Index Tracking ETF Analysis GUI

FIN3028 – Developer Showcase

Eoin Grant - 40364041

Index Tracking ETF Analysis GUI

Developer Showcase Project Documentation

Author: Eoin Grant **Student number:** 40364041 **Submission Date:** 12/12/2025

Contents

Introduction

This project is an index-tracking ETF Analysis tool designed to support junior traders in evaluating index-tracking ETFs for on-screen execution, creation, or redemption opportunities. The system retrieves ETF and benchmark data via APIs, computes key metrics including synthetic NAV, premium/discount, tracking error, and liquidity, and generates visualisations to support interpretation. It also uses a lightweight signal-generation decision tree to provide an initial trading recommendation based on these metrics.

The system is fully modular, with dedicated modules for data sourcing, analytical computation, visualisation, trading logic, and user interface (UI). A JSON configuration file ensures that user-defined parameters flow consistently through the analysis pipeline. The PyQt5-based Graphical User Interface (GUI) enables users to run analyses repeatedly, select an output directory, and review outputs through an intuitive interface.

This work aligns with Option B (Coding Focused) of the Developer Showcase guidelines, placing strong emphasis on clean module design, external data integration using APIs, effective visualisation with matplotlib, and sound development practices.

Installation

This project is designed to run within the Anaconda environment (https://www.anaconda.com/) in line with the course requirements. Most of the libraries used (such as pandas, numpy, matplotlib, and os) are included by default in Anaconda. Two additional external libraries must be installed manually: PyQt5, which is used for building the graphical user interface (https://wiki.python.org/moin/PyQt). yfinance, which is used to retrieve ETF and benchmark market data (https://ranaroussi.github.io/yfinance/). To install these libraries, open the Anaconda Prompt from the Anaconda Navigator and run: “pip install PyQt5” and “pip install yfinance” as shown below. If the terminal returns “Access is denied”, re-run Anaconda Prompt as Administrator and try again. No further installations are required.

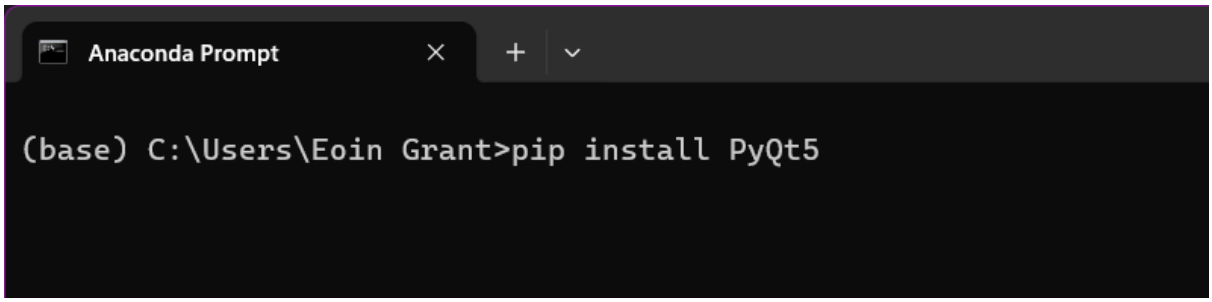

Figure 1 - Installing PyQt5 package

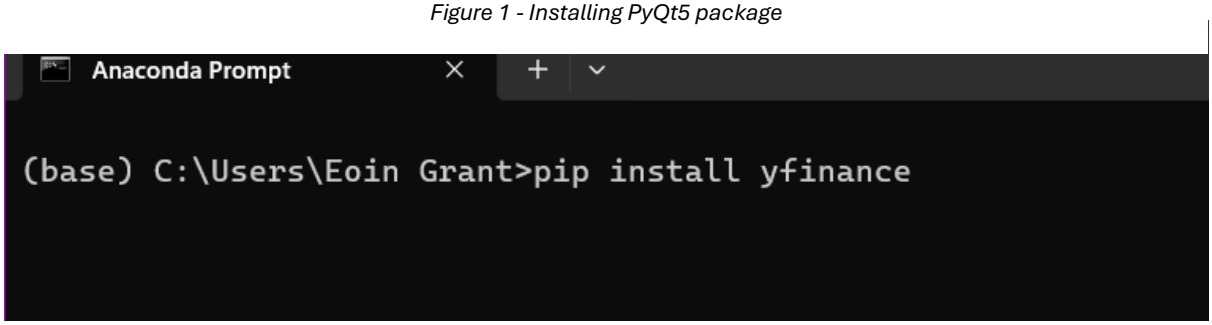

Figure 2 - Installing yfinance package

All market data used in the analysis is retrieved via the yfinance API. The above installation instructions assume a standard Anaconda environment with no previous package conflicts.

Project Structure

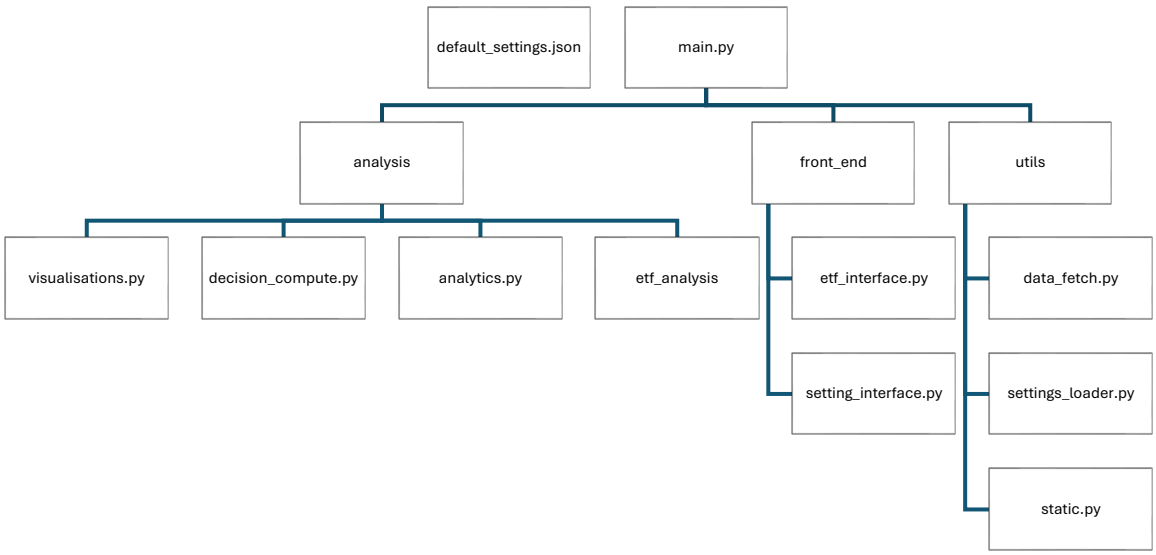

Figure 3 - Directory Structure

data_fetch.py: Responsible for sourcing all external data. This module retrieves ETF metadata, downloads historical price data, and includes defensive handling for missing or inconsistent fields returned by yfinance.

analytics.py: Performs the core financial computations. It calculates synthetic NAV, premium/discount, annualised tracking error, volatility, and liquidity scores using pandas and numpy. All analytical formulas are implemented here to keep the logic cleanly separated.

visualisations.py: Contains all chart-generation utilities used in the project. It provides generic line-plot and pie-chart functions, as well as a specialised tracking-error visualisation function. Charts are produced with matplotlib and exported to the user's selected output folder.

decision_compute.py: Encodes the project's trading-signal logic. Using threshold values for premium, liquidity, and tracking error (sourced from the JSON settings file), it produces a simple initial trading recommendation for on-screen execution decisions.

static.py: Holds static reference data, including benchmark mappings, currency metadata, and other constant lookup tables used throughout the analysis. These dictionaries were autogenerated via GenAI.

settings_loader.py: Handles loading, validating, and structuring the JSON configuration file. The application reads configuration from default_settings.json. Each setting is an

object with a “value” and a “type” key so the UI and analysis code can validate and convert values consistently. Supported type values are: int, float, string, bool, tuple, and choice. For choice settings, include an "options" array that lists the allowed values. The loader validates the settings and returns them as a dict.

etf_analysis.py: Acts as the project’s analysis orchestration layer. It fetches ETF and benchmark data, computes all analytical metrics, generates plots, builds the signal, and compiles the final text report. It exposes the central run_etf_analysis() function used by the GUI.

etf_interface.py: Defines the main PyQt5 GUI for running ETF analysis. It handles user input, folder selection, report display, exporting results, calling the analysis pipeline and launching the settings dialogue.

settings_interface.py: Defines the PyQt5 settings window. Dynamically constructs input fields from the JSON configuration file, validates edits and returns settings changes to the GUI.

main.py: The application entry point. Loads settings, initialises the GUI, and launches the user interface, providing a single starting point for running the entire system.

User Guide

One can run the application from Anaconda Prompt with “python main.py,” or from within Spyder (recommended IDE) with “main.py”.

To use the GUI:

1. Enter an ETF ticker symbol (e.g., SPY, QQQ, VUSA)
2. Select an output folder where charts should be saved.
3. Click Run Analysis
4. View the ETF report in the text window.
5. Use Optional buttons if needed.
    a. Export generated report (Excel, Clipboard, or text file)
    b. Open Output Folder

c. Reset Interface

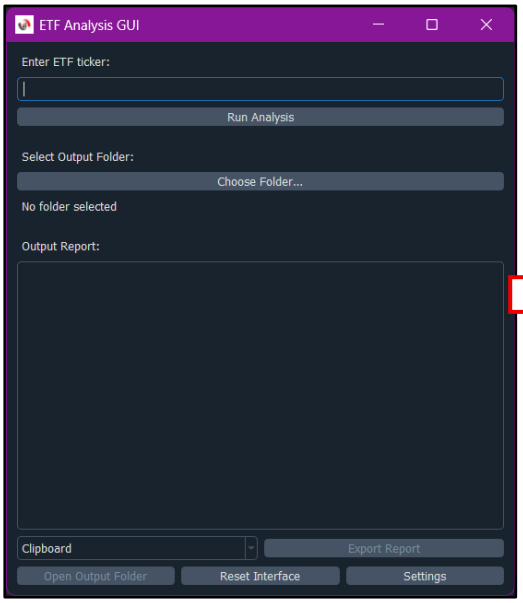

Figure 4 - Blank Interface

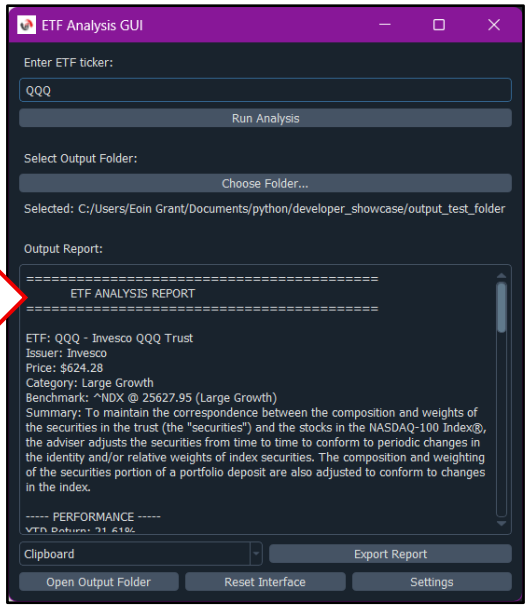

Figure 5 - Expected Output

The user may also wish to change the predetermined threshold settings (tracking error, liquidity, and premium) used to generate the initial trading signal, as well as the moving-average periods used to create the ETF price moving average and ETF rolling volatility plots. To do so, press the “Settings" button, these changes will be reverted on GUI restart. The user may also wish to change the default settings by editing the default_settings.json file. The user must ensure that any additions or changes conform with the consistent {"value": ..., "type": ...} format as described in the settings_loader.py section above. The default_settings.json file includes a few examples of how to add new settings values that aren’t currently used in the ETF analysis, such as the FTT rate, ARIMA, and GARCH model orders.

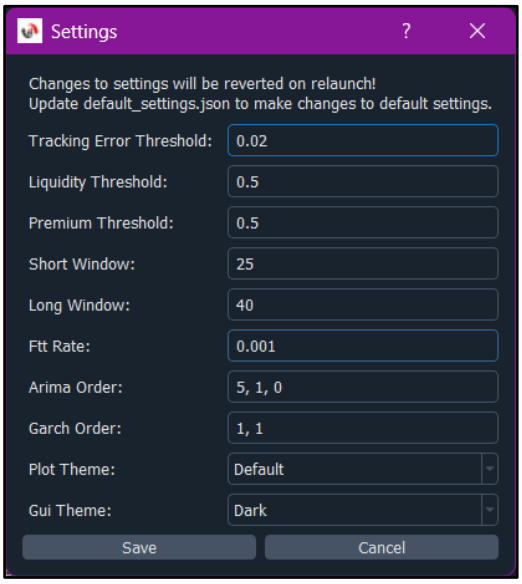

Figure 6 - Settings Window

Generated charts and results are saved in the user's chosen folder, see below.

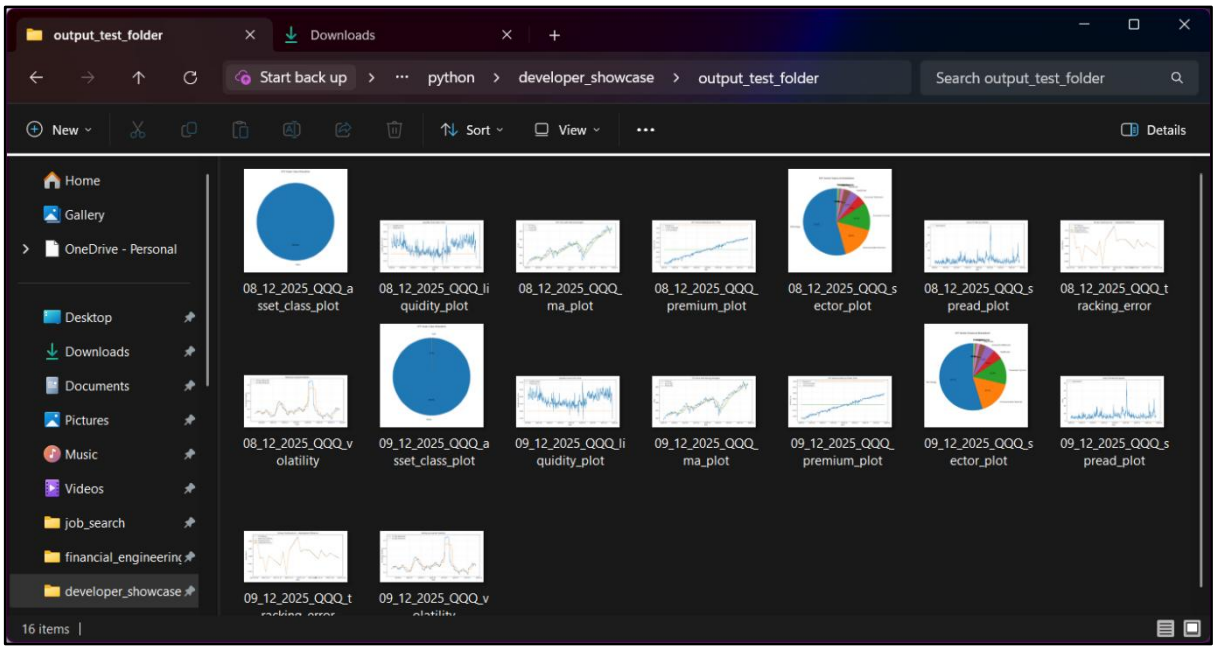

Figure 7 - Output Folder

The system outputs a detailed ETF report covering general metadata, performance metrics, premium/tracking metrics, liquidity metrics, fund/valuation details, and a trading recommendation. This report can be exported to the user's clipboard, a text file, or an Excel file, which will be saved in the user's chosen output folder.

Maintenance and Expansion

Maintenance

To keep this project effective and maintainable over time, several factors must be monitored. The system relies heavily on the yfinance API, whose data structure and available fields could change without notice. Such changes may break functionality that assumes the presence of specific fields, e.g., missing price fields, NAV values, or ETF category metadata. Defensive coding and regular testing will help navigate these changes.

The GUI front end also requires maintenance. Updates to PyQt may deprecate widgets or change certain behaviours that we expect, requiring adjustments to be made. As the interface grows, it would be beneficial to centralise styling in a stylesheet rather than modifying formatting in each widget. Alternatively, the UI could be built and expanded using PyQt Designer.

Static data, such as the ETF category mappings to relevant benchmark indices, and currency metadata, may also need periodic updates, particularly as new ETF categories emerge (e.g., crypto ETFs) or index tickers change. Storing this data in an external JSON

file or SQL database rather than hardcoding it would make updates cleaner and easier to manage.

Finally, long-term maintainability depends on good developer practices, i.e., consistent use of PEP 8 formatting, well-written docstrings, and maintaining modularity. Together, these will ensure that the system remains reliable, extensible, and straightforward for future developers to work with.

Expansion

The system offers a solid foundation for future expansion. A significant long-term improvement would be the integration of real, provider-level NAV data. Although the current version estimates synthetic NAV, free data remains lacklustre across ETFs. A more complete solution would involve scraping provider websites (e.g., Vanguard, BlackRock, SPDR) for complete daily holdings files, although would be labour-intensive due to non-standardised formats. A more scalable alternative would be to consume premium data feeds or APIs, such as those from Interactive Brokers (IB API), or, in an institutional setting, daily portfolio composition files (PCFs)/ daily NAV reports available through an Authorised Participant agreement. An authorised participant agreement is a contract allowing an authorised participant to create and redeem ETF shares, it also grants them access to these PCF files. With these richer datasets, the system could perform deeper holdings analysis with far greater accuracy.

A significant advantage of the current build is its modular structure and JSON-based configuration system. Because analytics, data sourcing, visualisation, and thresholds are all isolated into independent modules, new analytical techniques can be added with minimal friction. For example, advanced time-series modelling, such as ARMA/ARIMA, GARCH volatility estimation, could be integrated simply by creating a new analysis module and registering its parameters in the JSON settings.

Another valuable expansion would be that of multi-ticker batch analysis. This could take the form of a comparison of ETFs tracking the same index or a broader ETF screener that identifies the most efficient tracker for any given benchmark. The existing GUI settings could be extended to include screening filters to compare metrics.

Output functionality could also be expanded. Beyond the current chart and simple report export options, the system could generate complete formatted PDF reports that combine all the generated analytics, commentary, and visualisations, and/ or produce better-structured Excel workbooks. Also, enhancing the GUI with improved accessibility, theming, and tabbed plots would further enhance the user experience.

Finally, integrating a database such as SQLite or PostgreSQL would allow the system to store historical analytics, static reference data, and NAV data (if sourced). This would support historical tracking of liquidity, premium/discount behaviour, and tracking error,

allowing the tool to function not only as a point-in-time analyser but as a more comprehensive monitoring and research platform.

Reflection as a Developer

This project was definitely the most complete and technically challenging test of my Python skills to date; this experience has forced me to grow as a developer. I originally started this module with a very strong foundation in Python from my 12-month placement at Susquehanna International Group. I regularly used Python, SQL, and pandas for operational tools, analytics scripts, and GUI utilities. However, the open-ended nature and scale of this project pushed me well beyond the usual linear and task-specific scripts I had written in my experience so far.

A significant step in my development was learning to design and build a fully modular system, rather than a single self-contained script. By splitting this project into separate modules for data, analytics, visuals, settings, and the GUI it made me consciously think much more about how I structure my code and separate functionality. In doing so, I learned to keep different responsibilities separate to avoid mixing the interface with the underlying logic. This was a significant deviation from the smaller projects I worked on at SIG, where improvements often involved modifying simple scripts or writing new scripts for a specific task rather than engineering pipelines from scratch.

Although I was already familiar with pandas, numpy and various other common packages, the process of developing analytics for the many metrics that this tool covers forced me to apply these libraries in a far more analytical and rigorous way. In contrast to the operational scripts I built during my placement, where the goal was often to automate processes or present information efficiently (e.g., rebate estimation, SQL-driven analytics, and various daily market-making tools). This project meant creating analytics that will help with decision-making. Instead of just showing data, I had to build logic that analyses, interprets, and provides meaningful insights. This shift from operational tooling to analytical decision-support was a notable change in the very way I apply my Python toolset in a financial setting.

The most challenging yet rewarding part of the project was developing the front-end PyQt5 GUI for the analysis. This was a significant deviation from my SIG work, as mentioned, where GUIs were either pre-existing templates or built by one of the many in-house dev teams. Through this challenge, I had to learn how a GUI works, how widgets behave, how information flows through the interface, and how to connect it properly to the analysis running in the background. It also pushed me to learn more about object-oriented programming, using classes and objects. For example, the settings window forced me to think about inheritance and how shared attributes could be used to achieve better functionality. Once again, a significant deviation from my experience, and as such

significantly improved my confidence as it challenged my understanding of core Python concepts.

Another important learning point was working with a JSON settings file to configure the many parameters that the analysis pipeline uses to provide insights. Through incorporating the JSON and the adjustable settings panel in the GUI, I was able to build tools that allow the project to be easily updated and extended. Evolving from hard-coded thresholds, as they were in the early stages of this project, to configurable settings showed me how valuable dynamic solutions can be.

Another key area of developer growth came from the debugging and handling API inconsistencies. Unlike the complete internal SQL databases that I used during my placement, which were monitored and handled by other employees, yfinance often returns missing, malformed, or stale fields. This forced me to implement more rigorous error handling and input validation, all of which are essential in real-world development. Through these trails, I learned how to isolate problems, trace failures systematically, and subsequently use test cases to understand the issue at hand. Making me a more resilient and defensive developer.

Finally, this project helped me improve my ability to learn independently. Despite using module materials and lab examples, most of my breakthroughs came from reading documentation, experimenting with code, and trial-and-error. The use of AI tools supported this process as it allowed me to understand unfamiliar concepts, double-check ideas or spot errors more quickly. However, the real learning came from applying this guidance myself. By using AI in this way, I strengthened my own problem-solving skills and accelerated my development as opposed to bypassing it.

Overall, this project transformed me as a developer. Moving beyond writing scripts that process data into building a coherent, configurable, multi-module system with a graphical interface and extensible analytics. It strengthened both my technical abilities and my confidence. The skills I have developed will directly support my continued growth as I progress toward more advanced and technical roles in the finance industry.

Reflection as a Manager

Managing this project required me to turn a broad idea into a structured development plan which required adapting as the system evolved. Looking back at my JSON project plan and comparing estimated and actual effort, I can see how my planning matured throughout the project.

Most early tasks were completed ahead of schedule or within the expected time. This was mainly because I worked in focused sprints, freeing up valuable time for later in the

schedule. These early wins generated momentum and laid the foundations for the project.

However, the integration phase proved to be a real challenge. I had estimated four hours to merge modules into a cohesive pipeline, but it required eight, partly due to my early decision to include the originally optional GUI. This task meant ensuring that the many independent components and modules communicated reliably. Whether that meant handling missing API fields, consistent data structures, or connecting the GUI to the analysis logic. This taught me that my planning should treat integration as a continuous task that I should constantly keep in mind, as opposed to the last step.

A similar lesson emerged when I was building the GUI. What originally started as an optional add-on quickly became one of the most ambitious parts of the system, especially once I realised the value it could bring. What began as a simple interface mutated into a multi-component PyQt5 application with dynamic settings, robust error handling, and a JSON-driven configuration system. It took longer than I expected, but it significantly improved the tool's usability and professionalism, reminding me of the importance of managing scope and choosing upgrades that genuinely add value.

A more overarching shift occurred in how I approached planning more generally. During my placement at SIG, most of the Python tools I built were purpose-built scripts and not intended for generalisation. This mindset worked well in that environment, but it's not appropriate when designing a complete system such as this tool. This project challenged that habit and way of working. I had to think in terms of scalability, extensibility, and modular architecture, imagining not just the feature in front of me but how future components might interact with it. The JSON settings system and modular folder structure embody this shift and have changed how I plan.

My time management also changed. By working in uninterrupted blocks, I noticed I was far more productive than spreading tasks over days or weeks, as I had done while managing daily work during my placement. This approach helped me maintain momentum and uncover issues earlier. It also made it easier to reassess priorities. For example, delaying potential expansion, such as ARMA modelling, to instead strengthen the system's architecture so those features can be added later without refactoring.

Overall, this project strengthened my ability to plan realistically, manage increasing scope, anticipate integration challenges, and think beyond purpose-built solutions. It pushed me toward a more scalable, system-oriented way of working, which will only improve how I approach future projects.

Top Tips for Students

Such have been abbreviated in the updated JSON project plan.

Start early and iterate.

I initially set overly ambitious goals and struggled with getting visualisations into the GUI. Starting earlier gave me the flexibility to reassess the scope and focus on achievable goals. I also recommend working in focused "sprints," by dedicating continuous blocks of time to your project. This made it far easier to stay immersed in the project and maintain momentum than to dip in and out.

Modularise your code from the start.

Breaking functionality into modules instead of a single long script massively simplifies debugging and improves clarity. A modular structure also makes the project far more scalable, as you can extend or replace components without breaking the entire system.

Use AI strategically.

Use AI as a learning tool, not a shortcut. It's useful for exploring ideas, understanding unfamiliar functions or libraries, and checking your thinking. Make sure you understand the output fully before copying anything. Also be wary of it over engineering simple concepts or tasks.

AI Use

In Code

File Name	Full/ Partial Ownership
etf_analysis.py	Partial Ownership (see code comments)
anlytics.py	Full Ownership
decision_compute.py	Full Ownership
visualisations.py	Partial Ownership (see code comments)
etf_interface.py	Partial Ownership (see code comments)
settings_interface.py	Partial Ownership (see code comments)
data_fetch.py	Full Ownership
settings_loader.py	Full Ownership
static.py	No Ownership (see code comments, AI-generated)
default_settings.json	Full Ownership
main.py	Full Ownership

In Documentation

I used AI to refine the writing style and tone of certain parts of my documentation, I also used Grammarly to correct punctuation and spelling errors.

Conclusion

In summary, this project brought together data sourcing, analytics, visualisation, and a custom GUI to create a modular ETF analysis tool. The project has strengthened my technical skills, improved my approach to planning and design, and given me experience building a complete application rather than isolated scripts.