ChatGPT

Plan for Building HeartBeat.bot Automated Hockey Analytics System

Overview: HeartBeat.bot will be an autonomous agent that periodically gathers hockey information from trusted sources and synthesizes it into actionable content for our analytics platform. Inspired by Perplexity's automated research and report generation, HeartBeat.bot will perform web data collection and use an LLM to produce human-readable summaries and alerts [1] [2]. This will ensure that our platform provides up-to-date NHL roster moves, game highlights, team news, player performance updates, and daily league summaries with minimal manual effort. We will integrate this bot into the HeartBeat Engine's backend so it leverages our existing fine-tuned hockey analytics model and data infrastructure [3], while maintaining enterprise-grade reliability [4]. All AI-generated content will be reviewed by a human analyst before publication to guarantee accuracy and tone.

Objectives and Scope

The HeartBeat.bot will handle a **comprehensive range of daily updates** in the NHL, ensuring our platform's users never miss critical information. Key deliverables include:

- **Daily Transaction Alerts:** Automatically detect and post alerts for roster moves (trades, signings, call-ups, waiver claims, etc.) across all 32 teams. These alerts will appear on the analytics dashboard in near real-time whenever a transaction is recorded, with details like player names, teams involved, and contract/trade specifics.

- **Nightly Game Summaries:** After each game day, aggregate key information from every NHL game played. This includes final scores, standout performances (e.g. multi-goal games, goalie shutouts), and any notable incidents or milestones. The bot will collect game recap details from reliable sources and prepare brief summaries for each game.

- **Team-Specific Daily News:** For each NHL team, gather relevant news updates every day. This can range from roster changes and injury reports to travel schedules and insider rumors. The bot will maintain a per-team news digest (sourced from team official sites and trusted local outlets) so that users can see what's happening with any given team at a glance.

- **Player Performance Updates:** For every player in our system, provide an updated summary of their recent performance on their profile page. This might include statistics from recent games (e.g. "last 5 games: 3 goals, 4 assists") and highlights such as streaks or notable game performances. If a player had a major highlight (hat trick, record, award, etc.), the bot will annotate it on their profile. This keeps each player's page current with their latest exploits.

- **Daily League-Wide Digest (AI-Generated Article):** Every day, produce a **league-wide news roundup** – akin to Perplexity's Discover page – written by our LLM. This article will synthesize the day's most important events: big game outcomes, major transactions, injuries, quotes, and any

trending storylines. It will read like a concise news article covering the NHL overall, and be displayed prominently on the main analytics page. The content will be drawn from multiple sources (with the bot having "read" various news pieces), combined into a single coherent narrative by the LLM. A human will review this draft each morning before it goes live.

By achieving these objectives, HeartBeat.bot will function as an **always-on hockey analyst**, giving our users instant awareness of everything happening in the league, down to each team and player, without manual data entry.

Data Sources and Web Scraping Strategy

To ensure the information is reliable and comprehensive, HeartBeat.bot will **prioritize official and high-quality hockey sources** as input. We will start with a curated set of sources, and expand it with team-specific feeds as needed:

- **NHL Official Website (NHL.com):** This is the primary source for authoritative information. The bot will scrape the main NHL news section for league-wide announcements (e.g. league news, major milestones) and the official transactions page for roster moves. In addition, each team's official page on NHL.com (e.g. `nhl.com/hurricanes/` for Carolina) often contains news articles and press releases specific to that team. We'll parse those team pages (or their sub-sections like "News" or "Roster Transactions") daily to catch items like lineup changes, injury updates, and team announcements. Using the NHL's own data is ideal because it's timely and accurate; where possible we might use NHL's public APIs for structured data (similar to how we already fetch schedules via the NHL API [5]). For example, if there's an NHL endpoint for transactions or game recaps, the bot will utilize it; otherwise, HTML scraping will be employed.

- **Team-Specific Trusted Sources:** Beyond the official NHL sites, each team has local reporters, blogs, or official Twitter feeds that break news. In phase one, we'll focus on official and widely-read sites (to keep scope manageable), but we will design the bot so that each team can have a **configurable list of sources**. For instance, for the Montreal Canadiens, we might monitor the Montreal Gazette's sports section or a reputable blog for any analytical pieces or rumors; for another team, a well-known insider's feed or a fan site might be included. The plan is to structure a **source registry** (e.g. a JSON or YAML config) mapping each team to its preferred news sources. Initially, this could just be the official site and one general site like Daily Faceoff for all teams, then later we'll fill in team-by-team specifics as the user curates those lists. This modular approach makes it easy to plug in more sources per team over time.

- **Daily Faceoff and League-wide News Sites:** *DailyFaceoff.com* is a popular hockey site known for line combinations, rumors, and news. The bot will scrape Daily Faceoff's news section daily for any league-wide or team-specific tidbits (especially things like starting goalies, trade rumors, etc.). We may also include other league-wide sources such as TSN's NHL news feed, Sportsnet, or ESPN NHL section for a broader coverage. These will serve to cross-verify information and catch any important stories not covered on NHL.com (for example, trade rumors or analytical articles). Initially, however, NHL.com and team pages plus DailyFaceoff should capture most vital info.

- **NHL API and Internal Data**: We will leverage existing data sources where possible instead of scraping raw HTML if the information is accessible via API. For example, for game summaries, the NHL API provides game data (scoring summaries, stats) that we could use to generate our own recap if needed. Our HeartBeat Engine already integrates with the NHL API for rosters and schedules [6], so the bot can call similar endpoints to get live game results or player stats after games. This structured data can supplement the narrative from news articles. Also, since our platform maintains advanced metrics in Parquet files and a Pinecone knowledge base [3], the bot can pull contextual data (standings, player season totals, etc.) to enrich the updates (e.g., mentioning if a team's win puts them at first place in their division).

- **Web Scraping Methodology:** For HTML content, we'll implement scraping using **Python requests + BeautifulSoup** or a similar library to parse pages. We will ensure to respect `robots.txt` and not overload any site – scraping will be performed at modest intervals (and possibly at off-peak times like late night for daily digests). If any site offers RSS feeds (many news sites do), we can use those to easily fetch the latest articles without heavy scraping. The bot will extract the relevant text content (e.g. the body of news articles, or the snippet of a transaction blurb) and ignore multimedia content (since the user specified no need for video highlights at this time). **No video processing will be done** – we will strictly focus on textual information and maybe static images if ever needed for context.

- **Filtering and Relevance:** As the bot gathers content, it will filter for the information we care about. For example, when scanning a team's news page, it will look for keywords like "signed" or "trade" or "injury" to pick out roster moves and injury reports. For game recaps, it will identify key paragraphs describing the outcome and top performers. The goal is to trim the noise and keep only pertinent facts that would give our users an edge. Part of this filtering can be done with simple keyword rules, and part can be deferred to the LLM (which can summarize a long article into the key points). By pre-filtering, we also keep the amount of text fed into the LLM manageable.

In summary, HeartBeat.bot will gather data from **official NHL content (primary)** and **supplementary trusted media (secondary)**. The modular source list per team will ensure we eventually cover niche local info as needed. All scraping and API calls will be implemented in Python (for seamless integration with our project's Python backend [5]), and we'll reuse any existing fetching logic from our codebase where possible (for example, the way our `fetch_team_schedules.py` script hits the NHL API [5] can be mimicked for other NHL endpoints). This strategic data collection forms the foundation for the automation system.

Automated Workflow and Scheduling

To orchestrate these periodic tasks, we will introduce a **scheduler and worker system** into our backend – likely using **Celery** (with Redis or RabbitMQ as a broker) for robust task scheduling. Celery comes with a component called *Celery Beat*, which is a scheduler process that can kick off tasks at regular intervals [7]. This will allow HeartBeat.bot to run different scraping and processing jobs on a defined schedule, reliably and in parallel where needed. The reasons for choosing Celery include its flexibility in scheduling (support for crontab expressions, etc.), ability to distribute tasks across workers (for scalability), and integration with Python web frameworks.

We will set up **periodic tasks** (using Celery Beat or a similar scheduling mechanism) for each category of update. The tentative schedule and workflow for each task is as follows (we'll fine-tune exact times and frequencies):

- **Transaction Alerts Task:** Runs frequently (e.g. every hour, or every 30 minutes during peak transaction hours) to check for new roster transactions. This task will call an NHL transactions feed or scrape the NHL.com transactions page. Any new moves that weren't logged previously will be identified. For each new transaction, the bot will generate a short alert (e.g. "*Oct 15, 2025:* Montreal **acquired** F John Doe from Toronto in exchange for a 2026 3rd-round pick."). These alerts can be immediately pushed to the analytics page or stored in a buffer for the user to see. Frequent scheduling ensures near-instant awareness, but we will tune it to avoid redundant checks (e.g. maybe check hourly and always right after the daily NHL waiver wire deadline).

- **Game Results & Recaps Task:** Runs nightly, after all games are completed (e.g. 1:00 AM ET). This task will gather info for every game played that day. It might use the NHL API to get scores and stats, and also scrape the recap articles from NHL.com or team sites for any qualitative notes (like "Player X scored a hat trick to lead the team to victory"). The output will be a collection of **game summary snippets**. We'll store these per game (with game identifiers) and also use them to feed into the daily league-wide summary. If needed, this task can spawn subtasks per game for parallel processing, since Celery allows multiple worker processes. The game summaries might be 2-3 sentences each, focusing on key outcomes.

- **Team News Aggregation Task:** Runs daily (early morning, e.g. 6:00 AM ET). This goes through each team's dedicated sources (from the configured list) and picks up the latest news items for that team in the past 24 hours. For efficiency, instead of having 32 separate heavy jobs, the task can fetch all teams in one job (or a few in parallel batches) – we will optimize this by perhaps parallelizing per division or so. The result will be a structured collection like `{Team: [list of news updates]}`. Each update could be a one-liner or short paragraph. For example, for Team X: "Coach commented on Y's injury status (expected return next week)" or "Team signed Z (2-year entry-level contract)." These updates will be stored and also displayed on the team's page in the UI (perhaps under a "Latest News" section). This task ensures each team's info is fresh daily.

- **Player Performance Update Task:** Runs daily or perhaps after each game night (could be combined with the game results task). This will iterate through players who played that day (to limit scope) and update their recent performance metrics. We can utilize our internal analytics database for this: for example, query our Parquet stats to get each player's last 5 games stats or any new high. The bot can generate a sentence like "In the last week, Player X tallied 3 goals and 4 assists, and is currently on a 3-game point streak." If there's a news highlight (maybe the game recap noted this player's achievement), that can be incorporated. We might not need to run through **every single player** daily (which is thousands of players); instead, focus on players who have noteworthy events or who belong to the team the user is interested in. But since our site likely has profiles only for NHL roster players, a daily scan of all active roster players is feasible. We will update a field in the player's profile data (or a cache) with this summary text. This task may also be scheduled early morning, once game stats are final.

- **Daily Digest Article Task:** Runs early every morning (e.g. 7:00 AM ET) after all the above have collected data. This is the crown-jewel task that synthesizes everything. It will gather the outputs

from the game recaps, team news, transaction alerts, and any other major story, then invoke the LLM to compose a cohesive article. The process might look like:

- Collate key points: e.g. “Team A and Team B pulled off a major trade,” “5 games were played, including an upset win by a last-place team,” “Star player X had a 4-point night,” “Notable injury Y happened,” etc.
- Feed these points (along with short excerpts from sources for factual accuracy) into our LLM with a prompt to *"Write a concise, engaging summary of yesterday’s NHL happenings."* The LLM (which is trained on hockey context [3]) will draft an article that flows naturally, mentioning the highlights in a narrative form.
- The output article (maybe a few paragraphs) is saved as a draft. We will include references internally (like which sources were used for which fact) possibly as metadata, to assist the reviewer in verification.
- A human analyst (e.g. someone on our team each morning) can review this draft via an admin interface, make edits if necessary, and then approve it to publish on the site’s main page. By scheduling this at 7am, the content is ready right when users start their day.

The **scheduling infrastructure** will be configured carefully to manage these tasks. Celery Beat will manage the schedule and dispatch tasks to Celery workers. We’ll ensure that only one instance of each periodic task runs at a time to avoid duplication [8]. The worker processes can run in our backend server environment (for example, as separate Docker containers or threads) so as not to block the main API.

We chose Celery for its scalability – if our content volume grows, we can add more workers to handle scraping multiple sources simultaneously. For example, fetching all team news in parallel can speed up the cycle. Celery’s retry mechanisms are useful too: if a scraping task fails (say a site is temporarily down), Celery can automatically retry or we can schedule the next run without losing the schedule.

Alternate/Backup Scheduling: If for some reason Celery isn’t a good fit with our existing stack, a simpler alternative is using cron jobs on the server to trigger Python scripts (like those in our `scripts/` directory) at set times. However, given we likely want tight integration with our app and possibly to push notifications in real-time, an in-app scheduler is preferable. Celery + Beat offers the most control and integration (and is Python-native) [7], so we will proceed with that unless the architecture forces a change.

In summary, the HeartBeat.bot’s workflow will be highly automated through scheduled tasks that collect and process data around the clock. This ensures fresh content is generated without manual intervention, while still allowing a human final check where needed.

Integration with the HeartBeat Platform Backend

We will integrate HeartBeat.bot directly into our existing HeartBeat Engine backend, so it becomes a natural extension of our platform rather than a standalone silo. After reviewing the codebase and project structure, it’s clear that adding this functionality in the backend is feasible and beneficial:

- **Project Structure Placement:** We can create a new module or package (e.g. `backend/bot/` or `backend/tasks/`) to contain the scraping and scheduling logic. For instance, a `heartbeat_bot.py` module could define Celery tasks and orchestrate the flows described above. This keeps bot code organized separately from the API routes and orchestrator logic, but it will still

have access to the core settings and libraries of the project. Our backend is in Python (FastAPI for APIs, plus orchestrator code) so the bot code will be in Python as well, ensuring consistency.

- **Utilizing Existing Tools and Data:** The HeartBeat Engine already has an orchestrator with tools for retrieving information (e.g., `get_team_roster`, `get_schedule`) and a Pinecone vector database of hockey knowledge [3] [6]. We will leverage this where possible. For example, rather than writing a separate routine to figure out which games happened yesterday, we might call our `get_schedule` tool with yesterday's date to get the list of games (since the orchestrator can fetch schedule info via NHL API [9]). Similarly, to fetch player stats, we might use existing data clients (like functions in `parquet_data_client_v2.py` for recent game logs [10]) instead of reinventing the wheel. This integration ensures the bot is not duplicating functionality and remains **consistent with the rest of the system's data**.

- **Database/Storage Integration:** If our platform uses a database (SQL or NoSQL) to store content, we'll integrate the bot's outputs into it. For example, we might have tables for `transactions`, `team_news`, `player_highlights`, and `daily_articles`. The Celery tasks, after scraping and generating content, would upsert records into these tables. Then our existing API layer can simply read from the DB to serve the data to the frontend. If we don't have a formal DB set up, we can initially store these outputs in our `data/processed` directory as JSON files (similar to how schedule data is stored in JSON [11]) – this is quick to implement. However, for dynamic content like news that updates daily, a database or caching layer (Redis) would be more appropriate for query efficiency. We will likely introduce a simple persistence layer for this textual content.

- **New API Endpoints:** To display the aggregated content on the frontend, we will extend our FastAPI routes. For instance:

- An endpoint like `GET /api/v1/bot/daily_article` could return the latest daily summary article (and maybe previous ones if we want an archive).
- `GET /api/v1/bot/transactions` could return recent transactions (or this could be integrated into an existing endpoint for roster info).
- `GET /api/v1/teams/{team_id}/news` for team-specific news, and
- `GET /api/v1/players/{player_id}/highlights` for player updates.

These endpoints will pull from the data populated by HeartBeat.bot. We will define Pydantic models for the responses (e.g. a `NewsItem` model with fields like title, description, source, date). This approach aligns with how other parts of the analytics API are structured (for example, we have models for contracts and trades in `models/market.py` [12] [13]). By exposing this via API, our Next.js frontend or Streamlit app can fetch and render the info in the UI seamlessly.

- **UI Integration:** On the frontend side, we will add components to display the new content. The "analytics dashboard" page can be augmented to include a section for daily league news (populated by the daily article) and maybe a ticker or list of recent transactions. Team pages will show the team's news feed and player pages the performance blurb. We will ensure the content is presented with the same military-inspired styling as the rest of the UI [14] [15] for consistency. Since the user's question is backend-focused, the main point is that the backend will provide all needed data for these UI additions.

- **Human Review Mechanism:** Integration with the backend also allows us to build a simple review interface. For example, we can create an admin-only endpoint or page (protected by authentication) where the draft of the daily summary article is displayed along with an "Approve/Publish" action. This could simply toggle a flag in the database or move the content from a draft table to a published table. The human reviewer would use this interface each morning after the bot generates the article. Only after approval will the content be exposed to the public-facing endpoint. This ensures editorial control and catches any errors the LLM might have made.

- **Logging and Monitoring:** Following the project's professional standards for logging [4], we will add detailed logs for the bot's operations. Each Celery task will log its progress (e.g. "Fetched 5 new articles for Team X", "Daily summary generation completed in 8 seconds"). If any errors occur (e.g. a source is unreachable or the LLM fails to produce output), those will be logged and perhaps an alert can be sent to developers. This integration with our logging system means the HeartBeat.bot will be maintainable and issues can be diagnosed quickly in production.

- **Security and Load Concerns:** Running the bot within our backend means it has access to our environment and resources. We will sandbox any external calls (e.g. ensure requests time out and handle exceptions so a hanging web scrape doesn't crash anything). Also, we should be mindful of not letting the bot tasks consume all CPU/memory – hence using Celery workers that can be scaled and possibly run on separate machines if needed. Since our orchestrator is already async and high-performance [16], integrating Celery is complementary – Celery can handle the periodic jobs, while the orchestrator handles interactive queries. Both can coexist and even share data (e.g. orchestrator can query the news content if a user asks a question about it).

In conclusion, integrating HeartBeat.bot into the backend yields a cohesive system. The bot will feed new data into our platform's existing pipelines (even populating the Pinecone vector index for new knowledge, see below), and the end-user will experience it as just more features of HeartBeat (rather than an external feed). Python will be the core implementation language for all these components, aligning perfectly with our current tech stack [17].

LLM Summarization and Content Generation Framework

A crucial part of HeartBeat.bot is using an **LLM (Large Language Model)** to synthesize information into human-friendly text. We will build a framework that harnesses our fine-tuned hockey analytics model (DeepSeek-Qwen 32B) and/or other models to generate content in a reliable, controlled manner:

- **Choice of LLM:** Given that our orchestrator already uses a fine-tuned model (`DeepSeek-R1-Distill-Qwen-32B`) as the "central reasoning engine" [3], we should utilize this model for content generation to keep the domain knowledge strong. This model is trained on hockey analytics context, which likely includes knowing team names, player names, and common hockey terminology – ideal for writing recaps and news. We can access it via our existing SageMaker endpoint or orchestrator interface. In case this model is not well-optimized for long-form generation (since it's mainly for Q&A), we have a fallback to GPT-4 (or "gpt-4o-mini" per config) [18]. We can experiment: possibly use the fine-tuned model for shorter texts (player blurbs, game summaries) and GPT-4 for the longer daily article (to ensure fluency), or vice versa. Since cost and latency are considerations, we might try our primary model first and evaluate output quality.

- **Prompt Design:** We will craft prompts that guide the model to produce the desired output. For example, for the daily league-wide article, the prompt might be: *"You are a hockey journalist assistant. You have the following information from last night's NHL games and news: [bullet list of facts]. Please write a concise and informative recap of the day's major NHL events, in a professional yet engaging tone."* We will include key facts as bullet points or a short context paragraph. The model then "fills in" the narrative connecting those facts. Because our system values evidence and accuracy, we will ensure the model only has access to verified inputs (the bullet points will come directly from our scraped sources and stats). This approach is akin to how Perplexity's agent reads multiple sources and then writes a report [2] – we supply the sources to the model as context.

- **Multi-step Summarization:** For complex aggregation (like the daily article which has many pieces of info), we might use a two-step LLM process:

- **Micro-summaries:** First, use the LLM to summarize individual items if needed. For example, if a game recap article is very long, we can prompt the model to extract just the key outcome (who won and a key player) from that single source. Similarly, it could summarize a long team press release into one sentence. This could be done with a smaller model or even rule-based extraction, but the LLM might elegantly handle varying writing styles. We would do this for each game or news item as needed to distill the info.
- **Macro-synthesis:** Next, feed all those distilled points into the final prompt for the full article. This keeps the context within token limits and ensures the final model call is focused on already-filtered content. It's essentially performing the "dozens of searches, reads hundreds of sources" step separately, and the final model call does the "synthesizes into a clear report" [1] [2].

We'll implement these steps within our Celery task. LangChain or our own LangGraph pipeline could be used to manage these steps, but given we have a custom orchestrator, we might just call the model API directly with constructed prompts.

- **Tool-Assisted Generation:** Since our orchestrator has a **Response Synthesizer node** that is responsible for generating answers with citations [19], we can potentially reuse some of that logic. For example, the orchestrator might support a function like `orchestrator.generate_text(context)` or we can instantiate the model with the same settings. If the synthesizer automatically handles citation embedding, we may want to either leverage that or disable it for the article (the daily article might not need inline citations for readers, though internally we want to know sources). We will likely have the model output plain text (as opposed to Markdown, unless we want to stylize the text with bold team names or something). Ensuring a consistent tone and style is important – since this is an **analytics platform**, the writing should be factual and concise (somewhere between a news brief and an analytical note).

- **Ensuring Factual Accuracy:** The LLM will only be as accurate as the data it's given. Our framework will emphasize passing **ground truth data** into the model, rather than asking it to recall facts from memory (which might be outdated or hallucinated). By scraping official sources and using our up-to-date analytics data, we feed the model the exact details (scores, names, stats). The model's job is then mainly linguistic – to turn those into a nice narrative. We will instruct the model not to invent any information beyond what's given. If the model we use supports functions or a temperature setting, we will keep temperature low (to maintain determinism and avoid creative deviations). This

approach mirrors a Retrieval-Augmented Generation (RAG) setup: the model gets a knowledge context (from Pinecone or scraped text) and must stick to it [20].

- **Handling Different Content Types:** We will fine-tune prompts based on content:

- *Transactions:* These are straightforward; we might not even need the LLM to generate them. A template could do ("[Date]: [Team] [action] [player]…") since it's formulaic. However, if we want a more narrative style ("The Canadiens made a move today, acquiring X from Y…"), we could use the LLM with a prompt like "rephrase this transaction in one sentence". Probably a simple template is enough for alerts.
- *Game summaries:* We can use a template (Team A X, Team B Y, PlayerZ had 2 goals) or use the LLM to add a bit of flavor. Possibly: *"Summarize the following game result: Team A won 4-2 against Team B. Key events: [list]."* The LLM might then output something like "Team A defeated Team B 4-2, powered by two goals from Player Z, snapping Team B's three-game win streak." That reads well and adds context like streaks if provided. We should supply such context data (like streak info from our stats) to make it richer.
- *Team news:* If multiple minor news items for a team, we could either list them as bullet points or have the LLM merge them into a short paragraph ("For the Sharks: Player X returned to practice, while GM announced Y…").

- *Player performance:* This can probably be templated: e.g., "Over the last N games, [Player] has [stat line]." But for a more engaging blurb, the LLM can compare to averages or note improvements. Since our model is analytics-oriented, it might handle such comparison well if prompted ("Assess Player X's recent performance given these stats…").

- **LLM Framework & Libraries:** We will use our existing integration (likely via LangChain or direct API calls to SageMaker/OpenAI). The orchestrator already relies on LangGraph/LangChain, so we can extend that. For example, our code might have a utility to query Pinecone and then feed context + question to the model. We can adapt it: instead of a "question", we have a directive like "compose daily summary". Another approach is to script it imperatively in the Celery task: call Pinecone for relevant contexts (though since we gather the data ourselves, Pinecone might not be needed at generation time). We will likely not need a full autonomous agent (like an AutoGPT) here, because we're predefining the workflow. Simpler is better for reliability.

- **Content Storage and Post-processing:** Once the LLM returns text, we will do some post-processing:

- Ensure the output is within size limits (truncate if too long, though we'll prompt for a specific length).
- Sanity-check for any missing info or obvious errors (the human review is the safety net here).

- Possibly attach source references internally for our records (e.g. store which URLs were used, in case we want to show citations or for the reviewer to double-check). We might not display citations to end users on the article (as that could clutter the reading experience), but we will keep attribution info internally.

- **Human Tone and Style:** We should decide on the tone of the articles. Likely a neutral, factual tone with a slight analytical flavor (since the audience is hockey analytics-focused). We might instruct the model accordingly. During development, we can prompt the model with examples (few-shot prompting) if needed to calibrate style. Over time, if we accumulate many generated articles, we

could fine-tune a model on them for even more consistent style – but initially, prompt engineering and our existing fine-tune should suffice.

In sum, our LLM framework will transform raw scraped data into polished text using a controlled prompt-driven approach. We will **feed the model curated data and tasks** rather than letting it freely roam, which is similar to Perplexity's method of using an agent to read and then a report-writing phase [1] [2]. By integrating this with our existing model and toolchain, we ensure the generation is informed by our up-to-date hockey knowledge base and analytics. The result will be content that feels like it was written by an attentive hockey analyst, but actually produced by AI under our guidance.

Data Storage, Knowledge Base Integration, and Performance

Building on the integration discussion, here we detail how we'll store the bot's outputs and integrate them into our knowledge ecosystem:

- **Persistent Storage of Content:** Each type of content will be saved in an appropriate form:
- Transactions, team news, and player updates are well-suited to a **database or structured files** because they are discrete records. For instance, each transaction can be a record with fields (date, teams, player, description), each news item with (date, team, title, summary, source link), etc. A SQL database with tables (Transactions, News, PlayerHighlights, DailyArticle) would work. If our project doesn't yet use a DB, we could use SQLite for simplicity, or a document store like Mongo. However, since we already manage data in files (like Parquet and JSON), an interim solution could be writing JSON files for each day's output. For example, `data/processed/news/2025-10-15.json` containing all content for that day. This is easy to implement and version, but for scalability a DB is better. We'll likely start with a simple storage and then migrate to a proper DB as this feature matures.

- The **daily summary article** should be stored with a date or ID so we can retrieve past summaries if needed (and possibly show a history or allow the model to reference them). We can save the final reviewed version as an HTML or Markdown blob in the DB. The length is relatively short (a few paragraphs), so storage is trivial.

- **Integration with Pinecone (Vector DB for RAG):** One exciting advantage of having these textual summaries is that we can feed them into our vector knowledge base (Pinecone) to enrich the AI's knowledge for question answering. Our orchestrator's Pinecone retriever is designed to find relevant context for a query [21]. We will extend our pipeline that uploads data to Pinecone (perhaps similar to `scripts/upload_context_to_pinecone.py` we have). After HeartBeat.bot generates content (especially the daily article or any detailed analysis), we will create embeddings for those texts and upsert them into the Pinecone index. For example, the daily article can be broken into a few chunks and indexed with metadata like `date:2025-10-15`. If a user later asks "What happened yesterday with the Ottawa Senators?", the orchestrator could retrieve a chunk from the daily article that mentions the Senators (because of semantic similarity) and use it to answer [22]. This closes the loop between our automated content and our interactive Q&A system.

Additionally, team-specific and player-specific updates could be indexed with tags (team name, player name), so that queries about those entities pull in the latest info. This means our LLM won't hallucinate

outdated info about a player's status – it can retrieve the fresh update that our bot wrote. Maintaining this integration will make our platform truly feel "live" and context-aware.

- **Performance and Scalability:** Scraping dozens of sources and generating text daily has some cost, but it's quite manageable:
- Web requests: We'll implement polite delays and possibly do many in parallel (Celery concurrency). The volume (32 teams + a few league sites + transactions) is on the order of maybe 50 pages a day, which is fine. If some sites are slow, we might consider caching their responses or only diffing new content (e.g. remember the last article seen for each site, and skip if no change).
- LLM usage: The heavy LLM call is the daily article generation. That's one call per day (plus maybe smaller calls for micro-summaries). Even using a large model, it's acceptable. The player and game summaries might involve a moderate number of shorter LLM calls, but we can optimize by only calling when needed (e.g., we don't need an LLM to state "Team A beat Team B 4-3"; that can be templated). We will use the LLM where it adds value (fluency, combining info).

- Celery overhead: Our system will handle these tasks mostly in the background. We should ensure the periodic tasks don't overlap too much – we can schedule them such that they run sequentially in the early morning to avoid contention (e.g., game summary task finishes by 2am, player update by 2:30, daily article by 7am, etc.). If load becomes an issue, scaling out Celery workers or staggering tasks more is an easy fix. Celery's distributed nature is a plus for future scaling [7].

- **Error Handling and Redundancy:** Each step will include error checks. If a certain source fails one day, the bot should continue with others and perhaps try that source later. The human reviewer will also act as a fail-safe – if the daily summary is incomplete due to some data missing, they'll notice and can append info manually. Over time, as we trust the system, failures should become rare (and we can implement notifications to the devs if a task fails entirely). We'll also maintain backups of generated content (since it's small, maybe commit daily articles to a Git repository or cloud storage for record-keeping).

- **No Video for Now:** We reiterate that we will **not integrate video highlights at this stage** (point 3 of the user's clarification). This simplifies storage and processing. We might in the future attach images (like maybe a static chart or a photo if available via an API), but that's not in scope now. All outputs are textual (possibly with some basic Markdown).

- **Security Considerations:** Scraping external websites and integrating content needs some caution. We will parse and cleanse any HTML to avoid script injection. Only text content we intend to show will be stored. Also, if we display any part of the scraped text directly (we mostly will paraphrase via LLM or summarize, which mitigates copyright issues by not republishing large verbatim text). Nonetheless, since some content might be proprietary (like a full article text), we prefer the LLM to **summarize rather than copy** to stay on safe ground legally. Short factual statements (scores, transactions) are fine to use as is.

By putting these storage and integration pieces in place, HeartBeat.bot's contributions will persist and enhance the overall system's intelligence. Users will not only see the content on the front-end, but the behind-the-scenes AI will "know" the content as well, enabling more informed answers to ad-hoc questions.

Human Oversight and Review Process

Even though HeartBeat.bot automates content generation, **human oversight is a critical requirement** (point 1 from the user). We will implement a review workflow to ensure that all AI-generated content meets quality standards before being exposed to end users:

- **Draft vs Publish State:** Any long-form generated content (specifically the daily digest article) will initially be marked as a **draft**. The Celery task that creates the article can store it in a "drafts" table or file. It will not be served by the public API until approved. We will build a simple admin interface (could be a secured route in the frontend or even a command-line preview) for a human (likely our analytics team member) to read the draft each morning. They will check for factual accuracy (cross-check a couple of key points with the sources) and editorial quality (no weird phrasing or inappropriate language by the model).

- **Editing Capability:** The reviewer should be able to edit the content if needed. The admin interface can allow an edit box or we can instruct the reviewer to open the draft in an editor and tweak it. Common edits might be adding a missing detail, correcting a name spelling, or smoothing a sentence. Since the volume is low (one article per day), this is manageable. After editing, the reviewer hits "Publish", which moves the content to the published store. We'll log the fact it was reviewed by whom and when (for accountability).

- **Alert/Notification for Review:** We can automate a notification (email or Slack message) to the reviewer when a new draft is ready. For example, at 7:05am the system emails: "HeartBeat.bot has generated the daily NHL summary for Oct 15. Please review at [link]." This ensures the human is prompted to do their part in a timely manner.

- **Selective Trust for Other Content:** The shorter items (transactions, scores, etc.) are factual and in many cases will be directly scraped or templated, so they may not need heavy review. We can likely let those go live immediately (since they're essentially mirroring official announcements – e.g., if NHL.com says "Player X traded to Y", our alert is straightforward). The risk of error there is low except if our parsing fails. We will still glance at these feeds occasionally, and we'll have logging to catch anomalies (e.g., if the bot produced an alert that doesn't parse right, we can fix the template). But the major piece requiring human eye is the synthesized article where the model might accidentally omit context or mix facts from different games. With a quick read (a couple of minutes), the reviewer can validate it.

- **Feedback Loop:** If the human reviewer consistently finds a certain type of mistake (say the LLM tends to exaggerate a storyline or mis-order events), they can report that to the dev team. We can then refine the prompt or rules to eliminate that issue. Over time, the goal is the human rarely has to make changes – they become more of a final checker. We'll incorporate feedback to improve the LLM's output (via prompt tweaks or even fine-tuning if needed).

- **Fail-safe Publishing:** In the event the human doesn't review in time (say, on a certain day nobody is available by morning), we might have a policy whether to auto-publish or not. Initially, we would err on the side of not publishing unreviewed content. The content would simply remain as draft. The analytics page could either show nothing for that day or show "(Draft in progress...)". However, since

the user emphasizes human review, we will assume someone will check it daily. We can have a backup person in case the primary is out.

- **Auditing and Logs:** We will maintain a log of all content generated and the changes made by the reviewer. This is useful to trace back any incorrect info that slips through and to further train/improve the model. It also provides a layer of accountability – we'll know the AI said X, and the human changed it to Y, etc. These logs might be simply git diffs if we store articles in a version control, or entries in a moderation table.

By incorporating this human-in-the-loop stage, we combine **the efficiency of automation with the nuance of human judgment**. It de-risks the deployment of an AI writer in a professional analytics platform. Our goal is that to end users, the content appears seamlessly and reliably every day, but under the hood we've built a safety net where a human curator ensures nothing is off-base.

Implementation Steps and Timeline

To build HeartBeat.bot thoroughly, we can break the development into clear steps with an estimated sequence:

1. **Project Setup and Celery Integration:** *Duration: 1 day.* Install Celery (and a message broker like Redis) in our project. Configure Celery within our FastAPI app (creating a `celery_app` and Celery Beat schedule). Verify that we can run a simple periodic task (e.g., a dummy task that prints a log every minute) in our dev environment. This establishes the skeleton for scheduling.

2. **Source Configuration and Fetch Functions:** *Duration: 3-4 days.* Create a configuration mapping for data sources (initially NHL.com main, each team's NHL page, DailyFaceoff, etc.). Implement Python functions to fetch and parse each type of source:

3. `fetch_transactions()` – Scrape or call API for latest transactions.
4. `fetch_team_news(team)` – Scrape team's page (and any other sources for that team).
5. `fetch_game_results(date)` – Use NHL API or scrape summaries for all games on a given date.
6. `fetch_player_stats(player)` – Query our analytics data for recent performance (or use NHL stats API). Test these functions individually (perhaps using recent data) to ensure we correctly extract the needed info (e.g., list of transactions or structured game summary). We may write results to JSON for inspection.

7. **Celery Task Development:** *Duration: 2-3 days.* Write Celery tasks that use the above fetch functions, scheduled appropriately:

8. `task_collect_transactions` (periodic, frequent).
9. `task_collect_team_news` (daily).
10. `task_collect_game_summaries` (daily).
11. `task_collect_player_updates` (daily).

12. Ensure each task stores its results (maybe in a global cache or writes to file/DB). Since these tasks might run in separate worker processes, using a shared persistent storage (like writing to the DB) is important for passing data to the next stage.

13. Test each task in isolation by triggering it manually and verifying the stored outputs. For example, run `task_collect_team_news` and then query the DB to see that team news for each team was saved correctly.

14. **LLM Integration for Summaries:** *Duration: 2-4 days.* Using the outputs from the collection tasks, implement the summary generation:

15. Develop `task_generate_daily_article` that runs after the nightly data collection. It will load the necessary data (transactions, big game results, notable team news, etc.) and formulate the prompt for the LLM.
16. Use the OpenAI API or our model endpoint to get a draft of the article. Start with a conservative approach (e.g., GPT-4 with temperature 0.2) to see the quality. If using our Qwen-32B model via SageMaker, integrate the call similar to how orchestrator does it (possibly reusing `orchestrator.response_synthesizer` logic [19]).
17. Likewise, if needed, create smaller LLM calls for micro-summaries. For initial version, we might skip micro-summaries and rely on the model to handle the full context, but keep an eye on token limits.
18. Test this generation with sample data (we can use yesterday's real NHL info as a test case) and evaluate the output. Adjust the prompt until the structure and tone are acceptable.

19. Also implement any needed summarization for player or team updates if we choose to use LLM for those (likely not, these can remain mostly templated).

20. **Data Storage Implementation:** *Duration: 2 days.* Set up the database or file storage for the content:

21. Define Pydantic models for the content (e.g., a `DailyArticle` model with `date` and `content`, a `NewsItem` model, etc.) [23].
22. If using a database, create schema/migrations for the new tables. Otherwise, define file naming conventions and ensure the tasks write to those files.
23. Modify the Celery tasks to save their outputs properly. For example, after generating the daily article draft, save it as a draft record in the DB.

24. Implement functions to retrieve the latest published content for each category (these will be used by the API routes).

25. **API Route Extensions:** *Duration: 1 day.* Add FastAPI endpoints in `backend/api/routes` to expose the new data:

26. e.g., in `routes/analytics.py` or a new `routes/news.py`. Use the Pydantic models to shape responses.
27. These endpoints will read from the storage set up in step 5. For example, `get_daily_article()` returns the latest approved daily article.

28. Secure any admin endpoints (for draft review) with authentication. Possibly integrate with existing auth if any.

29. **Admin Review Interface:** *Duration: 2 days.* Implement a basic review UI:

30. If our frontend is Next.js, create an admin page (protected by login) that fetches the draft daily article and displays it with an edit option.
31. Or simpler, we can temporarily use an admin-only Streamlit page (since we have Streamlit in the project [24]) for internal use to list drafts and allow approval. Streamlit could be quicker for an internal tool.

32. Marking an article as approved could be as simple as a button that calls a backend endpoint to copy the draft to published status.

33. **Testing and QA:** *Duration: 2-3 days.* Do end-to-end tests of the entire pipeline:

34. Simulate a day's cycle: trigger transaction task, game summary task, etc., then daily article generation, then go through review step. Ensure everything flows (and that the schedule triggers tasks in the correct order).
35. Write unit tests for critical functions like parsing (to ensure we correctly parse HTML from sources).
36. Test error scenarios: e.g., what if a certain source is down – our task should handle exception and continue.

37. Also test the API endpoints (they should return the data in the expected format to the UI).

38. **Deployment and Monitoring:** *Duration: 1 day.* Deploy the updated system (likely in our dev environment first, then prod):

39. Ensure the Celery worker and beat scheduler are running on the server.
40. Monitor logs on the first few runs of each task. Verify that content appears on the front-end pages as expected.

41. Set up alerts if desired (e.g., if daily article generation fails, send email to devs).

42. **Iteration and Source Expansion:** *Ongoing.* After initial launch, gather feedback. Add more sources per team as the user provides them (this is just config changes mostly). Tune the LLM prompts if the human reviewers consistently tweak certain things. Over time, the process should require less manual intervention and the content quality will only improve.

By following these steps, we anticipate a working HeartBeat.bot system in a couple of weeks of development and testing. The result will be a pioneering automation within a sports analytics platform – a bot that tirelessly scans the hockey world and feeds our analytics engine with timely information.

Conclusion

HeartBeat.bot will elevate the HeartBeat Engine from a responsive Q&A system to a proactive analytics assistant. It combines **automated data collection, intelligent summarization, and seamless integration** with our existing architecture to deliver real-time hockey insights. By prioritizing official NHL data and reputable sources, we ensure accuracy; by using a scheduler and Python-based scrapers, we ensure

reliability and scalability; by leveraging our LLM as a synthesizer, we turn raw data into digestible narratives; and by keeping a human in the loop for oversight, we maintain quality and trustworthiness.

This plan aligns with the HeartBeat Engine's high standards (enterprise-grade, well-logged, and integrated [4]) and takes inspiration from state-of-the-art AI systems like Perplexity's autonomous research agent [1]. The end product will be the world's first advanced analytics automation platform in hockey – a system that not only answers questions but also continuously informs users of what they **should** know, as soon as it happens.

With HeartBeat.bot in place, our hockey analytics platform will truly live up to its name – providing the heartbeat of information that keeps teams and analysts a step ahead.

Sources:

- HeartBeat Engine Orchestrator README (project overview and architecture) [3] [4]
- HeartBeat Engine Data Fetch Example (NHL API schedule fetch script) [5]
- Perplexity AI Deep Research – autonomous search & report approach (analogy for our bot) [1] [2]
- Celery Beat Documentation – periodic task scheduler for Python (for implementing our scheduler) [7]

---

[1] [2] Introducing Perplexity Deep Research
https://www.perplexity.ai/hub/blog/introducing-perplexity-deep-research

[3] [4] [6] [9] [16] [17] [18] [19] [20] [21] [22] README.md
https://github.com/skywalkerx28/HeartBeat/blob/884dee8e963fde63653840a30ef4773f4a8d2451/orchestrator/README.md

[5] [11] fetch_team_schedules.py
https://github.com/skywalkerx28/HeartBeat/blob/884dee8e963fde63653840a30ef4773f4a8d2451/scripts/fetch_team_schedules.py

[7] [8] Periodic Tasks — Celery 5.6.0b1 documentation
https://docs.celeryq.dev/en/latest/userguide/periodic-tasks.html

[10] [14] [15] ANALYTICS_IMPLEMENTATION.md
https://github.com/skywalkerx28/HeartBeat/blob/884dee8e963fde63653840a30ef4773f4a8d2451/ANALYTICS_IMPLEMENTATION.md

[12] [13] [23] market.py
https://github.com/skywalkerx28/HeartBeat/blob/884dee8e963fde63653840a30ef4773f4a8d2451/backend/api/models/market.py

[24] run_app.py
https://github.com/skywalkerx28/HeartBeat/blob/884dee8e963fde63653840a30ef4773f4a8d2451/scripts/run_app.py