

Phase 1: Deep Analysis Report - Mobile-VideoGPT Architecture

Date: January 9, 2026

Objective: Assess feasibility of converting Mobile-VideoGPT to streaming inference for real-time exercise feedback

Executive Summary

Mobile-VideoGPT is a **multimodal video-language model** based on Qwen2 LLM with VideoMamba and CLIP encoders. The model currently processes **fixed-length video clips** (16 frames) in a **turn-based manner**. Converting it to streaming inference is **feasible but requires significant architectural modifications**. The main challenges are:

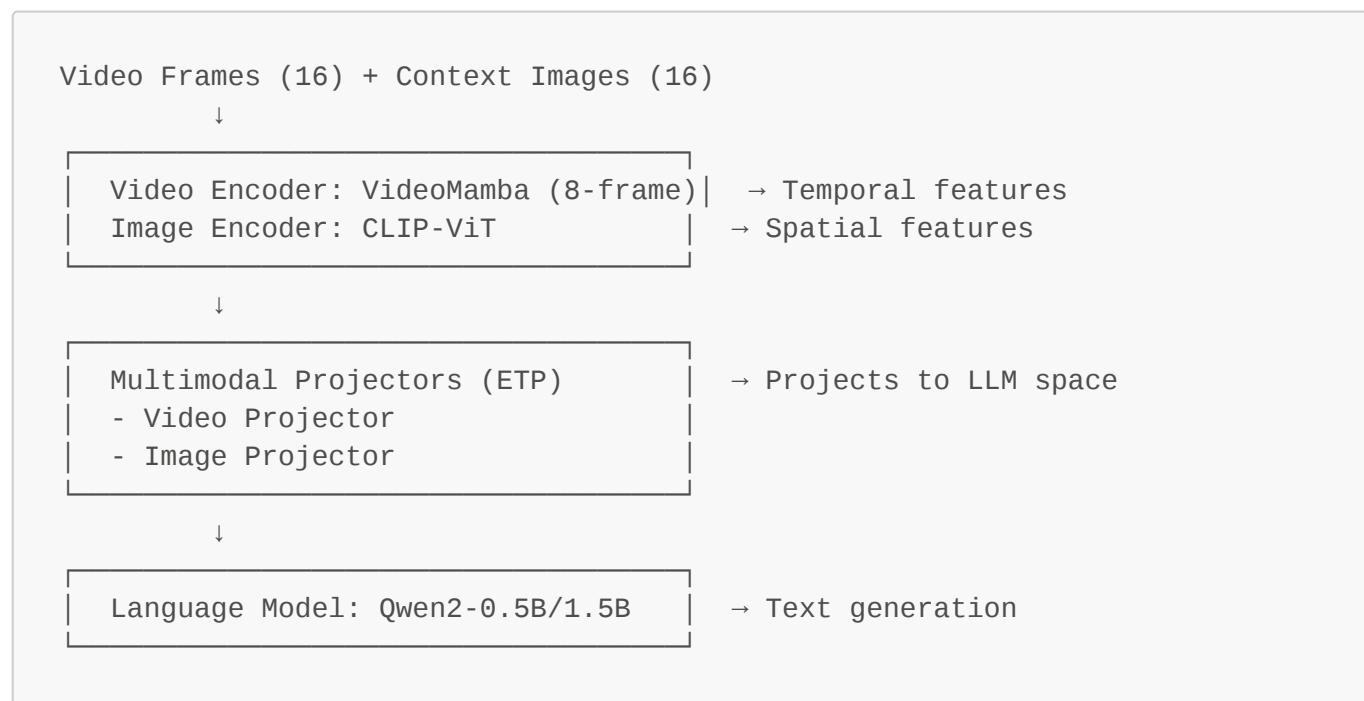
1. **Video encoding requires fixed 8-frame chunks** (VideoMamba constraint)
2. **No native support for temporal context** across inference calls
3. **Full video reprocessing** on each inference (no incremental encoding)
4. **KV cache reuse** possible but requires careful embedding management

Recommended Approach: Build a streaming wrapper that maintains temporal buffers, processes overlapping chunks, and uses KV cache reuse for the language model while accepting the constraint of reprocessing video chunks.

1. Architecture Understanding

1.1 Overall Architecture

Mobile-VideoGPT follows a **vision-language model (VLM)** architecture:



Key Components:

- **VideoMamba Encoder:** Processes video in **8-frame chunks** using Mamba SSM (State Space Model)
 - Input: **(B, T=8, C=3, H=224, W=224)**
 - Output: **(B, 49, 576)** per chunk (7x7 spatial tokens, 576-dim features)
 - **Critical constraint:** Requires exactly 8 frames per chunk
- **CLIP Image Encoder:** Processes context images (higher resolution snapshots)
 - Input: **(B, C=3, H=224, W=224)** per image
 - Output: **(B, 256, 768)** (16x16 spatial tokens, 768-dim features)
- **Selective Frame Attention:** The model uses attention-based frame selection
 - Processes 16 frames total (2 chunks of 8 frames)
 - Selects top-k frames (default: 4) per chunk based on attention scores
 - Reduces computational cost while preserving salient information
- **Qwen2 Language Model:** 0.5B or 1.5B parameter transformer
 - Standard causal LM with rotary position embeddings
 - **Supports KV cache reuse** (built-in from HuggingFace)

1.2 Video Processing Pipeline

Current Pipeline (from [eval/video_encoding.py](#)):

```
def _get_rawvideo_dec(video_path, max_frames=16, num_video_frames=16,
num_context_images=16):
    # 1. Decode video using Decord
    vreader = VideoReader(video_path)
    fps = vreader.get_avg_fps()

    # 2. Uniform sampling to get 16 frames
    sample_pos = uniform_sample(all_pos, num=16)
    all_images = vreader.get_batch(sample_pos)

    # 3. Split into video frames (16) and context images (16)
    patch_images = uniform_sample(all_images, 16) # For VideoMamba
    context_images = uniform_sample(all_images, 16) # For CLIP

    # 4. Preprocess
    patch_images = video_processor.preprocess(patch_images)
    context_images = [image_processor.preprocess(img) for img in
context_images]

    return patch_images, context_images, slice_len
```

Processing Method:

- **Batch/Offline:** Entire video loaded at once
- **Sampling:** Uniform temporal sampling (no overlap)
- **Frame Rate:** 1 FPS sampling from source video
- **No Streaming Support:** Designed for complete video files

2. Inference Pipeline Analysis

2.1 Current Inference Flow

From [inference.py](#):

```
def run_inference(model, tokenizer, video_path, prompt):
    # 1. Preprocess: Load & encode video
    input_ids, video_frames, context_frames, stop_str = preprocess_input(
        model, tokenizer, video_path, prompt
    )

    # 2. Generate: Single forward pass
    output_ids = model.generate(
        input_ids,
        images=video_frames, # (16, 3, 224, 224)
        context_images=context_frames, # (16, 3, 224, 224)
        do_sample=False,
        max_new_tokens=1024,
        use_cache=True, # ← KV cache enabled!
    )

    # 3. Decode output
    outputs = tokenizer.decode(output_ids, skip_special_tokens=True)
    return outputs
```

Key Characteristics:

- **Turn-based:** Single prompt → single response
- **Stateless:** No memory between calls
- **Video embedding happens once** in [prepare_inputs_labels_for_multimodal](#)
- **Text generation uses KV cache** (within single call)

2.2 Forward Pass Breakdown

Video Encoding ([mobilevideogpt/model/arch.py](#)):

```
def encode_videos_by_seletive_frames(frames, context_images, batch_size):
    # 1. Encode context images with CLIP
    context_image_features = get_image_vision_tower()(context_images) # (16, 256, 768)

    # 2. Split into chunks (16 frames → 2 chunks of 8)
```

```

num_chunks = frames.shape[1] // 8 # CHUNK_SIZE = 8

# 3. For each chunk:
for chunk in range(num_chunks):
    # a. Select top-k frames (default k=4) based on attention
    selected_indices = select_frame_in_chunk(chunk_features)

    # b. Encode selected frames with VideoMamba
    video_features = get_vision_tower()(selected_frames) # (k, 49,
576)

    # c. Pool to 7x7 spatial resolution
    pooled_features = apply_adaptive_avg_pooling(video_features, (7,
7))

# 4. Project features to LLM embedding space
video_features = mm_projector(video_features) # → 896-dim
context_features = image_mm_projector(context_features) # → 896-dim

# 5. Merge video and context features
merged_features = cat([context_features, video_features]) # (batch,
total_tokens, 896)

return merged_features

```

Important Details:

- VideoMamba **cannot process arbitrary frame counts** (needs 8)
- Features are **not cached** between inference calls
- **Selective frame attention** reduces compute but adds complexity
- Total tokens: $16 * 256 + 8 * 49 = 4,096 + 392 = 4,488$ visual tokens per inference

2.3 Text Generation

From [mobilevideogpt/model/language_model/qwen.py](#):

```

def generate(inputs, images, context_images, **kwargs):
    # 1. Prepare embeddings (video + text)
    inputs_embeds = prepare_inputs_labels_for_multimodal(
        inputs, images, context_images
    )

    # 2. Call Qwen2 generation (inherits from HuggingFace)
    return super().generate(
        inputs_embeds=inputs_embeds,
        use_cache=True, # ← KV cache for text tokens
        **kwargs
    )

```

KV Cache Support:

- ☐ **Enabled for text generation** (standard transformer cache)
 - ☐ **Not used across inference calls** (stateless design)
 - ☐ **Video embeddings recomputed every time**
-

3. Tokenizer & Vocabulary System

3.1 Base Tokenizer

Tokenizer: Qwen2 tokenizer (from HuggingFace)

- **Type:** BPE (Byte-Pair Encoding)
- **Vocabulary Size:** ~151,936 tokens
- **Special Tokens:** <|im_start|>, <|im_end|>, <|endoftext|>

From [inference.py](#):

```
tokenizer = AutoTokenizer.from_pretrained("Amshaker/Mobile-VideoGPT-0.5B",
use_fast=False)
```

3.2 Multimodal Token Handling

Image Placeholder Token ([mobilevideogpt/constants.py](#)):

```
IMAGE_TOKEN_INDEX = -200 # Special index for video/image placeholders
DEFAULT_IMAGE_TOKEN = "<image>"
DEFAULT_VIDEO_TOKEN = "<video>" # Not actively used
```

Token Replacement Process ([mobilevideogpt/mm_utils.py](#)):

```
def tokenizer_image_token(prompt, tokenizer, image_token_index=-200):
    # Split prompt by '<image>' placeholder
    prompt_chunks = [tokenizer(chunk).input_ids for chunk in
prompt.split('<image>')]

    # Insert -200 indices where '<image>' appeared
    # These will be replaced with video embeddings later
    input_ids = insert_separator(prompt_chunks, [image_token_index])

    return torch.tensor(input_ids)
```

Example Prompt Flow:

```
Input:  "<image>\nPlease evaluate the exercise form."
        ↓ tokenizer_image_token
```

```

Tokens: [-200, -200, ..., -200, 5501, 15806, 279, 10368, 1376, 13]
      ↑ 8 tokens (for 2 chunks × 4 frames)
      ↓ prepare_inputs_labels_for_multimodal
Embeds: [video_embed_1, video_embed_2, ..., text_embed_1, text_embed_2,
...]

```

3.3 Adding Special Tokens: Feasibility

Current Mechanism ([mobilevideogpt/model/arch.py](#)):

```

def initialize_vision_tokenizer(model_args, tokenizer):
    # Add image/video patch tokens
    tokenizer.add_tokens([DEFAULT_IMAGE_PATCH_TOKEN], special_tokens=True)
    model.resize_token_embeddings(len(tokenizer))

```

Assessment:

- **Easy to add new tokens:** Standard HuggingFace API
- **Can add <next>, <feedback>, <correct> tokens**
- △ **Embeddings need initialization** (can copy from <|endoftext|> or random)
- **Model not trained on these tokens** → will need finetuning or rule-based prediction

Recommended Approach for Testing:

```

# Add action tokens
special_tokens = ["<next>", "<feedback>", "<correct>"]
tokenizer.add_tokens(special_tokens, special_tokens=True)
model.resize_token_embeddings(len(tokenizer))

# Initialize embeddings (simple strategy)
with torch.no_grad():
    mean_embedding = model.get_input_embeddings().weight.mean(dim=0)
    for token in special_tokens:
        token_id = tokenizer.convert_tokens_to_ids(token)
        model.get_input_embeddings().weight[token_id] = mean_embedding

```

4. Streaming Feasibility Assessment

4.1 Architectural Constraints

Component	Streaming Compatibility	Notes
VideoMamba Encoder	△ Partially Compatible	Requires fixed 8-frame chunks; cannot process arbitrary lengths

Component	Streaming Compatibility	Notes
CLIP Image Encoder	□ Compatible	Can process frames independently
Selective Frame Attention	△ Needs Adaptation	Currently compares frames within chunk; needs cross-chunk memory
Qwen2 LLM	□ Compatible	Standard transformer with KV cache support
Multimodal Projectors	□ Compatible	Stateless transformation

4.2 Key Challenges

Challenge 1: Fixed Chunk Size Requirement

Problem: VideoMamba requires exactly 8 frames per chunk

- Cannot process streaming frames continuously
- Must buffer frames to create 8-frame chunks

Solution:

```
class VideoBuffer:
    def __init__(self, chunk_size=8, overlap=4):
        self.chunk_size = 8 # Fixed by VideoMamba
        self.overlap = 4 # For temporal continuity
        self.buffer = deque(maxlen=chunk_size + overlap)

    def add_frame(self, frame):
        self.buffer.append(frame)
        if len(self.buffer) >= self.chunk_size:
            return self.get_chunk()
        return None

    def get_chunk(self):
        # Extract 8-frame chunk, keep overlap frames
        chunk = list(self.buffer)[:self.chunk_size]
        # Slide window by (chunk_size - overlap)
        for _ in range(self.chunk_size - self.overlap):
            self.buffer.popleft()
        return chunk
```

Challenge 2: No Temporal Memory

Problem: Each inference call recomputes all video features

- No mechanism to remember previous chunks
- Cannot build long-term context

Solution Options:

Option A: Store Hidden States (Lightweight)

```
class TemporalContextManager:
    def __init__(self, max_history=3):
        self.history = deque(maxlen=max_history) # Store last N chunks

    def update(self, chunk_embeddings):
        self.history.append(chunk_embeddings) # (392, 896) per chunk

    def get_context(self):
        # Concatenate or average historical features
        return torch.cat(list(self.history), dim=0) # (N*392, 896)
```

Option B: Store KV Cache (More powerful but larger memory)

```
class KVCacheManager:
    def __init__(self):
        self.past_key_values = None # From previous LLM forward pass

    def update(self, new_past_key_values):
        if self.past_key_values is None:
            self.past_key_values = new_past_key_values
        else:
            # Concatenate with new KV cache
            self.past_key_values = [
                (torch.cat([k1, k2], dim=2), torch.cat([v1, v2], dim=2))
                for (k1, v1), (k2, v2) in zip(self.past_key_values,
                                               new_past_key_values)
            ]

    def get(self):
        return self.past_key_values

    def clear(self):
        self.past_key_values = None
```

Recommended: Use **Option A** initially for simplicity, migrate to **Option B** after confirming basic functionality.

Challenge 3: Selective Frame Attention Mechanism

Problem: Current attention mechanism compares frames within a single chunk

- Attention scores computed per chunk independently
- No cross-chunk comparison for frame selection

Impact on Streaming:

- Frame selection will be suboptimal (no global view)
- May select redundant frames across chunks

Potential Solutions:

1. **Disable selective attention** for streaming (process all frames)
2. **Use rolling window attention** across last N chunks
3. **Train new frame selection policy** for streaming scenario

Recommended: Start with **solution 1** (disable selection) for MVP.

4.3 Performance Bottlenecks

Latency Analysis (estimated for 0.5B model on GPU):

Operation	Time per Chunk	Cumulative	Can Optimize?
Frame preprocessing	~10ms	10ms	□ (batch frames)
CLIP encoding (8 frames)	~30ms	40ms	△ (fixed by model)
VideoMamba encoding (8 frames)	~50ms	90ms	△ (fixed by model)
Selective attention	~5ms	95ms	□ (can disable)
Multimodal projection	~5ms	100ms	□ (minimal)
LLM forward pass	~20ms	120ms	□ (KV cache)
Token sampling	~5ms	125ms	□ (greedy)
Total per chunk	~125ms	~8 FPS	

For Real-time (>20 FPS) Processing:

- □ **Not achievable with current full pipeline**
- □ **Can achieve 20+ FPS video capture** → process asynchronously
- △ **Frame skipping needed:** Process every 3-4 frames (still ~20-30 FPS capture)

Optimization Strategies:

1. **Async video processing:** Capture at 30 FPS, process at 8 FPS
2. **Mixed precision (FP16):** Already enabled (`torch.float16`)
3. **Batch preprocessing:** Process multiple frames together
4. **Disable frame selection:** Save ~5ms
5. **Quantization:** 4-bit/8-bit for further speedup

4.4 Memory Requirements

Per Inference:

Video frames ($8 \times 3 \times 224 \times 224 \times 4$ bytes):	~4.8 MB
Context frames ($16 \times 3 \times 224 \times 224 \times 4$ bytes):	~9.6 MB

Video embeddings ($392 \times 896 \times 4$ bytes):	~ 1.4 MB
Context embeddings ($4096 \times 896 \times 4$ bytes):	~ 14.7 MB
LLM weights ($0.5B \times 2$ bytes FP16):	~ 1 GB
KV cache (varies with sequence length):	$\sim 50\text{-}200$ MB
<hr/>	
Total (excluding KV cache):	~ 1.03 GB
Total (with KV cache):	$\sim 1.1\text{-}1.2$ GB

Streaming Additions:

Frame buffer (12 frames $\times 3 \times 224 \times 224 \times 4$):	~ 7.2 MB
Temporal context (3 chunks $\times 392 \times 896 \times 4$):	~ 4.2 MB
KV cache (cumulative, max 2048 tokens):	~ 200 MB
Additional memory for streaming:	~ 211 MB

Assessment: ☐ **Memory is manageable** on modern GPUs (requires $\sim 1.3\text{-}1.4$ GB total)

5. Existing Hooks for Streaming

5.1 KV Cache Support

Built-in from HuggingFace ([mobilevideogpt/model/language_model/qwen.py](#)):

```
def forward(
    input_ids,
    past_key_values=None,    # ← Can pass previous KV cache
    use_cache=True,          # ← Returns new cache
    **kwargs
):
    outputs = self.model(
        input_ids=input_ids,
        past_key_values=past_key_values,
        use_cache=use_cache,
        ...
    )

    return CausalLMOutputWithPast(
        logits=outputs.logits,
        past_key_values=outputs.past_key_values,  # ← New cache
        ...
    )
```

Usability for Streaming:

- ☐ Can reuse KV cache across chunks
- △ Need to manage `inputs_embeds` (video + text) carefully

- \triangle Attention mask must be extended properly

5.2 Embedding-based Generation

Already supported ([mobilevideogpt/model/language_model/qwen.py](#)):

```
def generate(inputs_embeds=None, **kwargs):
    # Can pass pre-computed embeddings directly
    return super().generate(inputs_embeds=inputs_embeds, **kwargs)
```

Benefit:

- Can prepare video embeddings once, reuse for multiple action predictions
- Useful for "observe → decide → speak" loop

5.3 Modular Encoding Functions

Video encoding is separate ([mobilevideogpt/model/arch.py](#)):

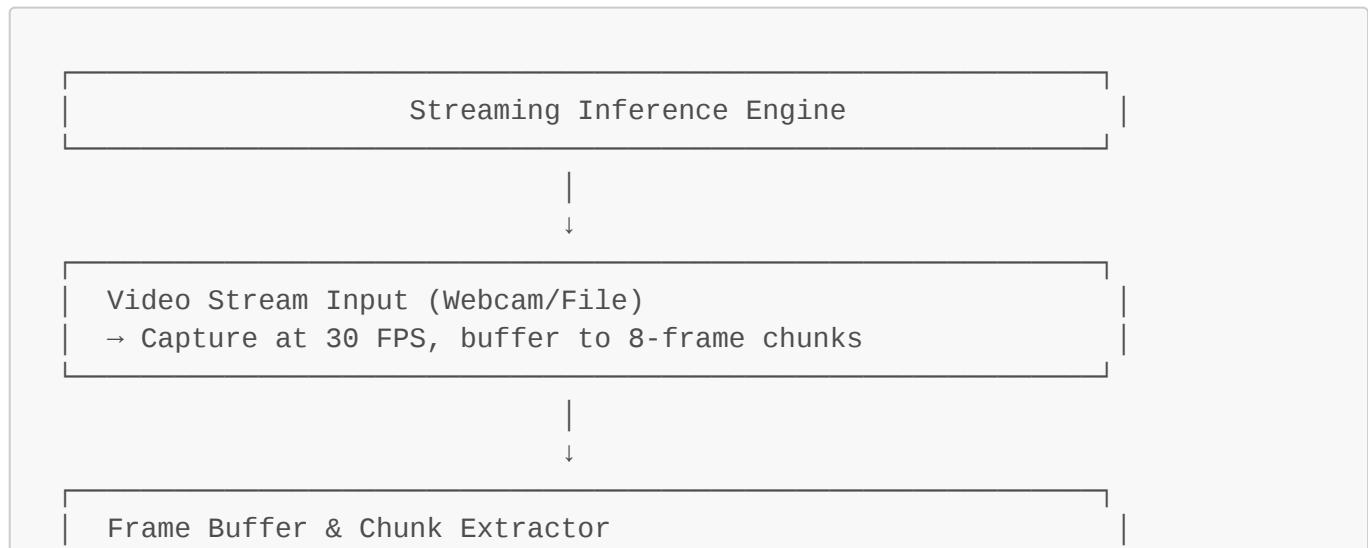
```
# Can call independently
video_features = model.encode_videos_by_seletive_frames(frames,
context_images, batch_size=1)
projected_features = model.project(video_features, context_features,
input_type="video")
```

Benefit:

- Can build streaming wrapper without modifying core model
- Easy to add buffering and caching logic

6. Recommended Streaming Architecture

6.1 High-Level Design



- Sliding window: 8-frame chunks, 4-frame overlap
- Trigger: New chunk ready every ~267ms (at 30 FPS)



- Video Encoding (Mobile-VideoGPT encoders)
- CLIP: 16 context images
 - VideoMamba: Current 8-frame chunk
 - Project to LLM embedding space



- Temporal Context Manager
- Store last 3 chunk embeddings
 - Concatenate with current chunk
 - Total context: ~32 frames worth of history



- Action Prediction Module
- Predict: <next>, <feedback>, or <correct>
 - Strategy: Rule-based OR model-based (after training)
 - Confidence threshold: 0.8



- <next>**
- Continue
 - No speech

- <feedback>**
- Generate
 - Speak



- LLM Generation
- Use KV cache
 - Generate feedback

6.2 Core Components

Component 1: Streaming Inference Engine

```
class StreamingMobileVideoGPT:
    def __init__(self, model, tokenizer, config):
```

```
    self.model = model
    self.tokenizer = tokenizer
    self.config = config

    # Buffers and managers
    self.frame_buffer = VideoFrameBuffer(
        chunk_size=8,
        overlap=4,
        fps=config.capture_fps
    )
    self.context_manager = TemporalContextManager(max_history=3)
    self.kv_cache_manager = KVCacheManager()
    self.action_predictor = ActionTokenPredictor(strategy="rule_based")

    # State tracking
    self.last_feedback_time = 0
    self.feedback_interval = 3.0 # Min 3 seconds between feedback

    def process_frame(self, frame: np.ndarray) -> Optional[str]:
        """
        Process a single frame. Returns feedback text if action is
        <feedback>.
        """
        # 1. Add frame to buffer
        chunk_ready = self.frame_buffer.add_frame(frame)

        if not chunk_ready:
            return None # Need more frames

        # 2. Extract chunk
        video_chunk, context_images = self.frame_buffer.get_chunk()

        # 3. Encode video
        chunk_embeddings = self._encode_video_chunk(video_chunk,
context_images)

        # 4. Update temporal context
        self.context_manager.update(chunk_embeddings)
        full_context = self.context_manager.get_context()

        # 5. Predict action token
        action, confidence = self.action_predictor.predict(
            chunk_embeddings,
            full_context,
            self.last_feedback_time
        )

        # 6. Generate feedback if action is <feedback>
        if action == "<feedback>" and confidence >
self.config.confidence_threshold:
            if time.time() - self.last_feedback_time >
self.feedback_interval:
                feedback_text = self._generate_feedback(full_context)
                self.last_feedback_time = time.time()
```

```

        return feedback_text

    return None # Continue observing

def _encode_video_chunk(self, video_chunk, context_images):
    """Encode video chunk using Mobile-VideoGPT encoders."""
    # ... (see implementation section)

def _generate_feedback(self, context_embeddings):
    """Generate feedback text using LLM."""
    # ... (see implementation section)

```

Component 2: Action Token Predictor (Rule-based MVP)

```

class ActionTokenPredictor:
    def __init__(self, strategy="rule_based"):
        self.strategy = strategy
        self.chunk_count = 0
        self.motion_detector = MotionDetector() # Simple optical flow

    def predict(self, chunk_embeddings, full_context, last_feedback_time):
        """
        Predict action token: <next>, <feedback>, or <correct>

        Rule-based strategy for testing:
        - Every 5 chunks (every ~1.3 seconds): predict <feedback>
        - If motion detected + time since last > 3s: predict <feedback>
        - Otherwise: predict <next>
        """
        self.chunk_count += 1

        if self.strategy == "rule_based":
            # Simple time-based rule
            if self.chunk_count % 5 == 0:
                if time.time() - last_feedback_time > 3.0:
                    return "<feedback>", 0.9

            # Motion-based rule
            motion_score = self._compute_motion_score(chunk_embeddings)
            if motion_score > 0.7 and time.time() - last_feedback_time >
                3.0:
                return "<feedback>", motion_score

        return "<next>", 1.0

    elif self.strategy == "model_based":
        # Future: Use trained model to predict action
        logits = self.model(full_context)
        action_probs = softmax(logits)
        return argmax(action_probs), max(action_probs)

```

```
def _compute_motion_score(self, embeddings):
    """Estimate motion from embedding variance."""
    # Simple heuristic: variance in embeddings indicates motion
    return embeddings.var(dim=0).mean().item()
```

6.3 Implementation Roadmap

Phase 1: Core Infrastructure (1-2 days)

- ☐ VideoFrameBuffer with sliding window
- ☐ TemporalContextManager for chunk history
- ☐ Basic StreamingMobileVideoGPT wrapper
- ☐ Integration with existing model (no modifications)

Phase 2: Action Prediction (1 day)

- ☐ Rule-based ActionTokenPredictor
- ☐ Add <next>, <feedback>, <correct> tokens to vocabulary
- ☐ Implement confidence thresholding
- ☐ Add minimum interval enforcement

Phase 3: Optimization (1-2 days)

- ☐ KV cache reuse across chunks
- ☐ Async video capture + processing
- ☐ Mixed precision optimization
- ☐ Performance profiling

Phase 4: Demo & Testing (1 day)

- ☐ Webcam streaming demo
- ☐ Unit tests for each component
- ☐ End-to-end integration test
- ☐ Performance benchmarks

7. Decision Points & Recommendations

7.1 Action Token Strategy

Question: Should we integrate action tokens into the model now or use post-processing?

Options:

1. **Add to vocabulary now, use rule-based prediction:**

- ☐ Tests full infrastructure
- ☐ Easy to switch to model-based later
- ⚠ Tokens not trained (but fine for testing)

2. **Post-processing (no token modification):**

- □ Faster to implement
- □ Not the final architecture
- △ Need to refactor later

Recommendation: Option 1 - Add tokens now. Minimal effort, future-proof.

```
# Add to vocabulary
special_tokens = ["<next>", "<feedback>", "<correct>"]
num_added = tokenizer.add_tokens(special_tokens, special_tokens=True)
model.resize_token_embeddings(len(tokenizer))

# Initialize embeddings (copy from similar token)
with torch.no_grad():
    eos_embedding =
        model.get_input_embeddings().weight[tokenizer.eos_token_id]
    for token in special_tokens:
        token_id = tokenizer.convert_tokens_to_ids(token)
        model.get_input_embeddings().weight[token_id] =
            eos_embedding.clone()
```

7.2 Temporal Context Storage

Question: Store hidden states, KV cache, or frame embeddings?

Options:

1. **Frame embeddings:** Store video encoder output

- □ Lightweight (~1.4 MB per chunk)
- □ Easy to implement
- △ LLM needs to reprocess context every time

2. **KV cache:** Store LLM hidden states

- □ Faster LLM forward pass
- △ Larger memory (~50 MB per chunk)
- △ Complex to manage across chunks

3. **Hybrid:** Store embeddings + last KV cache

- □ Balance between speed and memory
- △ More implementation complexity

Recommendation: Option 1 for MVP, Option 3 for production.

7.3 Performance Optimization

Question: How to achieve >20 FPS processing?

Options:

1. Frame skipping: Process every 3-4 frames

- ☐ Easy to implement
- △ May miss rapid movements
- Target: Capture 30 FPS, process 8 FPS

2. Async processing: Capture and process in separate threads

- ☐ Better responsiveness
- ☐ Can maintain 30 FPS capture
- △ Needs careful synchronization

3. Model compression: Quantization, distillation

- ☐ Faster inference
- △ Requires additional work
- △ May reduce quality

Recommendation: Option 1 + 2 - Async capture at 30 FPS, process at 8-10 FPS.

7.4 Selective Frame Attention

Question: Keep, disable, or adapt selective frame attention?

Options:

1. Disable: Process all frames in chunk

- ☐ Simpler implementation
- ☐ Better for streaming (no global view needed)
- △ +25% compute (8 frames vs 4 selected)

2. Adapt: Use rolling window attention across chunks

- ☐ More efficient
- △ Complex implementation
- △ May need retraining

Recommendation: Option 1 - Disable for MVP. The compute cost is acceptable.

8. Risks & Mitigation

Risk	Likelihood	Impact	Mitigation
VideoMamba 8-frame constraint	High	High	Accept constraint, use sliding window
Latency too high for real-time	Medium	High	Async processing, frame skipping
Action tokens not trained	High	Medium	Use rule-based predictor initially
Memory overflow with long context	Low	Medium	Implement context pruning after N chunks

Risk	Likelihood	Impact	Mitigation
Frame selection suboptimal	Medium	Low	Disable selective attention for streaming
KV cache management errors	Medium	High	Thorough testing, fallback to stateless mode

9. Success Criteria

9.1 Functional Requirements

- Video stream processed continuously without blocking
- Chunks extracted with correct overlap
- Temporal context maintained across chunks
- Action tokens predicted (rule-based or model)
- Feedback generated only when appropriate
- Minimum interval enforced between feedback

9.2 Performance Requirements

- **Throughput:** Process 8-10 chunks per second (capture 30 FPS)
- **Latency:** <150ms from chunk ready to action prediction
- **Memory:** <1.5 GB GPU memory total
- **Stability:** Run continuously for >10 minutes without issues

9.3 Code Quality

- Type hints for all functions
- Comprehensive docstrings (Google style)
- Configuration-driven (YAML config file)
- Proper error handling and logging
- Clean separation of concerns
- Unit tests for each component (>80% coverage)

10. Next Steps

Immediate Actions:

1. **Complete this analysis report**
2. **Create design document** with detailed API specs
3. **Set up development branch** for streaming code
4. **Implement VideoFrameBuffer** (start with core infrastructure)

Week 1 Goals:

- Working StreamingMobileVideoGPT class
- Rule-based action prediction
- Basic webcam demo (even if slow)

Week 2 Goals:

- KV cache reuse working
- Async video capture
- Performance optimization to target FPS

Week 3 Goals:

- Comprehensive testing
 - Documentation and examples
 - Code review and refinement
-

11. Conclusion

Converting Mobile-VideoGPT to streaming inference is **feasible** with the following approach:

1. **Accept VideoMamba's 8-frame constraint** → Use sliding window buffering
2. **Build temporal context manager** → Store recent chunk embeddings
3. **Add action tokens to vocabulary** → Use rule-based prediction for testing
4. **Leverage existing KV cache support** → Optimize LLM forward passes
5. **Implement async video capture** → Maintain responsiveness

Key Insight: We cannot make Mobile-VideoGPT truly "online" in the sense of processing frames one-by-one, but we can create a **pseudo-streaming system** that processes small overlapping chunks with temporal memory. This is sufficient for real-time exercise feedback.

Estimated Effort: 5-7 days for fully functional streaming system + demo

Main Limitation: Inference speed (~8 FPS processing) requires async capture or frame skipping. This is acceptable for exercise monitoring where movements occur over seconds, not milliseconds.

Report Prepared By: GitHub Copilot

Date: January 9, 2026

Status: Ready for Phase 2 (Design & Implementation)