# Designing a Claude Code–Style Financial CLI Assistant

## Overview

Building a **smart CLI assistant for financial workflows** requires combining large language models (LLMs) with finance-specific tools. The goal is similar to Anthropic’s **Claude Code** (a flexible, agentic CLI coding assistant) but focused on finance tasks instead of general coding[[1]](https://www.anthropic.com/engineering/claude-code-best-practices#:~:text=We%20recently%20released%20Claude%20Code%2C,Claude%20into%20their%20coding%20workflows). This assistant should let users query and analyze financial data, fetch market information, and model portfolios via natural language commands. Key requirements include:

* **Multi-LLM Support** – The system can route requests to different LLMs (GPT-4, Claude, local models, etc.) using a unified API/key management (the “Redpill” key system).
* **Financial Toolkit Integration** – Deep integration with the **OpenBB SDK** and related finance libraries to provide data and analytics (e.g. stock prices, technical indicators, portfolio metrics).
* **Local-First Design** – Runs locally by default for privacy (with Python environment and local data/LLM options), but extensible to server/cloud for heavier tasks or multi-user access.
* **Open Source & Modular** – A clean, modular architecture (CLI interface, model routing layer, tool integration layer, etc.) that can be released publicly, with security and licensing best practices in mind.

## Architecture and Modular Design

**High-Level Architecture:** The assistant can be structured in layers, each responsible for a part of the workflow:

* **CLI Interface Layer:** A command-line front-end that the user interacts with. This could be an interactive REPL-style prompt (e.g. assistant>) or commands invoked with flags. The CLI parses user input and triggers the AI logic. A framework like **Typer** or Click can be used to handle command parsing and subcommands (for example, a mode to enter an interactive chat vs. one-off commands). The CLI layer should remain thin, mostly forwarding user queries to the core AI engine and displaying results.
* **Core AI Engine (LLM Orchestration):** This is the “brain” – it routes the user’s request to an LLM and orchestrates tool usage. It includes:
* **Prompt Manager & Agent**: Constructs prompts that include relevant context or tool instructions. For financial use, the system prompt can define the assistant’s role (e.g. *“You are a financial research assistant with access to market data and analysis tools.”*). If using an *agentic* approach, the AI can decide to call toolkit functions (OpenBB) as needed. The design can draw from Claude Code’s agent model, where the LLM can issue actions (like running a tool or command) and get results back. This may be implemented via an event loop: the assistant model suggests an action (e.g. “fetch stock price for AAPL”), the engine executes that action using OpenBB, then feeds the result back into the model’s context before finalizing the answer.
* **Model Routing Layer:** Handles which LLM to use for a given query. This could be rule-based (e.g. always use a default model, or choose based on query size or user preference) or dynamic. The “Redpill router” approach implies a unified interface to multiple LLMs – for instance, an AI service class with multiple providers loaded (OpenAI API, Anthropic API, local model) and a mechanism to select or fall back between them. *For example, RedpillAI uses a dual-provider strategy: a primary custom model and OpenAI as fallback*[*[2]*](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L86-L90)*.* We can implement a similar chain: try the preferred model first, and if it fails or is unavailable, fall back to an alternate[[3]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L200-L209)[[4]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L210-L219). This increases reliability and allows using cheaper/free models when possible, with a backup for tough queries.
* **Financial Toolkit Integration Layer:** Modules that interface with financial data sources and libraries. The primary one is the **OpenBB SDK**, which provides access to 350+ financial data providers and analysis functions[[5]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L71-L79). This layer exposes functions like get\_stock\_price(ticker), get\_historical\_crypto(symbol), run\_technical\_analysis(data), etc., wrapping the calls to OpenBB or other finance APIs. By abstracting these as functions, the AI Engine can invoke them either directly (via code) or through an agent tool interface. For example, if the user asks “What is the current price of Bitcoin and how does it compare to last week?”, the AI could call an OpenBB function to get BTC price data and then perform the comparison. These functions should be designed to be **async-compatible** (since LLM calls and data fetches might be async). In a local CLI, they can run synchronously as well – Redpill’s OpenBB service shows an example of wrapping async OpenBB calls to run in a sync context safely[[6]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L104-L113).
* **Local vs. Server Agents:** By default, the above components run locally – the CLI and AI Engine on the user’s machine, calling local tools (and possibly local LLMs if configured). However, we can design an optional **server mode** or remote agent: the core AI engine could be exposed via an API (e.g. a FastAPI server) so that a user could connect a GUI or allow multiple clients to use the same backend. The architecture should separate the CLI frontend from the logic so that in the future a web interface or a VSCode extension could hook into the same engine. For instance, one could include a mode where the CLI spins up a background server for heavy computations or centralized logging. In local mode, all data and computation stay on the user’s machine (enhancing privacy). In server mode, ensure secure client authentication and rate limiting if it’s public-facing.

**Diagram – Modular Overview:** *(If we sketch the architecture, it would look like:)*

* **User CLI** → (Input) → **CLI Parser** → **AI Engine** (chooses model, builds prompt) → **LLM** → **AI Engine** (parses LLM output for actions) → **OpenBB/Finance Tools** → (results back) → **LLM** → **Output to User**.

This modular approach ensures each part (CLI, LLM interface, finance logic) can be developed and tested in isolation.

## Key Open-Source Components and Libraries

When building this from scratch, we can leverage several open-source components instead of reinventing the wheel:

* **LLM Model Wrappers:** To support multiple LLM providers, consider using a unified SDK or abstraction:
* *Direct SDKs:* The straightforward approach is to use provider-official SDKs (e.g. OpenAI’s openai Python package, Anthropic’s anthropic client) behind a common interface. You might write your own wrapper classes (e.g. OpenAIProvider, AnthropicProvider) each implementing a chat\_complete(prompt, \*\*kwargs) method. This is the approach in Redpill’s AI service, which defines provider classes and then iterates through them in a list[[3]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L200-L209). The benefit is control and minimal dependencies.
* *Multi-provider API/SDK:* Projects like **OpenRouter** or **LiteLLM** provide a single API that routes to many models. For example, *LiteLLM is an open-source library that lets you access multiple LLMs through one interface, using one unified API key and tracking usage across providers*[*[7]*](https://www.reddit.com/r/LLMDevs/comments/1irpjjk/too_many_llm_api_keys_to_manage/#:~:text=jellyouka). Using such a library could simplify integration, though it might abstract away some fine-tuning of prompts per model.
* *Local model support:* If offline use is important, consider libraries for running local models (like **llama-cpp** for LLaMA 2, or **Ollama** for managing local models). The CLI could detect if local model weights are available and use them when no API key is provided. Wrappers for HuggingFace Transformers pipelines or FasterTransformer can be used for local inference, though these may require significant resources for large models.
* **Prompt Orchestration & Agent Frameworks:** To implement “agentic” behavior (the LLM deciding to use tools), frameworks like **LangChain** or **Haystack** can help manage complex chains of prompts and tool calls. LangChain, for instance, has an agent tooling system where you can define tools (Python functions with descriptions) that the LLM can invoke via formatted prompts. This could accelerate development of the tool-using logic. However, these frameworks add complexity and may not be strictly necessary for a custom solution. An alternative is a lightweight custom approach: e.g., after getting an LLM’s response draft, check for certain triggers or special syntax that indicate a tool should be used, then handle it in code. The design should remain **flexible** and “unopinionated” about workflow, akin to Claude Code’s philosophy[[8]](https://www.anthropic.com/engineering/claude-code-best-practices#:~:text=Claude%20Code%20is%20intentionally%20low,develop%20their%20own%20best%20practices). This means the assistant shouldn’t be rigidly stuck in a single chain-of-thought loop; users might sometimes want a simple direct answer (no tools), and other times allow the assistant to autonomously perform multi-step analysis. A hybrid approach could let the user toggle an “agent mode” on or off (as some projects do). For example, the open-source **PAR GPT** CLI tool can run in a basic LLM mode or an agent mode with tool use[[9]](https://github.com/paulrobello/par_gpt#:~:text=,optimized%20resource%20cleanup%20and%20caching).
* **CLI Framework:** Use a Python CLI framework to handle arguments, subcommands, and help text. **Typer** is a great choice – it’s built on Click but uses Python type hints for clean syntax[[10]](https://github.com/fastapi/typer#:~:text=Typer%2C%20build%20great%20CLIs,Based%20on%20Python%20type%20hints)[[11]](https://www.realworldml.net/blog/let-s-build-an-ai-coding-assistant#:~:text=Let%27s%20build%20an%20AI%20coding,Ollama%20to%20run%20LLMs%20locally). Typer can easily define commands, options (like --model to choose an LLM, or --no-agent to disable agent tools), and even an interactive mode. For example, using Typer we can create a CLI app:

import typer  
app = typer.Typer()  
  
@app.command()  
def ask(query: str, model: str = typer.Option("gpt-4", "--model", "-m", help="LLM model to use")):  
 """Ask a financial question to the assistant."""  
 answer = run\_assistant(query, model\_name=model) # run\_assistant encapsulates AI Engine logic  
 print(answer)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 app()

This gives us a command ask "What is AAPL stock's P/E ratio?" --model claude out of the box. Typer also supports subcommands if we want to organize features (e.g. history command to show past queries, config to set API keys).  
Other CLI frameworks: *Click* (underlying Typer) or *Argparse* (in stdlib) could be used, but Typer’s developer experience and built-in help generation are superior for this use case.  
Additionally, libraries like **Rich** or **Textual** can improve the output formatting in the terminal. Rich can add color, tables, even progress bars. For a financial assistant, nicely formatted tables for stock data or colored alerts for significant changes can enhance UX. (PAR GPT uses Rich for terminal output styling[[12]](https://github.com/paulrobello/par_gpt#:~:text=,specific%20lazy%20loading%20system%20extending).)

* **Finance and Data Libraries:** Aside from OpenBB, consider other Python libraries depending on needs:
* **Pandas** for data manipulation (likely used under the hood by OpenBB too, but you might use it for custom calculations).
* **yFinance**, **Alpha Vantage**, etc., if directly accessing specific APIs (OpenBB already wraps many of these, so you might not need to import them separately).
* **Matplotlib/Plotly** if you plan to have the assistant generate charts or visualizations (these could be output as ASCII graphs or saved images if the CLI is later extended to a GUI). This might be beyond MVP, but worth noting as a possible extension (e.g. “plot the last 1 month price trend for TSLA”).

By using and comparing these components, you can accelerate development. For example, to orchestrate prompts and tools, one might start quick and simple (no heavy framework) and later, if needed, incorporate LangChain for more complex flows. The brief should weigh the trade-offs (e.g. LangChain’s power vs. complexity). In an **MVP, simplicity and reliability** of each component is key.

## Integration with OpenBB SDK (Financial Workflows)

**Why OpenBB:** OpenBB provides an extensive SDK to access financial data and analytical functions. It’s essentially the engine of the OpenBB Terminal (formerly Gamestonk Terminal) exposed as a Python package[[13]](https://medium.com/geekculture/openbb-sdk-do-your-investment-research-with-python-e68da42fedf3#:~:text=OpenBB%20SDK%3A%20Do%20your%20Investment,data%20sources%20from%20one). Integrating it means we instantly get the ability to pull data from Yahoo Finance, FinancialModelingPrep (FMP), Polygon, CoinGecko, and hundreds of other sources via a unified interface[[5]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L71-L79). We also get utilities for technical analysis, portfolio metrics, and more, without having to implement those from scratch.

**Best Practices for Integration:**

* **Setting Up API Keys:** Many OpenBB data providers require API keys (for example, FMP, Polygon.io, AlphaVantage for stocks; CoinGecko for crypto, etc.). We should allow the user to provide these in a config file or environment variables. The RedpillAI project demonstrates this by loading keys from the environment and assigning them to OpenBB’s credential manager at startup[[14]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L80-L88). For instance:

# Pseudocode: configure OpenBB with user API keys  
import os  
from openbb import obb  
if "FMP\_API\_KEY" in os.environ:  
 obb.user.credentials.fmp\_api\_key = os.environ["FMP\_API\_KEY"]  
# ... similarly for Polygon, AlphaVantage, etc.

Doing this early (when the assistant starts) ensures that subsequent OpenBB calls automatically use those keys. It’s good to wrap this in a function and call it on startup. Also, handle missing keys gracefully – OpenBB might have some free data without keys or you may notify the user which functionalities will be limited.

* **Using OpenBB Functions:** The OpenBB SDK functions are namespaced (for example obb.crypto.price.historical to get historical crypto prices, or obb.stocks.fa.ratios to get financial ratios). You can wrap commonly needed calls in the assistant’s tool functions. For example, to get a live price:

from openbb import obb  
def get\_stock\_price(ticker: str) -> float:  
 result = obb.stocks.price(ticker=ticker) # hypothetical function  
 return result.price if result else None

In practice, Redpill’s OpenBB service shows that a single “price” query may involve trying multiple providers. They configure a priority list of providers for each domain (crypto, equity, news) and attempt them in order[[15]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L72-L80)[[16]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L131-L139). For instance, to fetch a crypto price, try Yahoo Finance first, if that fails try FMP, then Polygon[[17]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L73-L81)[[18]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L133-L141). Implementing similar fallback logic ensures robustness if one API is down or returns incomplete data.

* **Asynchronous Calls:** The OpenBB SDK appears to support async operations (some calls might be asynchronous). In a CLI tool, you might call them synchronously for simplicity. But if performance is a concern (e.g. fetching data and querying LLM in parallel), leverage asyncio. For example, fetch multiple data points concurrently (stock price, news, etc.) before composing the answer. Redpill’s backend used an asyncio event loop and even a threadpool to run OpenBB calls without blocking[[6]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L104-L113). For an initial version, synchronous calls are fine – just be mindful if the CLI might hang on long data pulls, and possibly provide feedback (a spinner or message) to the user.
* **Data Volume and Formatting:** Some OpenBB functions return quite a bit of data (e.g. a full historical time series or a large table of financial ratios). The assistant should be prepared to **summarize or truncate** this data for the user, rather than dumping huge tables raw. This is where the LLM shines: you can fetch the raw data (perhaps as a Pandas DataFrame), then prompt the LLM to analyze or summarize it. For example, “Here is the last 30 days of TSLA prices: <dataframe> – Please calculate the 5-day vs 30-day average and tell me if the trend is upward.” The CLI could also support an option to output raw data (CSV or JSON) if the user explicitly wants it, but the default should be an informative summary.
* **OpenBB as a Foundation:** In practice, OpenBB could serve as the *backend “analytical engine”* while the LLM is the *frontend “language engine”*. A tight integration means the assistant can handle queries like *“Compare Apple and Microsoft’s P/E ratios and give me some insight”* by:
* Using OpenBB to fetch the P/E ratio of Apple and Microsoft via something like obb.stocks.fa.ratios("AAPL") and (... "MSFT") (which might return data including P/E, or using a specific function if available),
* Then feeding those numbers to the LLM with a prompt template that asks for comparison insight.
* Finally, printing the LLM’s answer, possibly with a small table of the numbers for transparency.
* **Error Handling & Validation:** Finance data can be messy. The integration layer should catch exceptions (e.g., symbol not found, API limit reached) and either retry with a fallback or inform the LLM that data is unavailable. This prevents the whole assistant from crashing on a bad query. For instance, Redpill’s get\_crypto\_price returns None if all providers fail and prints a warning[[19]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L153-L161) – our CLI could intercept that None and have the LLM respond, *“Sorry, I couldn’t retrieve that data.”* or attempt a different approach. Logging these events (perhaps to a debug log file) would help in improving reliability over time.
* **Extending Beyond OpenBB:** While OpenBB is the main toolkit, keep the integration layer open for other tools. For example, integrating a **calculator or NumPy** for custom calculations, or a **news API** to get headlines for a stock (OpenBB may have some news functionality too). If using an agent approach, each of these can be a “tool” the agent is allowed to call. During development, you might manually expose a few safe functions and gradually add more. Always test the outputs to ensure the LLM is using the tools correctly.

## Multi-LLM Support with Redpill API Routing

Supporting **multiple LLM providers securely** is crucial for flexibility. The “Redpill API key system” likely refers to a method of abstracting away individual model API keys behind a single routing service or configuration. In practice, here’s a strategy to achieve this:

* **Unified Configuration:** Use a config file (like a .env or a YAML) to store all the potential API keys: e.g. OPENAI\_API\_KEY, ANTHROPIC\_API\_KEY, etc., as well as a flag for which service to prioritize. For example, RedpillAI’s config has use\_redpill\_ai: True/False and both a Redpill API key and OpenAI key present[[20]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/config.py#L26-L34)[[21]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/QUICK_API_SETUP.md#L5-L14). In our CLI, we could have a setting like preferred\_provider = "redpill" or "openai" or "local", etc.
* **Redpill Router Usage:** If the user has a Redpill service (perhaps Redpill is an external API that itself routes to models), the CLI can simply send all prompts to the Redpill API endpoint. Redpill might choose the model based on its own logic (possibly it has endpoints to specify model or automatically pick). This offloads the multi-LLM handling to that service. However, we’d still allow direct usage of other providers in case Redpill is not available. For example, **primary**: Redpill (one API to rule them all), **fallback**: direct OpenAI or others. This is exactly how RedpillAI does it: *“Prefer Redpill.ai over OpenAI when available”*[[22]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/config.py#L32-L36), but if not, use OpenAI.
* **Custom Routing Logic:** If not using an external router service, implement the logic internally. A simple version is the **provider chain** pattern: try one provider, catch exceptions, then try next. The RedpillAI AIService snippet shows an async process\_conversation going through a fallback\_chain list of providers until one returns a response[[3]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L200-L209)[[4]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L210-L219). We can do similar (synchronously if not using async):

providers = [RedpillProvider(), OpenAIProvider(), LocalLLMProvider()]  
response = None  
for provider in providers:  
 try:  
 response = provider.generate(prompt) # e.g. a chat completion call  
 break # break on first successful response  
 except Exception as e:  
 print(f"Provider {provider.name} failed: {e}")  
 continue  
if not response:  
 raise RuntimeError("All LLM providers failed to respond")

In this pseudo-code, each provider class internally knows how to call its API (using the keys from config). This design makes it easy to plug new providers – just add a new class and add it to the list with appropriate priority.

* **Secure Key Storage:** Avoid hard-coding keys or putting them in the code repository. Use environment variables or an encrypted config file. The CLI can provide a command like assistant config set-key openai YOUR\_KEY which stores the key in a local config (possibly using the OS keychain or at least a file with proper permissions). If multiple keys are set, the user can switch the active model via CLI options or config. The RedpillAPI Quick Start suggests adding keys to .env and even provides a test script to verify connectivity[[21]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/QUICK_API_SETUP.md#L5-L14)[[23]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/QUICK_API_SETUP.md#L34-L42) – we can emulate that by including a --test-connection command that tries a simple completion on each configured provider.
* **Abstraction vs. Direct Control:** Decide whether to expose model choice to the user or make it automatic. A power user might want to explicitly say --model gpt-4 vs --model llama-2. We can map those choices internally to providers. For instance, --model llama-2 could route to a local LLaMA2 13B via LocalLLMProvider. In contrast, a user who doesn’t care could just rely on a default (which might be “whatever key you provided”). A nice approach is to have **sensible defaults with override options**. E.g., default to Redpill if its key is present (assuming it routes to a strong model), otherwise default to OpenAI if that key is present, etc., but allow an option or config setting to force one.
* **Security in Multi-Provider Context:** If this CLI is public/open-source, you must treat each API integration carefully:
* **Key Privacy:** Inadvertently sending a prompt intended for one API to another could leak a key if not careful (though unlikely if segregated). But more practically, ensure that if users share bug reports or logs, keys are scrubbed.
* **Usage Limits and Cost Control:** Provide settings for users to limit usage per provider (maybe warn if a response is over N tokens). The assistant could estimate cost of a query if model is known (some tools even show pricing info[[24]](https://github.com/paulrobello/par_gpt#:~:text=,file%2C%20url%2C%20and%20web%20search)). This is a nice-to-have feature for a later iteration to prevent surprise API bills during large analyses.
* **Dependency on External Services:** Redpill or OpenAI might not be available (network issues or service outages). The fallback system handles some of this. Additionally, offline models guarantee the assistant still functions (perhaps at reduced capability) when no internet or API is accessible. Document clearly to users which models run locally vs require internet, so they can make an informed choice.
* **Testing Across Models:** Because different LLMs have different strengths (for example, GPT-4 is excellent at detailed reasoning, Claude might handle longer context, a local model might have limited knowledge), test the assistant’s prompts and tool use across providers. You may need slight prompt tweaks or parameters per model. Your routing layer could adjust things like temperature or max tokens depending on provider defaults. Abstracting these differences is tricky but aim for a **common denominator interface** so that switching models doesn’t break the experience.

In summary, a Redpill-style multi-LLM system will make the CLI far more versatile. It ensures longevity (new models can be added) and gives users control over factors like cost, speed, and privacy (by choosing one model over another). By following the pattern of **primary + fallback providers**[[3]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L200-L209), we can achieve seamless switching – e.g. if the primary API errors out or times out, the user might only notice a slight delay as the next one takes over[[4]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L210-L219). Logging which model was used (in a verbose mode) can be helpful for transparency.

## Security, Privacy, and Licensing Considerations

When releasing a public-facing CLI tool that integrates AI and financial data, there are several important considerations:

* **Command Execution Safety:** If your assistant is truly *“Claude Code–style”*, it may execute shell commands or code as part of its agent behavior. This is powerful for automation (e.g. running a Python snippet to calculate something). However, it opens the door to accidental or malicious operations (imagine the model misinterprets a request and runs rm -rf \*). Anthropic’s Claude Code addresses this by requiring **user permission for any system-altering action**[[25]](https://www.anthropic.com/engineering/claude-code-best-practices#:~:text=By%20default%2C%20Claude%20Code%20requests,git%20commit). We should adopt a similar safety net:
* Maintain an **allowlist/denylist** of operations. For example, allow read-only actions (fetch data, run calculations) by default, but prompt the user to confirm if an action could modify files or system state. The user could override by explicitly allowing certain commands in a config (power users) or on a per-session basis.
* Provide a **“dry-run” or “explain” mode** for potentially dangerous actions, where the assistant says what it *wants* to do and asks for confirmation. This keeps the user in control.
* Consider sandboxing execution. For instance, if the assistant needs to run a piece of Python code (maybe to use a library not integrated or to do heavy computation), run it in a restricted environment. The PAR GPT tool implements a Docker sandbox for safe code execution[[26]](https://github.com/paulrobello/par_gpt#:~:text=,support%20for%20safe%20code%20execution). For our MVP, sandboxing might be too complex, but we can at least use Python’s exec carefully (restrict globals/builtins, timeouts) or subprocess with confined permissions.
* **Data Privacy:** Financial queries might involve sensitive data – e.g. “Analyze my portfolio (positions attached in a file) and suggest improvements.” If the user provides personal data, the assistant should not leak it. Running locally by default is a big advantage: none of the user’s data or queries need to leave their machine *unless* they opt to use an API model. We must communicate this clearly:
* Document that using cloud LLM providers will send prompts (which may include financial data) to those third-party APIs. Encourage anonymizing or using local models if data is highly sensitive.
* Possibly offer a mode to **mask identifiable info** (this is advanced, but an idea: automatically detect tickers or names and allow the user to replace them with placeholders for queries that go to cloud).
* Do not log full query contents by default, or if logging, store it only locally. If a server mode is used, ensure TLS and proper auth, and consider not storing queries on the server unless necessary.
* **API Key Security:** Users will be storing API keys (for LLMs, for OpenBB data sources, etc.). Best practices:
* Encourage use of environment variables or a local encrypted storage. If using a config file, instruct users not to commit it to any repo.
* If you provide a way to input keys via the CLI, ensure the input is not logged or echoed.
* When publishing the tool, *never include any real keys* (the example keys in the Redpill config[[20]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/config.py#L26-L34) are dummy placeholders). Also be mindful if you fork or reuse any code – scrub credentials.
* **Licensing:** Choose a license that is compatible with your dependencies and intended usage. OpenBB SDK is published on PyPI and is likely permissive (OpenBB Terminal was MIT License as far as known). Confirm that: if it were GPL, it could force your project to be GPL – but OpenBB’s website/blog suggests it’s meant for broad use, likely MIT. For any other libraries (LangChain is MIT, Typer is MIT), ensure your license is compatible. MIT or Apache-2.0 are good choices for open-source to allow commercial and community use. If you incorporate any code from others (like adapting part of RedpillAI or PAR GPT), respect their licenses (RedpillAI appears proprietary or not explicitly licensed on GitHub – if it’s not licensed, treat it as all rights reserved and do not copy code; just learn from it). PAR GPT is MIT licensed[[27]](https://github.com/paulrobello/par_gpt#:~:text=), so using snippets or ideas from it is fine under MIT with attribution.
* **Financial Data Licensing:** Note that some financial data sources have usage policies. If your tool fetches data (especially if caching or redistributing it), ensure compliance. For example, certain API free tiers are for non-commercial or personal use only. As an open-source tool, you might include disclaimers that the user is responsible for abiding by their data provider’s terms (since they plug in their own keys).
* **User Authentication & Permissions (if server mode):** If down the road you allow the CLI to run as a multi-user server (e.g. a chatbot service), implement authentication, and only allow trusted users to execute potentially dangerous operations. Logging user actions and having an audit trail is important in that scenario. For the local single-user CLI, the OS security (file permissions) is usually sufficient – just don’t create any additional vulnerabilities (like an insecure debug server open on a port).
* **Securing Outputs:** Be cautious about the model returning potentially problematic content. In a financial context, this could be erroneous advice or even compliance issues (e.g. financial predictions or recommendations might have legal implications). While a full discussion on AI output bias and accuracy is beyond scope, as a safety measure:
* Provide a disclaimer that this is a tool for research and not financial advice.
* Possibly allow the user to set the model to be more conservative or only factual. Using system prompts to remind the model to be truthful and avoid speculative recommendations is advisable.
* If the assistant integrates news or web data, beware of injecting unverified info. It might be wise to cite sources or at least present data before conclusions (similar to how this report itself cites sources). This transparency can build trust and allow the user to double-check the AI’s statements.

In summary, treat security and privacy as first-class concerns from the start, not as afterthoughts. The combination of a powerful LLM agent and access to system/financial data means we have to guardrails. Borrowing Claude Code’s approach of conservative defaults (ask for permission, allow only what’s needed) keeps things safe[[28]](https://www.anthropic.com/engineering/claude-code-best-practices#:~:text=By%20default%2C%20Claude%20Code%20requests,git%20commit). As we open-source the project, these considerations also boost confidence for others to use and contribute to the tool.

## Development Roadmap and MVP Features

Finally, outline a development plan with milestones from MVP to a robust assistant:

**Phase 0: Project Setup**  
- Choose the tech stack and set up version control, packaging, etc. Likely Python-based, so initialize a Python project with Typer and required libs. Set up the environment configuration (e.g. .env parsing for API keys). Draft a README with how to install and set keys – this helps clarify the setup process early.

**Phase 1: Basic Q&A with Single Model**  
- Implement a simple CLI command (e.g. ask) that takes a question and returns an answer from a hard-coded default LLM (say, OpenAI GPT-4, using an API key). At this stage, no tool usage – the goal is to validate the prompt/response flow. For example, the user asks “What is EBITDA?” and the assistant (GPT-4) returns a definition.  
- Ensure that the CLI prints the answer nicely. Test different output formatting (maybe use Markdown in answers and render via Rich, etc.).

**Phase 2: Integrate OpenBB for Data Retrieval**  
- Choose a couple of straightforward finance tasks to implement via OpenBB. For instance: - **Live stock/crypto price lookup** – user asks price of a ticker, the assistant fetches via OpenBB and includes it in answer. - **Basic financial metric** – e.g. P/E ratio or market cap for a stock, via OpenBB’s financial statements or ratios. - Implement these as either direct function calls triggered by keyword or as an agent tool. The simplest approach: after getting the user query, if certain keywords are present (like “price of” or “PE ratio”), call the appropriate OpenBB function, then feed both the question and the fetched data to the LLM to compose the answer. This is a form of prompt orchestration that can be rule-based initially.  
- Example: User: "Tell me the current price of BTC and ETH" → Code sees “price” + “BTC, ETH” → calls obb.crypto.price for both → gets prices → then prompts LLM: “The user wants the current price of BTC and ETH. BTC is $X, ETH is $Y. Respond with a brief statement.” → LLM answer → CLI prints “Bitcoin is $X and Ethereum is $Y as of now…”  
- Focus on making this flow reliable for the chosen tasks. This will prove that the assistant can use live data, which is a big MVP win.

**Phase 3: Multi-LLM Support**  
- Introduce the model routing layer. Implement classes for at least two providers (e.g., OpenAI and one open-source model or Anthropic Claude if available). Allow the user to specify --model or configure which to use. - If the Redpill service/API is available to you, integrate it here as the primary. If not, simulate a similar behavior by perhaps using OpenRouter or just manual selection. The key is that the architecture can easily add providers. - Test that both (or all) configured models can handle the tasks from Phase 2. Adjust prompts if needed per model. - Add fallback logic: e.g., if using Redpill primary and OpenAI secondary, cause a simulated failure of Redpill to see that OpenAI then answers. Verify it works as expected.  
- At this stage, also implement any needed **logging or monitoring** of model usage (just to a file or console) to help with debugging, and potentially to summarize token usage or costs (maybe optional verbose output).

**Phase 4: Expand Financial Capabilities**  
- Add more OpenBB-driven features: e.g., historical price trends and simple analysis (“How did AAPL perform in the last 6 months?” – fetch historical data and perhaps calculate return or volatility), portfolio analysis (“Given these 5 stocks with weights, what’s the overall performance?” – use OpenBB portfolio functions or manually compute if needed), and news queries (“Any recent news on Tesla?” – OpenBB or another API for news and summarize via LLM). - This phase is about broadening the skill set of the assistant. Each new capability might involve writing a new function in the integration layer and updating the prompt or agent logic to invoke it. - Ensure documentation is updated for each new feature (both inline in code and in user-facing docs). By now, you might start writing a usage guide for the README with examples of queries that work.

**Phase 5: Agent Mode and Autonomy**  
- If not already implemented as such, now consider a more **agentic approach** where the LLM decides on tool usage. Possibly integrate a lightweight LangChain agent or a custom loop: The model is prompted with a set of available tools (e.g. a JSON or markdown list of tools and when to use them). The user query is given, and the model’s output is parsed for tool commands (you can define a format like: ToolCall: get\_stock\_price("AAPL")). The engine executes the tool and returns the result to the model’s context, then the model produces final answer. - This is complex, so start with one tool (like price lookup) and one scenario. Evaluate if the model is smart enough with the given prompt to use it. This might require prompt engineering (and possibly fine-tuning in future). - Introduce a mechanism to avoid infinite loops or wrong tool calls. LangChain and others provide patterns for this (stop after N steps, etc.). - Also, implement the confirmation mechanism for any risky actions in agent mode. For example, if the agent tries to write to a file or execute code, intercept and ask user. This could simply be: if the chosen tool is not in a safe list (like anything beyond read-only data retrieval), pause and prompt user in CLI “Allow the assistant to run XYZ? (y/N)”.

**Phase 6: Hardening and Testing**  
- Write unit tests for key functions (e.g., test that get\_stock\_price returns a number for known tickers, test that model routing falls back correctly when primary fails by mocking an exception, etc.). - Test the CLI UX: how does it behave on invalid input or if OpenBB has no data? Make the error messages user-friendly. - Security review: try to “jailbreak” or confuse the agent in ways that could be harmful, and ensure the safeguards hold. For example, ask the assistant to execute a shell command in a trick phrasing and see if it wrongly executes without confirmation. Adjust allowlist or prompt to mitigate issues found. - Performance tuning: if responses are too slow, consider where you can optimize (perhaps caching last fetched data, using streaming responses from the LLM API if possible to show partial answer, etc.).

**Phase 7: Documentation and Release**  
- Prepare comprehensive documentation: a **README** with installation, setup of API keys, and example usage for each feature. Also include a **CHANGELOG** and roadmap for future ideas (showing the open-source community that the project is active and inviting contributions). - Add a **LICENSE** file (e.g. MIT license text) and ensure all dependencies are properly acknowledged. - If open-sourcing on GitHub, set up issue templates, and maybe CI for tests. Possibly deploy a PyPI package for easy install (pip install finance-cli-assistant for example). - Announce the release in relevant communities (perhaps OpenBB community, Reddit r/FinanceAI, etc.), being transparent about capabilities and limitations (especially that it’s not financial advice, etc.).

**MVP Feature Set:**  
By the end of the above roadmap, the MVP (Minimum Viable Product) should include at least:

* **Interactive Q&A**: The user can ask general finance questions (explanations of terms, etc.) and get answers from an LLM.
* **Real-time Data Fetching**: The assistant can retrieve current asset prices (stocks, crypto) and simple stats via OpenBB, and incorporate them into answers.
* **Basic Analysis**: It can perform straightforward analysis like comparing two assets, summarizing a price trend, or calculating a metric (with the help of the LLM for explanation).
* **Multi-Model Choice**: The user can configure which LLM to use (at least switch between two, e.g. a local and a cloud model). The system will handle the selection and any fallback.
* **Secure CLI Operations**: The CLI should have measures to prevent unintended destructive actions, and protect user’s secrets and data as discussed.
* **Extensibility Hooks**: Even if not fully fleshed out, the code structure should make it easy to add new tools (new OpenBB functions), new models, and possibly plug into other interfaces. This means having clear module boundaries and some documentation for developers.

Beyond MVP, features like advanced portfolio optimization, strategy backtesting, or voice input/output could be explored, but they can be future work once the foundation is solid. The outlined roadmap and design focus on getting a **functional, safe, and user-friendly CLI assistant** out the door, which users and contributors can then build upon.

By following this plan, we leverage proven components and prior art (Claude Code’s agent model, OpenBB’s data powerhouse, Redpill’s multi-LLM approach) to create a powerful financial assistant. The end result will be a tool that feels like having a market research analyst at your command line – one who can pull in fresh data, crunch numbers, and explain insights in real-time. With careful engineering and thorough testing, this CLI assistant can become an invaluable open-source project for tech-savvy investors, analysts, and researchers alike.

**Sources:**

* Anthropic Claude Code design and safety practices[[1]](https://www.anthropic.com/engineering/claude-code-best-practices#:~:text=We%20recently%20released%20Claude%20Code%2C,Claude%20into%20their%20coding%20workflows)[[29]](https://www.anthropic.com/engineering/claude-code-best-practices#:~:text=c,allowed%20tools)
* RedpillAI architecture and multi-LLM strategy[[3]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L200-L209)[[4]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L210-L219)
* RedpillAI OpenBB integration approach[[14]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L80-L88)[[16]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L131-L139)
* PAR GPT open-source CLI (Typer usage and security)[[12]](https://github.com/paulrobello/par_gpt#:~:text=,specific%20lazy%20loading%20system%20extending)[[26]](https://github.com/paulrobello/par_gpt#:~:text=,support%20for%20safe%20code%20execution)
* Community insight on multi-LLM management (LiteLLM unified API)[[7]](https://www.reddit.com/r/LLMDevs/comments/1irpjjk/too_many_llm_api_keys_to_manage/#:~:text=jellyouka)
* RedpillAI configuration for dual AI providers[[2]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L86-L90)[[30]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/config.py#L32-L35)
* OpenBB SDK capabilities and integration details[[5]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L71-L79)

[[1]](https://www.anthropic.com/engineering/claude-code-best-practices#:~:text=We%20recently%20released%20Claude%20Code%2C,Claude%20into%20their%20coding%20workflows) [[8]](https://www.anthropic.com/engineering/claude-code-best-practices#:~:text=Claude%20Code%20is%20intentionally%20low,develop%20their%20own%20best%20practices) [[25]](https://www.anthropic.com/engineering/claude-code-best-practices#:~:text=By%20default%2C%20Claude%20Code%20requests,git%20commit) [[28]](https://www.anthropic.com/engineering/claude-code-best-practices#:~:text=By%20default%2C%20Claude%20Code%20requests,git%20commit) [[29]](https://www.anthropic.com/engineering/claude-code-best-practices#:~:text=c,allowed%20tools) Claude Code Best Practices \ Anthropic

<https://www.anthropic.com/engineering/claude-code-best-practices>

[[2]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L86-L90) [[3]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L200-L209) [[4]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L210-L219) [[5]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md#L71-L79) architecture.md

<https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/docs/architecture.md>

[[6]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L104-L113) [[14]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L80-L88) [[15]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L72-L80) [[16]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L131-L139) [[17]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L73-L81) [[18]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L133-L141) [[19]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py#L153-L161) openbb\_service.py

<https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/services/openbb_service.py>

[[7]](https://www.reddit.com/r/LLMDevs/comments/1irpjjk/too_many_llm_api_keys_to_manage/#:~:text=jellyouka) Too many LLM API keys to manage!!?! : r/LLMDevs

<https://www.reddit.com/r/LLMDevs/comments/1irpjjk/too_many_llm_api_keys_to_manage/>

[[9]](https://github.com/paulrobello/par_gpt#:~:text=,optimized%20resource%20cleanup%20and%20caching) [[12]](https://github.com/paulrobello/par_gpt#:~:text=,specific%20lazy%20loading%20system%20extending) [[24]](https://github.com/paulrobello/par_gpt#:~:text=,file%2C%20url%2C%20and%20web%20search) [[26]](https://github.com/paulrobello/par_gpt#:~:text=,support%20for%20safe%20code%20execution) [[27]](https://github.com/paulrobello/par_gpt#:~:text=) GitHub - paulrobello/par\_gpt: CLI LLM / Agent multi-tool

<https://github.com/paulrobello/par_gpt>

[[10]](https://github.com/fastapi/typer#:~:text=Typer%2C%20build%20great%20CLIs,Based%20on%20Python%20type%20hints) Typer, build great CLIs. Easy to code. Based on Python type hints.

<https://github.com/fastapi/typer>

[[11]](https://www.realworldml.net/blog/let-s-build-an-ai-coding-assistant#:~:text=Let%27s%20build%20an%20AI%20coding,Ollama%20to%20run%20LLMs%20locally) Let's build an AI coding assistant - Real-World ML by Pau Labarta Bajo

<https://www.realworldml.net/blog/let-s-build-an-ai-coding-assistant>

[[13]](https://medium.com/geekculture/openbb-sdk-do-your-investment-research-with-python-e68da42fedf3#:~:text=OpenBB%20SDK%3A%20Do%20your%20Investment,data%20sources%20from%20one) OpenBB SDK: Do your Investment Research with Python - Medium

<https://medium.com/geekculture/openbb-sdk-do-your-investment-research-with-python-e68da42fedf3>

[[20]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/config.py#L26-L34) [[22]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/config.py#L32-L36) [[30]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/config.py#L32-L35) config.py

<https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/backend/app/config.py>

[[21]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/QUICK_API_SETUP.md#L5-L14) [[23]](https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/QUICK_API_SETUP.md#L34-L42) QUICK\_API\_SETUP.md

<https://github.com/Marvin-Cypher/RedpillAI/blob/4c5456c52f3bc4780d6eb17d77c1478f3767bca3/QUICK_API_SETUP.md>