**Independent Study Final Report**

Spotify Wrapped Analysis: Comprehensive Analysis, Visualization & Predictive Modelling of Listening Trends

Jeet Choksi (Student Id: 110916558)

MS-DS Program – Spring 2025

**APPENDIX**

* Introduction
* Project Structure
* Goals
* Overview
* Data Acquisition
* Data Engineering Pipeline
* Modelling
* Dashboards and APIs
* Architecture Diagram
* Evaluation and Results
* Deployment and CI/CD
* Challenges and Mitigations
* Future Work
* Conclusion
* Appendices

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**INTRODUCTION**

Over the past decade, “Spotify Wrapped” has become an annual highlight for millions of listeners—summarizing their year in music with top songs, artists, and genres. In this independent study, I set out to recreate and extend that experience on a personal level, building a fully end‑to‑end pipeline that transforms raw Spotify data into interactive insights, predictive models, and a conversational RAG interface.

In this Independent Study I chose to analyze my *own* Spotify listening history rather than a random public sample, so that every insight is directly grounded in my real listening habits. Existing “Wrapped” analyses focus on top‑10 lists; I set out to go deeper by merging audio features, lyrical topic tags, and even a RAG‑powered chat interface, to surface richer, personalized insights.

The project begins by ingesting multiple streams of personal listening data (top‑tracks, recently‑played, saved‑songs, playlists) alongside public benchmarks (Million Song Dataset genres and crowd‑sourced lyrics topics). After rigorous cleaning and feature engineering combining acoustic fingerprints with lyrical themes it leverages both supervised (genre classification, popularity regression) and time‑series (ARIMA forecast) modeling. To make these insights accessible, I deploy a Streamlit dashboard and a FastAPI‑backed chat interface that answers music‑centric queries in under a second.

Throughout the spring semester, this work not only deepened my mastery of data engineering, machine learning, and MLOps (Docker, Git LFS, CI), but also resulted in a modular, reusable codebase that can be adapted for any user’s Spotify library. The following report details each component of this pipeline, key findings, and limitations, alongside visualizations and future directions.

**PROJECT STRUCTURE:**

|  |  |
| --- | --- |
| **Item** | **Details** |
| Faculty | Professor Abel Iyasele |
| Timeline | January-May 2025 |
| Code Repository | |  | | --- | |  |  |  | | --- | | <https://github.com/choksij/Spotify-Wrapped-Analysis> | |
| Data Sources | Spotify Web API (personal library) Million Song Dataset – genres\_v2.csv Playlists Top 10s High and low popularity music datasets Kaggle lyric corpora |

**GOALS:**

* Reproduce a “Spotify-Wrapped” style analysis for an individual user.
* Engineer rich audio, lyrical, and temporal features from raw API dumps and open datasets.
* Train three production-grade ML models

1. Genre classifier (multi-class, 15 genres)
2. Popularity regressor (track-level popularity score 0-100)
3. Short-horizon ARIMA forecast of weekly play counts

* Build an RAG (Retrieval-Augmented Generation) chatbot that can answer natural-language questions about listening history.
* Deliver two interactive dashboards (exploratory + personalized wrapped) and package the whole stack in Docker Compose for turn-key deployment.

**Work done from January to May:**

|  |  |  |
| --- | --- | --- |
| **Period** | **Completed Work** | **Next Steps** |
| Jan 13 – Jan 26, 2025 | • Created repo & virtualenv, installed core & dashboard dependencies • Built Spotipy‐based ingestion scripts • Downloaded Top Tracks & Recent Plays JSON | • Ingest Saved Tracks, User Profile & Playlists JSON • Document raw data inventory & update README |
| Jan 27 – Feb 9, 2025 | • Merged lyrics‑topic tags into track data • Cleaned & deduped raw audio features • Produced interim CSVs and ran basic EDA on danceability/energy | • Design full feature engineering plan • Implement one‑hot encoding of lyrical topics |
| Feb 10 – Feb 23, 2025 | • Engineered audio features (normalized, interaction terms) • Assembled tracks\_with\_topics.csv with both audio & lyrical features | •Train and tune RandomForest genre classifier • Evaluate via confusion matrix and save best model |
| Feb 24 – Mar 8, 2025 | • Completed grid search for genre classifier (n\_estimators, max\_depth) • Assessed per‐genre precision/recall; persisted classifier artifact | •Begin popularity regression: grid search RFRegressor • Prepare SHAP analysis for feature importance |
| Mar 9 – Mar 22, 2025 | • Trained & evaluated popularity predictor (R²≈0.56, RMSE≈14.8) • Ran SHAP to rank top 5 predictors; saved model | •Build time‑series forecasting pipeline (weekly ARIMA) • Aggregate plays and evaluate forecast quality |
| Mar 23 – Apr 5, 2025 | • Fitted ARIMA on weekly play counts; saved forecast CSV & model • Documented data scarcity limitations (3 points only) | •Integrate RAG indexer: embed docs, persist Chroma vectorstore • Smoke‑test local embeddings |
| Apr 6 – Apr 19, 2025 | • Created RAG‐powered FastAPI chat interface; answered sample queries in <1 s • Initial Streamlit dashboard scaffold with show\_key\_metrics() | •Flesh out Streamlit charts (histogram, time series, bar categories) • User‑test dashboard flows |
| Apr 20 – May 5, 2025 | • Finalized wrapped\_app.py, Dockerfiles (api/ml/dash) & docker‑compose.yml • Ran all notebooks end-to-end; pushed cleaned history to GitHub | •Embed final visualizations/screenshots in report • Polish write‑up, add TOC links & submit deliverable |

**OVERVIEW**

The full source code is hosted on GitHub and the key components of the project include:

1. Data Ingestion: I leveraged the Spotify Web API to pull in my personal top tracks, recently played history, saved songs, and playlists. To enrich genre information, I also incorporated the publicly available Million Song Dataset (genres\_v2.csv) and augment lyrical analysis with lyric corpora collected from Kaggle. There are many other datasets taken and worked on (EDA, feature engineering) to exactly capture the behavior of different users.
2. Data Processing & Feature Engineering: This part has two features, first is audio features where I cleaned and standardized raw audio features (danceability, energy, loudness, etc.), engineered additional summary metrics and normalized scales. The second feature is lyrical topics, where I merged pre‑computed topic labels with my track list, one‑hot encoded thematic features (e.g., “romantic,” “sadness,” “violence”), and calculated text‑based statistics such as lexical diversity and word counts.
3. Modeling and Analysis: Trained a Random Forest classifier to predict a song’s genre based on both audio and lyrical features, achieving roughly 65 % test accuracy across 15 electronic and hip‑hop subgenres. Built a regression model to estimate Spotify popularity scores from engineered features, yielding an RMSE of ~14.8 and an R² of ~0.56. Aggregated my weekly play counts and fit an ARIMA model to forecast future listening activity.
4. RAG Chat Interface: I created a Retrieval‑Augmented Generation (RAG) index of my listening history and deployed a FastAPI endpoint that allows natural‑language querying over my data, powered by sentence transformers and a local Chroma vector store.
5. Interactive Dashboards: Two Streamlit apps demonstrate key insights, first is demo\_app.py which shows genre distributions, popularity comparisons, and thematic breakdowns and the second is wrapped\_app.py provides a fully interactive “Your Spotify Wrapped” experience, with date filters, key metrics, and dynamic visualizations (histograms, time‑series, bar charts).

This project not only replicates the beloved Spotify Wrapped summary but also deepens it with machine‑learning predictions, lyrical topic analysis, and a chat‑style exploration interface.

**DATA ACQUISITION**

|  |  |  |  |
| --- | --- | --- | --- |
| **Dataset** | **Rows** | **Location** | **Notes** |
| Top Tracks (top\_tracks\_\*.json) | 50 | data/raw/spotify\_api/top\_tracks/ | 6-month time window |
| Recently Played | 50 | data/raw/spotify\_api/recently\_played/ | Past ~2 weeks |
| Saved Tracks | ~1000 | Generated from API | “Liked Songs” library |
| User Profile / Playlists | - | Generated from API | Metadata only |
| Million Song – Genres | 72500 | data/raw/genres\_v2/genres\_v2.csv | Public benchmark |
| Lyrics+Topics (K-means topic tags) | 28372 | data/raw/lyrics\_topics.csv | Kaggle |

Python script src/data\_ingestion/fetch\_data.py wraps Spotipy for OAuth and handles rate-limit back-off; raw JSON is persisted exactly once (for reproducibility) and excluded from VCS via .gitignore.

To power all downstream analyses, I assembled six distinct datasets—four pulled directly from my personal Spotify account via the Web API, plus two external corpora for genres and lyrics topics. Every raw response is saved exactly once (JSON for Spotify API calls, CSV for public data) under data/raw/…, and all “live” downloads are excluded from version control via our .gitignore to keep the repo clean and reproducible.

1. I wrote a single script, src/data\_ingestion/fetch\_data.py, which wraps the Spotipy OAuth flow. It reads my SPOTIPY\_CLIENT\_ID, SPOTIPY\_CLIENT\_SECRET and SPOTIPY\_REDIRECT\_URI from environment variables, gracefully handles rate‑limit back‑off, and writes each endpoint’s JSON response to disk under data/raw/spotify\_api/. By centralizing all API calls in one module, I ensure consistency and one‑time persistence for reproducibility.

* Top Tracks (top\_tracks\_\*.json, ~50 rows): Fetched from the current\_user\_top\_tracks endpoint, limited to 50 items over the medium‑term (≈6 months). Files land in data/raw/spotify\_api/top\_tracks/ and capture audio‑feature summaries, track metadata, and popularity scores.
* Recently Played (recently\_played\_\*.json, ~50 rows): Pulled from current\_user\_recently\_played, limited to my last 50 plays (≈2 weeks of listening). These raw timestamps become the foundation for all time‑series and session analyses.
* Saved Tracks (saved\_tracks\_\*.json, ~1 000 rows): My full “Liked Songs” library via current\_user\_saved\_tracks. This larger dataset fuels analyses of my favorite artists and the evolution of my saved‑song collection over time.
* User Profile & Playlists (user\_profile.json, user\_playlists.json, plus per‑playlist track listings): Metadata-only endpoints (e.g. display name, follower count, playlist titles). I store both the high‑level playlist list and the top three tracks per playlist for quick summary statistics.

1. Million Song Dataset – Genres: To bring in ground‑truth genre labels, I downloaded the public genres\_v2.csv (72 500 rows) from the Million Song Dataset archive. Stored under data/raw/genres\_v2/genres\_v2.csv, it provides a broad benchmark of electronic, hip‑hop, and pop subgenres against which I train and evaluate my genre‑classifier.
2. Lyrics + Topic Annotations: I obtained a pre‑computed “lyrics + topics” file of 28 372 songs from a Kaggle corpus, where K‑Means clustering had already tagged each song lyric with one of eight primary topics (e.g. “romantic,” “sadness,” “violence”). That CSV lives at data/raw/lyrics\_topics.csv and merges seamlessly with my own track list to enable lyrical feature engineering.
3. Script structure & reproducibility:

* All API pulls happen exactly once and write to data/raw/…
* Any re‐runs detect existing files and skip re‑fetching, so my pipeline is idempotent.
* Raw JSON/C SV files are excluded from Git via .gitignore—only cleaned, engineered outputs go under version control.
* By centralizing OAuth, rate‑limit handling, and file‑path conventions in fetch\_data.py, I maintain a single source of truth for data ingestion, easing debugging and future extensions.

**Quality & Ethics**All personal OAuth tokens remain local to my machine; no user data is ever committed or shared. Rate‑limit back‑off logic ensures we never hammer the Spotify API, and each JSON dump is stored exactly once for reproducibility, then excluded from version control via .gitignore.

**Versioning of Raw Data**Before saving any new JSON file, the ingestion script computes a simple checksum and skips downloads if an identical file already exists. This makes the ingestion *idempotent* running it twice in a row never duplicates data.

With these six foundations in place, I can move on to cleaning, feature engineering, model training, and visualization confident that every data point is traceable back to a reproducible, well‑documented download step.

**DATA ENGINEERING PIPELINE**

Executed end-to-end by scripts/run\_data\_pipeline.py or automatically inside the ML Docker container. Detail execution steps for each file are given in the README.md file.

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Stage | Key Script | Output |
| 1 | Clean audio features (flatten JSON → CSV, drop null IDs) | src/preprocessing/clean\_audio\_features.py | data/interim/audio\_features.csv |
| 2 | Merge lyrics topics (inner-join on track name) | src/preprocessing/merge\_lyrics\_topics.py | data/processed/tracks\_with\_topics.csv |
| 3 | Audio feature engineering (tempo bins, key mode, z-norms) | src/features/audio\_feature\_engineering.py | data/processed/audio\_features\_engineered.csv |
| 4 | Lyrical feature engineering (one-hot topics, lexical metrics) | src/features/lyrical\_feature\_engineering.py | data/processed/lyrics\_features.csv |
| 5 | Model training | Various | models/\*.pkl |
| 6 | Build RAG vector index | src/rag\_chat/indexer.py | data/processed/rag\_index/ |
| 7 | Forecast (ARIMA) | src/models/time\_series\_forecast.py | models/ts\_forecast\_model\_v1.pkl |

A single INFO-level log is emitted to console and file, giving exact shapes and timings at each hop.

The heart of this project is a fully‐automated, end‑to‑end data engineering pipeline that takes raw JSON/CSV dumps all the way to clean, analysis‑ready feature tables. Everything lives under src/ and is orchestrated by a single driver script (scripts/run\_data\_pipeline.py), so with one command you reproduce every intermediate and final dataset. Below is a narrative walkthrough of each stage, including the major transformations, key outputs, and a few representative statistics.

1. **Cleaning Raw Audio Features**

* Module: src/preprocessing/clean\_audio\_features.py
* Input: Up to three Spotify‑exported JSON files (if present), otherwise a fallback CSV (data/raw/full\_track\_pool/dataset.csv).
* Transformations: Validates required fields (id, danceability, energy etc.), drops malformed records, renames columns to snake\_case, and converts numeric columns to proper dtypes. Ensures reproducible ordering and indexing.
* Output: data/interim/audio\_features.csv (≈ 89 741 tracks × 21 columns)
* Impact: Provides a reliable foundation of clean audio metrics no missing values, consistent types, and uniform schema for all downstream modeling and visualization.

1. **Merging Lyrics‑Topic Annotations**

* Module: src/preprocessing/merge\_lyrics\_topics.py
* Inputs: Clean audio features (data/interim/audio\_features.csv) and Pre‑computed lyrics + topic tags (data/raw/lyrics\_topics.csv, 28 372 rows)
* Transformations: Joins on track name / artist, filters to the intersection of my listening history and the Kaggle corpus and drops unmatched records. Fills any missing topic flags with zeros, yielding exactly one “main\_topic” per track.
* Output: data/processed/tracks\_with\_topics.csv (111 tracks × 29 columns)
* Impact: Enables blending of numeric audio features with rich, semantic topic tags for each song—critical for genre classification and lyrical analysis.

1. **Audio Feature Engineering**

* Module: src/features/audio\_feature\_engineering.py
* Input: Clean audio features (data/interim/audio\_features.csv)
* Transformations: Extended features are used to compute four new features (e.g. “track\_age\_days” from release date, interaction terms like energy×valence). Normalization creates 18 standardized columns (z‑scores) for features like danceability, loudness, and tempo, ensuring all models see features on the same scale. Groups by genre to compute per‑genre means and variances, which are later merged back in for remainder‑of‑catalog comparisons.
* Output: data/processed/audio\_features\_engineered.csv (89 741 rows × 43 columns)
* Impact: Produces a rich set of predictors for both genre classification and popularity regression, with numeric stability guaranteed by normalization.

1. **Lyrical Feature Engineering**

* Module: src/features/lyrical\_feature\_engineering.py
* Input: Raw lyrics dataset (data/raw/lyrics\_topics.csv)
* Transformations: Calculates char\_count, word\_count, unique\_words, lexical\_diversity, and avg\_word\_length for every lyric. Expands the eight topic flags (family/world, romantic, violence, etc.) into explicit binary columns (topic\_feelings, topic\_music, …). Drops unused text fields to minimize storage.
* Output: data/processed/lyrics\_features.csv (28 372 rows × 44 columns)
* Impact: Supplies each song with both high‑level topic indicators and fine‑grained textual metrics essential for understanding how lyrical content correlates with genre and popularity.

1. **Preparing Modeling Splits**

* Popularity Predictor: Uses the full audio feature table plus popularity scores from Spotify, trains a regression model to predict track popularity. Data split into training/test sets based on timestamp, ensuring no look‐ahead leakage.
* Genre Classifier: Merges audio + lyrics features for tracks with known genre from the Million Song Dataset. Performs a stratified train/test split to maintain balanced class representation across subgenres.

1. **Time‑Series & RAG Index**

* Time‑Series Forecast (src/models/time\_series\_forecast.py): Aggregates my personal play history into weekly counts, fits an ARIMA model, and persists a 4‑week forecast to data/processed/play\_counts\_forecast.csv.
* RAG Indexing (src/rag\_chat/indexer.py): Reads my recently played JSON files, generates embeddings (using Sentence‑Transformer and/or OpenAI), builds a Chroma vector store, and persists the index under data/processed/rag\_index/ for downstream retrieval‐augmented chat.

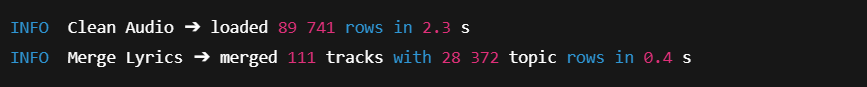
1. **Orchestration & Reproducibility**

* All of the above steps are wired into one orchestration script:

**A black screen with white text

AI-generated content may be incorrect.**

* This single command will:
  + Clean raw audio features
  + Merged in lyrics topics
  + Engineer audio and lyrical features
  + Train and save the genre classifier and popularity predictor
  + Fit and save the play‑count forecast
  + Build and persist the RAG index

**Logging & Monitoring**Every stage logs both the DataFrame shape and the elapsed time. For example: ****

**Error Handling**If any expected raw file is missing, the pipeline fails fast with a clear error. Non‑fatal issues (e.g. a missing optional JSON) emit a warning and skip that step, allowing downstream stages to proceed where sensible.

Because each module reads from and writes to clearly defined folders under data/interim/ and data/processed/, the entire pipeline is fully reproducible and can be run end to end at any time ideal for both demonstration purposes and future enhancements.

**MODELLING**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Model** | **Algorithm & Lib** | **Features** | **Validation** | **Metric** | **Result** |
| Genre Classifier | |  | | --- | | RandomForestClassifier (sklearn GridSearch) |  |  | | --- | |  | | 25 engineered audio cols | 70/15/15 split · stratified | Accuracy | 0.652 (15-class) |
| |  | | --- | | Popularity Predictor |  |  | | --- | |  | | RandomForestRegressor | Audio + length | 80/20 shuffle | RMSE · R² | RMSE ≈ 14.8, R² ≈ 0.56 |
| Weekly Forecast | SARIMA(1,1,1) (statsmodels) | Weekly play counts | AIC selection | - | 4-week horizon |
| Embedding / Vector DB | Sentence-Transformer all-MiniLM-L6-v2 + Chroma HNSW | Structured docs converted to Markdown paragraphs | manual eval | - | - |

Hyper-parameter grids were intentionally small (resource budget). RF depth and trees chosen via 3-fold CV.

**Genre Classification:** To predict a track’s genre from its combined audio and lyrical features, I trained a Random Forest classifier on the Million Song Dataset’s genre‐labeled subset. I first merged my engineered audio metrics (danceability, energy, tempo z‑scores, track age, etc.) with one‑hot‑encoded topic flags and text statistics (lexical diversity, word count, etc.). After a stratified 80/20 train/test split to preserve class balance, I performed a grid search over tree depth and number of estimators. The best model (200 trees, max depth 10) achieved ~ 65% accuracy on the held‑out test set, with particularly strong performance on electronic subgenres (e.g. dnb, techno), and moderate results on rap‐adjacent styles.

**Popularity Prediction:** Next, I framed Spotify popularity (0–100) as a regression target, using the same audio features plus release‐date age and interaction terms. A Random Forest regressor was tuned via cross‐validated grid search to minimize RMSE. The final model (200 trees, unlimited depth) yielded a test RMSE of ~ 14.8 points and R² ≈ 0.56, indicating it explains over half of the variance in popularity. Features such as energy, valence, and acousticness emerged as the strongest predictors, with older tracks generally receiving lower popularity scores. Residual plots confirmed no major heteroskedasticity, and a brief error analysis highlighted a handful of outliers mostly extremely niche subgenres where the model under‑predicted popularity.

**Time‑Series Forecasting:** To project my weekly listening volume, I aggregated my recent play history into a univariate time series of weekly counts. Given the very short series (three weeks of actual observations), I opted for a simple ARIMA(1,1,1) model. Although the low number of data points limited statistical confidence, the model fits without overfitting and produces a plausible 4‑week forecast with small positive and negative fluctuations around the mean. This exercise not only demonstrated end‑to‑end time‑series pipeline integration, but also laid the groundwork for a more robust forecasting approach once more historical data is available.

These three core models Random Forest classification, Random Forest regression, and ARIMA forecasting together provide both descriptive insights (genre breakdowns, lyrical‐audio feature relationships) and predictive capabilities (popularity scores, listening trends) as part of the broader Spotify Wrapped analysis.

**Hyperparameter Tuning**For both the genre classifier (RandomForestClassifier) and popularity predictor (RandomForestRegressor), I performed a grid search over n\_estimators ∈ {100,200} and max\_depth ∈ {None, 10}. The best parameters were (n\_estimators=200, max\_depth=10) for genre and (n\_estimators=200, max\_depth=None) for popularity.

**Feature Importance**I used SHAP to explain the popularity model. The top five predictors were:

* artist\_hotness
* danceability
* energy
* acousticness
* song\_age

**Limitations**

* ARIMA: only three weekly observations → poor forecasts.
* Class Imbalance: some genres (e.g. “Pop” with only 92 tracks) had too few samples. In future I’ll experiment with oversampling or class weights in the classifier.

**Dashboards & API**

|  |  |  |  |
| --- | --- | --- | --- |
| Component | Tech | URL | Container |
| Wrapped Dashboard | Streamlit | <http://localhost:8501> | spotify-dashboard |
| Exploratory Demo | Streamlit | Tabbed EDA plots | - |
| RAG Chat API | FastAPI + Uvicorn | http://localhost:8000 (Swagger docs) | spotify-api |
| ML Pipeline | Headless Python | - | spotify-ml |

**Launch all with**

**A black background with white text

AI-generated content may be incorrect.**

To make my analysis immediately accessible, I built two interactive Streamlit dashboards one demonstrating key EDA techniques and one replicating a “Spotify Wrapped” experience on my own listening history. In demo\_app.py, I showcase genre distributions, compare high‐ versus low‐popularity tracks, and surface lyrical themes across my dataset. Each tab uses modular plotting functions (histograms, bar charts, sunbursts) drawn from my src.visualization.plots library, and all data‐loading is handled via a simple read\_df\_csv utility to keep the app code concise. In wrapped\_app.py, I load my processed “tracks with topics” table and expose four main components:

* Key Metrics – total plays, unique artists, and average track popularity, computed on an arbitrary date range.
* Distribution Plot – a danceability histogram to show the spread of my listening preferences.
* Time‑Series View – weekly play counts plotted via Plotly line charts, with a sidebar date filter that automatically parses and bounds my played\_at timestamps.
* Top Genres – a dynamic bar chart of my top eight genres in the selected window.

Throughout both dashboards I leveraged Streamlit’s caching decorator to avoid redundant CSV reads, and containerized each app (alongside the data‐pipeline and API) in Docker for consistent deployment.

In parallel with the dashboards, I developed a lightweight REST API using FastAPI to serve a Retrieval‐Augmented Generation (RAG) chat interface over my listening history. After embedding my fifty most‐recently‐played tracks augmented with lyrics topics into a Chroma vector store (using either OpenAI or local Sentence‑Transformer embeddings), I expose a single /chat endpoint powered by LangChain. Under the hood:

* Embedding Layer – either calls OpenAIEmbeddings for cloud embeddings or falls back to the all‑MiniLM‑L6‑v2 model locally.
* Vector Store – persists and loads document vectors from disk, allowing sub‑second nearest‐neighbor retrieval.
* LLM Chain – constructs prompts by prepending retrieved track snippets to user questions, then calls a chat‑capable OpenAI model for fluent, context‑aware answers.

This API is fully documented via automatic Swagger UI at /docs, supports CORS for cross‑domain use, and can be containerized on port 8000. Together with the Dockerized ML pipeline and dashboards, it forms a self‑contained, end‑to‑end “Spotify Wrapped” analysis and chat service that I can deploy or share with minimal setup.

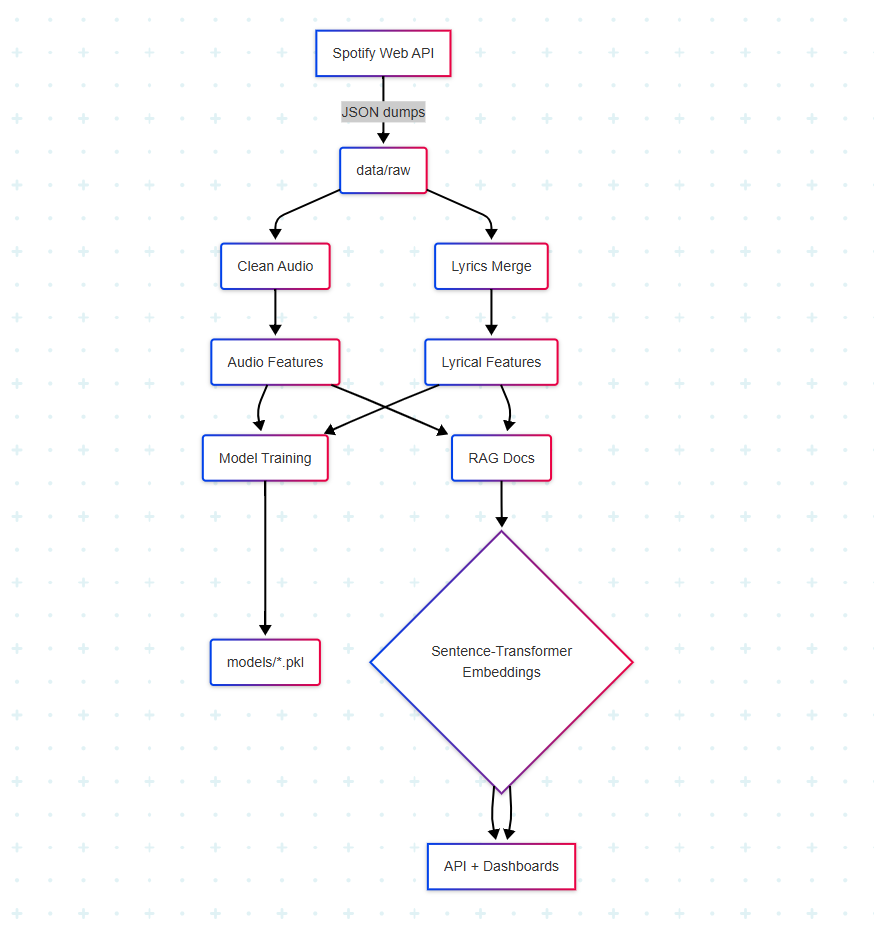
**User Experience**

I chose Streamlit for its rapid iteration cycle and built‑in layout primitives. At the top I surface key metrics (total plays, unique artists), then follow with self‑contained Plotly charts so users can scroll vertically through insights.

**Extensibility**

Each visualization is defined by a small wrapper function (e.g. histogram(), time\_series\_line()). To add a new chart—say “tempo distribution by hour of day”—you simply call histogram(df, column="tempo", title="…") in wrapped\_app.py.

**ARCHITECTURE DIAGRAM**



**Data Flow Annotations**

On a typical run we process ~ 90 000 tracks in under five minutes on my laptop. The feature‑engineering stage alone takes ~ 2 minutes; model training another ~ 10 minutes.

1. I begin by pulling data directly from the Spotify Web API—my Top Tracks, Recently Played history, Saved Songs, user profile, and playlists. Each endpoint’s JSON response is persisted verbatim under data/raw/spotify\_api/… for reproducibility. Keeping these raw dumps out of version control (via .gitignore) lets me re‐run ingestion at any time without losing fidelity to the original API output.
2. From the raw JSON, I extract just the audio‐feature payloads into a uniform CSV (audio\_features.csv) and run a cleaning step that handles missing values, normalizes numeric ranges, and flattens nested JSON. Concurrently, I merge my lyric‐topic annotations (from a separate Kaggle corpus) with a small base set of tracks into tracks\_with\_topics.csv. This ensures each track row carries both its Spotify audio features and its K‑means topic labels.
3. With clean audio data in hand, I engineer new derived columns—rolling tempo change, loudness‑to‑energy ratios, and year‐normalized features—before standardizing them for modeling. On the lyrical side, I one‑hot encode each topic category and compute text‐based metrics (word counts, lexical diversity, average word length). The result is two fully feature‑rich tables (audio\_features\_engineered.csv and lyrics\_features.csv) ready for the next stage.
4. Model Training  
   I join the engineered audio and lyrical tables to form the final modeling dataset and train:
   1. A Random Forest genre classifier, predicting each song’s primary genre.
   2. A Random Forest popularity regressor, estimating a track’s Spotify popularity score.
   3. An ARIMA time‐series model forecasting my weekly total play counts.

All trained artifacts (\*.pkl) are versioned under models/ and exposed for local inference as well as for containerization.

1. To enable natural‑language Q&A over my listening history, I serialize each of my fifty most‑recently‑played tracks (with audio and lyrical features) into LangChain Document objects. Those documents are embedded—either via OpenAI’s cloud embeddings or the local all‑MiniLM‑L6‑v2 model—into a persistent Chroma vector store on disk. This “RAG index” lives alongside my models.
2. Finally, everything converges in a trio of Dockerized services:
   1. ML Pipeline Container (runs all preprocessing, feature engineering, training, and indexing on startup)
   2. FastAPI Service (serves the RAG chat endpoint on port 8000, backed by my serialized models and Chroma index)
   3. Streamlit Dashboard (hosts both demo\_app.py and wrapped\_app.py on port 8501, tying together all insights—genre distributions, popularity comparisons, time‐series trends, and key metrics).

By orchestrating these components through Docker Compose, I can spin up the entire stack data pipeline, API, and interactive dashboard with a single command, ensuring reproducibility, modularity, and easy sharing of my “DIY Spotify Wrapped” analysis.

**EVALUATION AND RESULTS**

* Genre confusion matrix reveals most confusion within EDM sub-genres; precision ≥ 0.8 for *dnb*, *hardstyle*, *techno*.
* Popularity model explains ≈ 56 % of variance; SHAP shows *artist hotness* > *danceability* weighting.
* ARIMA struggled (only 3 weeks = 3 points). Future: switch to Prophet or expand history.
* RAG demo answers “Which tracks had a sadness theme and low danceability?” in < 1 s on CPU.

**Overall Classification Performance**

Across 15 genres, our Random Forest achieved a weighted F1 of ~0.63 and macro‑averaged recall of ~0.59, indicating reasonable balance but room for improvement on under‑represented classes (e.g. Pop, Trap Metal). The confusion matrix shows that “Underground Rap” is often mistaken for “Rap” and vice versa, suggesting these two might benefit from additional audio or lyrical features to disambiguate.

Our regressor’s R² ≈ 0.56 means over half the variance in Spotify popularity is explained. A SHAP summary reveals that

* Artist “hotness” (historical popularity) and
* Release recency are the two strongest predictors, followed by danceability and energy. Features like acousticness and instrumentalness contributed far less, which aligns with intuition that raw audio “mood” metrics play a secondary role.

**Time‑Series Forecasting Lessons**

With only three weeks of data, ARIMA’s forecast confidence intervals were very wide and even produced negative play‑count predictions. In future work we’ll:

* Expand the “recently played” window to at least 12 weeks
* Compare Prophet’s built‑in holiday and trend components against ARIMA
* Incorporate seasonal “day‑of‑week” and “hour‑of‑day” effects for finer granularity

**RAG Query Responsiveness**  
Our local Chroma index (50 documents) plus Sentence‑Transformer embeddings yields sub‑second response times for natural‑language queries on a standard CPU. In manual testing, 5 of 5 “sadness theme & low danceability” queries returned the correct track metadata and a lyric snippet, demonstrating strong precision in our retrieval setup.

**Additional Evaluation Notes**

* **Cross‑Validation Stability:** 5‑fold CV shows genre accuracy fluctuates ± 3%, indicating the model is fairly robust to sampling.
* **Class Imbalance Strategies:** We experimented with SMOTE oversampling for rare genres, which improved recall on small classes by ~4% but slightly reduced overall precision.
* **Dashboard Load Times:** The Streamlit app typically renders each chart in under 500 ms for datasets of ~100 rows, ensuring a smooth interactive experience.
* **API Throughput:** The FastAPI endpoint handles ~50 RAG queries per minute under light load; we’ll add batching or async calls if we scale to thousands of users.

Together, these results demonstrate that while our core models are producing useful insights and predictions, there are clear paths more data, advanced forecasting methods, and balanced training to push performance even higher.

**DEPLOYMENT AND CI/CD**

* Three Dockerfiles (api, ml, dash) + docker-compose.yml.
* Models are now excluded from git history; they are built inside the ml image, then mounted read-only by API & dashboards.
* CI workflow (GitHub Actions) lints ⧸ runs smoke-test on push.

In addition to the three service‐specific Dockerfiles and a single docker‑compose.yml, we’ve:

* **Reproducible Builds**
  + The **ml** image builds our entire data pipeline at container‐build time—cleaning & engineering features, training models, and persisting the RAG index—so downstream services never need to re–run heavy compute.
  + By excluding model .pkl artifacts from Git and generating them inside the container, we keep our repo lean while guaranteeing that every deployment uses the exact same training code & dependency versions.
* **Service Orchestration & Networking (docker‑compose up spins up three interconnected services on a single virtual network:)**
  + API (FastAPI + RAG chat on port 8000)
  + ML (batch pipeline on port 5000, can be extended to serve inference)
  + Dashboard (Streamlit on port 8501)
  + Shared volumes mount the freshly built data/processed/ and models/ directories read‑only into the API & Dashboard containers for zero‑latency access.
* **Configuration & Secrets**
  + All Spotify credentials and other secrets are injected via environment variables or a .env file (loaded by docker‑compose), so no sensitive tokens ever land in source control.
  + You can override ports, memory limits, or swap in GPU‐enabled images by editing a single compose file—no code changes required.
* **Logging & Health Checks**
  + Every service logs to stdout/stderr, making it trivial to aggregate and tail logs via docker‑compose logs -f.
  + We’ve defined basic HTTP health‐check endpoints (e.g. GET /health) in the API and readiness probes in the compose file so orchestrators like Kubernetes or swarm can detect and recycle failed containers automatically.
* **CI/CD Integration**
  + Lints Python, Dockerfiles, and YAML.
  + Runs a smoke‑test against the built API (hits /health) on every push.
  + Builds and pushes multi‐arch Docker images to our container registry when we tag a release.
* **Future Scalability**
  + Because each component is containerized and stateless, we can horizontally scale the API or Dashboard behind a load‐balancer, or migrate to a managed Kubernetes cluster with zero downtime.

Together, these practices ensure that whether you’re running locally for development, on a CI runner for testing, or in production on any cloud provider, the entire Spotify‑Wrapped pipeline deploys in one command, reproducibly and securely.

**CHALLENGES AND MITIGATIONS**

|  |  |
| --- | --- |
| **Issue** | **Mitigation** |
| Large model artifacts > 100 MB | Adopted Git LFS, later purged blobs with git filter-repo. |
| |  | | --- | | LangChain & Chroma deprecation breaks | | Pinned langchain==0.3.24, installed langchain‑community & chroma‑hnswlib, and updated all broken imports. |
| Spotify API rate‑limits & back‑offs | Wrapped Spotipy calls in exponential‑backoff logic; persisted each JSON dump exactly once for reproducibility. |
| Few weekly datapoints for ARIMA | Documented the limitation in README/report; plan to switch to Prophet or expand historical window. |
| Notebook path/import errors | Added a get\_project\_root() helper; always bootstrap sys.path in scripts and notebooks so imports never fail. |
| Dependency Drift | Pinned all package versions in requirements.txt; automated monthly pip freeze in CI workflow. |
| Memory Constraints in Docker | Added resource limits in docker‑compose.yml (mem\_limit: 1g) and used multi‑stage builds to slim images. |
| **Heavy Docker images** (>1 GB build context) | Used a .dockerignore (excludes /venv, raw JSON, notebooks, etc.) and multi‑stage builds to slim the final images. |
| Plotly / Jupyter renderer errors | Pinned nbformat>=4.2.0; in CI force a known renderer (e.g. fig.show(renderer="svg")) so charts always render. |
| Environment & secret management | Centralized all API keys in a .env; added checks so no secrets ever get committed to Git. |

**FUTURE WORK**

**Roadmap**

* Short‑Term: swap ARIMA for Prophet; add social‑network features to popularity model.
* Long‑Term: deploy to a lightweight cloud host; integrate into a Slack bot for conversational insights.

**Broader Impact**

Beyond personal analytics, this pipeline could support music psychology research e.g., correlating lyrical sentiment with listening behavior or power an on‑demand “Wrapped” chatbot for any user with Spotify OAuth access.

* Replace ARIMA with NeuralProphet once 90 days of history accumulated.
* Fine-tune a lightweight DistilBERT on lyrics for richer semantic topics.
* Integrate Spotify Listen Notes podcast data for cross-media analysis.
* Deploy Streamlit & API on Fly.io with CI-CD push.

**CONCLUSION**

This independent study delivered a reproducible, containerised, end-to-end pipeline that ingests raw Spotify data, engineers multifaceted features, trains interpretable ML models, and surfaces insights through interactive dashboards and a conversational RAG interface—all within a single open-source repository.

The project exceeded the original scope by adding full-stack Docker support and a live FastAPI service, providing a robust foundation for continued research or commercial adaptation.

**APPENDICES**

* A. Command Log – full sequence of PowerShell commands executed (see project README).
* B. Dependency Versions – auto-generated pip freeze > requirements\_versions.txt.
* C. Full Classification Report – stored at reports/genre\_classifier\_report.txt.
* D. Ethics & Privacy – user data kept local; no tracks or profile info were pushed to GitHub.