Video Summarization for Soccer Match

Review Report - 7

Uchit N M - PES1UG22CS661

Arya Gupta - PES1UG22CS111

Faculty Mentors:

Prof. Prasad H B prasadhb@pes.edu

Dr. Gowri Srinivasa gsrinivasa@pes.edu

Prof. Preet Kanwal preetkanwal@pes.edu

Table of Contents

List of Tables

List of Figures

Background:

This report documents the project progress made during the week of June 9th to June 13th, 2025. This period of work directly follows and addresses the outcomes and feedback received during our project review meeting with the Ittiam team on Friday, June 6th, 2025.

During the June 6th meeting, the PESU team presented the progress achieved to date, including demonstrations of initial summarized video results. The Ittiam team provided valuable feedback and defined key action items for the project.

The primary focus of our work since that meeting, as detailed in this report, has been to address these specific action items and refine the system based on Ittiam's requirements. The key areas of development and analysis covered are:

- Implementing a strict duration constraint for the final summarized video output.
- Evaluating system performance and transcription quality on video sources other than YouTube.
- Assess the challenge in building an End-to-End System.
- Preparing for broader testing across a diverse range of match videos.
- Estimation for Future Development using Open Source LLMs.

The subsequent sections of this report detail the work undertaken in these areas:

Section 1: Comprehensive Analysis of LLM Performance on a Constrained Video Duration focuses on addressing Ittiam's feedback regarding the video duration limit. This section details experiments with different Gemini Versions using sophisticated, multi-layered prompts designed to enforce the target 12-13 minute output length. This section examines the challenge of selecting and prioritizing video segments using AI, comparing the performance of the web-based Gemini 2.5 Pro against the earlier Gemini 2.0 (flash), with respect to time constraints and the maintenance of an 8-second contextual window. The results show that although Gemini 2.5 Pro consistently delivered the video summary within the 12-13 minute range, Gemini 2.0 Flash result fluctuates between the ranges 9-10 minutes or 14-15 minutes, but the output is never in the desired range i.e. 12-13 minutes. When compelled to provide the results within 12-13 minutes, Gemini 2.0 Flash retains

the exact event timestamp, but does not include the context(i.e. the build up to the event and post event frames).

Section 2: Comparison between Youtube Transcriber and Whisper directly addresses the feedback on non-YouTube video sources and transcription accuracy. This section presents a comparative analysis of transcription outputs from YouTube Transcriber And the Whisper framework to evaluate the semantic and structural similarities and differences between their results using standard metrics like (SBERT , TF-IDF etc.). The Comparison shows an SBERT, showing semantic similarity with values ranging from 0.8519 to 0.8990 and TF-IDF: indicating overlap in important terms, with values ranging from 0.7963 to 0.9900.

Section 3: End-to-End System Integration focuses on building a complete video summarization pipeline but more importantly, on understanding the challenges involved in connecting its different parts. The goal was to assess the real-world issues that come up when stitching together audio transcription, AI-based event detection, and video processing.

Section 4: In this section, we created a diverse test set to evaluate the end-to-end pipeline. It features a combination of our previous full-length international matches and a newly curated collection of club-level (FC) videos, spanning a range of years and teams.

Section 5: This section explores our attempt to replicate the output quality of Gemini using an open-source, locally hosted LLM. We walk through the architecture and design choices made—driven by the need to balance reasoning capability with resource constraints. While the pipeline was successfully implemented and technically functional, the section also highlights the reasoning and performance gaps that ultimately limited its effectiveness for constraint-based summarization tasks.

Section 1 : Comprehensive Analysis of LLM Performance on a Constrained Video Duration

The initial process for generating event timestamps was driven by a straightforward extraction prompt. Its sole function was to identify all occurrences of a predefined set of events within a transcript, with the total duration of the resulting summary being an emergent property, not a controlled variable.

The original prompt was structured as follows:

```
Extract all instances of the following events from the provided text, providing output *only* in the specified format. Do not include any introductory or explanatory
text.

**Events to Extract:**

*   **Goal:** When a goal is scored.
*   **Foul:** When a foul is committed.
*   **Replacement:** When a player substitution occurs.
*   **Missed Goal:** When a clear scoring opportunity is missed.
*   **Prologue:** Beginning of the match coverage.
*   **Epilogue:** End of the match coverage.

**Output Format:**

For each event, provide a single line in the following format:

`[start timestamp] - [end timestamp] - [team name] - [type] - [small descriptions]`

Where:

*   `start timestamp`: Timestamp 8 seconds *before* the event, or the beginning of meaningful build-up play if earlier.
*   `end timestamp`: Timestamp extending until all related context concludes (celebrations, VAR reviews, arguments, etc.).
*   `team name`: The team associated with the event (e.g., "Portugal", "Spain").
*   `type`: One of: "goal", "foul", "replacement", "missed goal", "prologue", "epilogue".
*   `small descriptions`: A brief, descriptive phrase (e.g., "Penalty Goal by Ronaldo", "Foul on Guesh").
```

Fig 1 : Original Prompt with contextual reference of 8 sec

This prompt, specified to cover the major event listed, instructs the model(API) to extract those events with a 8 second build up of the beginning of the event.

Following a key requirement from the June 6th Ittiam meeting, a significant change was mandated: the final summarized video output must be capped at approximately 12-13 minutes (720-780 seconds). This length better aligns with the style and conciseness of official broadcast highlights and required shifting the objective from simple event extraction to a more complex, curated selection process.

This introduced a critical analytical challenge: how would a Large Language Model (LLM) handle the inherent ambiguity between two conflicting instructions?

- The Comprehensive Rule: The existing instruction to find all instances of specified events and apply an 8-second context pre-roll to each one.
- The Selective Constraint: The new, overriding instruction to ensure the total duration of all selected clips fits within the strict time limit, implying some events must be omitted.

The central question for this analysis was whether the model (Gemini) would act as a literal extractor, blindly applying the 8-second rule to all events and violating the time cap, or as an intelligent editor, correctly interpreting the duration constraint as a higher-order instruction and selectively discarding lower-priority events to meet the target.

1.1 Objective and Context

The primary requirement was to cap the final summarized video output at **12-13 minutes (720-780 seconds)**. Meeting this goal required the model perform a complex editorial process identifying events, calculating their total duration, and selectively choosing/adjusting segments based on priority and the 8-second context constraint to fit the time Constraint.

This necessitated developing sophisticated, multi-layered prompts beyond simple event extraction enforcing the time constraint. This section details our analysis of different Gemini Version (2.0 Flash and 2.5 Pro) performance using these refined prompts, evaluating their ability to consistently achieve the desired duration while maintaining contextual requirements and event priorities, and presents the methodology and experimental results.

1.2 Methodology: Engineering a Multi-Layered Prompt

The original, simple event-extraction prompt was replaced with a sophisticated version containing three explicit, critical instructions:

Instruction Type	Detailed Command in Prompt
Duration Constraint	"The total combined duration of ALL extracted events must be **strictly between 12-13 minutes (720-780 seconds)**"
Prioritization Logic	If the total duration exceeds the limit, the model was instructed to prioritize events in the following order: 1. Goals (full context) 2. Prologue & Epilogue (full context) 3. Missed Goals 4. Fouls 5. Replacements
Self-Verification	"Before finalizing output, calculate total duration to ensure it falls within 720-780 seconds. Adjust event boundaries if necessary while adhering to the 8 sec pre set contextual checking."

Table 1 : New Prompt Analysis

1.3 Experimental Results & Detailed Model Analysis

The new prompt was tested across different Google Gemini models. The results revealed a significant difference in how each model interpreted and executed the complex instructions, behaving either as a literal Event Extractor or a sophisticated Automated Editor.

This model consistently prioritized comprehensive event identification over adherence to the duration constraint. It behaved as a diligent, albeit literal, event logger, excelling at finding events but not at culling them to fit a time budget. But covered all the relevant contextual aspects of the event.

Run no.	Output Total Duration	Outcome	Descriptive Analysis
Run 1	9m 33s (573s)	**Undershot**	Generated a lean collection of 15 high-priority events but made no attempt to "pad" the timeline to meet the 12-minute minimum. It delivered a valid but short highlight reel. Which was more precise to the event point.
Run 2	14m 03s (843s)	**Overshot**	Produced an exhaustive list of 18 events with generously long timestamps. It interpreted "full context" literally but the trimming logic to stay within the time limit was not within the constraint.
Run 3	15m 36s (576s)	**Overshot**	This was the most comprehensive extraction, identifying 41 distinct events. This demonstrates powerful extraction but a complete disregard for the duration constraint and prioritization hierarchy.

Table 2 : Result with Gemini 2.0 Flash with target time bound of 12-13 minutes

In contrast, the experiments with the more latest **gemini-2.5-pro (Web Based)** model successfully internalized the entire set of instructions, demonstrating a fundamental understanding of the end-to-end editorial task because of its reiteration and thinking capabilities aiding in cross calculation and verification of the total time bound of 12 to 13 minutes.

Run no	Output Total Duration	Outcome	Descriptive Analysis
Run 4 (Web UI)	12m 26s (746s)	**Success**	The model curated a list of 16 events, proving it understood its time budget. It included all high-priority events (goals) and then **selectively added events from lower tiers** (one foul, one missed goal) until the target duration was met.
Run 5 (Web UI)	12m 23s (743s)	**Success**	This was the most refined output, containing only 8 clips. The list was **focused on the core narrative**: the prologue, all six goals, and the epilogue. By making the stark editorial choice to omit all fouls and replacements, the model created the tightest possible high-impact summary while perfectly meeting the duration constraint.

Table 3 : Results with Gemini 2.5 Pro (Web Based) with with target time bound of 12-13 minutes

1.4 Conclusion

The experiment clearly demonstrates that model capability is the decisive factor in executing complex, multi-layered instructions.

Gemini 2.0 Flash acts as an Event Extractor extracting the timestamp of the event resulting in fluctuating between the ranges 9-10 minutes or 14-15 minutes, but the output is never in the desired range i.e. 12-13 minutes . When compelled to provide the results within 12-13 minutes, it retains the exact event timestamp, but does not include the context which includes the build up to the event and post event frames.

Whereas, with Gemini 2.5 Pro. It successfully integrates all prompt layers duration, prioritization, and context to construct a narrative package that fits the specific runtime. It makes calculated editorial decisions, delivering a final, ready-to-use event list that meets all creative and technical requirements in a single step, Consistently delivered the video summary within the 12-13 minute range

Links to Output Loggins: API_calls; Web_calls

Section 2 : Comparison between Youtube Transcriber and Whisper

This section compares automatic transcripts generated by YouTube transcriber and the Whisper model. Both tools convert speech from video content into text, and understanding how similar or different their outputs are is important for applications that depend on accurate transcriptions.

To evaluate their similarity, we used several standard metrics: **SBERT, TF-IDF, ROUGE-1 F1, ROUGE-2 F1, ROUGE-L F1,** and **BLEU**. These scores were calculated across transcripts from five full-length matches.

Match	SBERT	TF-IDF	ROUGE-1 F1	ROUGE-2 F1	ROUGE-L F1	BLEU
Portugal v Spain	0.8892	0.9120	0.8974	0.7891	0.8645	0.5628
France vs Croatia	0.8563	0.7963	0.6920	0.5080	0.5670	0.3445
Brazil v Germany	0.8845	0.9900	0.9114	0.8065	0.8704	0.5380
Brazil vs France	0.8519	0.9766	0.9130	0.8056	0.8691	0.5282
Belgium v Japan	0.8990	0.9556	0.8667	0.7212	0.7897	0.4939

Fig 2 : Comparison of Transcription Tools : Youtube vs Whisper

Each metric highlights a different aspect of the comparison:

- **SBERT** measures semantic similarity of given sentences of the documents. It was the highest among all metrics.
- **TF-IDF** and **ROUGE-1** F1 scores were also high. This shows a strong overlap in individual words and important terms used in both transcripts.
- On the other hand, **ROUGE-2** F1 and **BLEU** scores were lower. ROUGE-2 focuses on matching two-word sequences, and BLEU checks how precisely the sequences of words align. These lower scores suggest that while the content is similar, the sentence structure and word order often differ between the two systems.

In summary, YouTube and Whisper generally produce transcripts with similar meanings and vocabulary. Thus, they often express the same ideas using different phrasing or word arrangements.

Whisper Transcript Files: Whisper_transcripts

Section 3 : Assess the challenge in building an End-to-End System

Building an end-to-end video summarization pipeline came with notable integration challenges, particularly in ensuring smooth interoperability between three key functional blocks:

- The audio transcript generator
- The Gemini-based event timestamp extractor, and
- The video clipping and stitching component.

One of the primary challenges was the **consistent generation and structuring of intermediate outputs**, which were critical for reliably passing data between modules. Each block depended on a well-defined input format, so maintaining structural integrity across transitions was essential.

Another significant consideration was the **choice of the Whisper model** for transcription. Since transcription serves as the foundation of the pipeline, selecting a model that balances **transcription quality with performance** was crucial. This decision impacted both the accuracy of event extraction and the overall runtime of the system.

Finally, **parsing and converting Gemini's output into a structured JSON format** was a key step, enabling precise video clipping and overlaying of events based on detected timestamps.

With the current architecture, the system processes a 1 hour 45 minute full-length match video in approximately **25–35 minutes** using the **Whisper Small** model, and around **45 minutes** when using the **Whisper Large** model.

3.1 Overall System Architecture

The application is a client-server system running locally, composed of three main layers:

1. **User Interface (UI) -** : This is the user's entry point. It's a web application built with Streamlit that handles file uploads, displays processing progress, and presents the final results (videos, statistics, and interactive tools). It acts as the "client" that initiates and monitors tasks.

2. **Processing Pipeline -** : This is the core backend logic. It's a series of Python functions that perform the heavy lifting: video/audio manipulation with `ffmpeg`, transcription with `whisper`, and intelligent analysis with the `Gemini AI` API.
3. **File System (as a Database)**: The system doesn't use a formal database. Instead, it uses the local file system, specifically the `uploads/` directory, to store all data. Each processing task gets its own unique sub-folder, which holds the original video, all intermediate files, and the final outputs. This makes tasks self-contained and allows for easy caching and retrieval.

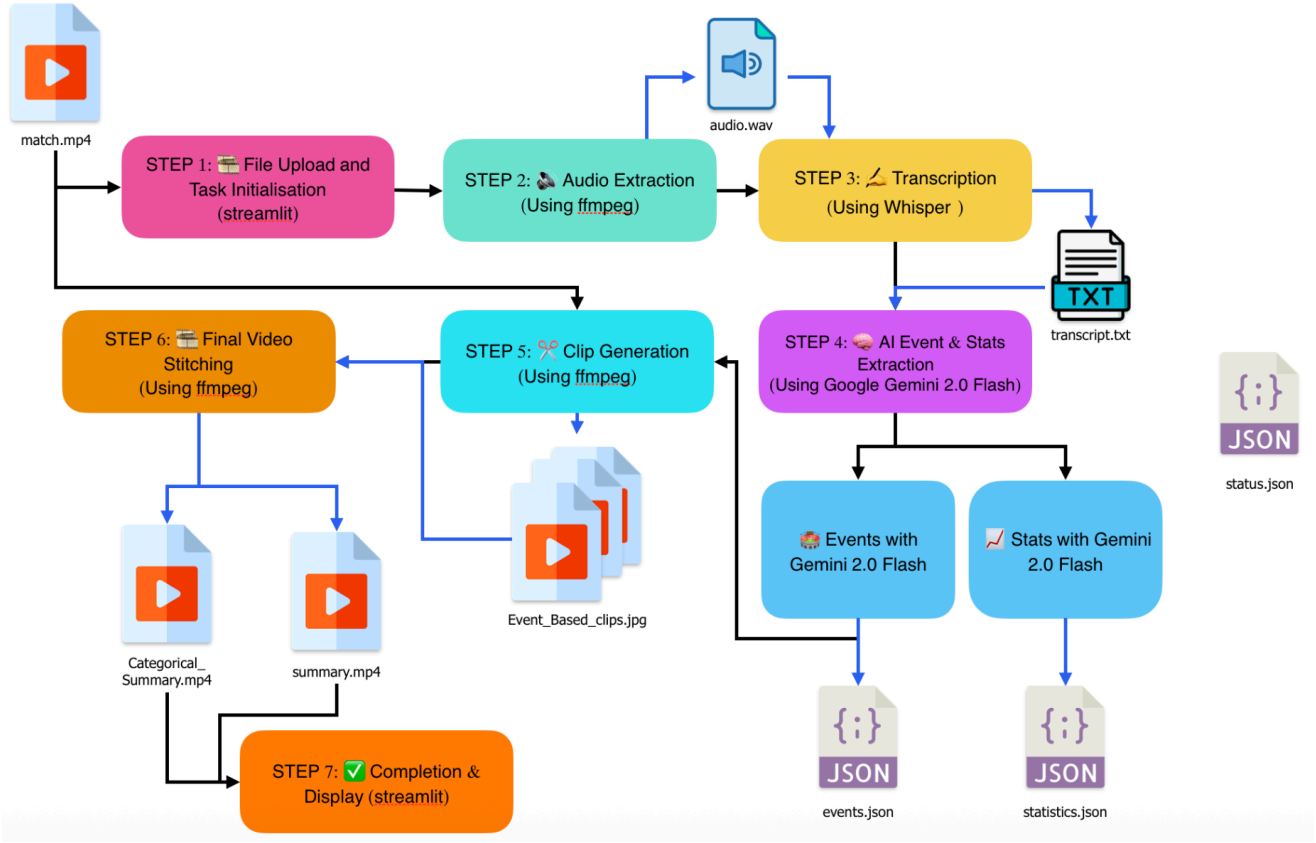

Fig 3 : Overall System Architecture

3.2 The Step-by-Step Data Flow: From Upload to Output

This section follows a single video file through its entire lifecycle in the system.

Step 1: Initiation (in `streamlit_app.py`)

1. **File Upload**: The user visits the Streamlit application and uses the `st.file_uploader` to select a video file (e.g., `match.mp4`). This file is initially held in memory by Streamlit.
2. **Generate Button Click**: The user clicks the "Generate Highlights & Stats" button.
3. **Task ID Creation**:
    - The app first writes the uploaded video to a temporary location: `uploads/match.mp4`.
    - It then calls `utils.get_file_hash()` on this file. This function reads the video file in chunks and computes its unique SHA-256 hash (e.g., `a1b2c3d4...`). This hash becomes the **`task_id`**.
    - A dedicated directory for this task is created: `uploads/a1b2c3d4.../`.
    - The video file is moved from `uploads/match.mp4` to its permanent task home: `uploads/a1b2c3d4.../match.mp4`.
4. **Cache Check**: The system checks if `uploads/a1b2c3d4.../status.json` exists and if its content indicates a "Complete" status.
    - **If YES (Cache Hit)**: The pipeline is skipped entirely. The app proceeds directly to the results display (Step 7), loading the pre-existing files.
    - **If NO (Cache Miss)**: The full pipeline processing begins.
5. **Pipeline Launch**:
    - `st.session_state.processing` is set to `True`.
    - A new background thread (`threading.Thread`) is started. Its target is the `pipeline_thread_target` function, which in turn calls the main orchestrator: `pipeline.run_full_pipeline()`.
    - The UI now enters a "Processing View," where it will periodically check the `status.json` file to update the progress bar for the user.

Step 2: Audio Extraction (in `pipeline.py`)

- **Function**: `extract_audio()`
- **Input**: The video file path (`uploads/<task_id>/match.mp4`).
- **Process**:
  - The `status_callback` updates `status.json` with `{ "status": "Extracting audio..." }`.
  - The `ffmpeg-python` library is used to read the input video.
  - It extracts the audio stream, converts it to a mono-channel (`ac=1`), 16kHz (`ar='16000'`) WAV file (`acodec='pcm_s16le'`), which is the optimal format for the Whisper model.
- **Output (Intermediate File)**: `uploads/<task_id>/audio.wav`.

Step 3: Transcription (in `pipeline.py`)

- **Function**: `transcribe_audio()`
- **Input**: The audio file path (`uploads/<task_id>/audio.wav`).
- **Process**:
  - The `status_callback` updates `status.json` with `{ "status": "Transcribing audio..." }`.
  - The `whisper.load_model("small")` function loads the pre-trained 'small' English model.
  - The model transcribes the entire `audio.wav` file. The result is a dictionary containing the full text and a list of "segments," each with a start time and text.
  - The code iterates through these segments and writes them to a text file, prepending a formatted timestamp `[hh:mm:ss]` to each line.
- **Output (Intermediate File)**: `uploads/<task_id>/transcript.txt`.

Note on Skipping: If the user provides their own transcript, Steps 2 and 3 are skipped, and the user's file is saved directly as `transcript.txt`.

Step 4: AI-Powered Event & Statistic Extraction (in `pipeline.py`)

This involves two separate, parallelizable calls to the Gemini AI.

A. Event Extraction for Highlights

- **Function**: `extract_events_with_llm()`
- **Input**: The transcript path (`uploads/<task_id>/transcript.txt`).
- **Process**:
  - The `status_callback` updates `status.json` with `{ "status": "Identifying key events with AI..." }`.
  - The entire content of `transcript.txt` is read and embedded into a large, detailed **prompt**. This prompt instructs the Gemini model to act as a sports analyst and identify specific events (Goal, Foul, Prologue, etc.). Crucially, it provides strict rules for timestamp formatting and a **total duration constraint** (12-13 minutes) to ensure the final highlight reel is a reasonable length.
  - The model's response, which is expected to be a list of events in plain text, is received.
  - A regular expression `r'\[(\d{1,2}:\d{2}:\d{2})\]\s*-\s*\[(\d{1,2}:\d{2}:\d{2})\]\s*-\s*([^-]+?)\s*-\s*([^-]+?)\s*-\s*(.+)'` is used to parse each line of the response, extracting the start time, end time, team, event type, and description.
- **Output (Intermediate File)**: `uploads/<task_id>/events.json`. This is a JSON array of event objects.

B. Statistic Generation

- **Function**: `generate_statistics_with_llm()`
- **Input**: The transcript path (`uploads/<task_id>/transcript.txt`).
- **Process**:
  - The `status_callback` updates `status.json` with `{ "status": "Generating match statistics..." }`.
  - The transcript is used in a **different prompt** sent to Gemini. This prompt asks the model to perform a statistical analysis and return the data *only* as a single, valid JSON object with a predefined structure (team stats, player events, etc.).
  - The model's text response is cleaned (removing markdown backticks) and parsed directly into a Python dictionary using `json.loads()`.
- **Output (Intermediate File)**: `uploads/<task_id>/statistics.json`. This is a single JSON object containing all the match stats.

Step 5: Clip Generation (in `pipeline.py`)

- **Function**: `create_clips_from_events()`
- **Input**: The events file (`uploads/<task_id>/events.json`) and the original video path.
- **Process**:
  - The `status_callback` updates `status.json` with `{ "status": "Creating individual highlight clips..." }`.
  - The code reads `events.json` and iterates through each event object.
  - For each event:
    1. It creates a subdirectory based on the event type (e.g., `uploads/<task_id>/clips/goal/`).
    2. It uses `ffmpeg-python` to cut a segment from the original `match.mp4` using the event's `start_timestamp` and `end_timestamp`.
    3. During this cutting process, it uses FFmpeg's `drawtext` video filter to burn a text overlay onto the clip (e.g., "GOAL: TEAM A").
    4. It also generates a small `.jpg` thumbnail for each clip by extracting the first valid frame.
- **Output (Intermediate Files)**: A collection of short video clips and their corresponding thumbnails, organized by event type.
  - `uploads/<task_id>/clips/goal/clip_1_goal.mp4`
  - `uploads/<task_id>/clips/goal/clip_1_goal.jpg`
  - `uploads/<task_id>/clips/foul/clip_2_foul.mp4`
  - `uploads/<task_id>/clips/foul/clip_2_foul.jpg`

Step 6: Stitching Final Videos (in `pipeline.py`)

- **Function**: `stitch_clips()`
- **Input**: A list of clip paths and a desired output path.
- **Process**:
  - This function is called multiple times by `run_full_pipeline`.
  - First, for the **chronological summary**:
    - The `status_callback` updates `status.json`.
    - It gets a list of *all* created clip paths, in order.
    - `ffmpeg-python`'s `concat` filter is used to join these clips (both video and audio streams) seamlessly into one video.
  - Second, for **category-based summaries**:
    - It iterates through the clips, grouping them by event type (e.g., all 'Goal' clips, all 'Foul' clips).
    - It calls `stitch_clips()` for each group, creating a separate summary video for each category.

- **Output (Final Files)**:
  - `uploads/<task_id>/summary_chronological.mp4`
  - `uploads/<task_id>/summary_goal.mp4`
  - `uploads/<task_id>/summary_missed_goal.mp4`

Step 7: Completion and Results Display (in `streamlit_app.py`)

1. **Final Status Update**: The `run_full_pipeline` function makes a final call to the `status_callback`, which writes the definitive `status.json` file: `{ "status": "Complete", "final_summary_filename": "summary_chronological.mp4", "completion_time": 754.32 }`.
2. **UI Switch**: The Streamlit app, which has been polling `status.json`, sees the "Complete" status. It stops the "Processing View" and `rerun`s.
3. **Display Results**: The app now displays the results page:
   - It reads `statistics.json` to populate the "Statistics Dashboard".
   - It reads `events.json` to populate the "Interactive Event Creator" table.
   - It uses `glob` to find all `summary_*.mp4` files in the task directory and displays them in video players with download buttons.
   - The `st.video()` component streams the final `summary_chronological.mp4` (and other summaries) for the user to watch.

3.3 Directory Structure of a Completed Task

```
uploads/
└── a1b2c3d4e5f6a1b2c3d4e5f6.../  (The Task ID folder)
    ├── match.mp4                       (Step 1: Original input video)
    ├── status.json                     (Step 1-7: Live status, then final 'Complete' state)
    ├── audio.wav                       (Step 2: Intermediate audio for Whisper)
    ├── transcript.txt                  (Step 3: Text output from Whisper)
    ├── events.json                     (Step 4: Parsed events from Gemini for highlights)
    ├── statistics.json                 (Step 4: Parsed stats from Gemini for dashboard)
    │
    ├── clips/                          (Step 5: Directory for all individual event clips)
    │   ├── goal/
    │   │   ├── clip_1_goal.mp4
    │   │   └── clip_1_goal.jpg
    │   ├── foul/
    │   │   ├── clip_5_foul.mp4
    │   │   └── clip_5_foul.jpg
    │   └── ... (other folders for replacement, missed_goal, etc.)
    │
    ├── summary_chronological.mp4       (Step 6: Final stitched chronological video)
    ├── summary_goal.mp4                (Step 6: Category-specific video)
    └── summary_foul.mp4                (Step 6: Category-specific video)
```

Fig 4 : Directory Structure of Complete Task

3.4 GUI Creation for the End-End Pipeline

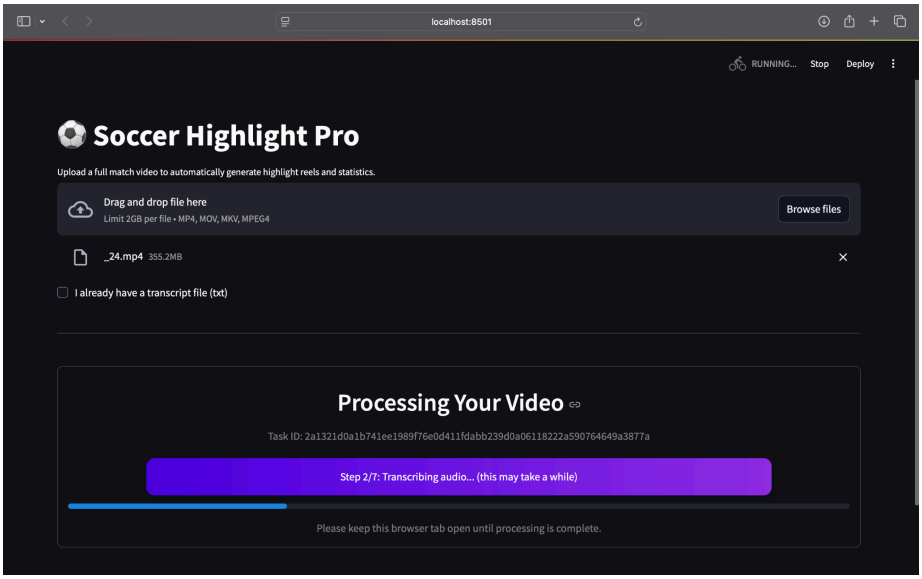

Fig 5 : Landing page for the GUI created

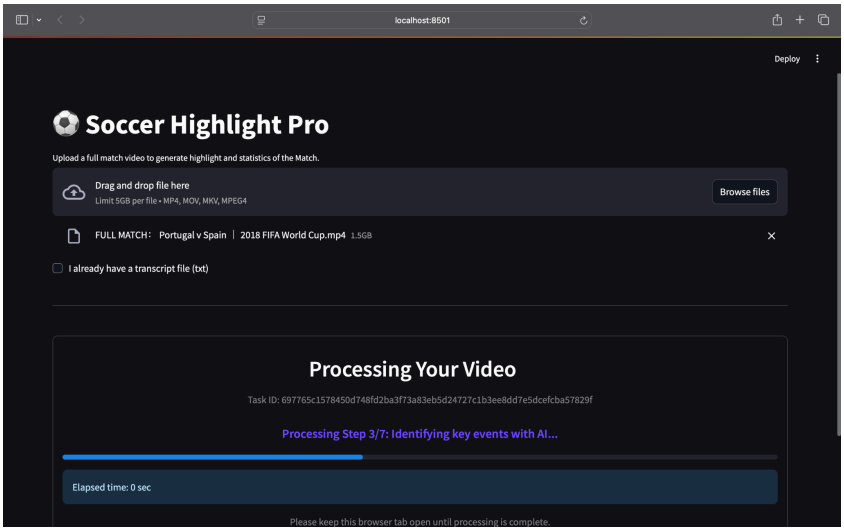

Fig 6 : Progress Dashboard for the integrated system

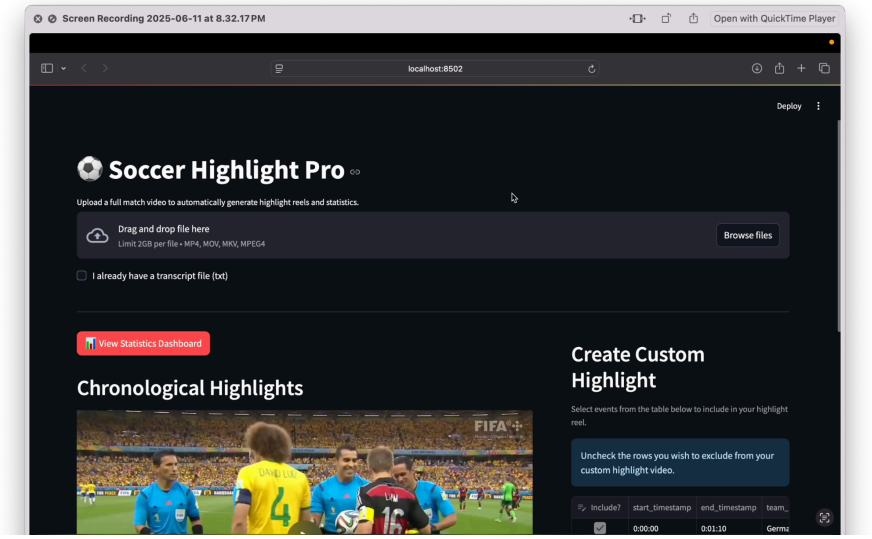

Fig 7 : Summary Dashboard shown after output generation

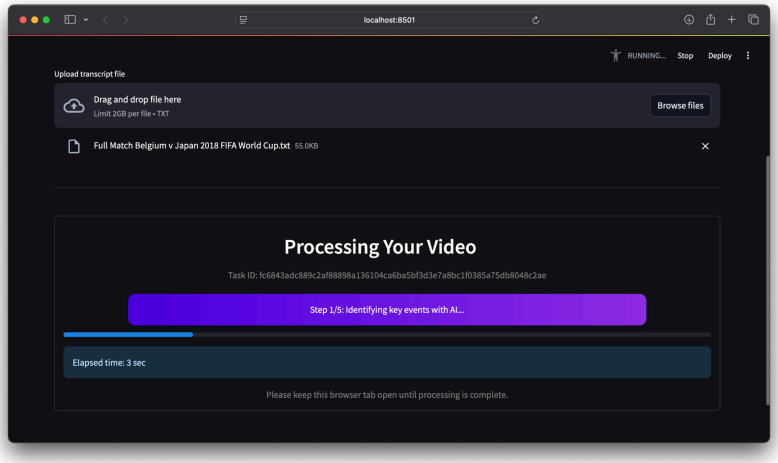

Fig 8 : Progress Dashboard if User uploads Video and Transcript

Demo Link: Pipeline_prototype_stable.mp4

Section 4 : Broader Testing

In this section, we created a diverse test set to evaluate the end-to-end pipeline. It features a combination of our previous full-length international matches and a newly curated collection of club-level (FC) videos, spanning a range of years and teams. The following table shows the comparative results between the generated and broadcasted highlights (Table 4).

Sr. No	Match	Transcript	Event.json	Generated Highlight	Broadcast Highlight	TD-IDF	SBERT Score
1	Manchester City v Liverpool	transcript.txt	events.json	summary_chronological.mp4	Man City vs Liverpool 03/01/2019- Premier League 2018/2019	0.9103	0.6681
2	Arsenal v. Manchester United	transcript.txt	events.json	summary_chronological.mp4	Arsenal vs Manchester United 3-2 Full Highlights 22/01/2023 - video Dailymotion	0.8330	0.6610
3	Belgium v Japan	transcript.txt	events.json	summary_chronological.mp4	Belgium 3-2 Japan	Extended Highlights	2018 FIFA World Cup	0.9229	0.6782
4	Brazil v Germany	transcript.txt	events.json	summary_chronological.mp4	Brazil 1-7 Germany	Extended Highlights	2014 FIFA World Cup	0.9450	0.6653
5	France v Croatia	transcript.txt	events.json	summary_chronological.mp4	France 4-2 Croatia	Extended Highlights	2018 FIFA World Cup Final	0.9184	0.8438

Table 4 : Comparative results between the generated and broadcasted highlights.

Based on the analysis, the generated highlights show a high degree of similarity to the broadcast highlights curated from different sources. The high TF-IDF scores, consistently above 0.83 and often exceeding 0.91, The SBERT Scores range from 0.66 to 0.84. This shows that the overall meaning and narrative flow are closely related, expected when describing the same sequence of events.

It is also important to note that for club matches, such as those from the Premier League, multiple highlight versions exist from different broadcasters and the clubs themselves. Unlike international tournaments with a unified broadcast feed, domestic league matches often have multiple highlight reels produced by different sources: the clubs themselves, the official league, and various broadcasters. Each of these sources brings its own contextual focus, leading to variations in their highlights. This makes a direct, one-to-one comparison challenging. Therefore, the model performance's strength lies in capturing the core contextual and narrative importance of events, rather than simply matching the editorial choices of one specific broadcast.

This variance between broadcast styles is demonstrated by an analysis of two different club-produced highlight transcripts for the “Liverpool 3-1 Man City match”. While a high Sentence similarity (SBERT) of 0.8432 confirms both describe the same core events, the significantly lower Keyword Similarity (TF-IDF) of 0.6094 their phrasing are different. Essentially, both sources tell the same content using different words, thus the model should aim to identify the objective event rather than mirroring a single subjective broadcast style.

Highlight - Liverpool FC - Liverpool 3-1 Man City | Fabinho's stunner helps Reds beat City | H...

Highlight - Man City FC - HIGHLIGHTS | Liverpool 3-1 Man City (Fabinho, Salah, Mane, Bern...

Section 5 : Experimenting with Open Source LLMs

5.1 : Background

The idea is to achieve outputs similar to Gemini using an Open Source LLM

A 70-billion parameter (Llama3:70B) model offered superior reasoning but its hardware requirements exceeded our available 24GB VRAM. Conversely, a 3-billion parameter (Llama3:3B) model lacked the necessary reasoning quality for the complex extraction task. The selection of the `llama3:8b` model was based on the balance it provides between reasoning capabilities and hardware requirements.

A direct processing approach was not feasible, as the transcript's ~21,000 tokens far exceeded the Llama 3 model's 8,192-token context limit. To solve this fundamental constraint, we implemented a Retrieval-Augmented Generation (RAG) architecture which enabled us to use the capable `llama3:8b` model, effectively balancing performance with power.

The project successfully resulted in a technically functional RAG pipeline. The system correctly ingests the transcript, builds a vector database, and retrieves relevant document chunks for the LLM. However, the output quality does not yet meet the project's requirements. The generated output is sparse, incomplete, and fails to adhere to the complex constraints laid out in the prompt.

The root cause of this quality gap is a fundamental capabilities mismatch. The prompt requires the model to perform a complex, multi-step reasoning task that includes constraint-based optimization—specifically, editing a list of events to fit a strict time budget. This level of cognitive load surpasses the reasoning capacity of the eight-billion parameter model.

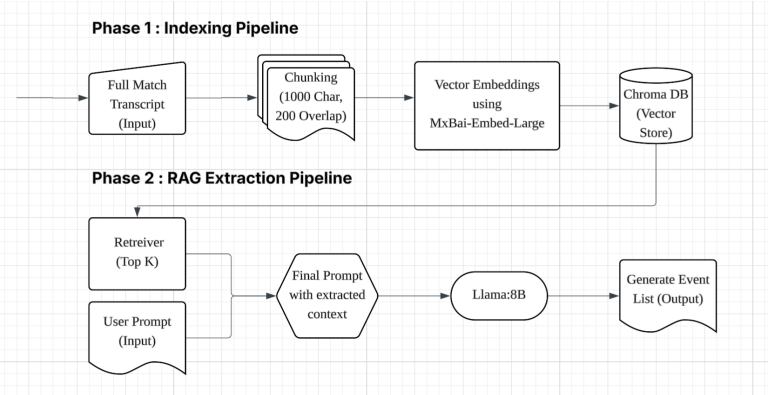

Fig 9 : System Architecture for the Naive RAG Approach

5.2 System Architecture & Data Flow

Core Components:

- **Ollama**: The server running the local LLMs (`llama3:8b`) and the embedding model (`mxbai-embed-large`).
- **LangChain**: The framework orchestrating the entire process, "chaining" together loading, splitting, embedding, retrieving, and prompting.
- **ChromaDB**: A local vector database that stores numerical representations (embeddings) of the text, enabling efficient semantic search.

The process is divided into two sequential pipelines:

5.2.1 The Indexing Pipeline (`indexing_pipeline.py`)

Its sole purpose is to convert a large text file into a searchable database.

Input: A single text file located at `data/transcript.txt`.

Step-by-Step Data Flow:

1. **Load Document**: The `TextLoader` reads the entire `transcript.txt` file into memory.
2. **Chunk Text**: The `RecursiveCharacterTextSplitter` breaks the long transcript into smaller, 1000-character chunks with a 200-character overlap. This is critical because the LLM has a limited context window and cannot process the entire transcript at once. This step creates a list of many small text documents.
3. **Embed Chunks**: The `OllamaEmbeddings` component, using the `mxbai-embed-large` model, converts each text chunk into a high-dimensional vector. This vector is a numerical representation of the chunk's semantic meaning.
4. **Store in Vector DB**: The `Chroma.from_documents` function takes all the text chunks and their corresponding embedding vectors and saves them to a new directory named `db/`. This directory now contains a searchable vector index of the entire transcript.

Output: A `db/` folder containing the ChromaDB vector store.

5.2.2 The Extraction Pipeline (`extraction_pipeline.py`)

This is the operational pipeline that is run to get the final result.

Input: The Chroma vector database in the `db/` directory.

Step-by-Step Data Flow:

1. **Initialize Components**: The script initializes the same models (`mxbai-embed-large`, `llama3:8b`) and connects to the existing `db/` vector store. A **retriever** is created to perform the search, configured to fetch the top 60 most relevant chunks (`k=60`).
2. **Retrieve Relevant Context (The "R" in RAG)**:
    - A broad, high-level query (`"The full football match transcript..."`) is created.
    - The retriever converts this query into an embedding vector and searches the database for the 60 text chunks with the most similar vectors.
    - The original text of these 60 chunks is retrieved and joined together into a single block of text called the `context`.
3. **Augment the Prompt (The "A" in RAG)**:
    - A detailed "prompt" is defined, containing strict instructions, rules, and formatting specifications.
    - The `context` retrieved in the previous step is injected into this prompt. This creates a complete, final prompt containing both the instructions and the data for the LLM to analyze.
4. **Generate the Response (The "G" in RAG)**:
    - This final, augmented prompt is sent to the local `llama3:8b` model via Ollama.
    - The LLM reads the instructions and the provided context and generates a response.
5. **Print Output**: The script prints the LLM's response directly to the console.

Expected Output: A list of formatted events printed to the terminal.

5.3 Root cause analysis for observed results

Although the pipeline functioned correctly on a technical level, the final result wasn't usable. Out of roughly 60 document chunks—about 50,000 characters of content—the model could only output three events.

The issue occurs due to the limits of the model being used (llama3:8b). The task required a series of complex reasoning steps—reading a large volume of content, finding several different kinds of events, assigning timestamps based on context, and then adjusting those timestamps to meet a strict overall time limit. The model could handle the first few steps to some degree, but it struggled when it came time to adjust and balance the timing across all events. This requires tracking multiple variables, comparing priorities, and making trade-offs—something this model isn't well-equipped to do.

The task is too complex for the model's reasoning capabilities. It couldn't juggle all the demands at once. A more advanced model with stronger reasoning abilities would be better suited for this kind of work.

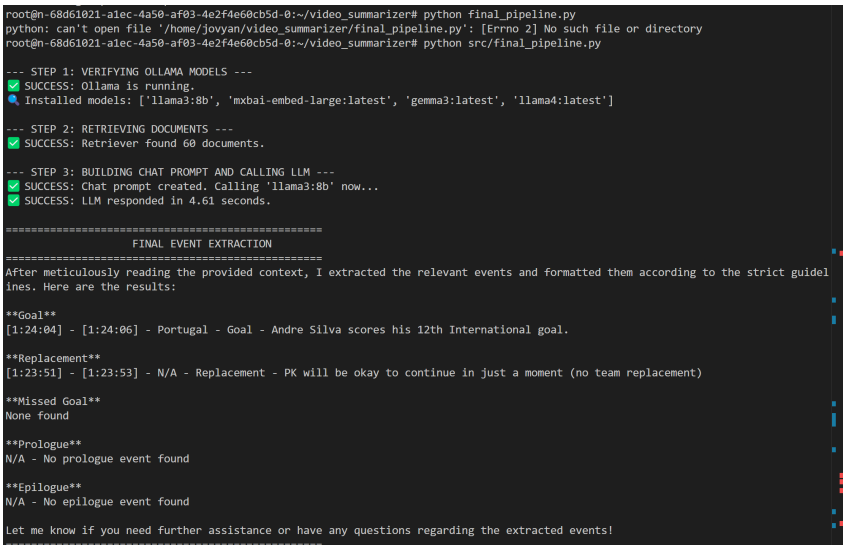

Fig 10 : Outputs with Llama3:8b RAG Approach

Conclusion and Future Work

Following the June 6th meeting with the Ittiam team, this week's work focused on implementing the feedback. Key progress was made in enforcing the 12–13 minute summarization constraint, aiming to produce consistent, context-rich summaries using Gemini. We also conducted a comparative evaluation of Whisper and YouTube Transcriber, which showed strong semantic alignment validating Whisper's suitability for non-YouTube sources.
Subsequently, we built and tested an end-to-end summarization pipeline, identifying key integration challenges across transcription, event detection, and video processing. The test set was expanded to include a broader mix of club-level and international matches. Finally, we attempted to replicate Gemini's output using a locally hosted open-source LLM, which, while technically functional, revealed limitations in reasoning and strict duration control.

In the coming weeks, we intend to focus on enhancing the performance of open-source LLMs, (LLaMA and Gemma), to better align with the summarization objectives. Following this, we plan to evaluate the effectiveness of off-the-shelf versus custom fine-tuned LLMs to determine the best fit for video summarization. In parallel, we aim to explore the integration of audio cue identification to improve event detection accuracy. We will Focus on improving generalization, Optimization and robustness of the end-to-end system across a wider variety of video sources and match contexts.