minor degradations introduced by quantization that might be obscured by coarser scales. Across all tasks and prompts, the study comprised over 300 paired evaluations, sufficient to identify consistent perceptual differences. Unlike standardized protocols (e.g., ITU-T Rec. P.910), this evaluation serves as supporting evidence for the quantitative results, specifically assessing whether quantization introduces perceptible degradation. Finally, no IRB approval was required, as all participants were anonymous and no personally identifiable or sensitive information was collected.

For the task of video captioning, we evaluated the generated captions on (a) Relevance: whether the caption in appropriate for the theme or essence of the video, (b) Accuracy: whether the caption has correct details about the objects present in the video, (c) Grammar and fluency: the linguistic correctness and naturalness of the caption, and (d) Clarity: whether the caption is unambiguous. Each participant was shown 10 videos with the captions generated from both the full-precision and quantized models.

Figure 3a shows the mean scores for the evaluated metrics, with error bars representing standard deviation. Mean scores for full-precision and quantized models differ by only 0.1–1 across the four metrics, while standard deviations range from 0.7 to 1.2 for both models, indicating that these differences are small relative to inter-participant variability. Overall, the results suggest negligible differences in caption quality between full-precision and quantized models, consistent with the quantitative findings in Table 1.

For video retrieval, we evaluated the relevance of retrieved videos to a given prompt using Relevance@k (top-k). Participants assessed how well the top-1, top-5, and top-10 retrieved videos matched the written prompt. Each participant generated 10 prompts and evaluated the corresponding results from both the full-precision and quantized models. Figure 3b shows mean Relevance@k scores in the range of approximately 7–9 for both models, with error bars indicating standard deviation. The absolute differences in mean scores between the full-precision and quantized models across top-k settings range from 0.10 to 1.52, while the standard deviations range from 0.75–1.02 for the full-precision model and 0.92–1.13 for the quantized model. The results indicate that the mean top-k scores for full-precision and quantized models fall within a similar range, with largely comparable score distributions across models, despite modest differences in some intermediate top-k settings. This suggests that quantization has minimal impact on video retrieval performance, as both models exhibit comparable score distributions.

Lastly, for the task of video assembly, the qualitative evaluation is based on (a) Adherence to the prompt: Whether the content of the assembled video matched the given prompt, (b) Progression of videos: Whether the sequence of videos in the assembly seems to be in a natural order, (c) Coherence of the assembly: Whether the style and content of the assembled video clips make sense together, and (d) Engagement: How interesting or compelling the assembled video is. Each participant was asked to write 10 prompts and evaluate the above metrics on the videos assembled by both the full-precision and quantized models. Figure 3c presents the mean scores for the evaluated metrics, with error bars representing standard deviation. Mean scores range from 6.8–8.3 for the full-precision model and 6.3–7.9 for the quantized model, with standard deviations of 0.81–1.17 and 0.77–1.19, respectively. The overlapping standard deviations indicate that quantization has a negligible effect on video assembly, with both models producing similar score distributions.

Therefore, across three tasks (video captioning, video retrieval, and video assembly) the qualitative evaluation shows that quantization has minimal impact on performance, with mean scores remaining within a narrow range across models and standard deviations of comparable magnitude, indicating no consistent or systematic perceptual degradation due to quantization.

Key Takeaway: Atom maintains human-perceived quality after quantization across video captioning, retrieval, and assembly tasks. Mean scores between full-precision and quantized models differ by only 0.1–1.5 (on a 1–10 scale), with comparable variability across participants and no consistent perceptual degradation observed despite reduced model precision.

7 Conclusion

We present Atom, an efficient on-device system enabling end-to-end video-language pipelines on commodity mobile devices. Its reuse-centric design modularizes and repurposes a single VideoLM across all pipeline stages, significantly reducing model loading latency and enabling practical deployment on devices with as little as 6GB of RAM. Evaluations on commercial smartphones show that, compared to sequential multi-model pipelines, Atom achieves higher performance (up to 3.7 CIDEr for assembly and 4.9 Recall@1 for retrieval) with minimal quantization degradation, up to 33% faster runtime, and qualitative gains fully consistent with quantitative results. By eliminating external servers, Atom also enhances privacy, reduces infrastructure costs, and supports offline or bandwidth-constrained use cases.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-2312396, CNS-2338512, IIS-2435822, and CCF-2449995. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Part of the work is also supported by Adobe gift funding.

References

[1] Max Bain, Arsha Nagrani, Gül Varol, and Andrew Zisserman. 2021. Frozen in time: A joint video and image encoder for end-to-end retrieval. In *Proceedings of the IEEE/CVF international conference on computer vision*. 1728–1738. doi:10.1109/ICCV48922.2021.00175

[2] David L. Chen and William B. Dolan. 2011. Collecting Highly Parallel Data for Paraphrase Evaluation. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL-2011)* (Portland, Oregon) *(HLT '11)*. USA, 190–200. doi:10.5555/2002472.2002497

[3] Tsai-Shien Chen, Aliaksandr Siarohin, Willi Menapace, Ekaterina Deyneka, Hsiang-wei Chao, Byung Eun Jeon, Yuwei Fang, Hsin-Ying Lee, Jian Ren, Ming-Hsuan Yang, et al. 2024. Panda-70m: Captioning 70m videos with multiple cross-modality teachers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 13320–13331. doi:10.1109/CVPR52733.2024.01265

[4] Xiangxiang Chu, Limeng Qiao, Xinyang Lin, Shuang Xu, Yang Yang, Yiming Hu, Fei Wei, Xinyu Zhang, Bo Zhang, Xiaolin Wei, et al. 2023. Mobilevlm: A fast, reproducible and strong vision language assistant for mobile devices. *arXiv preprint arXiv:2312.16886* (2023). https://arxiv.org/abs/2312.16886

[5] Aaron Grattafiori et. al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 https://arxiv.org/abs/2407.21783

[6] Lisa Anne Hendricks, Oliver Wang, Eli Shechtman, Josef Sivic, Trevor Darrell, and Bryan Russell. 2018. Localizing Moments in Video with Temporal Language.. In *Empirical Methods in Natural Language Processing (EMNLP)*. Brussels, Belgium, 1380–1390. doi:10.18653/v1/D18-1168

[7] Soyeong Jeong, Kangsan Kim, Jinheon Baek, and Sung Ju Hwang. 2025. VideoRAG: Retrieval-Augmented Generation over Video Corpus. arXiv:2501.05874 [cs.CV] https://arxiv.org/abs/2501.05874

[8] Won Jo, Geuntaek Lim, Gwangjin Lee, Hyunwoo Kim, Byungsoo Ko, and Yukyung Choi. 2024. VVS: Video-to-Video Retrieval with Irrelevant Frame Suppression. *Proceedings of the AAAI Conference on Artificial Intelligence* 38, 3 (Mar. 2024), 2679–2687. doi:10.1609/aaai.v38i3.28046

[9] Gang Li and Yang Li. 2023. Spotlight: Mobile UI Understanding using Vision-Language Models with a Focus. In *The Eleventh International Conference on Learning Representations*. Kigali, Rwanda. https://openreview.net/forum?id=9yE2xEj0BH7

[10] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. In *Proceedings of Machine Learning and Systems*, Vol. 6. 87–100.

[11] Zechun Liu, Changsheng Zhao, Forrest N. Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, Liangzhen Lai, and Vikas Chandra. 2024. MobileLLM: Optimizing Sub-billion Parameter Language Models for On-Device Use Cases. In *ICML*. Vienna, Austria, Article 1316. doi:10.5555/3692070.3693386

[12] Xinwei Long, Zhiyuan Ma, Ermo Hua, Kaiyan Zhang, Biqing Qi, and Bowen Zhou. 2025. Retrieval-Augmented Visual Question Answering via Built-in Autoregressive Search Engines. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 24723–24731. doi:10.1609/aaai.v39i23.34653

[13] Huaishao Luo, Lei Ji, Ming Zhong, Yang Chen, Wen Lei, Nan Duan, and Tianrui Li. 2022. Clip4clip: An empirical study of clip for end to end video clip retrieval. *Neurocomput.* 508 (Oct. 2022), 293–304. doi:10.1016/j.neucom.2022.07.028

[14] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. 2024. The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits.

[15] Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. Llm-pruner: On the structural pruning of large language models. *Advances in neural information processing systems* 36 (2023), 21702–21720.

[16] Andrew Or, Apurva Jain, Daniel Vega-Myhre, Jesse Cai, Charles David Hernandez, Zhenrui Zhang, Driss Guessous, Vasiliy Kuznetsov, Christian Puhrsch, Mark Saroufim, and Supriya Rao. 2025. TorchAO: PyTorch-Native Training-to-Serving Model Optimization. https://openreview.net/forum?id=HpqH0JakHf

[17] Yeonhong Park, Jake Hyun, SangLyul Cho, Bonggeun Sim, and Jae W. Lee. 2025. Any-precision LLM: low-cost deployment of multiple, different-sized LLMs. In *Proceedings of the 41st International Conference on Machine Learning (ICML '25)*. JMLR.org, Vienna, Austria.

[18] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Hong Kong, China, 3982–3992. doi:10.18653/v1/D19-1410

[19] Sharath Turuvekere Sreenivas, Saurav Muralidharan, Raviraj Joshi, Marcin Chochowski, Mostofa Patwary, Mohammad Shoeybi, Bryan Catanzaro, Jan Kautz, and Pavlo Molchanov. 2024. LLM Pruning and Distillation in Practice: The Minitron Approach. *arXiv preprint arXiv:2408.11796* (2024). https://arxiv.org/abs/2408.11796

[20] Khalid Tahboub, Neeraj J. Gadgil, and Edward J. Delp. 2015. Content based video retrieval on mobile devices: How much content is enough?. In *2015 IEEE International Conference on Image Processing (ICIP)* (Quebec City, QC, Canada). IEEE Press. doi:10.1109/ICIP.2015.7351071

[21] Yannis Tevissen, Khalil Guetari, and Frédéric Petitpont. 2024. Towards Retrieval Augmented Generation over Large Video Libraries. In *2024 16th International Conference on Human System Interaction (HSI)* (Paris, France). 1–4. doi:10.1109/HSI61632.2024.10613524

[22] Yi Wang, Kunchang Li, Xinhao Li, Jiashuo Yu, Yinan He, Guo Chen, Baoqi Pei, Rongkun Zheng, Zun Wang, Yansong Shi, et al. 2024. Internvideo2: Scaling foundation models for multimodal video understanding. In *European Conference on Computer Vision* (Milan, Italy). Springer, Berlin, Heidelberg, 396–416. doi:10.1007/978-3-031-73013-9_23

[23] Shengqiong Wu, Hao Fei, Leigang Qu, Wei Ji, and Tat-Seng Chua. 2024. Next-gpt: Any-to-any multimodal llm. In *Forty-first International Conference on Machine Learning* (Vienna, Austria). JMLR.org, Article 2187, 32 pages. doi:10.5555/3692070.3694257

[24] Daliang Xu, Wangsong Yin, Xin Jin, Ying Zhang, Shiyun Wei, Mengwei Xu, and Xuanzhe Liu. 2023. LLMCad: Fast and Scalable On-device Large Language Model Inference.

[25] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. 2025. Fast On-device LLM Inference with NPUs. *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking* (2025). doi:10.1145/3669940.3707239

[26] Haiyang Xu, Qinghao Ye, Ming Yan, Yaya Shi, Jiabo Ye, Yuanhong Xu, Chenliang Li, Bin Bi, Qi Qian, Wei Wang, Guohai Xu, Ji Zhang, Songfang Huang, Fei Huang, and Jingren Zhou. 2023. mPLUG-2: a modularized multi-modal foundation model across text, image and video. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA). JMLR.org, Article 1614, 21 pages. doi:10.5555/3618408.3620022

[27] Jiaqi Xu, Cuiling Lan, Wenxuan Xie, Xuejin Chen, and Yan Lu. 2023. Retrieval-based Video Language Model for Efficient Long Video Question Answering. arXiv:2312.04931 https://arxiv.org/abs/2312.04931

[28] Jun Xu, Tao Mei, Ting Yao, and Yong Rui. 2016. MSR-VTT: A Large Video Description Dataset for Bridging Video and Language. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 5288–5296. doi:10.1109/CVPR.2016.571

[29] Dawei Yan, Pengcheng Li, Yang Li, Hao Chen, Qingguo Chen, Weihua Luo, Wei Dong, Qingsen Yan, Haokui Zhang, and Chunhua Shen. 2025. TG-LLaVA: Text Guided LLaVA via Learnable Latent Embeddings. *Proceedings of the AAAI Conference on Artificial Intelligence* 39, 9 (Apr. 2025), 9076–9084. https://ojs.aaai.org/index.php/AAAI/article/view/32982

[30] Guoxing Yang, Haoyu Lu, Zelong Sun, and Zhiwu Lu. 2023. Shot Retrieval and Assembly with Text Script for Video Montage Generation. In *Proceedings of the 2023 ACM International Conference on Multimedia Retrieval* (Thessaloniki, Greece) *(ICMR '23)*. Association for Computing Machinery, New York, NY, USA, 298–306. doi:10.1145/3591106.3592247

[31] Zhihui Yin, Ye Ma, Xipeng Cao, Bo Wang, Quan Chen, and Peng Jiang. 2024. Text-Video Multi-Grained Integration for Video Moment Montage. https://doi.org/10.48550/arXiv.2412.09276

[32] Haotian Zhang, Allan D. Jepson, Iqbal Mohomed, Konstantinos G. Derpanis, Ran Zhang, and Afsaneh Fazly. 2021. Personalized Multi-modal Video Retrieval on Mobile Devices. In *Proceedings of the 29th ACM International Conference on Multimedia (MM '21)*. Association for Computing Machinery, New York, NY, USA, 1185–1191. doi:10.1145/3474085.3481545

[33] Mingjin Zhang, Xiaoming Shen, Jiannong Cao, Zeyang Cui, and Shan Jiang. 2024. EdgeShard: Efficient LLM Inference via Collaborative Edge Computing. *IEEE Internet of Things Journal* 12, 10 (2024), 13119–13131. doi:10.1109/JIOT.2024.3524255