ChatGPT

Plan for Implementing a Hockey Video Clip Retriever

1. Recognizing and Routing Clip Queries

The orchestrator will first identify when a user's request involves video clips. Our **Intent Analyzer** already classifies queries for clips using keywords like *"clips," "highlights," "video," "shifts"* etc. [1]. For example, a query *"Show me my shifts from the last five games"* or *"show me all ozone clips from the last five games"* will be categorized as `QueryType.CLIP_RETRIEVAL` and flagged to use the clip retrieval tool [1]. Once the intent is recognized, the **Smart Router** will include the **ClipRetrieverNode** in the tool execution plan (alongside any other needed tools). This ensures the orchestrator routes the query to our new clip retrieval logic. We'll add a tool specification for clip retrieval (e.g. `"retrieve_clips"`) in the **tool registry** so the LLM's planning can select it when appropriate. All of this happens before any attempt to fetch data – the system is simply deciding *"this query needs video clips"* and preparing to handle it accordingly.

2. Parsing Queries into Clip Search Parameters

Once the ClipRetrieverNode is invoked, it will parse the natural language query into structured search parameters. Our implementation will leverage the helper methods we've begun in `clip_retriever.py`. The node will extract:

- **Player Names** – e.g. "my shifts" will resolve to the user's own name if the user is a player [2]. The parser scans for any Canadiens player names or nicknames in the text (we maintain a list of MTL player names/aliases) [3] [4]. It adds the matched player(s) to the search criteria. For instance, "Show me Suzuki's goals from last game" would pick up **Nick Suzuki** as the player of interest. If the query uses "my" or "me," we check the `user_context` – if the logged-in user is a player, we use their name [2].

- **Event Types** – The parser will map hockey terminology to event categories. We have a dictionary of keywords for goals, assists, saves, hits, penalties, zone entries/exits, etc. [5] [6]. For example, "dzone exits" contains *"exit"*, which we map to the event type `zone_exits`. "Ozone" (offensive zone) isn't explicitly listed, but we interpret it as offensive-zone plays – likely **zone entries or offensive sequences**. We might enhance the parser to catch "offensive zone" or slang like "ozone" as a cue to gather clips of offensive zone play (e.g. zone entries, offensive possessions). If the query uses generic terms like "highlights" or "shifts" without a specific event, our logic will treat it as a broad request and not filter to one event type [7]. In those cases, we aim to retrieve all notable clips matching the other criteria (e.g. all types of plays by that player in the timeframe).

- **Timeframe** – The parser detects phrases like "last game", "last 5 games", "this season", etc., and converts them to a `time_filter` code [8] [9]. For example, "last five games" → `last_5_games`.

This will later tell our search to restrict clips to those games. If no explicit range is given, we default to recent games or all available data (depending on context).

- **Opponent/Team Filters** – If the query mentions an opponent (e.g. "vs Toronto" or "against Boston"), we capture that as well [10] [11]. The parser looks for NHL team names or "vs X" patterns to populate an `opponents` filter. For example, *"power play clips vs Toronto"* would include opponent = Toronto in search params.

All these extracted pieces – players, event_types, time_filter, opponents – will form a `ClipSearchParams` dataclass [12]. We'll also set a limit on number of clips (default 10) to avoid overloading the response. At this stage we log the interpreted search parameters for transparency (e.g. *players = [Suzuki], events = ["zone_exits"], time = "last_5_games"* [13] [14]). This confirms we correctly understood the query.

3. Searching the Play-by-Play Data for Relevant Events

With structured search params, the ClipRetriever will next find all matching video snippets. This involves two layers of data: (a) our **play-by-play analytics dataset** (with event timestamps and details), and (b) the **video files** for those moments. We have comprehensive game event data stored in Parquet/JSON from our NHL extractor, which includes for every event: period, period time, game time, and a *timecode* timestamp [15]. The **timecode** is crucial – it represents the real-world elapsed time in the video (including stoppages) at which the event occurred. Using this data, we can pinpoint where in a video file a given event happens.

Event Lookup: Given the search parameters (player, event type, timeframe, etc.), we will query our event data to retrieve all matching events: - For example, *"all of my d-zone exits clips from last game"*: We filter the last played game's events for that player where the event category is *zone_exit*. Our **Parquet Analyzer** tool or a direct pandas query can do this efficiently since each event in the timeline has fields like player name and event labels (our extracted **tendency timeline** CSV/JSON contains entries like `name: Suzuki, event: zone_exit, periodTime: 12:34, timecode: 753.2s` etc.). We might integrate a function in ClipRetriever to query the Parquet dataset for relevant events, or reuse `query_game_data` to fetch those rows. The result is a list of events (each with game identifier, period, timestamp). If the query was *"last 5 games"*, we'd gather events across the five most recent games. Our extraction includes an easy way to filter by game dates; for instance, we can get the dates of the last N games and filter events by those dates [16] [17]. If an opponent is specified, we further filter to games against that team.

Pre-Indexed Clips vs On-Demand: We have a filesystem of pre-cut clips for common events (goals, assists, etc.) organized under `data/clips` by season, player, and game [18] [19]. The ClipRetriever will first utilize the `ClipIndexManager` to see if requested clips already exist in our library. For example, if the query is "Suzuki's goals from last game," and we have files like `suzuki_goal_10-12.mp4` in the folder for that game, the index will find those [19]. The index manager scans the `players/[player]/vs_team_date/` directories for matching filenames, and it caches metadata of all clips (clip ID, player, event type, game date, description, etc.) [20] [21]. We will leverage this to quickly retrieve known clips. The filtering function then applies the search params: matching player name, event type, opponent, and game date range [7] [22]. It also applies our `time_filter` like *last_5_games* to restrict results to recent games [23] [17].

However, many granular actions (like “zone exits” or full “shifts”) may not be pre-cut as individual clip files (since storing every zone exit for every player would be a huge number of files). For those, we’ll implement **on-the-fly clip generation**. If the initial clip index search yields nothing or not enough results for the query, our retriever will fall back to the event list from the analytics data: - We take each relevant event that doesn’t have a file and **generate a clip by cutting from the game footage**. Using the event’s period and timecode, we know which period video file to use and the exact timestamps. For example, an event in *Period 2 at timecode 350.5s* means we go to the second period’s video and seek to 5:50 into that file. We will likely maintain raw game videos split by period (e.g., `game1234_period1.mp4`, `game1234_period2.mp4`, etc.) as source material. The note *“game clips will already be filtered by periods”* implies we have each period isolated, making it easier to seek within a 20-minute segment rather than a full game.

- We will use **FFmpeg** to perform the cut. Our system already uses ffmpeg for generating thumbnails [24] [25], so we know ffmpeg is available. To cut a clip, we’ll run a command (via subprocess or a Python binding) with `-ss` (start time) and `-to` (end time or `-t` duration). For each event, we might capture a short window around it – for instance, start 3-5 seconds *before* the event and end a few seconds after, so the clip has context. In cases of *“shifts”*, which represent longer intervals of a player’s ice time, we’d use the exact shift start and end times (our data extraction provides shift start/end down to the second [26]). That could result in a 40-second to 1+ minute clip per shift, which is fine. We just need to ensure the file sizes remain reasonable. FFmpeg will output an MP4 clip for each segment. We’ll name these dynamically similar to our naming convention (e.g., `suzuki_zone_exit_1P_12-34.mp4` for a clip from 1st period at 12:34). We can either save them to the `data/clips` directory (and possibly update the index cache) or serve them as ephemeral files via the API. Caching them on disk is wise for performance – if a coach asks the same query later, we won’t recut the same clip. Our ClipIndexManager’s cache can be invalidated or updated to include newly created clips as needed (we can call `discover_clips` again or add to its `clip_cache`).

- **Security Filtering:** Before finalizing the list of clips, we apply role-based access rules. For example, if a player (user role PLAYER) asks for “show me Cole Caufield’s clips,” the system should restrict clips to ones the user is allowed to see. Currently, players can only see their own clips by policy [27]. Coaches/Analysts can see all team clips [28]. The ClipRetriever will use the `user_context` to enforce this, likely by filtering out clips that the user shouldn’t access (we can reuse `_user_can_access_clip` logic from the API [29] during or after retrieval). This ensures the results comply with our role-based access control. If no clips remain after filtering, the user will get a polite “No video clips available” message.

4. Video Clip Extraction and Assembly

For each identified event, we execute the extraction as described: 1. **Identify Source Video**: Determine the correct game and period. We’ll have a consistent naming scheme or a lookup to find the right video file. For instance, if the event metadata says game date 2024-10-12 vs Toronto, period 3, we find the file `2024-10-12_MTL_vs_TOR_period3.mp4` (exact path depends on how we store raw game footage). It’s implied we have these files accessible on the server. If they are not already segmented by period, we may pre-split them to speed up seeking.
2. **Calculate Timing**: Use the event’s timecode to seek in the video. The `timecode` is essentially the number of seconds from the start of that period’s broadcast. We will subtract a few seconds for a lead-in (if available – ensure not to go before 0). For example, an event at timecode 350s – we might cut from 340s to

355s to capture 5 seconds before and after. For a **shift**, we'll use the shift's exact start and end times (from our shift data) as the clip boundaries.
3. **FFmpeg Clip Command**: Run ffmpeg with arguments like:

```
ffmpeg -i game_period3.mp4 -ss 340 -to 355 -c:v libx264 -c:a copy -y
suzuki_zone_exit_20241012_clip1.mp4
```

This seeks to 5:40 (340s) and outputs 15 seconds until 5:55 (355s). We use `-c:a copy` (copy audio) and `-c:v libx264` to encode video (we can also attempt stream copy for video if cut aligns on keyframes to speed it up). We run this asynchronously in Python; since our orchestrator is async, we can `await` the subprocess. If multiple clips need cutting, we might do them sequentially or concurrently limited by CPU. Given typically <10 clips, sequential cutting with efficient encoding should be acceptable (especially if source videos are local and short). 4. **Thumbnail Generation**: Optionally, generate a thumbnail for each clip (we can reuse our `ThumbnailGenerator.generate_thumbnail` which grabs a frame at 5s into the clip [30]). This will give a quick preview image for the UI. Our ClipRetriever already calls `thumbnail_generator` in the API when a clip is requested [31]. We might integrate thumbnail paths into the ClipResult.
5. **Metadata assembly**: For each clip, create a **ClipResult** object with relevant info [32]. This includes a unique `clip_id`, title, player name, game info (opponent and date) and event type. We can fill in a descriptive title like "Nick Suzuki – Zone Exit vs TOR (2024-10-12)" and maybe a short description (e.g. "Clears defensive zone with controlled exit"). If our event data provides a textual description (some NHL data does, or we could template one from event details), we add that. The **file_url** will point to our API endpoint for the video (e.g. `/api/v1/clips/<clip_id>/video`) and similarly a thumbnail URL [33]. The ClipRetriever's `process` method already collects the clip results into the state [14]. We'll ensure that newly generated clips are accessible via the same API. In practice, we might need to write the new clip file to `data/clips/...` and then the existing `/api/v1/clips/{id}/video` route can find it by scanning the updated index [34] [35]. (Alternatively, we could bypass writing to disk and stream it directly with a `StreamingResponse`, but writing to disk lets us reuse the existing serve logic and cache for later.)

Throughout extraction, we must handle errors gracefully (ffmpeg failure, file not found, etc.). Errors will be caught and logged [36]. If a clip fails to generate, we'll mark that tool execution as failed in the agent state (so the LLM knows something went wrong if it needs to respond with an apology or alternate info) [37]. But in normal cases, we expect success.

5. Orchestrator Integration and Tool Workflow

In the LangGraph orchestrator, after intent analysis and routing, the **ClipRetrieverNode** will run (likely in parallel or sequence with other tools). The orchestrator's design calls tools and accumulates their results in an `AgentState`. For clip retrieval, the node returns a `ToolResult` object containing the list of found clips and the parameters used [14]. We include also a record of how long the retrieval took (for logging/performance) [38]. This ToolResult is added to the state's `tool_results` list [39].

We also place the actual `ClipResult` list into a known location in state (e.g. `state["analytics_data"]["clips"]`) [39] so that the Response Synthesizer can access it easily when formulating the answer. The ClipRetrieverNode will generate simple "citations" metadata too – e.g. `[clip_database:3_clips][games:vs Toronto (2024-10-12)][players:Nick Suzuki]` to denote

the sources of the clips [40] [41]. This is analogous to how our system cites data or knowledge sources, but here it's citing the internal clip database and context of the clips. These citations can be surfaced in the UI or included in the LLM's answer for transparency.

Crucially, we will update the **Tool Registry** to include the clip retriever tool, specifying what data it produces. Likely we treat video clips as a form of **visual or contextual data**. We might tag it as producing `visual` (since it yields media) or define a new tag like `TAG_CLIPS`. This way, the orchestrator knows if other tools need to wait for it (though in most cases clip retrieval can run independently or in parallel with data queries). We will configure it to not run in parallel with the final answer generation (since we want the clips ready before the model writes the answer). The orchestrator's scheduler will then include `retrieve_clips` in the function call sequence when the query demands it, much as it would include `query_game_data` for stats or `pinecone.search` for knowledge.

Additionally, we consider **performance**: since video generation could be slower than pure data lookup, we should monitor this. The orchestrator's logging will note how many ms each tool took [38]. If clip retrieval becomes a bottleneck, we might spawn it concurrently with, say, a stats query so the model can crunch numbers while clips are being cut. The LangGraph's design for multi-step reasoning (with Qwen3 planning) allows the LLM to decide to call clip retrieval only when needed. In some cases, the LLM might first call a data tool to figure out *which clips* are needed (e.g. determine which games were the last 5 games) and then call `retrieve_clips` with specific parameters. Our system should handle such multi-turn tool usage gracefully.

6. Response Synthesis and Presentation

After clips are retrieved (and any other analytical tools have run), the orchestrator moves to the **Response Synthesizer**. This is where the fine-tuned LLM (DeepSeek-Qwen model) produces a final answer by considering all accumulated data. We will modify the prompt or the system message to let the LLM know that video clips are available in `state["analytics_data"]["clips"]`. The model doesn't actually "see" the binary videos, but it has metadata: player names, events, what happened, etc. It can use that to form a meaningful answer.

The LLM will likely produce an answer that references the clips and describes them. For instance, if the query was *"Show me all of my d-zone exit clips from the last game"*, an ideal response might be:

> **"Here are the defensive zone exits you made in the last game vs Toronto (Oct 12, 2024):**
> – **Exit 1 (1st Period, 12:34)** – You retrieved the puck deep in the zone and skated it out past the blue line with control. (*Video Clip 1*)
> – **Exit 2 (2nd Period, 05:10)** – You made a quick outlet pass from the defensive corner to start a rush. (*Video Clip 2*)
>
> *These clips highlight your successful exits under pressure.* [40] "

The model can't *see* the clip content, but our system knows the context of each clip (we can supply brief descriptions as above). We might prefill the description in ClipMetadata (e.g. "carried puck out of DZ" if our data labels the event outcome). In absence of detailed descriptions, the LLM might generate a generic statement per clip, but since it's been fine-tuned on hockey data, it should be able to infer a bit from the

event type and context. The key is that it will enumerate the clips in a clear, **formatted list** or paragraphs for easy reading. We will enforce that the model output remains text-focused (the README notes *"Plain text only, no Markdown or asterisks; clear hyphen bullets"* for output style [42]). However, since the user specifically requested Markdown formatting in answers, our final system might allow some formatting or at least well-structured text. In any case, the UI will handle embedding the actual video players.

On the **frontend**, we will use the `VideoClipsPanel` component to display the clips returned. The LLM's answer will be accompanied by the list of `ClipData` objects (via the API response). The client code already expects a list of clips and will render each with a thumbnail and play button [43] [44]. Our job is to ensure the API (or websocket) delivers the clip list. We might extend the response JSON to include a `clips` field alongside the text answer. The UI can then call `toAbsolute()` on the `file_url` and embed the videos [45] [43]. In practice, when the orchestrator finishes, it could return something like:

```
{
  "answer": "Here are your clips ... (text)…",
  "clips": [ {clip_id: ..., file_url: ..., thumbnail_url: ...}, {...} ]
}
```

The UI then shows the text in the chat bubble and the videos in a gallery below it. This achieves *"seamless video clip embedding in responses"* as outlined in our roadmap [46]. Importantly, the LLM's text will also include the **citations** we generated, indicating the source of the clips (e.g. `[clip_database:3_clips]` `[games:vs Toronto 2024-10-12]`). This is similar to Perplexity.ai showing sources for its answers – in our case the sources are internal (the clip database and game data) [40] [41]. It adds credibility to the answer, and the UI can display those references appropriately (maybe as footnotes or tooltips).

The **synthesizer** will also tailor the tone of the answer to the user's role. If it's a player asking about "my shifts," the response might be encouraging and focused on their performance. If it's a coach asking about a player's clips, the response might be more analytical ("In these clips, you can see the player tends to carry the puck out rather than pass. This was effective 4 out of 5 times, except on one failed clear in the 3rd period."). Because the LLM has access to both the video evidence and the statistical context (via other tools), it can combine them in the narrative. The result is a rich, multi-modal answer: text + video support.

7. Example Query Walk-throughs

Let's apply this design to the example queries:

- **Q: "Show me my shifts from the last five games."**
  Understanding: The user (assume a player) wants to review all their shifts (each shift = the time from getting on ice to getting off) over the past 5 games.
  Process: The intent analyzer sees "my shifts" -> Clip Retrieval [1]. ClipRetriever parses it: finds "my" => user's name (e.g., Cole Caufield) [2]; event type "shifts" is a generic term (we treat it as requesting full shifts, not filtering out any event) [47] [7]; time_filter "last_5_games" [48]. The node then queries the shift data from the last 5 games. Our extracted data has each shift's start and end times for the player [26]. We gather those intervals (perhaps 20-25 shifts total for 5 games). We then proceed to cut each shift's video: for each interval, find the period(s) it spans (usually a shift is within one

period), and use ffmpeg to cut from shift start timecode to shift end timecode. We might generate, say, up to 25 clip files (if that's too many to show at once, we could limit or perhaps combine by game). The ClipResults are created with titles like "Game vs TOR 2024-10-12 – Shift 1" etc., and description "Shift duration 0:45, started in defensive zone, ended with a line change after a whistle." The LLM then can summarize: *"You had 5 games of shifts loaded. Over the last 5 games, you played 23 shifts. Here are clips of each shift. Notably, your average shift length was 45 seconds. Clip 3 (Oct 10 vs BOS) shows your longest shift where you were on for 1:15...",* etc. It will list the clips in order per game. The UI presents a grid of these clips (likely scrollable). If 25 is too many, we might have the LLM or the system restrict to, say, a few highlights or allow pagination. But the architecture supports retrieving that many if needed (the ClipRetriever `limit` parameter could be increased or set by query, e.g. "show me all" might override to fetch all shifts).

- **Q: "Show me all ozone clips from the last five games."**
  Understanding: Likely the user (coach or analyst) wants offensive-zone play clips for a particular player or the team. The query isn't explicitly specifying a player – possibly it implies *"my team's offensive zone clips"* or if a player is logged in, *their* offensive plays. We'd clarify context: If a player asks, "my ozone clips" means their offensive zone actions. If a coach asks just "all ozone clips," maybe they mean the team's collectively. Assuming a player context for simplicity: "my ozone clips" -> player's offensive zone sequences.
  Process: "ozone" triggers the idea of offensive zone. Our parser might not directly catch "ozone" unless we extend it, but it will catch the word "zone" which could mistakenly map to `zone_entries` [6]. That's actually reasonable – zone entries are a key offensive zone action. We should also consider offensive-zone *possessions* or *shots*. We might interpret this query as *zone entries that resulted in offensive zone possession*. Our ClipRetriever could combine multiple event types: zone_entries, maybe shots or scoring chances in the offensive zone. For a robust solution, we could use our **Whistle-to-Whistle sequences** from the data: find sequences where the team spent time in the offensive zone (OZ) and maybe had a positive outcome. But to keep it simpler: we retrieve all **zone entry** clips by that player/team in the last 5 games. The search params: player = user (if applicable), event_type = `zone_entries`, time_filter = last_5_games. If the user is a coach and no specific player mentioned, we might return team-level clips – possibly we have some team clips stored under `games/` or `vs_opponents/` directories for offensive zone plays (though not likely). In that case, we might retrieve clips from the `games/` folder labeled "game_highlights" that are offensive plays. If none pre-exist, we generate: for each game in last 5, for each zone entry event by the team, cut a short clip around that entry. That could be many clips (teams have dozens of entries per game). We likely would filter further, maybe only entries by key players or entries leading to shots. We might leverage our data to find *successful* zone entries (carry-ins). Perhaps limit to 5-10 clips total that exemplify ozone play.
  Result: The LLM could respond with something like, *"In the last 5 games, the team executed several offensive zone entries that led to extended pressure. Here are a few examples:* (then bullet list of clips from various games, with opponent and date). *Clip 1: vs BOS (Oct 10) – a clean carry-in by Caufield resulting in a setup in the slot. Clip 2: vs TOR (Oct 12) – a dump-and-chase where Slavkovsky recovers the puck in the corner..."* and so on. Each clip gives the coach visual evidence of offensive zone performance. The answer might also summarize overall insight: *"Overall, you've had a high success rate entering the zone with control in these games."*

- **Q: "Show me all of my d-zone exits clips from the last game."**
  Understanding: The user (player likely) wants every defensive zone exit they made in the most

recent game.
Process: Intent: Clip retrieval. Parse: “my” -> player’s name; “d-zone exits” -> event_type `zone_exits` (we map “exit” to zone_exits) [6] ; time_filter “last_game” [49] . Search events: we filter the last game’s events for that player where event is zone_exit. Suppose we find 3 such events (e.g., two successful exits, one failed). We check ClipIndex – unlikely we pre-saved generic defensive plays, so we generate new clips from last game’s video. We know the game (last game date), and for each exit event, we have a timecode. We cut maybe ~8-second clips for each (from a couple seconds before the exit to just after the puck leaves the zone). We get, say, 3 clips.
Result: The LLM’s answer might be: *“Here are all your defensive zone exits from the Oct 12, 2024 game vs Toronto:*” and then list each with period and context. *“- 1st Period (10:05): Retrieved the puck behind the net and skated it out along the boards (successful zone exit).

- 2nd Period (7:20): You banked the puck off the glass and out of the zone (successful, under pressure).
- 3rd Period (14:45): Attempted a stretch pass from the d-zone, but it was intercepted at the blue line (failed exit).” *The model can highlight which were successful vs not if it knows the outcomes (our data flags exit success/failure). It then might add a coaching tip like,* “Overall, in the last game you were 2/3 on exit attempts under pressure.”* And importantly, each bullet corresponds to an actual video clip the user can watch.

Through these examples, we ensure the system not only retrieves the correct clips but also **contextualizes** them. This aligns with our inspiration from Perplexity – we’re building a retrieval-augmented LLM, but instead of just text snippets, we retrieve **video clips** as the supporting “documents.” The LLM then provides a cohesive answer with those clips as evidence.

8. Ensuring Efficiency and Future Enhancements

To make this clip retrieval feature robust and fast, we will: - Preprocess and index as many common clips as feasible (goals, assists, saves, special teams highlights) so those queries return near-instantly from disk.
- Use caching for generated clips: once a clip is cut, store it (and perhaps even pre-generate thumbnails) so subsequent queries reuse it. Our ClipIndexManager already caches index results for 5 minutes [50] [51] ; we can extend that to update when new clips are added.
- Only generate what’s needed: the query parsing helps narrow down exactly which moments to retrieve (player, games, event types), preventing unnecessary video processing.
- Monitor performance: Since our target is ~<5s even with vision analysis [52] , we’ll test that typical clip queries (maybe 3-5 clips generated) can be done within a couple of seconds on the server. If needed, we might spawn parallel ffmpeg processes or restrict clip length to keep it quick.

Finally, integrating this with our **Qwen-VL** vision model is a future possibility. For example, Qwen-VL could analyze a critical frame from each clip to double-check which player is which or to generate a richer description of the play. However, in the immediate term we avoid complex computer vision – we rely on known event metadata to identify the action (per the requirement *“without using computer vision to track players”*). This keeps the solution simpler and faster, leveraging structured data instead of real-time vision processing.

In summary, we will extend the HeartBeat orchestrator to include a **Clip Retriever** that maps natural language requests to specific video moments using our detailed play-by-play metrics. By cutting those

moments from game film (which we have segmented by period for efficiency) and returning them as clips, the LLM can then present a richly informative answer. This system will enable queries like *"show me my highlights"* or *"display our power-play entries vs Toronto"* to be answered with actual video evidence, much like Perplexity provides source-backed answers – bringing our hockey analytics assistant to a new level of interactivity and insight [46]. The plan above ensures that this is done systematically: from intent recognition, through data retrieval, clip generation, and finally to a well-formatted, insightful response for the end-user.

Sources: The design draws on our existing HeartBeat codebase and roadmap – the orchestrator's intent patterns [1] and clip tools [53], the clip directory schema [19], our analytics extraction (for timecodes) [15], and planned video integration features [46] – to achieve seamless clip retrieval in the service of advanced hockey analysis.

---

[1] intent_analyzer.py
https://github.com/skywalkerx28/HeartBeat/blob/884dee8e963fde63653840a30ef4773f4a8d2451/orchestrator/nodes/intent_analyzer.py

[2] [3] [4] [5] [6] [8] [9] [10] [11] [13] [14] [36] [37] [38] [39] [40] [41] [47] [48] [49] [53] clip_retriever.py
https://github.com/skywalkerx28/HeartBeat/blob/884dee8e963fde63653840a30ef4773f4a8d2451/orchestrator/nodes/clip_retriever.py

[7] [12] [16] [17] [20] [21] [22] [23] [32] [33] [50] [51] clip_models.py
https://github.com/skywalkerx28/HeartBeat/blob/884dee8e963fde63653840a30ef4773f4a8d2451/orchestrator/models/clip_models.py

[15] [26] COMPREHENSIVE_EXTRACTION_SYSTEM.md
https://github.com/skywalkerx28/HeartBeat/blob/884dee8e963fde63653840a30ef4773f4a8d2451/COMPREHENSIVE_EXTRACTION_SYSTEM.md

[18] [19] README.md
https://github.com/skywalkerx28/HeartBeat/blob/884dee8e963fde63653840a30ef4773f4a8d2451/data/clips/README.md

[24] [25] [30] thumbnail_generator.py
https://github.com/skywalkerx28/HeartBeat/blob/884dee8e963fde63653840a30ef4773f4a8d2451/orchestrator/utils/thumbnail_generator.py

[27] [28] [29] [31] [34] [35] clips.py
https://github.com/skywalkerx28/HeartBeat/blob/884dee8e963fde63653840a30ef4773f4a8d2451/backend/api/routes/clips.py

[42] README.md
https://github.com/skywalkerx28/HeartBeat/blob/884dee8e963fde63653840a30ef4773f4a8d2451/orchestrator/README.md

[43] [44] [45] VideoClipsPanel.tsx
https://github.com/skywalkerx28/HeartBeat/blob/884dee8e963fde63653840a30ef4773f4a8d2451/frontend/components/hockey-specific/VideoClipsPanel.tsx

[46] [52] HEARTBEAT_ENGINE_ROADMAP.md
https://github.com/skywalkerx28/HeartBeat/blob/884dee8e963fde63653840a30ef4773f4a8d2451/HEARTBEAT_ENGINE_ROADMAP.md