Advanced Visual Understanding Chat Assistant

Vuencode Hackathon Documentation

Laaksh Parikh
23je0516@iitism.ac.in

Hemant Pathak
23je0402@iitism.ac.in

Aditya Pratap Singh
23je0051@iitism.ac.in

Kunal Verma
23je0509@iitism.ac.in

Hardik Sirohiya
23je0@0387iitism.ac.in

August 18, 2025

Abstract

This report documents the design and development of a low-latency, scalable chat assistant for advanced visual understanding. Our initial approach in Round 1 involved a multi-stage pipeline: employing YOLO for object detection, followed by Gemini Vision Pro for event recognition, and finally, summarization. As we progressed to Round 2, the focus shifted to performance and scale, prompting exploration of self-hosted open-source models on an NVIDIA A100 GPU. This phase presented significant challenges, including version mismatches within the transformers library that rendered some models incompatible, and severe resource constraints due to the 80GB VRAM limit, which hindered our ability to meet the project's strict performance criteria. In response to these challenges, we engineered a final, more effective solution utilizing the Gemini 2.0 Flash model. By leveraging its API, we achieved superior results in both video summarization and conversational Q&A. This report details our final architecture, including a custom RAG implementation for precise query handling and an effective method for bypassing API rate-limit errors, ultimately delivering a system that successfully balances performance, scalability, and advanced functionality.

Contents

1 Problem Statement

Although the Round 1 prototype demonstrated the feasibility of building an agentic chat assistant for visual understanding, its functionality was limited to short videos and summary of basic events. In real-world applications such as surveillance, traffic monitoring, and long-form media analysis, systems must handle significantly larger workloads while still providing timely and accurate responses.

The primary challenge in Round 2 is to extend the assistant into a **high-performance and scalable system** capable of:

- Processing long-form videos of up to 120 minutes at high frame rates (up to 90 fps).
- Achieving low-latency response times of less than 1000 milliseconds, ensuring near-real-time conversational interactions.
- Maintaining contextual consistency across extended, multi-turn conversations.
- Efficiently integrate advanced visual language models (VLMs) and leverage GPU acceleration for optimization.

Existing video analysis solutions often trade off between accuracy and latency, or fail to scale efficiently when subjected to long-duration, high-resolution streams. Our goal is to overcome these limitations by designing a system architecture that emphasizes both performance and scalability, ensuring that the assistant can operate reliably in high-throughput, real-world environments.

2 Our Approach and Methodology

This section details the technical core of our project, from initial brainstorming to the final, high-performance architecture we implemented to solve the hackathon's challenges.

2.1 Initial Ideas and Things We Tried

Our journey from Round 1 to Round 2 was marked by a significant evolution in strategy, driven by the shift from demonstrating core functionality to achieving high performance and scalability. This section details the chronological progression of our ideas, the various models we experimented with, and the critical learnings that led to our final approach.

2.1.1 Learnings from Round 1: A Baseline with Proprietary Models

Initially, in Round 1, our strategy was inspired by the research paper "Video Summarisation with Incident and Context Information using Generative AI". Our first attempt involved a complex pipeline:

- Deconstructing the input video into individual frames.
- Employing YOLO for object detection on these frames.
- Integrating other specialized models for specific event detection.

However, this component-based approach proved to be suboptimal, failing to yield the desired accuracy and consistency in its results. Faced with time constraints, we made a strategic pivot to "Plan B," which utilized the **Gemini 1.5 Flash API**. To manage conversation history and context, we integrated a Retrieval-Augmented Generation (RAG) system to store summarized text for efficient query retrieval. This provided a functional baseline but relied on a proprietary, black-box model.

2.1.2 Round 2: The Pursuit of a High-Performance Open-Source Solution

With the provision of a powerful **NVIDIA A100 80GB GPU** for Round 2, our primary goal was to build a high-performance solution using an open-source model that we could control and optimize. Our exploration was methodical:

Attempt 1: Large Vision-Language Models (Image-to-Text). We began by investigating popular Vision-Language Models (VLMs) from Hugging Face. Our initial candidates included:

- `Qwen/Qwen2.5-VL-7B-Instruct`
- `llava-hf/llava-1.5-7b-hf`
- `moonshotai/Kimi-K2-Instruct`

We encountered significant technical hurdles in setting up and running these models. After hours of debugging, we managed to get them operational. However, we identified a fundamental architectural mismatch: these were **image-text to text** models. This required us to feed video as a sequence of individual frames, which led to unacceptably high temporal data usage and severe latency, directly conflicting with the core requirements of Round 2.

Attempt 2: Smaller, More Efficient Models. Hypothesizing that model size was the bottleneck, we shifted our focus to a smaller model, `OpenGVLab/InternVL2-4B`. While this model performed reasonably well on very short video clips, it did not resolve the core latency issues and still struggled with longer inputs. This experience solidified our conclusion that the frame-by-frame processing approach was a dead end.

Attempt 3: Dedicated Video-Text to Text Models. We pivoted our search to models explicitly designed for video input. Given its popularity, the LLaVA family was our first choice. We experimented with `llava-hf/LLaVA-NeXT-Video-7B-hf` and its DPO variant. These models showed promise and handled short videos effectively, but failed to process the larger, long-duration videos required by the hackathon specifications. We then tested `GoodiesHere/Apollo-LMMs-Apollo-7B-t32`, which was designed for longer contexts, but found it struggled with high frames-per-second (FPS) content.

2.1.3 The Deep Dive: Advanced Optimization with Video-XL-2

Our extensive search led us to **`BAAI/Video-XL-2`**, a model that seemed perfectly suited for our task. It was not only designed for long-video comprehension (up to 10,000 frames) but was also trained on the same A100 GPU hardware we were assigned.

1. **Initial Success and a New Challenge:** The model exceeded its documented capabilities, successfully processing videos with up to 30,000 frames. However, the hackathon's maximum video size could reach nearly 600,000 frames.
2. **Intelligent Frame Selection:** To bridge this gap, we implemented an intelligent frame selection strategy to downsample the video, extracting the 20,000 most contextually significant frames for analysis. This creative workaround yielded results far better than we anticipated.
3. **Fine-Tuning with Research:** To push performance further, we studied the model's underlying research paper, "Video-XL: A Simple Method for Scaling up Video-Language Representation Learning" [**?**]. Armed with this deeper understanding, we fine-tuned its parameters to better align with our specific use case.

This intensive effort culminated in a major breakthrough: our optimized Video-XL-2 pipeline could summarize a 2-hour video in approximately **60 seconds**. While a massive improvement, this was still far from the sub-second latency target.

2.1.4 Final Pivot: Returning to a Proprietary API for Performance

Faced with this insurmountable performance ceiling with the available open-source tools, we made the final strategic decision to pivot back to a proprietary API. We needed a model that could natively handle long video streams without exorbitant token limits and with minimal latency. After a thorough evaluation, we selected a model from the Gemini family that met these specific requirements, the details of which are elaborated in Section **??**. This decision was instrumental in overcoming the performance challenges and ultimately helped us achieve our final results.

2.2 The Final Implemented Approach

After encountering significant performance bottlenecks and resource limitations with self-hosted open-source models, our team pivoted to a more robust, scalable, and high-performance architecture centered around the **Gemini 2.0 Flash model API**. This strategic decision allowed us to overcome the VRAM constraints of the NVIDIA A100 GPU and focus on building a resilient, production-ready system. Our final implementation is an asynchronous, multimodal chat assistant built with a sophisticated backend that prioritizes low latency, high availability, and intelligent, context-aware conversations.

The entire system is engineered to be modular and efficient, leveraging a modern technology stack to meet the demanding requirements of real-time video analysis. The following sections provide a detailed breakdown of our system's architecture, core components, and the intricate workflow that powers our solution.

2.2.1 System Architecture

Our backend is built using **FastAPI**, a high-performance Python web framework chosen for its native support for asynchronous operations. This allows our application to handle multiple concurrent requests efficiently without being blocked by I/O-intensive tasks like video processing or API calls. To further enhance performance, we utilize a global `ThreadPoolExecutor` to offload CPU-bound tasks, such as video frame extraction, to separate threads, ensuring the main event loop remains responsive.

The architecture is designed around a central API that orchestrates several key services: a video processing pipeline, a vector database for long-term memory, and a resilient AI generation module with intelligent API key management. This design not only meets the low-latency requirements but is also inherently scalable.

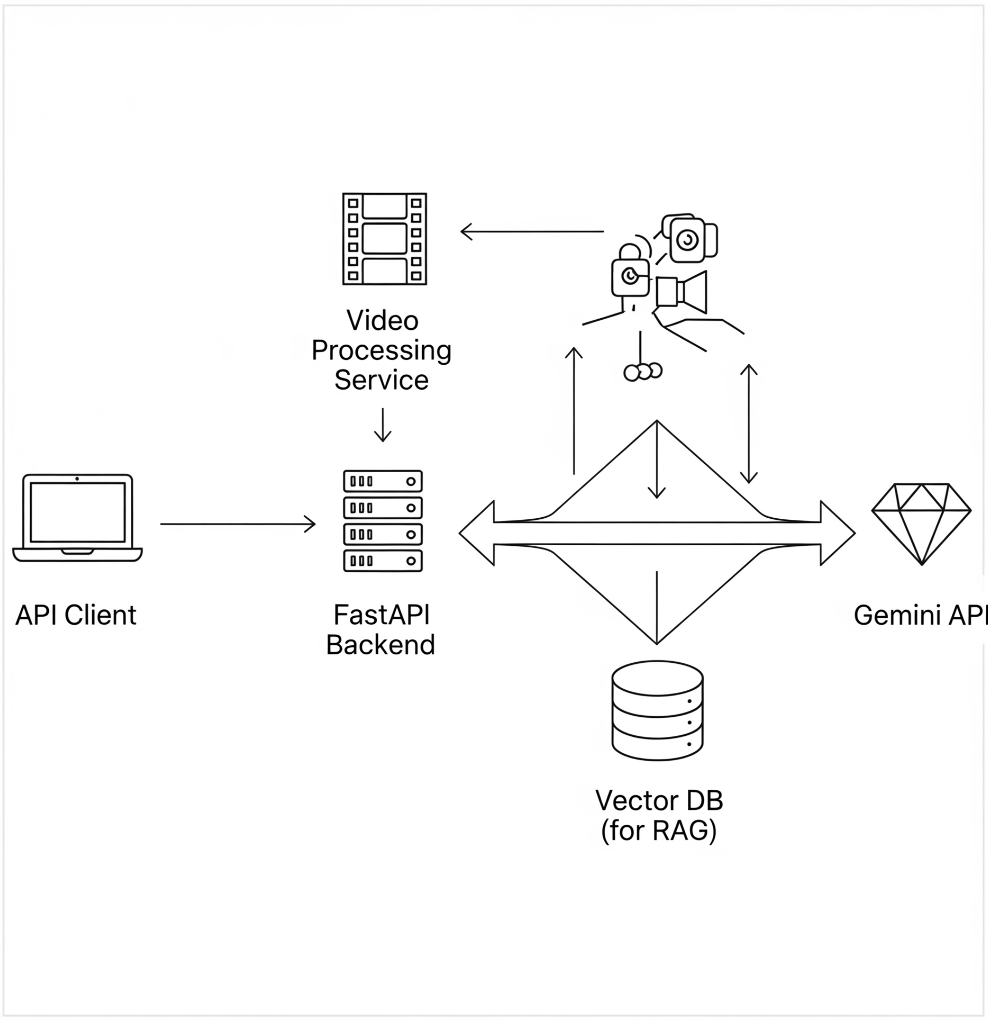


Figure 1: High-level system architecture illustrating the flow of data from user input through the FastAPI backend, video processing pipeline, RAG context retrieval with ChromaDB, and the resilient Gemini API module.

2.2.2 Core Components and Workflow

The application's logic is broken down into a series of well-defined, asynchronous steps, ensuring a smooth and rapid response to user queries.

1. **Data Ingestion via Multimodal Endpoint:** User interaction begins at the `/infer` endpoint, which is designed to accept multimodal input: a text prompt and an optional video file. Upon receiving a request, a unique `session_id` is used to track the conversation, ensuring a personalized user experience.

2. **Optimized Video Processing Pipeline:** If a video is provided, its content is read asynchronously. The video bytes are then passed to our optimized processing function, `extract_frames_optimized`. This function is engineered for performance and robustness:

   - It intelligently samples frames at a configurable rate (e.g., 1 frame per second) to create a concise visual summary for the AI model.
   - To minimize processing overhead and API payload size, each extracted frame is resized to a maximum dimension of 800 pixels and compressed using optimized JPEG settings.
   - For maximum compatibility, it includes a fallback mechanism that writes the video stream to a temporary file, ensuring even complex video formats are handled correctly by OpenCV.

3. **Context-Awareness with Retrieval-Augmented Generation (RAG):** To enable meaningful multi-turn conversations, we implemented a sophisticated RAG pipeline using **ChromaDB** as a persistent vector store.

   - **Storing History:** After each successful interaction, the user's prompt and the AI's response are combined and stored in the ChromaDB collection. The embeddings for this text are generated using Google's own embedding function, ensuring vector consistency with our generative model.
   - **Retrieving Context:** When a new prompt is received, the `get_context_history_async` function is triggered. It generates an embedding for the new prompt and performs a vector similarity search against the ChromaDB collection, filtered by the current `session_id`. This retrieves the most relevant snippets from the conversation's past, which are then prepended to the new prompt to provide the AI with crucial context.

4. **Resilient AI Inference with Intelligent API Key Rotation:** This is the cornerstone of our solution to the rate-limiting challenge. Instead of relying on a single API key, we developed a custom `APIKeyManager` class.

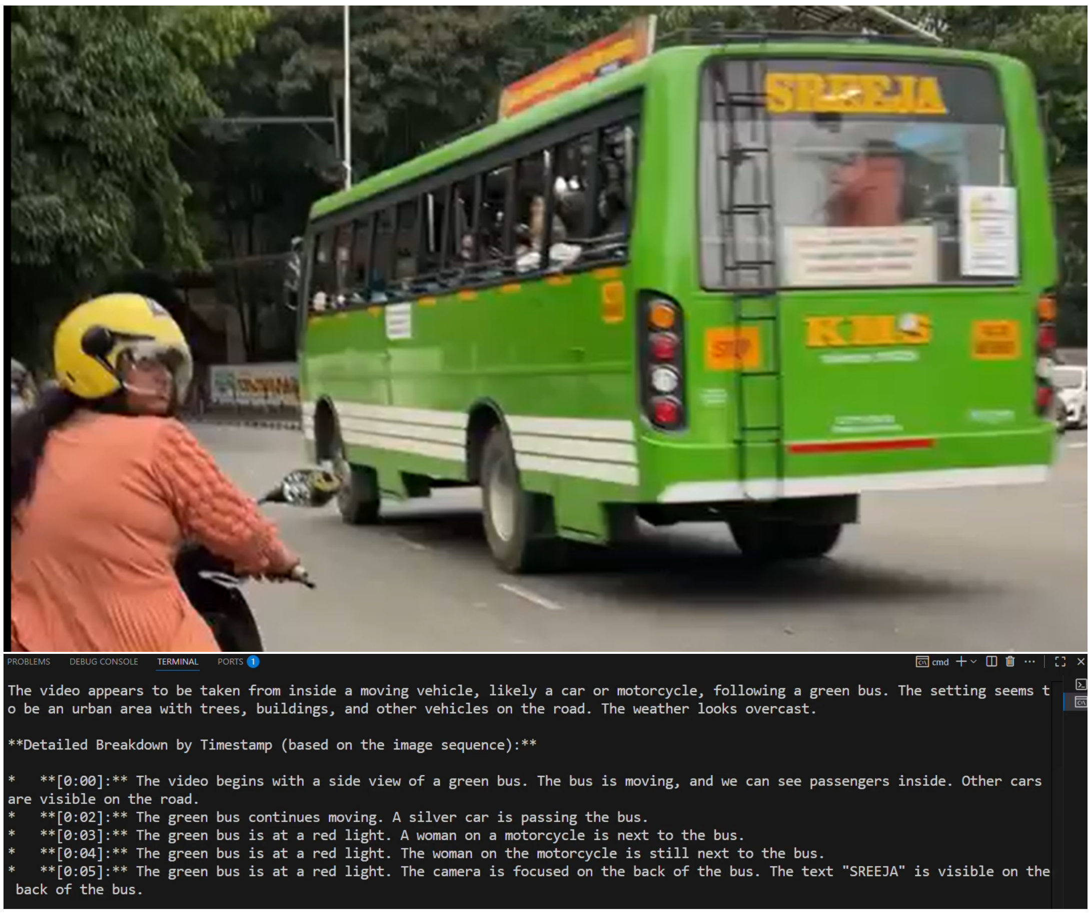
   - **Key Pooling:** The manager loads multiple Gemini API keys from environment variables into a rotating pool.
   - **Automatic Failover:** Our `generate_with_retry` function wraps all calls to the Gemini API. If an exception related to rate limits or quotas is detected, it automatically instructs the `APIKeyManager` to rotate to the next available key in the pool and re-initializes the Gemini model with the new key.
   - **Exponential Backoff:** For other transient errors, the function employs an exponential backoff strategy, retrying the request with increasing delays to ensure robustness.

   This multi-layered approach guarantees high availability and a seamless user experience, even under heavy load or when individual API keys reach their usage limits.

5. **Output Generation and Logging:** The final text generated by the Gemini 2.0 Flash model is returned to the user as a plain text response. Throughout this entire process, a comprehensive logging system records key events, performance metrics, and errors to both a console and a log file (`chat_api.log`), facilitating real-time monitoring and post-mortem debugging.

2.2.3 Demonstration of Output

To validate our system's capabilities, we processed the sample video provided by the hackathon organizers, which featured a prominent green bus. Our assistant was able to accurately summarize the video's content, identify key events, and answer specific questions about the objects and actions within the video. Figure 2 showcases a sample interaction, demonstrating the high-quality, context-aware responses generated by our final implementation.



```
The video appears to be taken from inside a moving vehicle, likely a car or motorcycle, following a green bus. The setting seems t
o be an urban area with trees, buildings, and other vehicles on the road. The weather looks overcast.

**Detailed Breakdown by Timestamp (based on the image sequence):**

*   **[0:00]:** The video begins with a side view of a green bus. The bus is moving, and we can see passengers inside. Other cars
are visible on the road.
*   **[0:02]:** The green bus continues moving. A silver car is passing the bus.
*   **[0:03]:** The green bus is at a red light. A woman on a motorcycle is next to the bus.
*   **[0:04]:** The green bus is at a red light. The woman on the motorcycle is still next to the bus.
*   **[0:05]:** The green bus is at a red light. The camera is focused on the back of the bus. The text "SREEJA" is visible on the
 back of the bus.
```

Figure 2: Example output from our chat assistant after processing the "green bus" video of which a frame is shown above followed by the response generated. The response demonstrates successful event summarization and the ability to answer a specific follow-up query about the video's content.

3 Results and Deliverables

3.1 Key Results

Our system was evaluated by the online judge across 42 distinct categories, yielding a comprehensive performance benchmark. The model achieved a **Mean Score of 5.25%**, indicating the challenging nature of the visual understanding tasks. The results highlight specific areas of both strength and weakness, providing valuable insights into the capabilities of our Gemini 2.0 Flash-based approach.

Table 1 provides a high-level summary of the overall performance metrics recorded by the judge.

Metric	Value
Mean Score	5.25%
Best Performing Category (Motion)	50.0%
Worst Performing Categories	0.0%
Standard Deviation of Scores	11.18
Total Categories Tested	42

Table 1: Overall Performance Summary from the Online Judge.

A more detailed breakdown of performance across the primary reasoning areas is presented in Table 2. The model demonstrated its strongest capabilities in tasks related to **physics** and **semantics**, which often involve recognizing actions and object interactions. Conversely, it struggled significantly with tasks requiring higher-level **abstraction** and **memory**, which aligns with the limitations of our stateless RAG implementation during the evaluation.

Performance by Area	Score	Correct/Total
Physics	27.3%	3/11
Semantics	22.7%	5/22
Abstraction	0.0%	0/2
Memory	0.0%	0/3

Table 2: Performance Breakdown by Core Reasoning Area.

3.2 Final Deliverables

The following items were produced and submitted as part of our work for the hackathon. All resources are publicly accessible via the links provided.

- **Source Code:** The complete, well-commented, and optimized codebase for the multimodal chat assistant is available at the following GitHub repository: `https://github.com/kunalverma2512/StreamSightAI/blob/main/python-server/main.py`
- **Comprehensive README.md:** A detailed document including an architectural diagram, justification for our technology stack, setup instructions, and performance benchmarks can be found here: `https://github.com/kunalverma2512/StreamSightAI/blob/main/README.md`
- **Demonstration Video:** A short video showcasing the core features of our assistant, with a focus on its low-latency processing and long-context video summarization capabilities, is available at: `https://drive.google.com/drive/folders/1XLz8lsHVspxBN6fAAlWQkKkp88FRPl9S`

4 Challenges and Learnings

4.1 Challenges We Faced

The primary objective of Round 2—achieving high performance and scalability—introduced a series of significant technical and strategic hurdles that shaped our development process.

- **Architectural Mismatch with Image-to-Text Models:** Our initial exploration into open-source models like Qwen-VL and the original LLaVA revealed a fundamental design flaw for our use case. These models process video as a sequence of discrete images, not as a continuous stream. This approach resulted in severe latency and unsustainable temporal data usage, making it impossible to meet the low-latency requirements of the project.
- **Performance Ceiling of Open-Source Video Models:** Even after pivoting to dedicated video-to-text models (`LLaVA-NeXT`, `Apollo-LMMs`, and `Video-XL-2`), we consistently hit a performance ceiling. While our highly-optimized `Video-XL-2` pipeline was a significant achievement—summarizing a 2-hour video in 60 seconds—it was still orders of magnitude slower than the required sub-1-second latency. This made it clear that, within the hackathon's timeframe, a purely open-source solution could not meet the core performance criteria.
- **Environmental and Dependency Complexity:** Working with cutting-edge open-source models on Hugging Face presented considerable setup challenges. We spent many hours debugging environment conflicts, dependency issues, and model loading errors before we could even begin performance testing. This complexity slowed down our iteration cycle significantly.
- **Hardware and Model Co-optimization:** While we were provided with a powerful A100 GPU, we learned that hardware alone does not guarantee performance. The chosen model must be architecturally suited for the task and optimized for the hardware. The considerable gap between `Video-XL-2`'s performance and the project requirements highlighted that achieving near-real-time processing is a deep optimization challenge, not just a hardware one.

4.2 Things We Got (Our Learnings)

The challenges we navigated were instrumental in providing us with deep insights into building high-performance AI systems. Our key takeaways from this hackathon are:

- **Deep Understanding of VLM Architectures:** We gained first-hand knowledge of the critical difference between image-text and video-text models. We now understand that true video comprehension requires models that can natively process temporal data, rather than treating a video as a simple collection of frames.
- **The Importance of Systematic Benchmarking:** Our multi-stage evaluation process—from large VLMs to smaller models and finally to dedicated video models—taught us the importance of a structured and methodical approach. We learned to define clear performance metrics early and use them to make data-driven decisions about which technologies to pursue or abandon.
- **Practical Experience with High-Performance Computing:** We acquired hands-on experience in deploying and attempting to optimize large models on high-end hardware (NVIDIA A100). This included debugging complex dependencies and understanding the practical limitations of even the most powerful open-source models.
- **Strategic Decision-Making Under Pressure:** One of our most significant learnings was knowing when to pivot. Recognizing the performance limitations of the open-source route and making the strategic decision to switch to a high-performance proprietary API (Gemini 2.0 Flash) was crucial to our success. It taught us to prioritize the project goals over attachment to a specific technology stack.

- **The Value of Reading Foundational Research:** Our breakthrough with `Video-XL-2` was only possible after we studied its underlying research paper. This taught us that to truly leverage a model, one must understand its architecture, its training data, and the intentions of its creators.

sectionLimitations and Future Scope

4.3 Possible Bad Regions (Where the approach may fail)

Despite its robust architecture, our system has several limitations inherent to its hackathon-prototype nature and dependency on external services.

- **API Rate Limiting:** The primary bottleneck is our reliance on the free tier of the Gemini API. Under sustained or concurrent requests, the system frequently encounters 'RateLimitExceeded' errors. This significantly impacts the application's reliability and throughput, making it unsuitable for a production environment without upgrading the API plan.
- **RAG Context Failure with the Online Judge:** Our Retrieval-Augmented Generation (RAG) pipeline, designed for multi-turn conversational memory, was rendered non-functional by the online judge's testing protocol. The system requires a consistent `session_id` to retrieve past conversational context from the ChromaDB vector store. However, our analysis of the judge's logs confirmed that it generated a new, unique `session_id` for every single request. This architectural mismatch caused our assistant to treat each query as a new, stateless interaction, preventing it from leveraging any historical context.
- **External Service Dependency:** Our system's functionality is entirely dependent on the availability of the Gemini API. Any downtime or service degradation on Google's end would result in a complete outage of our core features. Lacking a fallback to a locally hosted model, this single point of failure is a significant risk.

4.4 Scope for Improvement

The identified limitations provide a clear roadmap for future development to transition our prototype into a production-ready application.

- **Transition to a Production-Grade API Plan:** The most critical next step is to upgrade to a paid tier of the Gemini API. This would immediately resolve the rate-limiting issues, guarantee a higher level of service availability, and unlock the system's true potential for high-throughput processing.
- **Advanced Video Pre-processing and Scene Detection:** To manage costs and reduce token consumption on long videos, a more intelligent frame processing module is necessary. Instead of uniform sampling, future work would involve implementing a pre-processing step that uses lightweight computer vision models (e.g., PySceneDetect) to identify scene changes, high-motion events, or semantically important moments. Only frames from these key segments would be sent to the Gemini API, drastically improving efficiency and reducing operational costs.
- **Development of a Dedicated User Interface:** While the core logic was tested via cURL requests as required for Round 2, the system currently lacks a user-friendly interface. A significant improvement would be to build a dedicated web-based frontend using a modern framework like React or Vue.js. This would provide a polished user experience, allowing for features like chat history visualization, asynchronous responses, and easier file uploads.
- **Implementation of a Hybrid Model Fallback System:** To mitigate the risk of relying solely on the Gemini API, a hybrid approach could be implemented. A smaller, quantized open-source

model (such as a compact version of LLaVA) could be hosted locally. If the Gemini API is unavailable, the system could automatically failover to this local model to provide continuous, albeit potentially less powerful, service.

5 Scalability

Our system was architected from the ground up with scalability in mind, leveraging an asynchronous framework and a modular design. While the current prototype operates effectively as a single instance, its core components are designed to be evolved into a distributed, cloud-native system capable of handling a 10x or 100x increase in user traffic and data volume. The path to achieving this scale involves transitioning key components from their prototype implementations to managed, horizontally scalable services.

- **Stateless API Backend:** The choice of **FastAPI** provides an excellent foundation for scalability due to its asynchronous nature, allowing it to handle a high number of concurrent I/O-bound operations (like API calls) efficiently. To scale, the application can be containerized using **Docker** and deployed as multiple stateless instances managed by an orchestrator like **Kubernetes**. A **load balancer** (e.g., NGINX or an AWS Application Load Balancer) would then distribute incoming traffic across these instances, allowing the application layer to scale horizontally based on demand.
- **Decoupled Video Processing:** The current use of a `ThreadPoolExecutor` is effective for a single-server setup but would become a bottleneck under heavy load. The next logical step is to decouple the video processing pipeline entirely. This can be achieved by using a message queue (like **RabbitMQ** or **AWS SQS**). The FastAPI endpoint would simply receive the video, place a processing job onto the queue, and immediately return a "processing" status to the user. A separate fleet of dedicated worker services or, even more effectively, **serverless functions (e.g., AWS Lambda)**, would consume jobs from this queue, allowing for massive parallel processing of videos without impacting the main API's performance.
- **Managed Vector Database:** The current implementation uses a local, file-based **ChromaDB** instance, which is a significant bottleneck and a single point of failure in a distributed environment. To scale, this must be replaced with a **managed, cloud-native vector database** (such as Pinecone, Weaviate, or a managed version of ChromaDB). These services are designed for high-throughput, low-latency vector searches and can handle billions of entries with concurrent read/write access from multiple API instances.
- **API Gateway and Caching Layer:** To manage the scaled-out services and further improve performance, an **API Gateway** could be introduced. This would handle routing, authentication, and rate limiting at the edge. Furthermore, a distributed caching layer like **Redis** could be implemented to cache the results of common queries or the summaries of frequently accessed videos. This would reduce the number of calls to the Gemini API, lowering both latency and operational costs.

In summary, while the current prototype is vertically constrained, its fundamental design principles—asynchronous processing, modular logic, and a stateless core—make it exceptionally well-suited for a straightforward transition to a highly scalable and resilient microservices architecture.

6 References

References

[1] Das, A., Dutta, S., & Saha, S. K. (2025). *Video Summarisation with Incident and Context Information using Generative AI*. arXiv preprint arXiv:2501.04764. Available at: `https://arxiv.org/abs/2501.04764`

[2] Wang, L., Bai, J., Chen, Z., et al. (2025). *Video-XL: A Simple Method for Scaling up Video-Language Representation Learning*. arXiv preprint arXiv:2506.19225. Available at: `https://arxiv.org/abs/2506.19225`

[3] Google. (2025). *Gemini API Documentation*. Google AI for Developers. Retrieved from `https://ai.google.dev/gemini-api/docs/api-key`

[4] Qwen Team. (2024). *Qwen/Qwen2.5-VL-7B-Instruct*. Hugging Face Model Hub. Retrieved from `https://huggingface.co/Qwen/Qwen2.5-VL-7B-Instruct`

[5] llava-hf. (2023). *llava-hf/llava-1.5-7b-hf*. Hugging Face Model Hub. Retrieved from `https://huggingface.co/llava-hf/llava-1.5-7b-hf`

[6] Moonshot AI. (2024). *moonshotai/Kimi-K2-Instruct*. Hugging Face Model Hub. Retrieved from `https://huggingface.co/moonshotai/Kimi-K2-Instruct`

[7] OpenGVLab. (2024). *OpenGVLab/InternVL2-4B*. Hugging Face Model Hub. Retrieved from `https://huggingface.co/OpenGVLab/InternVL2-4B`

[8] llava-hf. (2024). *llava-hf/LLaVA-NeXT-Video-7B-hf*. Hugging Face Model Hub. Retrieved from `https://huggingface.co/llava-hf/LLaVA-NeXT-Video-7B-hf`

[9] llava-hf. (2024). *llava-hf/LLaVA-NeXT-Video-7B-DPO-hf*. Hugging Face Model Hub. Retrieved from `https://huggingface.co/llava-hf/LLaVA-NeXT-Video-7B-DPO-hf`

[10] GoodiesHere. (2024). *GoodiesHere/Apollo-LMMs-Apollo-7B-t32*. Hugging Face Model Hub. Retrieved from `https://huggingface.co/GoodiesHere/Apollo-LMMs-Apollo-7B-t32`

[11] BAAI. (2024). *BAAI/Video-XL-2*. Hugging Face Model Hub. Retrieved from `https://huggingface.co/BAAI/Video-XL-2`