ChatGPT

Building a Future-Proof Portfolio Investment Strategist

Data Normalization: Focus on Core Logic

Use in-house solutions for data prep. You don't need to deploy an external MCP server just to normalize or standardize your data – that would add unnecessary complexity. The open-source *Model Context Protocol* (MCP) servers in the repo (e.g. an MCP tool for Yahoo Finance data [1]) are great for fetching or interacting with external resources, but basic data normalization (cleaning formats, merging datasets, scaling values, etc.) is typically simpler to handle directly in your code. In other words, treat data normalization as a **trivial but essential** step in your pipeline that you implement internally. This ensures you maintain full control over data quality and format, and it keeps the early development focused on core logic rather than integrating extra services. Your intuition is correct – get your core processes working reliably first, then consider fancy add-ons later.

Leverage libraries, not servers, for now. For example, if Mosychlos is written in Go, you might use Go libraries (or embed a Python snippet if needed) to parse CSVs, handle missing values, or standardize date formats. This keeps your toolchain lean. Unless a specific MCP server offers a *huge* advantage in data prep (which is uncommon), it's better to invest effort in refining your own normalization routines. You can always add MCP-based tools later if a particular data source or transformation is hard to implement from scratch, but initially keep things straightforward. In summary: **focus on core logic and processes rather than external servers for data cleanup**, as you suspected.

Integrating External MCP Tools into Mosychlos

Don't limit yourself to the given list – think value-added. The awesome-mcp-servers list is a goldmine of capabilities (from database access to web browsing to finance APIs). Exposing some of these tools via Mosychlos could enhance its power – for instance, hooking in a Yahoo Finance MCP server to pull live market data [2] or a trading API MCP for future automation. However, *simply embedding others' tools is not a silver bullet.* Use them only where they clearly save you time or add functionality you can't easily build. For example, if you need real-time stock prices or SEC filings, an MCP server from the **Finance & Fintech** category might be useful [3]. But if a capability is trivial to implement directly (or not critical right now), don't complicate your stack with an external tool.

Add tools thoughtfully and plan for the future. Mosychlos can expose external tools (via an MCP client or similar) as needed, but each integration has overhead (running the MCP server, handling auth, etc.). Since this project is currently for your personal use (single-user), you have the freedom to hard-code some integrations (like calling an API directly) to move fast. That said, keep an eye on the *future SaaS vision*: design Mosychlos's architecture in a modular way so that replacing a direct API call with an MCP tool later is straightforward. In fact, **you could eventually convert Mosychlos itself into an MCP server**, exposing its functionality via the standard protocol. (MCP is an open protocol that standardizes how AI models interact with tools [4].) This means in the future other AI clients could interface with Mosychlos as a "portfolio

analysis tool." The good news is that migrating Mosychlos into a standalone MCP service should be trivial if you've built your core engines cleanly – essentially just wrapping your existing logic with MCP's interface. So, **consider both approaches**: use direct integration now for simplicity, but structure your code so you can modularize or "plugin-ize" features later. Not only should you leverage existing tools when they add clear value, but also invest your time where it counts: robust analytics, user experience, and unique features that differentiate your strategist. These will be more worth your money and time than over-engineering tool integrations early on.

Roadmap for a Solid & Futuristic Portfolio Strategist

To build a comprehensive and forward-looking portfolio investment strategist, proceed in structured phases. Here's a detailed roadmap:

1. **Foundation – Project Architecture & Data Model:** Start by setting up a clean project structure. Define the core components: an engine interface, orchestrator, and data models for portfolios and market data. For example, outline how you'll represent holdings, transactions, market prices, etc. Ensure your `internal/engine/orchestrator.go` (or equivalent) can sequence tasks and pass along a central data object (your "bag" of data). Aim for a modular design so components can be added or swapped easily. Since you anticipate future multi-user SaaS use, **design with scalability in mind**: abstract user-specific data and keep state handling flexible so you can later introduce user accounts or a database without massive refactoring. At this stage, also set up version control, basic CI tests, and any needed project infrastructure.

2. **Data Ingestion & Normalization:** Implement robust data ingestion pipelines. Identify your data sources (e.g. Yahoo Finance for prices, perhaps an internal CSV for portfolio holdings, and maybe economic indicators or news feeds for context). Build connectors or use APIs to fetch this data. Then, focus on **data normalization** – unify formats (dates, currencies), handle missing or outlier values, and ensure consistency. This might involve writing parsing functions or using libraries to convert raw inputs into structured tables or objects. By the end of this step, you should have a reliable way to go from raw data (portfolio positions, price history, etc.) to a clean, standardized dataset ready for analysis. Test this on sample data to verify that all inputs are correctly normalized.

3. **Core Analytical Engines Development:** Now, create the key analytical "engines" as separate modules or classes. Each engine should focus on a specific aspect of portfolio analysis – akin to having specialized analysts on a team [5] [6]. For example:

4. **Performance Engine:** Calculates historical performance metrics for the portfolio and its assets (CAGR, cumulative returns, volatility, Sharpe ratio, drawdowns, etc.). It might compare portfolio returns to a benchmark index to gauge alpha.
5. **Risk Analysis Engine:** Evaluates risk exposures – e.g. compute the portfolio's volatility, VaR, beta to market, sector or asset-class concentrations, and correlations between holdings. If the portfolio is multi-asset, assess factors like equity vs bond allocation, or for stocks, use beta or factor exposures.
6. **Fundamental Analysis Engine (optional/future):** If your strategy involves fundamentals, this engine could pull in P/E ratios, earnings growth, or other company financials for each holding and flag any concerns. (This might use an MCP server to fetch data from financial databases, but only if needed.)

7. **Technical/Sentiment Engine (optional/future):** For a *"futuristic"* edge, you might include an engine that gauges market sentiment (using news headlines or social media via an API or LLM) or technical indicators (trend signals, momentum scores for each asset). This is not crucial in early versions, but it's something to plan if you want AI-driven insights beyond pure numbers.

Develop and **unit-test each engine in isolation**. Given an input dataset (from step 2), each engine should output its findings in a structured way (e.g., a JSON or a section of the "bag" containing results like *"Portfolio volatility = X%, Sharpe = Y"*). This modular approach ensures each piece is trustworthy. It also mirrors a multi-specialist approach – like having separate AI agents for risk, performance, etc., which is a proven design for complex AI systems [7]. By the end of this phase, you'll have a collection of analysis outputs covering various angles of the portfolio.

1. **Engine Orchestration & Workflow:** With individual engines built, configure the orchestrator to chain them into a coherent workflow. The orchestrator (possibly your `engine/orchestrator.go`) should call each engine in a logical sequence, passing along the data and accumulating results. For example, the flow might be: *Ingest Data* → *Normalize Data* → *Performance Engine* → *Risk Engine* → *Other Engines…* → *Aggregate Results*. Ensure that intermediate results from one engine can be used by the next if needed (the "bag" of data can hold results like metrics that subsequent engines might reference). If some engines are independent, you could run them in parallel, but serial execution is fine initially for simplicity. The goal is to have a single command (e.g. `mosychlos analyze`) trigger the full pipeline end-to-end. This chained execution is very much in line with common AI orchestration practices – MCP servers and frameworks like LangChain often execute multiple tools in sequence and pass intermediate results along [8]. Once this works, you effectively have an automated "analyst workflow" that takes raw data and produces a thorough analysis.

2. **Report Generation & Narrative (LLM Integration):** Now, build the component that turns engine outputs into a digestible report. This is where Large Language Models can shine. You might have an **LLM-based Report Generator engine** that takes all the quantitative outputs (from the performance, risk engines, etc.) and produces a human-friendly narrative – almost like an Investment Committee write-up. For instance, it can highlight key findings ("Portfolio return YTD is 8%, outpacing the S&P 500 by 2% [9], with volatility slightly above average. The portfolio is heavily weighted in tech, which contributed to both high returns and risk."). Design prompt templates for each section of the report (performance summary, risk assessment, etc.), and feed the LLM the relevant stats using these templates. Since this is for personal use now, you can use a powerful model (GPT-4, Claude, etc.) via API to generate rich analysis text. Ensure to **fact-check the LLM output** against your engine data (you don't want the model hallucinating numbers – one way to mitigate this is to provide the numbers in the prompt and ask it to reason on them rather than letting it guess). The output of this step is a nicely structured report, possibly in Markdown (so it can include bullet points, tables, charts if any, etc.). At first, you might generate a static text report. In the future, you could render it in a web UI or PDF, but focus on content first. By the end of this step, Mosychlos should be able to produce a full "Portfolio Analysis Report" combining quantitative metrics and qualitative insights.

3. **Optimization & Recommendation Engine:** This is a key piece for a "futuristic" strategist – not just analyzing the portfolio's past and present, but suggesting future actions. Develop an engine (or set of engines) for portfolio optimization. This could start simple: for example, use **Modern Portfolio Theory** to suggest an optimal asset allocation given the holdings' historical returns and covariances, or use goal-based heuristics (e.g., reduce any position exceeding X% of the portfolio to improve

diversification, etc.). More advanced: incorporate an optimization library or algorithm to propose trade ideas (e.g., "reduce Tech from 40% to 30%, increase Healthcare…") that would improve some metric (risk-adjusted return, drawdown, etc.). There are open-source projects like *Double Oracle* that perform sophisticated rebalancing and tax-loss harvesting [9] – you might not need that level of complexity initially, but be aware of these capabilities for future enhancements. Start with a basic optimizer: perhaps target an improved Sharpe ratio or realign the portfolio to a target asset allocation. Have the optimizer engine output concrete suggestions (e.g., "sell 5% of Asset A, buy Asset B instead"). Importantly, treat this as a separate step *after* analysis, since it will likely use inputs from the analysis (current weights, risk metrics, etc.). This engine sets the stage for turning analysis into actionable strategy.

4. **"Investment Committee Conclusion" Synthesis:** Once optimization suggestions are available, integrate them into the final report. You might create a concluding section that reads like an investment committee's decision or recommendations. This could be generated by the LLM as well, using a prompt template that takes the optimization engine's output and writes a rationale for the proposed changes. For example, *"Considering the analysis above, the investment committee recommends reallocating the portfolio toward a more balanced mix. Specifically, we suggest trimming positions in X due to their high valuation and volatility, and increasing exposure to Y, which offers defensive characteristics…*". This would resemble how a human strategist concludes a review meeting. You can implement this as part of the report generation engine or as a small additional LLM call focused on the conclusion. The key is that by now you have both the analytical evidence and the recommended actions, so the LLM can merge them into a coherent, authoritative summary.

5. **Testing & Iteration:** With all major components in place (data ingest, multiple analysis engines, optimization, and report generation), thoroughly test the entire pipeline. Use historical or sample portfolios to see if the outputs make sense. Check each engine's numbers for accuracy. Also solicit feedback (even if just your own judgment for now) on the report readability and usefulness. Iterate on engine algorithms (maybe your risk engine needs to add a metric, or your optimize engine needs a constraint) and on prompt phrasing (to get clearer or more concise LLM output). This stage may also involve implementing **logging** – log every step's output, and especially log the prompts & responses of the LLM. This will help debug any weird conclusions and will be invaluable as you refine the system [10]. Treat each run as a chance to improve both the code and the content.

6. **Multi-User Architecture & SaaS Transition (Future):** As a forward-looking step, start planning how to turn this into a SaaS product. This includes selecting a technology stack for a web service (you might keep the Go backend and expose REST/GraphQL APIs for analysis, or run Mosychlos as an MCP server). Think about multi-tenancy: how will you segregate and secure different user's data? Likely you'll introduce a database to store user portfolios, accounts, and analysis results. Also consider an authentication system and an interface (web UI or at least an API that a front-end can call). At this stage, also revisit your orchestrator and engines to ensure they are stateless per run (so that running analyses for different users in parallel won't conflict). The good news is if you followed clean modular design, converting your single-user tool into a web service should be straightforward – mostly adding wrappers and a user management layer. You might also integrate a task queue if you expect heavy analyses, so jobs can run asynchronously. Additionally, plan for **scalability** (containerizing the app, or using cloud functions for each run, etc.) and **observability** (monitoring performance, errors). Essentially, this phase is about taking the solid core you built and making it production-ready for multiple clients. Keep in mind aspects like rate limiting (especially if using third-

party APIs), error handling (what if an API call fails mid-analysis?), and maintaining compliance if you store financial data.

7. **Continuous Improvement & AI Advancements:** Finally, a futuristic strategist should evolve with technology. Keep an eye on new MCP tools or AI models that could enhance Mosychlos. For instance, as more finance-specific LLMs or tools emerge, you can integrate them for better insights. You might introduce a **learning component** – e.g., use reinforcement learning or user feedback to fine-tune recommendations over time. Also consider adding **scenario analysis engines** (simulate how the portfolio might perform under various market conditions or shocks) and **stress tests** as additional modules. These could be both algorithmic and LLM-driven (an LLM could explain, for example, *"If a recession hits, expect X to drop Y%..."* based on training data or provided context). Always incorporate user feedback loops – if this becomes a SaaS, user interaction data can tell you which recommendations were followed or which parts of the report users find most useful. That insight can guide you to refine the models (maybe the next step is a *chatbot interface* where users can query "why did you recommend selling Asset X?" and the system can answer). The roadmap is iterative: as you add features, continuously test and ensure the system remains coherent and reliable. With this phased approach, you'll have built a **solid foundation** first, and gradually layer in the **futuristic** elements (like AI-driven insights, multi-agent orchestration, advanced optimization) in a manageable way.

Engine Chaining and Orchestrator Pattern

Engine chaining is a sound pattern for this application. The fact that you have an `orchestrator.go` chaining multiple engines is actually aligned with best practices in AI tool orchestration. In the MCP context, chaining tools or models in sequence (with outputs feeding into the next step) is a key feature [8]. It allows complex tasks to be broken into simpler, specialized steps – exactly what you're doing. For example, scanning an email, then doing a lookup, then writing a summary can be three tools chained [11]; similarly, you're chaining data prep → analysis → summary, etc. This modular pipeline (also akin to the *pipeline design pattern*) tends to be **easier to debug and extend**. Each engine does one job and passes its result on. If something goes wrong or needs improvement in the workflow, you can pinpoint which engine in the chain is responsible and fix or swap it.

Benefits of the orchestrator approach: It centralizes the flow of logic. All the high-level sequence is in one place (the orchestrator), making it clear how data moves through the system. Adding a new engine is as simple as writing it and then invoking it in the orchestrator in the right spot. The pattern also supports conditional logic if needed – for instance, in the future you might say "if the portfolio has crypto assets, run the crypto analysis engine next." You can handle that branching in the orchestrator cleanly without messing up individual engines.

Potential improvements or alternatives: If your chain grows very large or complex, you might consider an orchestration framework or a rules engine, but that might be overkill. Some AI orchestration frameworks (like LangChain's chains or CrewAI, LangGraph, etc.) provide abstractions for branching, memory, retries, etc., but since you're coding in Go with your own orchestrator, you have full control. That's fine – simplicity is good here. Another angle: if some engines are independent of each other (say a sentiment analysis vs a risk calc), you could execute them in parallel to speed things up, and then join results. Go's concurrency could allow that. But again, premature optimization isn't needed if each run is fast enough sequentially.

In summary, **the engine chaining pattern you have is a good one and likely doesn't need a radical change**. Just ensure your orchestrator remains organized: it should handle errors gracefully (e.g., if one engine fails, do you abort or skip it?), and possibly allow configuration of which engines to run (maybe via flags or config file) so you can easily toggle certain analyses on/off. This pattern is flexible enough to accommodate future growth – for instance, integrating an MCP server call can simply be another engine in the chain (or an engine that delegates to an MCP client). Many robust AI systems use exactly this kind of orchestrator or multi-step approach (for instance, an AI coding assistant might break a task into design, implementation, testing steps [12] [13] – analogous to your separate analysis engines). As long as the chaining is well-documented and each engine's input/output is clear, you're on the right track. No need to change it just for the sake of change.

Key Engines to Develop and How to Chain Them

You should prioritize building certain engines that cover the end-to-end needs of a portfolio strategist. Here are the key ones and how they link together in the workflow:

- **Data Ingestion Engine:** Responsible for pulling in all necessary data – portfolio holdings, asset prices, maybe economic indicators. This engine interfaces with data sources (APIs, databases, files) and outputs standardized data structures. It runs first in the chain. Ensure it gathers not just current snapshot data but historical data if needed for trend analysis. This engine's output (say, a dictionary or struct containing portfolio holdings and a table of historical prices for each asset) feeds the rest of the pipeline.

- **Data Processing/Normalization Engine:** (If needed, this can be part of ingestion or immediately after.) It takes the raw data and performs cleaning steps: sorting, aligning time series, currency conversion, etc. By the end, you have "analysis-ready" data. In many implementations, the ingestion and normalization might be done in one engine, which is fine – the key is that before any complex analysis, this step ensures consistent inputs. Once this is done, the orchestrator passes the clean data to all analysis engines.

- **Performance Analysis Engine:** This engine focuses on returns and performance metrics. Given the clean data, it might calculate things like: one-year return, year-to-date return, CAGR since inception, best/worst month, etc. It should also calculate benchmark comparisons (if a benchmark index or risk-free rate is provided). Additionally, it can compute contribution analysis (which holdings contributed most to performance) and perhaps basic projections (like forward dividend yield if relevant). This engine's results could be stored in the bag as a set of metrics and insights, which the report generator will later turn into prose.

- **Risk Analysis Engine:** After performance, chain in the risk engine. This computes metrics such as portfolio volatility, Sharpe ratio (needs the risk-free rate input), Sortino ratio, maximum drawdown, value-at-risk, etc. It also examines allocation: e.g., asset class breakdown, sector distribution, geographic distribution – anything that speaks to diversification or concentration risks. If the portfolio is equity-heavy, it might compute beta vs the market; if multi-asset, maybe calculate correlation between assets. The risk engine might also flag any position that exceeds a certain threshold (e.g. "Stock ABC is 25% of the portfolio, which is a concentration risk"). Its outputs are risk metrics and any flags/warnings.

- **Portfolio Health/Diagnostics Engine:** This could be a simple rule-based engine that runs after risk to check for common issues. For instance, check if the portfolio aligns with the user's stated objectives or constraints (if those are known – in personal use you might input something like "target 60/40 stock/bond"). It could also ensure no asset is violating some rule (e.g., no single stock > 10% if that's a rule). Basically, a compliance or policy check engine. This is more relevant in a professional context, but including it adds a "committee-like" thoroughness – an investment committee often checks if the portfolio violates any guidelines. If you implement this, it would output a list of any issues or confirm that all's well.

- **Optimization/Recommendation Engine:** (Later in the chain, possibly triggered in an "optimize" run.) This engine takes the outputs of performance and risk (and the raw data) and formulates suggestions. It could be as straightforward as mean-variance optimization to get an efficient portfolio, or as custom as you want (e.g., based on your own criteria). It might use a third-party solver or just greedy heuristics. For now, you can implement a basic version: for example, if one asset has very high risk contribution, recommend reducing it; if an asset has low correlation and good returns, maybe recommend increasing it. As you progress, you might incorporate more advanced techniques or even machine learning (for instance, some researchers use reinforcement learning for portfolio optimization, especially in crypto [14] [15] – a futuristic idea for you down the road). When this engine runs, it should produce a clear set of recommended actions (like target weights or specific trades).

- **LLM-Based Analysis Engine(s):** These would be the engines that convert numbers to narratives. You might have one LLM engine per section: e.g., **Performance Commentary Engine** (takes the performance metrics and writes a summary paragraph), **Risk Commentary Engine** (does the same for risk findings), and **Recommendation Rationale Engine** (explains the optimization suggestions). Alternatively, you can have one LLM engine that takes *all* results and produces a full report, but that prompt might be very complex. A middle ground is to use multiple smaller prompts (one per topic) and then concatenate the outputs into one report. Chaining wise, you'd run the LLM engines after all the data-oriented engines have populated the bag with metrics. Each LLM engine should be given only the data it needs for its section (to keep prompts focused and within token limits). This separation also mirrors specialization: e.g., you prompt the LLM differently when "wearing the hat" of a risk analyst vs a performance analyst. That often yields better, more focused writing.

- **Report Assembly Engine:** Finally, you might have a step that assembles all pieces (raw data, metrics, LLM-generated text) into a final deliverable format. This could be as simple as concatenating strings and formatting, or as elaborate as generating a PDF or interactive HTML. In the chain, this is the last step – it takes everything from previous steps and produces the output to the user. If you're returning a Markdown or text report, this engine might insert charts or tables as needed (for example, a table of holdings or a chart of performance – if you plan to generate any visuals, you'd do it here, possibly by calling a plotting library or an MCP server that generates charts). Since for now a text/markdown output might suffice, the assembly engine will just structure the final report (sections, headings, inserting the content from LLM engines, etc.).

When chaining these engines, **maintain a logical flow**: data → analysis → insights/recommendations → final output. Each engine's output feeds naturally into the next engine or into the final report. Also consider dependencies: for instance, the optimize engine might depend on risk metrics (if optimizing for Sharpe, it needs volatility from risk engine) – so ensure the orchestrator orders them correctly or consolidates needed

info before optimization. By focusing on these engines, you cover the gamut of what a human portfolio strategist would do: gather data, evaluate performance, assess risk, decide on changes, and communicate the plan. This multi-engine design is essentially a multi-agent system tackling a complex task, which is indeed how cutting-edge AI investment platforms are being built [7]. Each engine is your "specialist" in the chain, and the orchestrator is the coordinator ensuring they work in concert. This structure will make your code easier to extend (e.g., add a new analysis engine later without disturbing others) and even possible to distribute in the future (you could run some engines on separate threads or servers if needed).

Adding an "Optimize" Command for Final Recommendations

It's a great idea to introduce a separate `optimize` command (and corresponding engines/orchestrator sequence) to produce the **Investment Committee's Conclusion** style output. In practice, this might reuse a lot of the `analyze` pipeline and then extend it. For example, you could have the `optimize` command run everything `analyze` does *plus* run the optimization engine and an additional LLM step for the conclusion. Architecturally, you can implement this by either (a) having the optimize orchestrator call the analyze orchestrator internally (to gather all analysis first), or (b) factoring your orchestrator so that common steps are in a helper that both commands invoke. The key is to avoid duplicating the entire analysis pipeline logic between analyze vs optimize – keep it DRY by reusing engines.

Purpose of the optimize stage: The "Investment Committee Conclusion" should synthesize analysis into actionable advice. It's somewhat a different mode than pure analysis: it's prescriptive. By separating it as a command, you give yourself flexibility – a user (or you) might sometimes just want to analyze the portfolio without any suggestions (non-intrusive analysis), whereas other times you want the full advisory output. From a design standpoint, this separation is clean: `analyze` populates the data and descriptive insights, and `optimize` builds on that to answer "So what should we do?". Many professional processes mirror this – first the analytics, then the committee decides on changes.

Implementing the optimize engines: As discussed, the main new engine is the **Optimization Engine**, which we detailed earlier. Another might be a **Scenario/Stress Test Engine** that could be part of optimization phase – e.g., before making a conclusion, test how the proposed changes might perform under certain scenarios (rising rates, market crash, etc.) to ensure they're robust. This could be a future addition. For now, the optimization engine's output (proposed trades or new allocations) can be passed to the LLM for generating the conclusion. You might craft a prompt like: *"You are an Investment Committee summarizing your decision. Given the following recommended changes to the portfolio [list of changes] and the analysis above, produce a conclusion explaining the rationale."* This will prompt the model to write something that reads like the minutes or summary of a committee meeting – i.e., a final verdict on the strategy.

Engine chaining for optimize: The chain might look like: *(reuse analyze chain)* → *Optimization Engine* → *Conclusion LLM Engine*. It could even be interactive in the future (e.g., the LLM could simulate a bit of back-and-forth ("The committee noted X, however considered Y, and decided…"), but that can get complex. A simpler approach is fine: just a clear final recommendation section.

By adding an `optimize` command, you keep responsibilities separated. The analyze report might already include some observations like "Cash is high" or "Allocation drifts noted", but the optimize will explicitly say "Therefore, do A, B, C." This mirrors how an analyst report differs from an investment plan. Technically, it's absolutely doable with your current setup: you mostly plug in one extra engine and one extra LLM step.

Since your orchestrator is flexible, you can either create a new orchestrator sequence or adjust the existing one based on a mode. Either way, **having a distinct optimize phase is advisable** for a full-featured strategist. It makes your portfolio tool truly proactive (not just descriptive).

One consideration: ensure that the optimize command outputs are clearly distinguished and perhaps more formally formatted (because in a multi-user SaaS, recommendations might even trigger actions or at least be something the user will scrutinize heavily). For instance, listing trade suggestions in a table format (Asset, Buy/Sell, Quantity or % change) could be useful. Your LLM conclusion can then reference that table ("as shown in the table above, the committee suggests…") if you integrate text and data well. This added command will position Mosychlos closer to tools used by actual advisors – providing not just analysis but also **decision support**. Given that even advanced platforms are heading this direction (autonomous agents that not only analyze but also act or recommend actions [16]), you'll be ahead of the curve in creating a truly **futuristic portfolio strategist**.

Managing Prompts and Templates in the Application

Organizing your prompts and templates well is crucial for maintainability, especially as you add more LLM-driven features. Here are some insights and best practices:

- **Centralize your prompt templates.** All prompt strings (especially large ones) should be defined in one place or in a clearly structured manner – *do not scatter prompt text throughout your code*. This could mean having a dedicated directory (e.g., `prompts/`) with files like `performance_analysis.txt`, `risk_analysis.txt`, `conclusion.txt`, etc., or a single config file where each prompt is a constant. The goal is that if you (or future team members) need to tweak how a question is asked of the LLM, there's a single obvious place to do it. Many developers use simple templating systems – even something like Jinja2 templates or Go's text/template – so you can include placeholders for dynamic data (like `{portfolio_return}`) in the prompt files and fill them in at runtime. This approach avoids mixing code logic with large blocks of text and makes it easier to see the structure of your prompts at a glance. One community member put it well: having **"one place where all the data is assembled into a prompt, instead of having it all over the place"** is key [17].

- **Use descriptive, modular templates.** Break prompts into reusable pieces if possible. For instance, if every prompt needs a section describing the portfolio context (like holdings summary), you can maintain that as one template and insert it into others. This way, if you change the way you format the portfolio summary in text, you update one template and it reflects everywhere (a sort of DRY principle for prompts). There are advanced techniques like "exploding prompts" which use dependency graphs to compose prompt parts from a directory structure [18], but you may not need that complexity initially. Still, keep the idea of *template composition* in mind. Perhaps have a template for the intro, one for metrics bullet points, one for conclusion, etc., and combine them. This will also make it easier to manage prompt versions.

- **Employ version control and iteration for prompts.** Treat your prompt templates as first-class artifacts that evolve. It can be useful to keep a changelog of how you modify them and note which versions yield better LLM responses. In fact, consider embedding a version identifier in complex prompts as a comment, so when you log the outputs you know which prompt version produced it. This relates to prompt **versioning and diff tracking**, which is emerging as a best practice in prompt

engineering [10]. If your prompts are in separate files, they'll naturally be under git version control – use that to your advantage to see diffs when you refine a prompt's wording. Sometimes a small change (like adding "Step-by-step:" or changing the order of instructions) can significantly affect output; having a history helps you understand what worked best.

- **Log prompts and responses for debugging.** Especially in a complex application like this, you want to know exactly what was sent to the LLM and what it replied. Implement logging that records each prompt (perhaps with sensitive data masked if needed) and the model's answer. This will be invaluable when an output is off – you can trace back to see if the prompt was the issue. It's also helpful for auditing and improving prompts. Many robust AI systems include prompt/response logging as part of their operational best practices [10].

- **Template design best practices:** When writing the prompts themselves, follow known guidelines to get the best results. Clearly instruct the model of its role (e.g., "You are an investment analyst…" at the top of the prompt), use bullet points or numbered lists in the prompt where appropriate to structure the model's output (LLMs respond well to structured input), and provide examples if needed (few-shot prompting) to guide style. Keep prompts as concise as possible but also **unambiguous**. If you have multiple variables to inject (numbers, lists of holdings, etc.), make the format easy for the LLM to parse – for instance, you can present data as a mini-table in the prompt or as bullet points with consistent labeling. Since you might use different models in the future (perhaps switching to an open-source LLM for cost savings), design prompts to be model-agnostic: avoid relying on any one model's quirks and stick to straightforward instructions that any reasonably capable model can follow.

- **Organize by context or engine.** It often makes sense to name/organize templates according to the engine or phase they're used in. For example, templates related to performance analysis can be in one file or group, those for risk in another. If you foresee supporting multiple languages or tones, you could even structure directories by language or scenario. The idea is that as your number of templates grows, a clear naming convention and hierarchy prevents chaos. For instance: `prompts/performance/summary_en.txt` could be a file, and maybe later `prompts/performance/summary_fr.txt` if you ever support French output. This way, your code can select templates dynamically (based on user or context) without hardcoding strings. It's a bit forward-looking, but since you mentioned SaaS future, such organization might pay off.

- **Utilize prompt management tools (if needed):** Since your project is currently small-scale, manual template management is fine. But be aware there are emerging tools (like LangChain's prompt templates, or open-source libraries like *Pezzo*, *PromptLayer*, etc.) that help manage and even A/B test prompts. There are also community solutions where prompts are stored as TOML or in databases for live updating. You might not need these now, but knowing about them is useful. For example, some developers use Jinja with a state machine (like Burr) to handle complex prompt logic and appreciate the **visibility at each stage of prompt assembly** it provides [19]. The takeaway: as your prompt logic gets more complex (maybe multi-step formatting or conditional prompts), consider leveraging such libraries to reduce reinventing the wheel.

In conclusion, treat prompt templates as a dynamic, central part of your application. Organize them in a way that any change is easy to make and review. Maintain an eagle-eye on how prompts perform – since the LLM's output quality will heavily influence user perception of Mosychlos's intelligence. Good prompt

hygiene (clear structure, centralized management, and logged outputs) will make your application far easier to maintain and improve. By following these practices, you'll ensure that as Mosychlos grows, you won't be hunting through code for that one stray string that's making the AI say something odd – you'll know exactly where to look and how to adjust it.

---

1 2 3 4 GitHub - punkpeye/awesome-mcp-servers: A collection of MCP servers.
https://github.com/punkpeye/awesome-mcp-servers

5 Solving a Portfolio Analysis Problem with LangGraph and Python | by Can Demir | Towards AI
https://pub.towardsai.net/solving-a-portfolio-analysis-problem-with-langgraph-and-python-d65072e5d9da?gi=2fd543ac1c2f

6 7 14 15 Multi-Agent AI Architecture for Personalized DeFi Investment Strategies | by Jung-Hua Liu | Medium
https://medium.com/@gwrx2005/multi-agent-ai-architecture-for-personalized-defi-investment-strategies-c81c1b9de20c

8 10 11 Can multiple models be chained or orchestrated together? - MCP Server Docs
https://socradar.io/mcp-for-cybersecurity/core-concepts/can-multiple-models-be-chained-or-orchestrated-together/

9 16 GitHub - doublehq/oracle
https://github.com/doublehq/oracle

12 13 GitHub - EchoingVesper/mcp-task-orchestrator: A Model Context Protocol server that provides task orchestration capabilities for AI assistants
https://github.com/EchoingVesper/mcp-task-orchestrator

17 19 What's your favorite approach to managing prompt templates? : r/LocalLLaMA
https://www.reddit.com/r/LocalLLaMA/comments/1egvdix/whats_your_favorite_approach_to_managing_prompt/

18 Exploding Prompts Now Available as Open Source | Proofpoint US
https://www.proofpoint.com/us/blog/engineering-insights/exploding-prompts-available-open-source