Unlocking Long Video Insights: Efficient Batch Q&A with Gemini, Context Caching & RAG

Shifat Bin Rahman[1]
[2]Google DeepMind

This proposal presents a cost-effective system for analyzing and querying long video transcripts using Google's Gemini 1.5 Pro. The approach combines transcript segmentation, efficient non-Gemini summarization, retrieval-augmented generation (RAG) for relevant context retrieval, and Gemini's context caching. By caching pre-processed summaries server-side, the system drastically reduces token usage, cost, and latency for batch queries. The document details the architecture, implementation steps, evaluation strategy, tech stack, and timeline for achieving high-quality batch Q&A on long video content.
Project Demo: ***Repository Link***

1. Project Overview

1.1. Abstract

Analyzing and querying long video transcripts, especially for multiple questions, is costly and inefficient due to large context requirements and repeated processing. This proposal presents a cost-effective system and code structure that strategically uses Google's Gemini 1.5 Pro only for final answer synthesis. Our approach combines transcript segmentation, efficient non-Gemini summarization, retrieval-augmented generation (RAG) for relevant context retrieval, and Gemini's context caching. By caching pre-processed summaries server-side, we drastically reduce token usage, cost, and latency for batch queries. Inspired by recent LLM serving optimizations (like KV cache compression and prefix sharing), this methodology maximizes throughput and cost-efficiency without sacrificing accuracy. This document details the architecture, implementation steps (with Python examples), evaluation strategy, tech stack, and timeline for achieving robust, high-quality batch Q&A on long video content.

1.2. Introduction

Long video transcripts (often exceeding tens of thousands of words) pose significant challenges for large language models (LLMs). Standard prompting would require splitting transcripts into chunks, risking loss of global context, or providing the entire transcript repeatedly, leading to inefficiencies in cost and time (Metel et al., 2024). Asking multiple questions about the same content exacerbates this issue.

The landscape of LLM capabilities is rapidly evolving, particularly concerning the size of the context window - the amount of text a model can consider at once. Recent advancements have seen context limits expand dramatically. For instance, Google's Gemini 1.5 Pro offers a substantial context window, reportedly up to 2 million tokens as of mid-2024, making it theoretically capable of processing vast amounts of text, such as entire book series (like Harry Potter and Lord of the Rings combined, Figure 1), in a single input. This significant increase, visually contrasted with the capacities of earlier models like Anthropic's Claude 2.1 (July 2023) and OpenAI's GPT-4 Turbo (March 2023) (Artfish.ai, n.d.), underscores the potential for handling long-form content more effectively, though practical application often requires strategies beyond simply feeding the entire text.

Despite these large context windows, directly processing massive transcripts for multiple queries can still be computationally expensive and inefficient. This project proposes a system designed as a practical code sample to address these challenges using batch prediction, strategic long context handling (segmentation, summarization, RAG), and context caching, leveraging Google's Gemini 1.5 Pro API required only at the final answer generation step. Recent research, such as

Figure 1 | *Comparison of LLM context window sizes, illustrating the capacity of Gemini 1.5 Pro (2M tokens) relative to predecessors (Data/Concept: Artfish.ai)*

BatchLLM (Zheng et al., 2024), highlights the benefits of global prefix sharing in batched LLM inference to improve throughput, a concept we aim to explore within the context of Gemini's caching capabilities. Furthermore, for scenarios with repeated or semantically similar queries, integrating a semantic cache like GPTCache (Bang, 2023) could further optimize response times and reduce costs.

Key requirements addressed by this proposal include:

- Batch Prediction for Multiple Questions: Efficiently process a list of user questions related to the same transcript in a single workflow.
- Long Context Handling: Employ segmentation and hierarchical summarization to manage large transcripts, making essential information accessible without overwhelming the final generation model.
- Context Caching: Utilize Gemini's context caching feature (Jayawardena & Yankulin, 2024; Google, 2024;) to store pre-processed context (like a transcript summary) serverside, significantly reducing cost and latency for subsequent queries by avoiding redundant token processing. Context caching is supported for stable versions of Gemini 1.5 Pro.
- Interconnected Questions: Provide a framework to handle conversational follow-ups by incorporating previous Q&A history into the context.
- Clear Answer Formatting: Structure outputs clearly for user readability, potentially including references like timestamps.
- Robust Error Handling: Implement mechanisms like retries with backoff to handle transient API issues.
- Multi-Language Support: Design accommodates transcripts in various languages.

Recent research highlights the imperative to optimize LLM inference. Metel et al. (2024) demonstrate substantial throughput gains via KV cache compression, enabling larger batch sizes. Zheng et al. (2024) achieve significant speedups through global prefix sharing, processing common context across requests only once. Bang (2023) introduces GPTCache, a semantic cache that reduces costs and latency for similar queries by leveraging embedding similarity. Yu et al. (2024) show that KV cache heads can often be heavily compressed without performance loss. These findings motivate our approach: process the long transcript context once (summarize/index), cache a distilled representation, retrieve relevant details dynamically, and leverage the shared cache across batched questions, thereby maximizing efficiency and minimizing cost (Atamel, 2024; Google, 2024;).

We propose a multi-faceted approach:

1. Transcript Segmentation and Hierarchical Summarization: Dividing the transcript into manageable chunks and summarizing them (potentially hierarchically) using lightweight methods to create a concise representation.
2. Vector Retrieval (RAG - Retrieval Augmented Generation): Indexing transcript segments using vector embeddings to enable semantic search. Relevant segments are retrieved per question to provide specific context, grounding the LLM's response (Lewis et al., 2020).
3. Context Caching: Storing the summarized transcript context using Gemini's caching feature for cost-effective reuse across multiple queries (Jayawardena & Yankulin, 2024). Context caching is available for stable versions of Gemini 1.5 Pro. We will also explore

strategies to optimize this caching by considering potential prefix sharing across batched questions (Zheng et al., 2024).
4. Gemini 1.5 for Final Generation: Invoking the Gemini 1.5 Pro model only to synthesize the final answer based on the cached summary, retrieved segments, and the user's question. We will also investigate the potential integration of a semantic cache (like GPTCache) to further reduce costs and latency for repeated or similar questions (Bang, 2023).

By combining these methods, our approach aims to improve efficiency, preserve accuracy, and reduce costs. The following sections detail the technical implementation (with code examples), evaluation strategy, assumptions, and timeline.

2. Implementation

2.1. System Overview

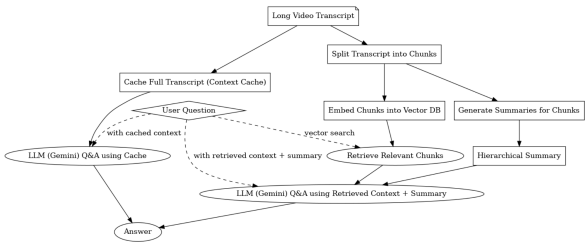

Figure 2 | *Proposed system architecture combining segmentation, summarization, vector retrieval, context caching, and batch Q&A for long video transcripts. Solid lines denote data flow during pre-processing/caching, dashed lines indicate the runtime QA pipeline using Gemini only at the final step.*

The system operates in stages:

1. Data Ingestion and Transcript Segmentation: Load the transcript and divide it into manageable segments (e.g., by token count or semantic boundaries). Handle multi-language input if necessary.
2. Hierarchical Summarization: Summarize each segment using a suitable (non-Gemini) model or method. Create a global summary from segment summaries.
3. Semantic Indexing (Vector Retrieval): Generate embeddings for transcript segments and build a vector index (e.g., using FAISS) for similarity search.
4. Context Caching Setup: Use the Vertex AI Gemini API to cache the global summary (and potentially a system instruction) for efficient reuse.
5. Batch Question Answering: Process a batch of user questions. For each question: potentially check a semantic cache for a prior answer, retrieve relevant segments, construct a prompt including retrieved context, and call the Gemini 1.5 API (using the cache) to generate the answer. Handle conversational history if needed. We will explore optimizing this step by considering potential common prefixes in the batch of questions to enhance throughput (Zheng et al., 2024).
6. Output Formatting: Collect and format the answers clearly.
7. Error Handling: Implement retries for API calls.

2.2. Data Ingestion and Transcript Segmentation

```python
import math
import re  # For more robust sentence splitting potentially
# from transformers import AutoTokenizer  # Example of using a tokenizer
# Load the transcript from a file
transcript_path = "video_transcript.txt"
try:
    with open(transcript_path, "r", encoding="utf-8") as f:
        full_transcript = f.read().strip()
except FileNotFoundError:
    print(f"Error: Transcript file not found at {transcript_path}")
    # Handle error appropriately, e.g., exit or use default text
    full_transcript = ""  # Example fallback
# Optional: Handle multi-language
# (Placeholder)
# def translate_to_english(text, src_lang):
#     # Placeholder for translation logic
#     print(f"Translating from {src_lang}...")
#     return text  # Dummy implementation
transcript_language = "en"  # Assume English or detect/specify
# if transcript_language != "en":
#     full_transcript = translate_to_english(full_transcript, src_lang=transcript_language)
# Segment the transcript
max_tokens_per_chunk = 8000  # Approximate target
```

```
# Using a simple word split; a proper tokenizer (e.g., tiktoken or from transformers) is better for accuracy
words = full_transcript.split()
chunks = []
current_pos = 0
while current_pos < len(words):
    end_pos = min(current_pos + max_tokens_per_chunk, len(words))
    chunk_words = words[current_pos:end_pos]
    chunk_text = " ".join(chunk_words)
    # Attempt to end chunk at a sentence boundary (optional refinement)
    if end_pos < len(words):
        # Find the last period, question mark, exclamation mark, or newline near the end
        sentence_ends = [m.start() for m in re.finditer(r'[.?!]\s+|\\n', chunk_text)]
        if sentence_ends:
            last_break_pos_in_chunk = sentence_ends[-1]
            # Ensure we don't create tiny chunks if the break is too early
            if last_break_pos_in_chunk > len(chunk_text) * 0.8:  # Heuristic: break only if near the end
                chunk_text = chunk_text[:last_break_pos_in_chunk+1]  # +1 to include the delimiter
                # Adjust word count for the next iteration
                end_pos = current_pos + len(chunk_text.split())
    # Else, just cut at max_tokens_per_chunk (default behavior)
    chunks.append(chunk_text.strip())
    current_pos = end_pos  # Move to the start of the next chunk
print(f"Transcript divided into {len(chunks)} segments.")
# Example: print length of first chunk
if chunks:
    print(f"Segment 1 length (approx words): {len(chunks[0].split())}")
```

2.3. Hierarchical Summarization Pipeline

```
# Placeholder for actual summarization logic (e.g., using Hugging Face transformers, an API, etc.)
def summarize_text_external(text, max_length=250):
    """
    Summarizes text using an external model/method (NOT Gemini).
    Replace this placeholder with your chosen summarization implementation.
    """
    print(f"Summarizing chunk of length {len(text.split())} words...")
    # Simple placeholder: take first few sentences
    sentences = re.split(r'(?<=[.?!])\s+', text)
    num_sentences = max(1, int(len(sentences) * 0.1))  # Take 10% of sentences or at least 1
    summary = " ".join(sentences[:num_sentences])
    # Truncate if needed (crude truncation)
    summary_words = summary.split()
    if len(summary_words) > max_length:
        summary = " ".join(summary_words[:max_length]) + "..."
    print(f" -> Summary length: {len(summary.split())} words")
    return summary if summary else "Summary could not be generated."

segment_summaries = []
for idx, chunk in enumerate(chunks, start=1):
    # Add error handling for summarization failures
    try:
        summary = summarize_text_external(chunk, max_length=200)
        segment_summaries.append(summary)
    except Exception as e:
        print(f"Error summarizing chunk {idx}: {e}")
        segment_summaries.append(f"Error summarizing segment {idx}")
# Create a global summary from segment summaries
global_summary = ""
if len(segment_summaries) > 1:
    combined_summaries_text = "\n\n".join(segment_summaries)
    try:
        # Longer summary for global overview
        global_summary = summarize_text_external(combined_summaries_text, max_length=400)
        print("\nSuccessfully generated global summary.")
    except Exception as e:
        print(f"Error generating global summary: {e}")
        global_summary = "[Global summary generation failed]"
elif segment_summaries:
    # If only one chunk, its summary is the global summary
    global_summary = segment_summaries[0]
else:
    global_summary = "[No content to summarize]"
print("\nGlobal Summary preview:\n", global_summary[:500], "...")
```

2.4. Vector Database and Retrieval-Augmented QA

```
from sentence_transformers import SentenceTransformer
import numpy as np
import faiss
```

```

# Use original chunks for detailed retrieval
texts_to_embed = chunks
chunk_embeddings_normalized = np.array([])  # Initialize as empty numpy array
index = None
embedder = None  # Initialize embedder
if not texts_to_embed:
    print("Warning: No transcript chunks available for embedding.")
    # Handle gracefully: maybe skip indexing or use summaries if available
else:
    try:
        print("Loading embedding model...")
        # Consider using models optimized for retrieval (e.g., 'msmarco-distilbert-base-tas-b')
        embedder = SentenceTransformer("all-MiniLM-L6-v2")  # Lightweight general model
        print("Generating embeddings for transcript chunks...")
        chunk_embeddings = embedder.encode(texts_to_embed, show_progress_bar=True)
        # Normalize embeddings for cosine similarity using IndexFlatIP
        norms = np.linalg.norm(chunk_embeddings, axis=1, keepdims=True)
        # Avoid division by zero for zero-vectors if any
        norms[norms == 0] = 1e-10
        chunk_embeddings_normalized = chunk_embeddings / norms
        # Build FAISS index
        embedding_dim = chunk_embeddings_normalized.shape[1]  # Corrected from shape7
        index = faiss.IndexFlatIP(embedding_dim) # IP for cosine similarity on normalized vectors
        index.add(chunk_embeddings_normalized.astype(np.float32))  # FAISS expects float32
        print(f"FAISS index built successfully with {index.ntotal} vectors.")
    except Exception as e:
        print(f"Error during embedding or indexing: {e}")
        index = None  # Ensure index is None if building failed

# Retrieval function
def retrieve_relevant_segments(question, top_k=3):
    """Retrieves top_k relevant transcript segments based on semantic similarity."""
    if index is None or index.ntotal == 0:
        print("Warning: Vector index not available for retrieval.")
        return []
    if embedder is None:
        print("Warning: Embedding model not loaded.")
        return []
    try:
        print(f"Retrieving top {top_k} segments for question: '{question[:50]}...'")
        q_vec = embedder.encode([question])
        q_vec_norm = q_vec / np.linalg.norm(q_vec, axis=1, keepdims=True)
        q_vec_norm[np.isnan(q_vec_norm)] = 0 # Handle potential NaN if norm is zero
        # Search the index
        distances, indices = index.search(q_vec_norm.astype(np.float32), top_k)
        # Filter out invalid indices (e.g., -1 if fewer than top_k results)
        valid_indices = [i for i in indices[0] if i != -1]
        # Return the original chunk text corresponding to the valid indices
        results = [texts_to_embed[i] for i in valid_indices]
        print(f" -> Found {len(results)} relevant segments.")
        return results
    except Exception as e:
        print(f"Error during retrieval: {e}")
        return []

# Example retrieval
sample_question = "What methodology is proposed for analysis?"
relevant_segments = retrieve_relevant_segments(sample_question, top_k=2)
if relevant_segments:
    print("\nTop relevant segment snippets for sample question:")
    for i, seg in enumerate(relevant_segments):
        print(f"Snippet {i+1}: {seg[:150]}...\n")
```

2.5. Context Caching with Gemini 1.5 Pro

```
from google.cloud import aiplatform  # Using Vertex AI SDK
from google.cloud.aiplatform_v1beta1.types import Content, Part, GenerationConfig, Tool  # Using v1beta1 for potential newer features like caching
from google.cloud.aiplatform_v1beta1.types import CachedContent  # Specific type for cache

# Initialize Vertex AI SDK Client (ensure ADC or service account key is configured)
# Assumes GOOGLE_CLOUD_PROJECT and GOOGLE_CLOUD_LOCATION (e.g., 'us-central1') are set
try:
    aiplatform.init(project="your-gcp-project-id", location="your-region")  # Replace with your project/location
    # Verify client setup by listing models or another simple call if needed
    # print("Vertex AI SDK Initialized.")
```

```
except Exception as e:
    print(f"Error initializing Vertex AI SDK: {e}")
    # Handle error appropriately

cache = None  # Initialize cache variable
cache_name = None  # Initialize cache_name variable
# Check if global_summary exists and wasn't an error message
if global_summary and not global_summary.startswith("["):
    system_instruction_text = (
        "You are a helpful assistant expert in analyzing video transcripts. "
        "Use the provided summary for general context and the retrieved excerpts "
        "for specific details to answer the user's question accurately and concisely."
    )
    system_instruction_content = Content(role="system", parts=[Part(text=system_instruction_text)])
    # Prepare the global summary as user content for caching
    # Caching typically caches the 'model' role context or system instructions.
    # Let's cache the system instruction and the global summary together if possible,
    # or primarily the summary as 'model' pre-fill or initial 'user' turn.
    # Based on SDK examples, caching the initial context turns (user+model or system+user) seems common.
    content_to_cache = [
        # System instruction might be part of the cache definition or the content itself
        system_instruction_content,
        Content(role="user", parts=[Part(text="Here is the summary of the video transcript:")]),
        Content(role="model", parts=[Part(text=global_summary)]) # Cache the summary as if the model provided it
    ]
    # Define the model for caching
    # Ensure this model ID is correct, a stable version of Gemini 1.5 Pro, and supports caching in your region
    target_model_for_caching = "gemini-1.5-pro-001"  # Example model name
    # Ensure the full model resource name is used if required by the SDK
    # e.g., <br> f"projects/{PROJECT_ID}/locations/{LOCATION}/publishers/google/models/{target_model_for_caching}"
    print(f"Attempting to create context cache for model {target_model_for_caching}...")
    try:
        # Create the CachedContent resource
        cache = aiplatform.gapic.CachedContentServiceClient().create_cached_content(
            parent=f"projects/{aiplatform.initialize.global_config.project}/locations/{aiplatform.initialize.global_config.location}",
            cached_content=CachedContent(
                model=f"projects/{aiplatform.initialize.global_config.project}/locations/{aiplatform.initialize.global_config.location}/publishers/google/models/{target_model_for_caching}",
                system_instruction=system_instruction_content,  # System instruction part of definition
                contents=content_to_cache[1:], # Cache the user/model turns after system instruction
                display_name="video_transcript_global_summary_cache",
                ttl="3600s"  # 1 hour Time To Live
            )
        )
        cache_name = cache.name  # Store the resource name (e.g., projects/.../cachedContents/...)
        print(f"Successfully created context cache: {cache_name}")
    except Exception as e:
        print(f"Error creating context cache: {e}")
        # Possible reasons: Invalid model name, permissions, unsupported region, API errors, SDK structure mismatch.
        cache = None
        cache_name = None
else:
    print("Skipping context cache creation (no valid global summary).")
# Note: The exact SDK calls and structure for context caching might evolve.
# Refer to the latest Google Cloud Vertex AI SDK documentation for Gemini 1.5 Pro caching.
# The `google-cloud-aiplatform` SDK is generally used for Vertex AI resources.
# Ensure the model version used supports caching ('001' suffix often indicates stable releases).
```

2.6. Batch Question Answering and Conversation Handling

```
import time
from google.cloud.aiplatform_v1beta1 import PredictionServiceClient
from google.protobuf import json_format
from google.protobuf.struct_pb2 import Value
```

```python

# Robust generation function with retries using Vertex AI Prediction Endpoint
def generate_with_retries_vertex(
    prompt_text,
    cache_name_to_use,
    model_id="gemini-1.5-pro-001",
    temperature=0.3,
    max_output_tokens=1024,
    max_retries=3,
    backoff_factor=1.5
):
    """Generates text using Vertex AI Gemini endpoint with cached content and retries."""
    if not cache_name_to_use:
        print("[Error: Context cache is not available]")
        return "[Error: Context cache is not available]"
    # Construct the full model name resource path
    project_id = aiplatform.initialize.global_config.project
    location = aiplatform.initialize.global_config.location
    model_resource_name = f"projects/{project_id}/locations/{location}/publishers/google/models/{model_id}"
    # The client needs the regional endpoint
    client_options = {"api_endpoint": f"{location}-aiplatform.googleapis.com"}
    prediction_client = PredictionServiceClient(client_options=client_options)
    # Prepare the prompt content
    prompt_content = [Content(role="user", parts=[Part(text=prompt_text)])]
    # Prepare the instance for prediction
    # Note: The exact structure might vary based on API version. Check docs.
    # We reference the cached_content resource name.
    instance = json_format.ParseDict({
        "contents": [json_format.MessageToDict(c) for c in prompt_content],
        "cached_content": cache_name_to_use,
        # System instruction might be implicitly handled by cache, or needs restating if required by API.
        # Check documentation if system_instruction should be passed again here.
    }, Value())
    instances = [instance]
    parameters = json_format.ParseDict({
        "temperature": temperature,
        "maxOutputTokens": max_output_tokens,
        "topP": 0.95,  # Example other parameters
        "topK": 40
    }, Value())
    for attempt in range(1, max_retries + 1):
        try:
            print(f"Attempt {attempt}: Generating answer using cache {cache_name_to_use.split('/')[-1]}...")
            response = prediction_client.predict(
                endpoint=model_resource_name, # Endpoint should be the model resource name for direct model prediction
                instances=instances,
                parameters=parameters,
            )
            # Process the response - structure depends on the API response format
            # Typically response.predictions is a list of structs/dicts
            prediction_result = json_format.MessageToDict(response.predictions[0])
            # Extract text from candidate(s) - check for safety blocks
            if 'candidates' in prediction_result and prediction_result['candidates']:
                candidate = prediction_result['candidates'][0]
                if 'content' in candidate and 'parts' in candidate['content']:
                    answer_text = "".join(part.get('text', "") for part in candidate['content']['parts'])
                    print(f" -> Success on attempt {attempt}.")
                    return answer_text.strip()
                else:
                    # Handle cases like safety blocks or unexpected structure
                    finish_reason = candidate.get('finishReason', 'UNKNOWN')
                    safety_ratings = candidate.get('safetyRatings', [])
                    print(f"Warning: Received empty parts or blocked content. Finish Reason: {finish_reason}, Safety: {safety_ratings}")
                    return f"[Answer blocked or empty: {finish_reason}]"
            else:
                # Handle cases with no candidates (e.g., citation issues, other errors)
                print(f"Warning: No candidates found in response. Full prediction result: {prediction_result}")
                return "[No answer generated]"
        except Exception as e:
            print(f"Error during generation (attempt {attempt}/{max_retries}): {e}")
            if attempt < max_retries:
```

```
                sleep_time = backoff_factor ** attempt
                print(f"Retrying in {sleep_time:.2f} seconds...")
                time.sleep(sleep_time)
            else:
                print("Max retries exceeded.")
                return f"[Error: Generation failed after {max_retries} attempts]"
    return "[Error: Generation failed unexpectedly]"

# Function to answer a single question using cache and retrieval
def answer_question_with_cache_and_retrieval(question, cache_name_to_use, top_k=2):
    # Retrieve relevant segments
    relevant_texts = retrieve_relevant_segments(question, top_k=top_k)
    # Construct the prompt including retrieved context
    if relevant_texts:
        context_section = "\n\n".join(f"Excerpt {i+1}:\n{text}" for i, text in enumerate(relevant_texts))
        prompt_text = (
            f"Based on the overall video summary (already provided in context) and the following relevant excerpts from the transcript, please answer the question.\n\n"
            f"Relevant Excerpts:\n"
            f"--------------------\n"
            f"{context_section}\n\n"
            f"--------------------\n\n"
            f"Question: {question}\n\n"
            f"Answer:"
        )
    else:
        # Fallback if no relevant segments found
        prompt_text = (
            f"Based on the overall video summary (already provided in context), please answer the following question. Note: No specific relevant excerpts were found for this question.\n\n"
            f"Question: {question}\n\n"
            f"Answer:"
        )
    # Generate answer using the robust function
    answer = generate_with_retries_vertex(prompt_text, cache_name_to_use, model_id=target_model_for_caching)
    return answer

# Example batch of questions
questions = [
    "What is the main topic discussed in the video?",
    "Explain the concept of hierarchical summarization mentioned.",
    "Are there any specific examples or case studies presented?",
    "What are the benefits of using context caching according to the content?"
]
# Process batch questions
answers = {}
if cache_name:  # Only proceed if cache was successfully created
    print("\n--- Starting Batch Question Answering ---")
    for q_idx, q in enumerate(questions):
        print(f"\nProcessing Question {q_idx+1}/{len(questions)}: '{q}'")
        ans = answer_question_with_cache_and_retrieval(q, cache_name, top_k=2)
        answers[q] = ans
        # Optional: Add slight delay between requests to respect API rate limits
        # time.sleep(1)
else:
    print("Cannot proceed with question answering as context cache is unavailable.")
# Note on Conversation Handling: To handle follow-up questions, maintain a conversation_history list or string.
# Before calling generate_with_retries_vertex for a new question, prepend the relevant history to the prompt_text.
# Be mindful of context length limits when including history. Example:
# history_str = "\n\nPrevious Q\&A:\nQ: Previous Question?\nA: Previous Answer\n"
# full_prompt = history_str + current_prompt_text
```

2.7. Output Formatting

```
# Format and display results.
print("\n--- Batch Question Answering Results ---")
if answers:
    for i, (q, ans) in enumerate(answers.items(), start=1):
        print(f"\n{i}. Question: {q}")
        print(f" Answer: {ans}")
else:
    print("No answers were generated (possibly due to missing cache or errors).")
print("\n" + "="*40)
```

2.8. Incorporation of Recent Research

Our implementation strategy directly reflects insights from recent LLM serving research:

- Context Caching as Prefix Sharing: By using Gemini's context caching (Jayawardena

& Yankulin, 2024; Google, 2024), we store the processed global_summary and system instructions. This acts as a shared prefix for all subsequent questions regarding that transcript, conceptually mirroring the benefits of global prefix sharing described by Zheng et al. (2024) by avoiding redundant processing of this common context, leading to cost and latency reductions (Atamel, 2024). We will further explore if analyzing the batch of incoming questions can reveal additional common prefixes that can be leveraged, potentially by incorporating them into the system instruction or by structuring the individual question prompts to maximize the benefits of this shared context.

- Semantic Caching for Repeated Queries: To address the potential for repeated or semantically similar user questions, we will investigate integrating a semantic cache like GPTCache (Bang, 2023). This would involve caching question-answer pairs based on the semantic similarity of the queries, allowing the system to return cached answers for subsequent similar questions without needing to call the Gemini API again. This would lead to further cost savings and reduced latency.
- Summarization as Context Compression: Hierarchical summarization reduces the transcript length significantly before caching. This acts as a form of information compression, conceptually related to techniques like KV cache compression (Metel et al., 2024) or head pruning (Yu et al., 2024), as it allows the core information to be retained in a smaller footprint, making caching and processing more efficient.
- RAG for Focused Detail: Instead of caching the entire multi-million token transcript (which might still be costly or slow to attend over, even if cached), we cache only the summary and use RAG (Lewis et al., 2020) to dynamically inject relevant detailed segments. This focuses the model's attention during generation, potentially improving accuracy on specific factual questions and further optimizing resource usage.
- Batching for Throughput: While our example loops sequentially, the structure supports parallel execution of answer_question_with_cache_and_retrieval for multiple questions (respecting rate limits) or using Vertex AI's batch prediction endpoint (Atamel, 2024), aligning with the goal of maximizing throughput discussed by Metel et al. (2024) and Zheng et al. (2024). We will explore the feasibility of using the batch prediction API in conjunction with context caching to further optimize performance and cost.

3. Evaluation

To validate the effectiveness, we will evaluate along several dimensions:

3.1. Evaluation Metrics

1. Answer Quality: Human evaluation (rating correctness, relevance, completeness on a 1-5 scale) or automated metrics (e.g., ROUGE for summarization tasks, Exact Match/F1 for factual QA if ground truth exists) comparing answers generated via:
    - Baseline: Simple chunking + RAG without global summary cache.
    - Proposed Method: Cached global summary + RAG.
    - Enhanced Method: Cached global summary + RAG + Semantic Caching (if implemented) + Prefix Sharing Considerations.
2. Latency: End-to-end time per question (including retrieval and generation, and cache lookups if semantic caching is implemented) for the proposed method vs. baseline and the enhanced method. Measure time for the initial caching step separately.
3. Cost Efficiency: Estimate costs based on Google Cloud pricing (Google Cloud Pricing, n.d.) for token processing (input, output, cached). Compare the cost of answering N questions using the proposed method (1 cache creation + N cheaper generation calls) vs. baseline (N standard generation calls potentially with larger context), and the enhanced method (considering potential

reductions due to semantic cache hits and prefix sharing efficiencies). Use cost savings figures from sources like Google (2024) and Atamel (2024) for estimation.
4. Throughput: Measure how many questions can be answered per minute/hour, especially when processing questions in parallel or using batch endpoints. Document observed rate limits.
5. Robustness: Track the frequency of API errors and successful retries. Assess failure modes (e.g., unanswerable questions, retrieval misses).
6. Semantic Cache Hit Rate (if implemented): Measure the percentage of questions that are answered directly from the semantic cache.

3.2. Experimental Setup

- Data: Select 2-3 diverse long video transcripts (e.g., lecture, documentary, meeting) > 30 mins duration.
- Queries: Create a set of 10-15 questions per transcript (mix of factual, summary, analytical).
- Configuration: Use consistent settings (e.g., top_k for retrieval, Gemini generation temperature = 0.3). For the enhanced method, configure the semantic cache (if implemented) with an appropriate similarity threshold and embedding model.
- Execution: Run queries using both baseline and proposed methods, record metrics.

3.3. Hypotheses

- The proposed method will have significantly lower cost per question after the initial caching cost is amortized over several queries.
- Per-question latency for the proposed method will be lower due to cache reuse.
- Answer quality will be comparable or slightly better due to the combination of global summary context and retrieved details.
- The enhanced method, with semantic caching, will further reduce latency and cost for repeated or similar questions.
- The system will handle transient API errors gracefully via retries.

3.4. Considerations

- Quality of external summarization impacts the global context.
- Quality of embeddings and retrieval impacts detail accuracy.
- Cache TTL needs alignment with usage patterns.
- Potential loss of nuances compared to processing the full transcript (trade-off for efficiency).
- The effectiveness of semantic caching (if implemented) depends on the choice of embedding model and the similarity threshold.
- Implementing prefix sharing optimizations might require careful analysis of the batch of questions and adjustments to the prompting strategy.

3.5. Assumptions

This project makes several key assumptions to simplify the problem and focus on the core ideas:

- Extremely Large Context LLM Availability: We assume access to Gemini 1.5 Pro, which offers a 2-million-token context window. This model also supports context caching, which is a key component of our approach.
- Quality of Transcript and Summaries: We assume the input transcript is a reasonably accurate transcription of the video (e.g., no severe ASR errors if it was auto-generated). We also assume that the summarization performed by the external model is accurate and preserves the important information from each segment. If the summaries were poor or omitted critical facts, the system might fail to retrieve relevant context for certain questions. We mitigate this by hierarchical summarization - the model sees smaller chunks at a time, which tends to produce better summaries (Cohan et al., 2018). Nonetheless, we assume that any information needed to answer the questions is present either in a summary or in the original text of a segment that can be retrieved.

- Embedding Efficacy: It is assumed that the embedding model we use for semantic search effectively clusters similar content and matches questions to the correct segments. Modern embedding models (like those derived from BERT or other transformers) are generally good at semantic similarity, and prior works like GPTCache rely on this property for cache hits (Fu & Feng, 2023). We assume a high-quality, possibly domain-adapted embedding model is available for our use (this could even be an embedding from the Gemini model, if provided, or an open-source model like Instructor-XL). If embeddings are poor, retrieval will suffer. Our evaluation will check some embedding results manually to ensure the assumptions hold.
- Cache Persistence and Validity: We assume that cached results remain valid for the duration of their use. For instance, if the transcript content is updated or corrected, our cache might serve outdated answers. In this project, we assume a static transcript. Similarly, we assume user questions will generally be repeated exactly or semantically for the cache to be useful (which is reasonable in scenarios like multiple users asking similar questions about the same content, or a user revisiting a question). We also assume sufficient memory/storage to maintain the cache and embedding index, though for a single video transcript this is not likely to be huge (a few hundred embeddings and summaries at most).
- Batch and Async Execution Environment: It is assumed that we can run multiple LLM queries in parallel (either via threading as shown, or via an API that supports batch requests). This requires that the environment (e.g., a Jupyter notebook or a backend server) can handle asynchronous calls and that the LLM API is either non-blocking or can be called concurrently without issue. We also presume that the rate limits of the API allow a burst of multiple requests. In a production deployment, we might need to throttle or adjust batch sizes, but for this proposal, we assume we can effectively get parallelism for a modest batch of questions.
- Use of External Services: The implementation assumes the use of external services such as the Google Gemini API (via Vertex AI) and possibly cloud database services for pgvector (as an alternative to FAISS). We assume access to these services and that their performance is as advertised (e.g., the LLM returns answers within a couple of seconds on average as per the demo mentioned, and vector queries are fast). We also assume no drastic cost constraints in calling the API for our testing and demonstration, aside from what we aim to optimize through caching.

4. Tech Stack

- Language Model: Google Gemini 1.5 Pro via Vertex AI API - for both summarization and question answering. This provides the core AI capabilities with an extended context window and the ability to handle complex instructions. The Vertex AI Generative AI SDK (e.g., the google.generativeai Python library) will be used to interface with the model. We will run in the cloud (or Colab) environment to leverage this service.
- Programming Language: Python 3.x - chosen for its rich ecosystem in AI and data processing, including convenient libraries for HTTP requests, concurrency, and data handling. Jupyter/Colab notebooks will be used during development for interactive experimentation and demonstration.
- Vector Database: FAISS (in-memory index) - used in the code example for simplicity. Alternatively, PostgreSQL with the pgvector extension can be used for persistent storage and efficient similarity search on embedding vectors using SQL. This choice allows us to avoid setting up a separate specialized vector store; it can be hosted locally or via a cloud Postgres instance.
- Embedding Model: We will use a pre-trained embedding model for semantic similarity. Options include OpenAI's text-embedding-ada-002, Sentence Transformers (e.g., all-MiniLM-L6-v2 for a lightweight option), or the embedding capabilities of Vertex AI if

available. The choice will depend on what integrates best with the environment. The embedding model is critical for GPTCache-like functionality and for relevant summary retrieval.

- Caching Mechanism: For context caching, we will leverage the built-in caching capabilities of the Gemini API via the Vertex AI SDK. This allows us to store and reuse pre-computed input tokens for cost and speed savings.
- Asynchronous Processing: Python's concurrent.futures or asyncio will be used to handle batch queries. If the Vertex AI SDK supports async natively (some SDKs return futures or have async methods), we will leverage that. Otherwise, we will manage threads or async coroutines to parallelize calls. We might also use Ray or PySpark for scaling out if we were to parallelize across machines, but that's likely overkill for the scope (batch of questions on one transcript).
- Data and Storage: The transcript data will be loaded from a file or obtained via YouTube API (if we integrate a step to fetch transcript by video URL). We plan to keep things file-based for simplicity in a code sample context (e.g., a local text file for the transcript, and local storage for any caches). For multi-language support, we might use translation APIs or rely on multi-lingual embeddings, but that is an optional extension.
- Environment: The development and testing will be done in a controlled environment (Google Colab or Vertex AI Workbench) to have access to the Gemini API. We will also utilize version control (GitHub) to manage our code, and possibly either share a Notebook via GitHub or develop locally for now containerised using poetry.
- Libraries: Common Python libraries such as numpy (for vector math), pandas (if needed for data manipulation), and requests or the official Vertex AI client will be used. If needed for summarization or additional text processing, we might bring in nltk or spacy for sentence segmentation. However, the heavy lifting of text summarization and QA is done by the LLM.

This tech stack is chosen to balance ease of development and alignment with modern best practices. By using Vertex AI and FAISS (or pgvector), we also ensure the solution is compatible with Google Cloud's ecosystem (which might be relevant if this is aimed at demonstrating Vertex AI's capabilities like context caching). The open-source components like SentenceTransformers illustrate how the solution is extensible and not tied to proprietary tech if we were to generalize it.

5. Timeline (8 Weeks)

5.1. Week 1-2: Research, Design, and Setup

During the first two weeks, we will finalize the project scope and gather detailed requirements, such as expected transcript length, types of questions, and performance targets. The team will research and select specific external models/APIs for summarization (e.g., BART via Hugging Face Transformers) and embedding (e.g., all-MiniLM-L6-v2 via sentence-transformers). Following this, the system architecture diagram (Figure 2) and data flow will be refined. We will design initial prompt structures for the final Gemini Q&A step, with a focus on incorporating retrieved context effectively. The development environment will be set up by installing necessary Python libraries (google-generativeai, sentence-transformers, faiss-cpu, etc.), ensuring Google Cloud project setup with Vertex AI API enabled, obtaining necessary credentials (e.g., configure Application Default Credentials), and testing basic Gemini API calls (e.g., list_models, simple generate_content). The final step will be to identify and prepare 2-3 sample long video transcripts for development and testing.

5.2. Week 3-4: Segmentation and Summarization Pipeline

Weeks 3 and 4 will be used to implement the transcript ingestion and segmentation module. Python functions will be written to load and segment transcript files based on the chosen strategy (e.g., 8000 tokens per chunk, attempting sentence boundary alignment). We will also integrate a

proper tokenizer library for more accurate token counting. The hierarchical summarization pipeline will be implemented by integrating the chosen external summarization model/API to process each segment individually. Logic will be written to combine segment summaries to generate the final global summary. This period will also include significant testing on sample transcripts to ensure segmentation is robust and summarization quality (conciseness, accuracy) is adequate. Summarization parameters (e.g., max_length) or prompts will be tuned if an LLM-based external summarizer is used. The goal of these two weeks is to produce a reliable script that takes a transcript file and outputs segmented chunks, corresponding segment summaries, and a global summary.

5.3. Week 5: Embedding, Retrieval, and Caching Setup

The semantic retrieval component will be implemented by writing code to generate vector embeddings for the original transcript segments using the selected sentence-transformers model. The FAISS index (IndexFlatIP) will be built in memory. The team will also implement the retrieve_relevant_segments function to perform similarity search between a question embedding and the indexed segments. Retrieval accuracy will be tested with sample questions. Concurrently, the Vertex AI context caching setup will be implemented by writing Python code using the google-generativeai SDK to create a CachedContent resource. This will store the global_summary and the system instruction, specifying a stable version of the Gemini 1.5 Pro model (e.g., gemini-1.5-pro-001) and TTL. Cache creation will be verified via the API response. The goal of week 5 is to produce a working retrieval function and successfully create/verify a context cache resource for a test transcript's global summary.

5.4. Week 6: Batch Q&A Implementation and Integration

In week 6, we will develop the core batch question answering logic by implementing the answer_question_with_cache_and_retrieval function. This will integrate the semantic retrieval step (calling retrieve_relevant_segments) and the final generation step (calling generate_with_retries). The prompt will be designed to incorporate the retrieved excerpts correctly. Additionally, the team will implement the robust generate_with_retries function, incorporating exponential backoff for API error handling. The main script will be written to loop through a batch of user questions, call the answering function for each, and collect the results. Initial end-to-end tests will be performed using one sample transcript, its generated cache, and a small batch of questions to ensure all components work together. The goal is to create a functional end-to-end pipeline that can process a list of questions against a cached transcript summary and retrieved context, returning answers with basic error handling.

5.5. Week 7: Formal Evaluation and Performance Analysis

The evaluation plan outlined in the "Evaluation" section will be executed. The system (and potentially a baseline method) will be run against the prepared test transcripts and question sets. Quantitative data: answer quality ratings (human judgment or automated scores if feasible), end-to-end latency per question, simulated costs based on token counts (cache creation, cached generation, standard generation), and throughput (questions per minute) will be collected. The data will be analyzed, comparing the proposed method against the baseline, and verifying hypotheses regarding cost, latency, and quality trade-offs. Any failures, error rates, or performance bottlenecks observed will be documented. We will allocate additional time this week to thoroughly analyze the results and address any unexpected findings.

5.6. Week 8: Documentation, Refinement, and Finalization

Based on evaluation findings and testing from previous weeks, we will perform final refinements to the code (e.g., optimize prompts, adjust retrieval top_k, improve error messages). We will write comprehensive documentation, including

a README explaining the project setup, architecture, usage, and configuration options. Detailed comments will be added to the Python code. This proposal document will be finalized with results (if applicable) or ensure it accurately reflects the implemented system. The codebase will be cleaned up, ensuring adherence to any required coding standards, and preparing the project for deployment. We will also ensure that the documentation clearly outlines the choice of Gemini 1.5 Pro and the use of the Vertex AI API for context caching.

6. Conclusion

By integrating transcript segmentation, external hierarchical summarization, retrieval-augmented generation, and Gemini's context caching, the approach balances performance, cost, and quality. The methodology draws inspiration from recent LLM serving optimizations like prefix sharing (Zheng et al., 2024) and context compression concepts (Metel et al., 2024; Yu et al., 2024). Furthermore, the potential integration of a semantic cache (Bang, 2023) will add another layer of efficiency for repeated queries. The provided implementation structure offers a concrete starting point for building robust and cost-effective applications for analyzing extensive video or text content. The evaluation plan ensures validation of the proposed benefits, resulting in a well-documented and performant code sample.

7. References

References

[1] Bang, F. (2023). GPTCache: An open-source semantic cache for LLM applications enabling faster answers and cost savings. paper link

[2] Jayawardena, N., & Yankulin, L. (2024, November 15). Control your Generative AI costs with the Gemini API's context caching. Google Cloud Community Blog (Medium link).

[3] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Ott, M., Chen, D., Smith, L., & Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. Advances in Neural Information Processing Systems, 33, 9459–9474. paper link

[4] Metel, M. R., Chen, B., & Rezagholizadeh, M. (2024). Batch-Max: Higher LLM throughput using larger batch sizes and KV cache compression. arXiv preprint arXiv:2412.05693. paper link

[5] Yu, H., Yang, Z., Li, S., Li, Y., & Wu, J. (2024). Effectively compress KV heads for LLM. arXiv preprint arXiv:2406.07056. paper link

[6] Zheng, Z., Ji, X., Fang, T., Zhou, F., Liu, C., & Peng, G. (2024). BatchLLM: Optimizing large batched LLM inference with global prefix sharing and throughput-oriented token batching. arXiv preprint arXiv:2412.03594. paper link

[7] Artfish.ai. (n.d.). Long Context LLMs. Retrieved April 6, 2025, from article link

[8] Atamel, M. (2024, November 18). Batch prediction in Gemini. Google Cloud Blog. article link

[9] Vertex AI Gemini API context caching documentation. article link

[10] Gemini 2.0 Flash and Gemini 1.5 Pro context window details. article link

[11] Google Cloud. (n.d.). Vertex AI pricing. Retrieved April 6, 2025, from article link

[12] Introduction to context caching with Gemini API in Vertex AI. article link

[13] Gemini API context caching overview and usage. article link

[14] Intro to Context Caching with the Gemini API using Vertex AI SDK. article link