ChatGPT

Index

1. Engines

Engine Orchestration Overview: Mosychlos uses a chain of specialized **engines** to perform portfolio analysis in stages. Engines execute sequentially within a single AI session context – sharing state via a global **SharedBag** – so that each engine builds on the prior results [1]. The orchestrator loops through a configured list of engines, calling each engine's `Execute` method in turn [2]. All engines contribute to the *same conversation context* with the AI model, achieving a "single-session multi-step" analysis [3]. Crucially, each engine writes its output into the SharedBag (global state), making it available to subsequent engines [4]. This design yields a **single-context pipeline**: for example, *Engine1* → *Engine2* → *Engine3*, each adding to the conversation and state, rather than isolated agents [5].

Engine Design Principles: Each engine has a **single responsibility** focusing on one analytical domain [6]. Engines produce two forms of output: (1) a **human-readable report** and (2) **structured data** (JSON) for programmatic use [7]. Internally, engines implement a common interface: they expose a `Name()`, list any **Dependencies()** (engines that should run before them), and an `Execute(...)` method to perform analysis [8]. The result of an engine is typically an **EngineResult** implementing getters for summary, recommendations, timestamp, and a data payload [9]. In practice, engines instruct the AI to return a JSON object conforming to a Go struct schema (using the structured output feature) [10] [11]. This ensures each engine's analysis is captured in a well-defined output format.

Core Engines in Mosychlos: The current engine suite covers different aspects of portfolio analysis [12]:

- **RiskEngine:** Assesses portfolio risk exposure and concentration. **Responsibilities:** Calculate metrics like position concentration (Herfindahl index), sector/geographic exposure, liquidity risk, etc., and provide risk mitigation recommendations [13]. **Inputs:** Portfolio data and any prior gathered market data (if available in SharedBag). **Outputs:** Structured risk analysis results (risk scores, identified risk factors, recommended hedging or rebalancing actions) [14]. The RiskEngine requires no prerequisite engines (pure analysis on the portfolio) [15]. **Implementation:** It uses a **prompt template** for a "Portfolio Risk Analysis Expert" persona with instructions to output a JSON report (with sections for Summary, Risk Breakdown, Recommendations, etc.) [16] [17]. Before invoking the AI, it sets up tool constraints (budget for tool calls) and constructs the user prompt via a PromptManager [18] [19]. The

AI's response is expected to match a Go struct (e.g. `InvestmentResearchResult`) defined for investment analysis [11]. The engine then stores the JSON output in the SharedBag under a key like `KRiskAnalysisResult` [20]. **Limitation:** The RiskEngine itself does not decide what engine comes next – it simply writes its result. The orchestrator determines the sequence (e.g., RiskEngine runs before any engine that depends on risk output).

- **NewsEngine:** Performs news intelligence and sentiment analysis on relevant news for the portfolio. **Responsibilities:** Fetch and summarize market news, analyze sentiment and potential impact on portfolio holdings [21]. It often involves **multi-phase** processing – e.g. using a tool to retrieve news articles then summarizing them [22]. **Inputs:** Possibly a list of portfolio tickers or sectors to focus the news search. **Outputs:** News summaries and sentiment metrics in structured form (e.g. key headlines, sentiment scores, commentary). **Dependencies:** None – it can run as a data-gathering engine at the start [23]. **Implementation:** The NewsEngine likely first invokes the `newsapi` tool to get headlines, then includes those results in the AI prompt for summarization. It uses a prompt template focused on "Market News Analyst" behavior and may produce JSON fields like top news themes and sentiment indicators. **Limitation:** The engine produces its summary and cannot trigger further analysis on its own. For example, if the news indicates a specific risk, the engine cannot branch; it relies on subsequent engines (or a final synthesis) to incorporate that insight.

- **AllocationEngine:** Analyzes current asset allocation vs. optimal targets. **Responsibilities:** Evaluate the portfolio's allocation across asset classes/sectors, compare against benchmarks or profile targets, and propose optimization or rebalancing actions [24]. **Inputs:** Portfolio composition and possibly outputs from RiskEngine (for risk-adjusted allocation) [25]. **Outputs:** Structured data on allocation percentages, diversification metrics, and recommended adjustments (e.g. "increase bonds to X%"). **Dependencies:** Requires RiskEngine to have run (to incorporate risk metrics into allocation decisions) [25]. **Implementation:** It retrieves the risk analysis from SharedBag (`KRiskAnalysis` data) if available [26], and uses a prompt persona like "Asset Allocation Expert." The AI is prompted to output recommendations in JSON (e.g. suggested new allocation breakdown, rationale). **Limitation:** Similar to others, the AllocationEngine runs when invoked by the orchestrator sequence. It cannot, for instance, decide to re-run risk analysis or fetch additional data on its own – it assumes required data (risk metrics, etc.) are already present in SharedBag.

- **ComplianceEngine:** Checks regulatory and policy compliance of the portfolio. **Responsibilities:** Evaluate the portfolio against regulatory constraints or investment policy statements (jurisdiction limits, concentration limits, prohibited assets) [27]. **Inputs:** Portfolio holdings and any compliance rules (possibly provided in profile). **Outputs:** Compliance status, any violations found, and recommendations to resolve issues (e.g. flags if portfolio exceeds some threshold or holds disallowed securities). **Dependencies:** None – it's a standalone rule-based analysis [28]. **Implementation:** Likely uses a ruleset (could be encoded in prompt or code) to identify compliance issues. The engine's prompt might be a "Compliance Officer" persona listing any violations and remedial steps in structured form. **Limitation:** It runs unconditionally if included in the pipeline. If the portfolio is simple or rules are not provided, it might produce trivial output – currently there's no dynamic skip based on context.

- **ReallocationEngine:** Synthesizes all prior analyses into final recommendations (the "synthesis" or decision-making engine). **Responsibilities:** Propose concrete portfolio changes and strategy, combining outputs of risk, news, allocation, compliance, etc. [29]. This engine acts like an *Investment*

Committee, collating perspectives. **Inputs:** All analysis results from previous engines (gathered from SharedBag: risk findings, news insights, allocation plan, compliance notes). **Outputs:** A comprehensive investment strategy or rebalancing plan, often as a structured report (e.g. final recommended trades or allocations with justification). **Dependencies:** It depends on RiskEngine, NewsEngine, and AllocationEngine (and presumably any others with data to contribute) [30]. It should run last to produce the final output. **Implementation:** The ReallocationEngine's prompt may impersonate a "Portfolio Manager" or an investment committee that has access to *all prior engine outputs*. For example, it might instruct the AI: "Given the risk analysis, news analysis, allocation review, etc. (which are in context), provide final actionable recommendations." The engine likely merges relevant SharedBag data into the prompt (or relies on the fact that the AI session has seen all prior content). The output is a structured plan (JSON) – e.g. recommended buys/sells, rationale, and maybe a summary section [31] [32]. **Limitation:** The ReallocationEngine assumes all necessary input is available from previous steps. It cannot selectively decide to omit an engine's input or query an additional source. Its success relies on the completeness of preceding engines' outputs.

Integration via NEXT_PATTERN: Currently, engines are wired in a fixed sequence defined by the orchestrator (or a config file listing engine order). An engine cannot choose or skip the next engine – the flow is linear and predetermined [33]. For instance, if the pipeline is [NewsEngine → RiskEngine → AllocationEngine → ReallocationEngine], each will run in order regardless of intermediate results. The orchestrator does not support dynamic branching like "if high risk then run Engine X" – this is a noted limitation. The proposed **NEXT_PATTERN** aims to allow more dynamic flow control in the future (see Section 4), but as of now engines themselves do not implement a `GetNext` mechanism. They integrate by *accumulating shared context* rather than by controlling sequence. In summary, each engine focuses on generating a structured output given the current SharedBag state, and the orchestrator simply moves to the next engine in list [34]. This simplicity ensures a clear, single-context conversation, but it means **no engine can alter the pipeline order** (e.g. skip or invoke additional engines) – a design choice to keep the multi-step analysis straightforward [35].

2. Tools

Tool System Overview: *Tools* are modular data provider integrations that supply external data (market data, financials, news, etc.) to the AI engines [36]. Each tool implements a standard Go interface (Name, Key, Description, Definition, Tags, Run) enabling it to be registered with the AI client for function calls [37]. Tools encapsulate calls to external APIs or databases and return the results in a normalized JSON format that the AI can understand. As of August 2025, Mosychlos has **9 active tools** covering market data, fundamentals, economics, news, and more [38]. Key tools include **Yahoo Finance (YFinance)** for market prices, **Financial Modeling Prep (FMP)** for fundamentals, **FRED** for macroeconomic data, **NewsAPI** for news, and a basic **Weather** tool [39].

Normalization and Structured Output: All tools follow a similar pattern: they fetch data from their source, then **normalize it into a JSON structure** (often via marshaling into Go maps or structs) before returning it as a string. This ensures the AI receives structured output consistently across tools. Every tool's `Run(ctx, args)` method typically parses its input JSON, calls an API client, then post-processes the result into a JSON string. Common post-processing steps include converting the response into a generic map and adding a `metadata` section (with timestamp, source, etc.) to the output [40] [41]. This approach yields self-descriptive outputs, for example: a tool result might look like:

```
{
  "data": { /* tool-specific data here */ },
  "metadata": { "timestamp": "2025-08-23T08:00:00Z", "source": "fmp", "symbol":
"AAPL" }
}
```

Such structured outputs allow easier chaining and interpretation by engines.

Below we detail each major tool (SEC EDGAR, YFinance, FMP, FRED) including their normalization logic, output schema, and primary API endpoints:

SEC EDGAR Tool (Regulatory Filings – *planned*)

Purpose: The SEC EDGAR tool will provide access to company filings and related regulatory data (e.g. 10-K/Q reports, insider trades, ownership filings). It's listed as a future integration in the roadmap [42]. The design outlines multiple functionalities: a **filings retrieval** (fetching and parsing SEC filings text/structures), **company facts** (key financial metrics from XBRL data), **insider trading** (Form 3/4/5 data on insider transactions), and **ownership** info (institutional holdings from 13F filings) [43].

Normalization: The SEC EDGAR tool would likely collate raw filing data into structured JSON. For example, a query for "filings for Company X in 2021" might return a JSON with a list of filings (each with fields like date, form type, description, and extracted key values). Similarly, an insider trading query might return a list of insider transactions with fields (insider name, relation, date, buy/sell, amount). The tool is expected to implement `Run` such that it transforms the EDGAR API's raw XML/JSON into **Go structs** and then into a JSON string output. Given the pattern of other tools, it will likely include a `"metadata"` field indicating data source ("sec_edgar"), and possibly the target company's identifier (CIK or ticker).

Go Output Types: We anticipate Go types like `Filing` struct (fields: form type, date, report text or key values), `InsiderTrade` struct, etc., defined to mirror EDGAR data. The `sec_edgar/tool.go` implementation will marshal these into JSON. For example, the output might be `{ "filings": [ { "form": "10-K", "filed": "2025-03-01", ...}, ... ], "metadata": { "source": "sec_edgar", "company": "XYZ Corp", "timestamp": "..." } }`. This aligns with the proposed file breakdown [44].

Key Endpoints: The SEC EDGAR API provides JSON data for filings and company facts. Likely endpoints to be used: - **Company Submissions** (to list all filings for a company by CIK), - **Company Facts** (financial statement data in JSON for XBRL-tagged facts), - **Daily Transactions** (for insider trades), and possibly **Ownership filings** (13D/G or 13F summaries for institutional investors). For example, the tool might call `/api/company/CIK/filings` to get recent filings, or `/api/xbrl/companyfacts/CIK.json` for standardized financial metrics. These would then be parsed into the output structures. The exact internal endpoints file plan (filings.go, company_facts.go, etc.) suggests one function per major data category [45].

Status: As of now, `sec_edgar` is in the planning phase – references in configuration and docs exist [46], but a fully implemented tool is forthcoming. Once implemented, it will fill a crucial gap by providing regulatory data for engines (e.g., allowing an engine to analyze fundamental filings or insider sentiment).

YFinance Tool Suite (Yahoo Finance Market Data)

Purpose: The YFinance suite is a collection of tools providing comprehensive market data from Yahoo Finance [47]. There are **5 sub-tools** under the YFinance umbrella [47] [48]: - `yfinance_stock_data` – Real-time and historical price data (OHLCV) for stocks [49] [50].
- `yfinance_stock_info` – Company profiles, key financial metrics, and fundamentals [51] [52].
- `yfinance_dividends` – Dividend history and yield analysis [53] [54].
- `yfinance_financials` – Financial statements (income, balance sheet, cash flow) for a given stock [55] [56].
- `yfinance_market_data` – Broad market indices and sector performance data [57] [58].

This suite covers free stock and market data, replacing the need for third-party finance libraries.

Normalization: All YFinance tools share a **consistent output format**. Each returns a JSON object with a top-level `status` ("success" or "error"), the input symbol(s) echoed, a `data` field containing the tool-specific results, and a `metadata` field for context [59] [60]. For example, a successful Stock Data call might output:

```
{
  "status": "success",
  "symbol": "AAPL",
  "data": { ... price and volume data ... },
  "metadata": { "timestamp": "2025-08-23T08:00:00Z", "source": "yahoo_finance",
"tool": "yfinance_stock_data" }
}
```

This uniform structure is explicitly documented in the YFinance README [59] [60]. The **normalization logic** typically involves taking Yahoo's API response (often a nested JSON) and unmarshaling it into a Go struct, then re-marshaling to `map[string]any` to add metadata and ensure no extraneous fields. For instance, in the Stock Data tool, after fetching the data, they convert the entire response struct to a map and insert a `"results_count"` field, then wrap it with status and metadata before final JSON output [61] [62]. This approach prevents manual field-by-field mapping – preserving all Yahoo data in the output.

Go Output Types: The YFinance package defines specific response structs for each API: e.g. a `ChartResponse` for price data, a `QuoteSummary` for stock info, etc. These are part of the `pkg/yfinance` client. The tools use these types – for example, `stockData.Chart` struct – but ultimately output generic JSON. Key fields from Yahoo are preserved (like OHLCV arrays for historical prices, or fundamental ratios for stock info). The Go types are strong (with numeric types, etc.), but by converting to `map[string]any` the final JSON is flexible and directly consumable by the LLM [61] [63].

Key Endpoints: The tools call Yahoo's public web APIs. Notably: - **Historical data:** Yahoo's *Chart API* (e.g. `/v8/finance/chart/{symbol}`) is used for `stock_data`. In fact, a recent fix for `yfinance_stock_info` involved using the chart API to retrieve profile data as well [64], because the usual summary endpoint required auth. - **Stock info:** originally would use Yahoo's *QuoteSummary API* (e.g. `/v10/finance/quoteSummary/{symbol}`), but due to 401 errors, they switched to an alternative (the chart

endpoint) [64]. The tool now likely pulls company info from fields in the chart response or a combination of endpoints. - **Dividends:** could leverage the Chart API with an appropriate query parameter to get dividend events over a period. - **Financials:** Yahoo provides financial statements via *Yahoo Query Language* or another internal API (possibly also under `/v10/finance/` endpoints). The tool might scrape or use an unofficial JSON source to gather quarterly/annual statements. - **Market data:** This may call multiple symbol quotes (e.g. ^GSPC, ^DJI, ^IXIC for indices) in one request or sequentially. Yahoo's quote API can retrieve batch quotes for indices and sectors. It might also use special endpoints or just the chart API for indices.

Implementation Notes: The YFinance tools require no API key (Yahoo is free) [65]. They handle rate limiting implicitly by respecting Yahoo's service limits (added realistic headers to mimic a browser) [64]. All five tools have a comprehensive test suite and are fully validated [66] [67]. The consistent design of YFinance outputs serves as a model for other tools – providing a stable JSON schema to the AI.

FMP Tool (Financial Modeling Prep – Stock Fundamentals)

Purpose: The FMP tool connects to the Financial Modeling Prep API to fetch fundamental financial data for stocks [68]. FMP provides a range of financial information – company profiles, financial statements, key metrics, sector and industry data, and analyst estimates. In Mosychlos, the FMP integration currently focuses on **company profiles and fundamentals** for given tickers. It can retrieve things like company description, sector, market cap, P/E ratio, etc., and is expandable to other endpoints (financial statements, metrics, etc.).

Normalization: The FMP tool is designed to handle **batch ticker requests**. The `Run` method expects a JSON input with a list of ticker symbols (e.g., `{"tickers": ["AAPL","MSFT"]}`) [69]. It will iterate over each ticker, fetch data via the FMP client, and aggregate results into a **map from ticker to data** [70] [71]. After fetching, the tool converts the results to JSON as follows: it marshals the Go structs into JSON, unmarshals to a map, then inserts a `"metadata"` entry with context (timestamp, source `"fmp"`, the list of tickers, and count) [40] [41]. Finally, it re-marshals to a JSON string to return [72]. This ensures all fetched data plus metadata are returned in one JSON. For tickers that fail (e.g., API error), it logs a warning and puts an `"error"` field for that ticker in the map rather than failing the whole call [73] – making the output robust to partial failures.

Go Output Types: The FMP client defines several result structs in `pkg/models`: e.g. `FMPCompanyProfile` for profile data, `FMPFinancialStatement` for statements, `FMPKeyMetrics` for metrics, etc. In the current implementation, the tool primarily uses `FMPCompanyProfile`. So the output map's values are essentially `FMPCompanyProfile` objects (with fields like company name, CEO, industry, beta, etc.) serialized to JSON [74] [75]. If expanded, the tool could include additional fields or nested objects (for instance, an entry for financial statements might include an array of statements). By marshaling to a map, all struct fields become JSON keys automatically, so the output for each ticker is comprehensive. An example output snippet for `{"tickers":["AAPL"]}` could be:

```
{
  "AAPL": {
    "symbol": "AAPL",
    "companyName": "Apple Inc",
    "sector": "Technology",
```

```
    "industry": "Consumer Electronics",
    "beta": 1.23,
    "price": 175.80,
    "...": "..."},
  "metadata": { "source": "fmp", "tickers": ["AAPL"], "count": 1, "timestamp":
"2025-08-23T08:00:00Z" }
}
```

This map-of-tickers format makes it easy for the AI to retrieve data for multiple symbols in one call.

Key Endpoints: The FMP client code shows multiple API endpoints it can hit: - **Profile:** `/v3/profile/{symbol}` – returns basic company info (used by default in `fetchWithClient`) [74]. - **Financial Statements:** `/v3/{statementType}/{symbol}` e.g. `/v3/income-statement/AAPL` (with a limit parameter) – returns arrays of statements [76]. - **Key Metrics:** `/v3/key-metrics/{symbol}` – returns a list of key financial metrics (like ROE, debt/equity) [77]. - **Analyst Estimates:** `/v3/analyst-estimates/{symbol}` – forecasted estimates for earnings, etc. [78]. - **Stock Price (Quote):** `/v3/quote-short/{symbol}` – quick quote (price and volume) [79]. - **Market Cap:** `/v3/market-capitalization/{symbol}` – current market cap info [80].

In the present Mosychlos tool, when `Run` is called, it uses `GetCompanyProfile` for each ticker (which hits the `/profile/` endpoint) [74]. The existence of other methods suggests the tool could be extended or parameterized to retrieve those as needed. For now, the focus is on profile fundamentals and any critical metrics available therein.

Usage & Status: The FMP tool is **active and registered** (requires an FMP API key) [81]. In `config.default.yaml`, one can configure the FMP API key and caching. The tool has caching and rate-limit wrappers applied if configured (like all tools) [82] [83], meaning it won't slam the API beyond allowed rates. With normalization adding a metadata block and handling multiple tickers, the FMP tool is well-suited to gather fundamentals for an entire portfolio in one go. This data can then feed engines like Risk or Analysis engines that need fundamental context.

FRED Tool (Federal Reserve Economic Data – Macroeconomics)

Purpose: The FRED tool integrates macroeconomic indicators from the Federal Reserve's FRED API [84]. It allows Mosychlos to pull in data like GDP, inflation, interest rates, or other economic series to inform the investment analysis. The current implementation focuses on **regional economic data** by utilizing FRED's GeoFRED capabilities (in particular, an example of per-capita income by region). However, it is structured to support general indicator queries with some preset defaults.

Normalization: The FRED tool's `Run` takes a JSON input specifying parameters such as `series_group`, `date`, `region_type`, `units`, `frequency`, and `season` [85] [86]. By default, it assumes `series_group: "882"` (which corresponds to *Per Capita Personal Income* dataset) if not provided [87]. When executed, it calls the FRED client to get regional data for the given series group and date [88]. The raw response from FRED (likely a nested structure of regions and values) is then converted into a flat map: keys are region codes and values are data points (with associated info). Specifically, the code loops through `geoData.Meta.Data` which maps date -> list of observations, and constructs a map where each region

code maps to a `FREDGeoRegionalSeries` struct containing the value and metadata for that region [89] [90]. The result is a map like `{ "AL": { "region":"Alabama", "value": "40350", ...}, "AK": { ... }, ... }`. This map is then augmented with a top-level `"metadata"` containing the query parameters and timestamp [91] [92]. Finally, it's marshaled to JSON and returned [93].

Go Output Types: The tool uses a custom struct `FREDGeoRegionalSeries` (as seen in code) with fields such as region Code, Region name, SeriesID, Value (as string, since some values might be null or stringly), Units, Frequency, and Date [90]. Each entry in the output map is one such struct serialized to JSON. The metadata includes the original inputs (series_group, date, region_type like *state* or *msa*, units (Dollars/Percent), frequency, seasonal adjustment) and source. If the tool were extended to generic series, we might have a different struct (e.g., for time series array), but currently it's tailored to regional snapshots.

Key Endpoints: The FRED API endpoint in use is the **GeoFRED Series Group** API. Based on the parameters: - It likely calls something like: `/fred/series/group?series_group_id=882&...` to get all state data for per-capita income on a given date. The code sets `series_group=882` by default and expects a date like "2022-01-01" (annual frequency) [94] [95]. This suggests they're using the GeoFRED **"Series Group"** query which returns values for all regions in that group as of the specified date. - For other series, FRED has `/fred/series/observations` for time series. They might integrate that in future for time-series data (e.g., get GDP over time). - The docstring description hints at specific series group IDs for common indicators: *Per Capita Personal Income (882)*, *GDP (249)*, *Unemployment Rate (158)*, *Federal Funds Rate (115)* [96]. These numbers suggest predefined groups or categories. It appears they constrain usage to certain known IDs (like only 882 is guaranteed working in current code) [85]. In effect, the tool is currently somewhat specialized: it's demonstrated for pulling one group's data (882) on a given date for all states (region_type=state).

Output Example: An example output for the default would be a JSON where each U.S. state (by abbreviation code) maps to its per-capita income value:

```
{
  "AL": { "Code":"AL", "Region":"Alabama", "SeriesID":"PCPI_AL",
"Value":"43000", "Units":"Dollars", "Frequency":"Annual", "Date":"2022-01-01" },
  "AK": { "Code":"AK", "Region":"Alaska", "SeriesID":"PCPI_AK",
"Value":"57000", ... },
  ...
  "metadata": {
    "timestamp": "2025-08-23T08:00:00Z",
    "source": "fred_geofred",
    "series_group": "882",
    "date": "2022-01-01",
    "region_type": "state",
    "units": "Dollars",
    "frequency": "a",
    "season": "NSA"
  }
}
```

This output gives a snapshot of a macro metric across regions. Engines might use such data for regional investment analysis or economic context.

Usage & Expansion: The FRED tool is active (requires an API key) and can be configured in the YAML (API key, caching) [97]. In the current form, it's somewhat constrained to the example dataset. A logical enhancement is to allow specifying a particular series ID to retrieve a time series or a national-level indicator (e.g., "GDP" series). The plan is to eventually support more general FRED queries. Nonetheless, even this example shows the system's ability to integrate alternative data: e.g., providing economic indicators that an engine could plug into an analysis (for instance, an engine might query FRED for the latest unemployment rate or interest rate and interpret its impact).

Tool Endpoints & Focus: To summarize endpoints: - Using **GeoFRED series group** endpoint for broad data distributions (as implemented). - Could use **Standard FRED series** endpoints for single series if extended. The documentation in code suggests series like GDP might be handled via known IDs, perhaps in a future iteration or alternate mode.

All tool outputs are ultimately accessible to the AI through function-calling. For example, the AI can call `yfinance_stock_data` with a symbol and get structured price data, or call `fred` tool with parameters to get a macro snapshot. The consistency of JSON output and presence of metadata in each ensures that when multiple tool outputs are combined, the source and context remain clear.

Endpoint Summary Table (per tool):

- *SEC EDGAR:* Planned – e.g., `/submissions/{CIK}.json`, `/facts/{CIK}.json`, etc. for filings and XBRL data.
- *YFinance:* Yahoo Finance unofficial API – e.g., `/v8/finance/chart/{symbol}` (prices, also used for profiles), possibly `/v8/finance/spark`, etc., and batch quote endpoints for indices.
- *FMP:* Official FMP API – `/v3/profile/{sym}`, `/v3/income-statement/{sym}`, `/v3/key-metrics/{sym}`, etc. [74] [98].
- *FRED:* Official FRED API – `/fred/series/group` (for Geo data by group) and potentially `/fred/series/observations` (for time-series by ID).

Each tool is configured with API keys or base URLs in the Mosychlos config. They also benefit from a common infrastructure: caching (to avoid repeated calls) and rate limiting wrappers can be applied transparently [82] [83]. All active tools are registered at startup so the AI can call them by name [99].

Current Standalone Usage: Without chaining, tools operate independently. The AI model decides when to call a tool based on its prompt and the function definitions. For example, in an analysis prompt the AI might call `newsapi` to get news then proceed to analyze. However, the AI's ability to chain tool calls in one session is limited by its prompting – it might not always know to call one tool after another automatically. This leads to the motivation for a more explicit tool chaining architecture.

3. Prompt Management

Current Prompting System: Mosychlos employs a structured prompt management approach, using **template-based prompt builders** and enforceable output schemas. Each engine has an associated prompt

(often split into a *system prompt* defining the role/persona and a *user prompt* with the task). These prompts are managed by a `PromptBuilder` or specialized managers (e.g., a RegionalPromptManager for localized prompts) [100] [101]. For instance, the RiskEngine uses a hard-coded system message to shape the AI's persona ("You are a Portfolio Risk Analysis Expert...") and outlines the content and format required [16]. The actual portfolio data or query is then inserted via the PromptBuilder (which likely fills in details like portfolio holdings or risk focus into the user prompt) [102]. By centralizing prompt templates (in files or code), the system ensures consistency in how tasks are asked of the AI.

Structured Output Enforcement: A major feature of the prompt strategy is the use of **OpenAI's function calling / JSON schema** capabilities to get structured outputs. Mosychlos leverages a generic `Ask[T]` mechanism where the expected type `T` (a Go struct) is provided, and a JSON schema is generated from it [103] [11]. The AI's response must conform to this schema. For example, for risk analysis, the `InvestmentResearchResult` schema (containing fields for summary, findings, risk factors, etc.) is attached to the prompt as the desired format [11]. This guides the model to output a JSON matching the Go struct (with pretty indentation etc. as configured). If the model deviates or produces invalid JSON, the system can detect it and possibly retry. This strategy significantly improves reliability of downstream parsing and allows each engine's output to be directly consumed (stored in SharedBag or passed to other components) without manual extraction.

Comparison of Prompt Libraries: In designing this system, one could consider existing prompt/agent orchestration libraries (such as LangChain, etc.). However, Mosychlos primarily implements its own lightweight prompt management tailored to Go. The benefit is fine-grained control and integration with Go types (via the schema generation) [103]. High-level "prompt libraries" often used in Python (like LangChain or PromptToolkit) are not directly available in Go, so the team built custom managers. This provides flexibility: prompts can be defined in YAML or Go code (some prompt templates are likely stored under `internal/prompt/templates` for easy editing). The **trade-off** is more manual prompt engineering. Given the complexity of tasks (financial analysis, multi-part reasoning), this custom approach is beneficial – it aligns with the project's domain-specific needs (e.g., using an **Investment Committee** persona is very specific) without generic library overhead.

That said, the team is always evaluating improvements. For example, one could integrate a library to manage long conversations or memory, but since Mosychlos uses a single-session approach, a simple shared context suffices. The current system already implements a form of *Chain-of-Thought prompting*: e.g., instructing engines to reason step-by-step or cover multiple perspectives (the system prompt often enumerates what the AI should do, as seen in RiskEngine's instructions to provide assumptions, calculations, etc. in the output) [104].

Summarization Challenges: One key area of prompt management is handling **large content** (e.g., lengthy news articles, many search results). Directly inserting very large texts into the prompt can hit token limits or overwhelm the model. The roadmap addresses this by proposing to use **WaterCrawl and OpenAI batch processing** for summarization:

- **WaterCrawl Integration:** *WaterCrawl* is an AI-enabled web crawling and content extraction framework [105]. The plan is to leverage WaterCrawl to fetch relevant web content (for example, multiple news articles or filings) and possibly do preliminary AI processing on them. WaterCrawl can systematically extract text from URLs and even use AI to summarize or structure it. By using this, Mosychlos can offload some summarization tasks: e.g., crawl 10 news articles about a company and

return summaries. These summaries (already distilled) can then be fed into the Mosychlos analysis engine prompt instead of raw articles. This two-stage approach ("crawl then analyze") improves efficiency and keeps prompts concise. Essentially, WaterCrawl would act as a specialized tool or pre-processing step to provide *LLM-ready summaries* of large documents.

- **OpenAI Batch Calls for Summaries:** Another approach is using OpenAI's API to summarize chunks of text in parallel. Mosychlos's AI client includes a `BatchManager` that likely supports sending multiple requests concurrently [106] [107]. The idea is to split a long document or a set of documents into smaller parts and have the model summarize each part simultaneously, then combine those summaries. For instance, if there are 100 pages of SEC filings to review, the system could create 10 requests (each summarizing 10 pages) and execute them as a batch, then have the engine merge the summarized points. This significantly speeds up processing compared to a sequential prompt. It also improves quality because intermediate summaries keep each prompt focused.

OpenAI's API doesn't natively have a "batch summarization" endpoint, but Mosychlos can implement it at the client level with concurrency. Using batches also helps manage token usage by processing pieces individually. The **Waterfall or Map-Reduce summarization pattern** emerges: first-layer summarization (parallel), then a second-layer that synthesizes those summaries. Mosychlos plans to incorporate such patterns for tasks like extensive **financial report analysis or comprehensive news scanning**.

Prompt Libraries vs In-house: In summary, the current in-house prompt framework is quite robust (structured personas, JSON outputs, tool function calling). The enhancements revolve around feeding the prompt pipeline with **pre-digested information** so that the final prompts remain within manageable length. WaterCrawl can serve as a library/service to get that pre-digestion (especially for web content). If integrated, one might treat WaterCrawl as a tool that given a URL or query will return a summary (possibly via its own AI backend) – effectively making it a specialized summarization tool. This would complement Mosychlos's capabilities, allowing it to reason over summaries instead of raw text.

Additionally, as multi-agent capabilities are considered (from the FinRobot enhancement plan), prompt management will need to handle **multiple personas interacting**. The plan is to keep this within the single session via prompt engineering (e.g., the prompt explicitly says "Analyst A says this… Analyst B responds…") [31]. This avoids needing an external multi-agent orchestration library – instead, carefully crafted prompts can simulate a panel of agents in one model's output. Libraries like LangChain's agents might not easily allow that single-context multi-persona style; Mosychlos's approach of engine chaining with persona prompts is actually a simpler solution [35].

In conclusion, prompt management in Mosychlos balances **structured rigidity** (schemas, fixed personas) with **flexibility** (extensible prompts, external summarization). The roadmap includes using tools or parallelization to summarize content before it hits the main prompt, ensuring the AI's attention is on analysis rather than raw data digestion.

4. Tool Chaining Architecture (NEXT_PATTERN Model)

In the current design, each tool is invoked independently – the AI decides which tool to call and uses its output directly in the conversation. **There is no built-in mechanism for one tool to automatically trigger another**. For example, if the AI wants to analyze a company deeply, it might call `sec_edgar` to get filings

and then separately call `yfinance_stock_data` for market data, but it's the AI's logic (in the prompt/response) orchestrating this sequence. This can be limiting: the AI might not always realize a second tool call is needed, or it might not format the output of the first tool appropriately for the second. To address this, we introduce a **Tool Chaining architecture** – a structured way to link tools so that they execute in a defined sequence, passing data along.

Concept of NEXT_PATTERN: The idea is to allow each tool (or engine) to specify what tool(s) should follow after it, creating a chain or graph of actions. Concretely, this involves extending the `models.Tool` interface with a method such as `GetNext(prevTool string, prevOutput any) (nextTool string, nextArgs any)` (or a slice of next actions) that determines the next step based on the previous tool's output. This pattern is akin to a workflow: each step knows the potential next steps. We call it **NEXT_PATTERN** because each tool/engine can declare a list of "next" tools to consider.

Defining Next Steps: One approach is to give each tool a configurable `Next` **list** of tool names. For example, we could declare: SEC EDGAR tool's next list includes "yfinance_stock_data" (to get market prices for tickers found in filings) and "newsapi" (to get recent news if a significant filing is found). In code, this could be a field like `Tool.NextTools []string`. When constructing a tool, we might pass in the next tools it should trigger. The design calls for **variadic constructor arguments** or builder methods to set up these chains. E.g., one might do:

```
secTool := NewSecEdgarTool().WithNext("yfinance_stock_data").WithNext("newsapi")
```

or pass them in New:

```
NewSecEdgarTool(nextTools...).
```

This statically associates a sequence. Each engine/tool could thus have a pre-defined `[]NextSteps` slice.

Dynamic Decision with GetNext: While a static chain is useful, often whether to call the next tool depends on output content. That's where the `GetNext(prevToolName, prevOutput)` logic comes in. After tool A runs, the chaining controller will call tool A's `GetNext` (or have a mapping of A→next) to see what to do. This function can inspect `prevOutput` (already parsed into a Go type or map) to decide on the next action. For example: - The SEC EDGAR tool's `GetNext` might parse the output JSON for a list of ticker symbols mentioned in filings. If tickers are present, it returns `("yfinance_stock_data", {"symbol": <firstTicker>, "period":"1y","interval":"1d"})`. This means "after EDGAR, call YFinance for that ticker's price history". If multiple tickers are relevant, it might even schedule multiple next calls or choose one (perhaps the portfolio's own ticker). - Or, a simpler logic: always call YFinance after EDGAR (no condition) – which could be set without parsing output, just a rule. - Another example: imagine a `FinancialStatementsTool` that pulls financial statements. Its `GetNext` could automatically trigger a `RatiosTool` to calculate financial ratios from those statements.

Tool Wrapper for Chaining: To implement chaining without altering the core AI client too much, we can introduce a **ToolChain wrapper**. This would be a decorator that wraps any standard `Tool` and manages the sequence. The wrapper's `Run` would: 1. Execute the base tool's `Run(args)` to get output. 2. Parse

the output (e.g., `json.Unmarshal` to a map or struct). 3. Call the base tool's `GetNext(currentToolName, output)` to get the next tool and args (if any). 4. If a next tool is specified, look up that tool in the registry (all tools are registered in a map) and call its `Run` with the provided args. 5. Potentially repeat this process if tools can chain multiple in a row (like a linked list). This could continue until `GetNext` returns nil/no next tool.

The wrapper could also accumulate outputs if desired – for instance, combining results from multiple tools into one JSON. But an easier approach is to simply return the final tool's output (or have the final output contain all intermediate results in a structured way). One design is to nest outputs, e.g., EDGAR's output could carry a field that includes the subsequent YFinance data when chained, giving a composite result.

Example Workflow (chained tools): Suppose we set up: SEC EDGAR → YFinance. If an engine calls the SEC EDGAR tool (via the AI or orchestrator), the ToolChain wrapper will ensure the YFinance call happens immediately after EDGAR. So the AI (or engine) only requested one tool, but behind the scenes two API calls occurred. The final data might look like:

```
{
  "sec_edgar": { ... filings data ... },
  "yfinance_stock_data": { ... price data ... },
  "metadata": { "chain": ["sec_edgar", "yfinance_stock_data"], "primary":
"sec_edgar" }
}
```

(This is one possible structure if we combine outputs.) The AI would then have both filings and price data available at once, without needing to issue a second function call itself. This makes the AI's job easier – it asked for filings and magically also got stock prices.

Generalizing to Engines: The NEXT_PATTERN concept isn't limited to tools. We could allow engines to similarly declare next engines. However, since the EngineOrchestrator already sequences engines, dynamic engine jumps are more complex and not implemented yet. The initial focus is on tools because tools often represent data retrieval that naturally chains (output of one is input to another). Engines are higher-level and usually we know their order upfront (plus they all share context, so less need to branch). Thus, the chaining wrapper will primarily orchestrate *tool chains within an engine's execution*. In effect, it's extending the AI's tool-calling mechanism with a deterministic script.

Implementation Details:
- We extend `models.Tool` interface (or create a sub-interface) with `GetNext(prevName string, prevOutput any) *NextAction`. `NextAction` could be a struct holding the next tool name and arguments for it. If no next action, it returns nil.
- Each tool that wants chaining can implement `GetNext`. Simpler: have a default implementation that just returns the first from a `NextTools` list (ignoring output), and override in specific tools for conditional logic. - The tool registry can compose chains. For instance, we might create a **composed tool** that wraps multiple calls. Alternatively, the orchestrator/AI client can be made aware of chaining: after a tool execution, check if it implements chaining and loop accordingly. - A **chain budget** might be wise – e.g., prevent infinite loops by limiting chain depth. But if chains are static or acyclic by design (which they should be), this isn't an issue. - Logging and error handling: If Tool A succeeds but Tool B in chain fails, how to handle? Possibly treat

it as overall failure, or return partial data with an error field. We should define that. Perhaps include an `"error"` in the combined output for Tool B if it fails but still return what Tool A got, so the AI is aware.

Use Cases:
1. **Data Enrichment:** EDGAR → YFinance → FMP chain: Filings give tickers, use YFinance to get current market data for those tickers, then use FMP to get financial metrics, all automatically. This provides an engine (like InvestmentResearchEngine) a one-stop call to gather all relevant data.
2. **Fallbacks:** If a primary tool fails or is empty, `GetNext` could direct to an alternate. For example, a `polygon` market data tool could list `yfinance` as next if Polygon has no data for a ticker – automatically trying Yahoo as a backup.
3. **Multi-step calculations:** A "ChartsTool" might generate a chart image and then automatically invoke a "UploadTool" to upload that image and get a URL. The chain hides complexity from the AI – it just gets back a URL to the chart.

Chaining and AI Orchestration: By formalizing tool chains, we reduce the reasoning burden on the AI. Instead of the AI figuring out "I should call Tool B after Tool A", the system ensures Tool B is called and its data included. This also means the prompt to the AI can be simpler (fewer back-and-forth turns). It becomes more like a traditional pipeline of functions. Essentially, we're moving some logic from the AI's "thought process" into the system's deterministic process. This can improve reliability (the AI can't forget to call a needed tool) and efficiency (fewer GPT tokens used deciding on calls).

Limitations and Considerations:
- We must parse `prevOutput` reliably. Since each tool returns a JSON string, the chaining logic will parse it into `map[string]interface{}` or the known output struct. This requires the output to indeed be valid JSON (our structured output design helps here).
- If output is large, passing it to the next tool might require picking specific parts. For example, EDGAR output could be huge (full text of filings). The `GetNext` should extract only what Tool B needs (e.g., a list of symbols or a summary) to form next args, rather than pass the entire output. So careful coding in `GetNext` is needed to avoid inefficient data passing.
- Tool chaining should not conflict with AI's own function calling. One approach: the AI still calls the first tool in a chain as if standalone, and the chain wrapper seamlessly handles subsequent calls. From the AI's perspective it made one function call and got a richer response. This is backward-compatible and does not require changing how prompts are written (the function definition of Tool A remains the same, but the returned result contains more info).
- In debug or verbose mode, it should be clear that multiple tools ran. Logging each step of the chain is important for observability. Perhaps the combined output includes which tools were invoked (as in the metadata "chain" list in the example above).

The NEXT_PATTERN model thus brings a more **deterministic, rule-based workflow** on top of the AI's flexible reasoning. It is a form of **tool orchestration layer**. We can start by chaining obvious pairs (like filings → market data) and later extend to more complex graphs. Over time, as more tools are added (SEC, alternative data, etc.), chaining will help manage complexity by automating multi-step data gathering. In effect, we're giving Mosychlos a built-in recipe for using its tools, rather than leaving all the planning to the LLM. This makes the system more **predictable and faster**.

5. Implementation Ideas & Limitations

While Mosychlos's architecture is powerful (single-context engine chaining, unified tool interface, structured outputs), there are some limitations and areas for improvement that guide the roadmap. Below we discuss these along with ideas to address them:

Engine Orchestrator Behavior: Currently, the **EngineOrchestrator** runs engines in a fixed sequence and stops on any error [108]. This all-or-nothing sequential execution is straightforward but not resilient. One limitation is lack of **error recovery or skipping** – if one engine fails (e.g., a tool within it errors out or the AI fails to produce output), the whole pipeline aborts. An enhancement would be to catch engine errors, log them to SharedBag (perhaps under an `engine_errors` key), and continue with the next engine if possible. This way, a non-critical failure (say NewsEngine failing to fetch data) wouldn't derail the final outcome.

Static Pipeline vs. Conditional Logic: As noted, engines cannot make decisions about which engine runs next. The pipeline order is typically hard-coded or configured, and every engine in the list executes. In practice, some engines might not always be relevant – for example, a ComplianceEngine might not need to run for certain portfolios or might only need to run for portfolios in regulated accounts. Currently, there's no mechanism to skip it based on context. Introducing simple conditions (perhaps via engine `Dependencies()` or a pre-check using SharedBag state) could streamline the workflow. One idea is to allow an engine to declare prerequisites in SharedBag (e.g., "only run if SharedBag KComplianceRules exists"). This would be a form of engine chaining logic (a simpler analog to tool chaining) that orchestrator could honor.

Parallel Execution: The architecture documentation suggests that independent engines *could* run in parallel [109] (e.g., risk and news analysis can be concurrent). The current implementation, however, runs them sequentially by default [110]. Exploiting concurrency could speed up analysis – for instance, run NewsEngine and RiskEngine at the same time, since they don't depend on each other, then synchronize before AllocationEngine which needs risk's output. Implementing this requires orchestrator to detect independent engines (perhaps via the `Dependencies` list) and use goroutines or asynchronous calls. It's a to-do item to make the orchestrator smarter in scheduling. The limitation is mainly performance; logically, results would be the same, but faster execution can reduce user wait time especially if some engines (like those calling many tools) are slow.

Context Length Management: By using a single AI session for all engines, the context (conversation) grows with each engine's output appended. This is great for continuity, but it risks hitting token limits for long analyses. Right now, engines simply accumulate context, and the final engine sees everything. If the number of engines or length of outputs increases (consider adding more detailed analysis engines), we might need to prune or summarize intermediate results in SharedBag before feeding to later engines. Currently, there is no automatic summarization or context truncation. An idea is to perform a **mid-pipeline summary** – e.g., after two or three engines, have the AI produce a concise summary of the state so far, replace the detailed content with that summary in context (while still keeping detailed data in SharedBag for reference if needed). This is complex and not yet implemented, but it's a potential solution to context bloat.

Multi-Persona / Multi-Step Reasoning: The orchestrated single session already simulates a multi-agent discussion by sequential prompts [5]. However, to fully realize FinRobot-like capabilities, engines may need

to incorporate prompts with **multiple persona dialogue in one engine**. For example, the ReallocationEngine could be extended to have the AI role-play a committee conversation (rather than a single assistant voice) [31]. This requires prompt tweaks but not structural changes. The limitation is mostly on prompt engineering side – ensuring the AI can output a coherent multi-perspective analysis within one JSON. The roadmap's **Phase 1** and **Phase 2** (from the Engine Chaining analysis) already target this: adding multi-perspective prompts and ensuring context from previous engines is used [111] [112]. The plan is to refine prompts rather than code to achieve this, which is in progress.

Tools Integration and Expansion: A limitation was that some data was missing or had to be faked due to lack of tools. The Phase 3 enhancements include **adding more tools** (e.g., advanced financial data, alternative data) [113] [114] so that engines have richer info to draw on. For instance, earlier if the AI needed SEC filings data, it wasn't available – the engine might have just skipped that detail or hallucinated. Once the SEC EDGAR tool is in place, that limitation is removed and engines can rely on real filings. Similarly, adding tools for crypto data, Reddit sentiment, etc., will broaden the analysis. The key is orchestrating their use via ToolConstraints (ensuring engines know they exist). The limitation here was simply development time – the system is built to integrate tools, and many are planned (as listed in Tools Inventory), so it's more about implementation completion.

Report Generation and Output: Right now, after engines finish, the results are in SharedBag and logs. The example final output might be the JSON from ReallocationEngine stored under `KAnalysisResults` or similar. To present this to end users (e.g., in a PDF or interactive report), a separate **ReportGenerator** component is considered. There is an outline for generating professional reports (including charts, tables) from the analysis [115]. In fact, Mosychlos has a `report` package (with a renderer) and supports output formats like Markdown and PDF. A limitation is that the integration between analysis and report generation might not yet be fully automated (the `GenerateReports` function in orchestrator is stubbed out [116]). Implementation idea: once analysis is done, automatically feed the collected results into a reporting engine that uses templates to format a PDF or Markdown. This would close the loop, delivering a polished output to users. Currently, users may have to manually inspect JSON or logs, which is not ideal. So, improving the **end output formatting** is on the roadmap (professional report templates and possibly a `pdf_generator_tool` as mentioned in FinRobot plan [117]).

OpenAI API Usage Limits: The project recognized it lacked proper **rate limiting for OpenAI calls** [118]. While not a user-facing feature, this is an important implementation detail. Hitting OpenAI's limits could cause failures mid-analysis. They have a design for a middleware to handle rate limits and retries [119] [120]. Implementing that will improve reliability under heavy usage. Until then, the limitation is that if too many calls are made too fast (e.g., in a batch of engines), they might get rate-limited by OpenAI. The interim measure is possibly to slow down or limit parallel calls manually. This is being addressed as per the `openai-rate-limiting-architecture.md` plan.

Tool Chaining Implementation: The concept of tool chaining (Section 4) is planned but not yet live. So currently, any chaining of tools relies on the AI's own decisions. This is a limitation because the AI might not chain optimally. The idea to create a `ToolChain` wrapper and the `GetNext` method will have to be coded and tested. Challenges to implement include how to represent the combined output (as discussed) and how to integrate it so that from the AI's perspective, it's just calling one function. This might involve registering the chain as a new "meta-tool" or intercepting calls to certain tools. While design is clear, careful implementation is required to maintain backward compatibility with existing function call structures. The team will likely prototype this with a simple chain (like EDGAR → YFinance) and expand from there.

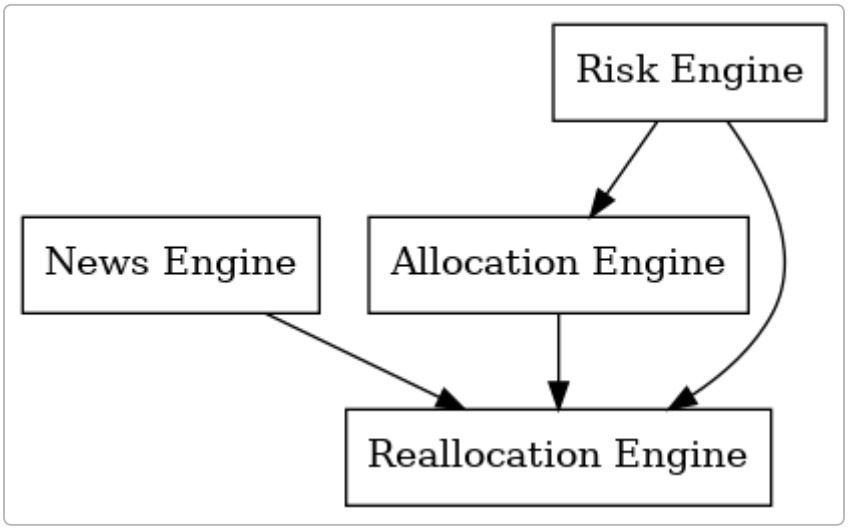
Security & Robustness: Minor points: the system must ensure that one engine's output doesn't maliciously or accidentally break the next (structured output helps here). Also, if the AI's output JSON is slightly invalid, currently an engine might throw an error on parsing and stop. Adding more tolerant parsing or automatic correction (maybe by re-asking the AI with format correction) would be useful. That's more of an AI response validation improvement than architecture, but it's part of making the system robust.

In summary, **current limitations** such as rigid engine ordering, lack of internal branching, sequential execution, and partial tool coverage are recognized and being actively worked on. The **roadmap implementation ideas** center on: - **Dynamic chaining** (for tools now, perhaps engines later) to inject more logic into the workflow (making the system smarter about using data). - **Parallelization and performance** improvements for speed. - **Enhanced data integration** (new tools, better use of external info like web content via summarization). - **Better outputs** (automated report generation, multi-format support).

The system's strength is its single-session design which naturally accumulates knowledge across engines [35] – that will be preserved. Most improvements build on this strength: e.g., using summarization to keep context manageable, or adding multi-persona in-session dialogues to enrich that single session. By iterating on these, Mosychlos aims to achieve a very comprehensive, automated investment research process that is both **flexible** (due to AI reasoning) and **structured** (due to system-imposed workflows).

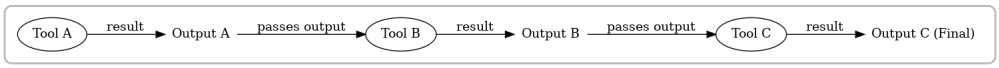
6. Diagrams

To illustrate the architecture, below are two diagrams: one for **Engine Chaining and Dependencies**, and one for the **Tool Chaining workflow**.



Engine orchestration and dependencies. **Diagram 1: Engine Chain:** This diagram shows a typical engine pipeline and how certain engines depend on others' outputs. All engines share the same context and state, but the **Risk Engine** output feeds into the **Allocation Engine** (risk analysis is needed to optimize allocation), and the **News Engine** (as well as Risk and Allocation results) feed into the final **Reallocation Engine** which synthesizes everything. This reflects the configured execution order [33] [30]. (Compliance Engine, not shown here, would similarly run independently and its results could also feed the final engine or reporting stage.) The orchestrator ensures that dependent engines (Allocation depends on Risk) run after their prerequisites [121]. Solid arrows indicate flow of data (via SharedBag) rather than direct function calls – e.g. the

Reallocation Engine reads the outputs of News, Risk, and Allocation from SharedBag to produce the final analysis.



Tool chaining via NEXT_PATTERN. **Diagram 2: Tool Chaining Sequence:** This diagram depicts how the proposed NEXT_PATTERN tool chaining works in a linear sequence. **Tool A** runs first and produces **Output A**. Immediately, the chain logic uses Output A to determine the next step and invokes **Tool B**. Tool B then produces **Output B**. The chain continues to **Tool C**, and so on. In this schematic, arrows "result" show the output generation, and "passes output" indicates the hand-off of data to the next tool's input. For example, Tool A could be `sec_edgar` fetching filings, Tool B could be `yfinance_stock_data` using the ticker from Output A, and Tool C could be `fmp` getting fundamentals – all triggered in one flow. The final **Output C** would contain the culmination of all steps. The AI or engine that initiated Tool A doesn't need to explicitly call B or C; the chain wrapper manages it. This ensures a deterministic multi-tool workflow: each tool's output is normalized and fed forward. The result to the AI can be a merged structure or just the final output, depending on implementation. The NEXT_PATTERN design thus creates a **tool pipeline** similar to a UNIX pipe (`ToolA | ToolB | ToolC`), but in the context of AI function calls.

These diagrams highlight the separation of concerns: engines form the high-level analytical pipeline (with each engine potentially calling multiple tools internally), and tool chaining provides fine-grained data gathering sequences within or across engines. By combining these, Mosychlos achieves a layered orchestration – from portfolio-level analysis stages down to automated multi-tool data retrieval. The visual flows underscore how adding chaining capability will transform a sequence like *Engine → (Tool1, Tool2, …)* into a richer graph of actions, all while maintaining structured interactions and a single coherent AI session.

References: The engine chain design is documented in Mosychlos's engine README and FinRobot mapping [33] [122], and the idea of sequential multi-tool workflows is inspired by planned enhancements in the project documentation [123]. The diagrams encapsulate those concepts in a simplified form for clarity.

---

[1] [2] [3] [4] [5] [31] [32] [35] [111] [112] [115] engine-chaining-single-context-analysis.md
https://github.com/amaurybrisou/mosychlos/blob/dde6b2005935822034e19a8bb344e7b7f5278f13/docs/engine-chaining-single-context-analysis.md

[6] [7] [8] [9] [12] [13] [14] [15] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [33] [34] [109] [121] README.md
https://github.com/amaurybrisou/mosychlos/blob/dde6b2005935822034e19a8bb344e7b7f5278f13/internal/engine/README.md

[10] [11] [16] [17] [18] [19] [20] [102] [104] risk.go
https://github.com/amaurybrisou/mosychlos/blob/dde6b2005935822034e19a8bb344e7b7f5278f13/internal/engine/risk/risk.go

[36] [37] [38] [39] [42] [43] [44] [45] [47] [48] [64] [66] [67] [68] [84] [97] [99] [113] [114] tools-inventory.md
https://github.com/amaurybrisou/mosychlos/blob/dde6b2005935822034e19a8bb344e7b7f5278f13/docs/tools-inventory.md

[40] [41] [69] [70] [71] [72] [73] fmp.go
https://github.com/amaurybrisou/mosychlos/blob/dde6b2005935822034e19a8bb344e7b7f5278f13/internal/tools/fmp/fmp.go

[46] [81] [82] [83] tool_registry.go
https://github.com/amaurybrisou/mosychlos/blob/dde6b2005935822034e19a8bb344e7b7f5278f13/internal/tools/tool_registry.go

49 50 51 52 53 54 55 56 57 58 59 60 65 README.md
https://github.com/amaurybrisou/mosychlos/blob/dde6b2005935822034e19a8bb344e7b7f5278f13/internal/tools/yfinance/README.md

61 62 63 stock_data.go
https://github.com/amaurybrisou/mosychlos/blob/dde6b2005935822034e19a8bb344e7b7f5278f13/internal/tools/yfinance/stock_data.go

74 75 76 77 78 79 80 98 client.go
https://github.com/amaurybrisou/mosychlos/blob/dde6b2005935822034e19a8bb344e7b7f5278f13/pkg/fmp/client.go

85 86 87 88 89 90 91 92 93 94 95 96 fred.go
https://github.com/amaurybrisou/mosychlos/blob/dde6b2005935822034e19a8bb344e7b7f5278f13/internal/tools/fred/fred.go

100 101 investment-research-engine.md
https://github.com/amaurybrisou/mosychlos/blob/dde6b2005935822034e19a8bb344e7b7f5278f13/docs/investment-research-engine.md

103 ai-architecture-refactoring-proposal.md
https://github.com/amaurybrisou/mosychlos/blob/dde6b2005935822034e19a8bb344e7b7f5278f13/docs/ai-architecture-refactoring-proposal.md

105 WaterCrawl - Modern Web Crawling Framework
https://watercrawl.dev/

106 107 108 110 116 orchestrator.go
https://github.com/amaurybrisou/mosychlos/blob/dde6b2005935822034e19a8bb344e7b7f5278f13/internal/engine/orchestrator.go

117 122 123 finrobot-enhancement-plan.md
https://github.com/amaurybrisou/mosychlos/blob/dde6b2005935822034e19a8bb344e7b7f5278f13/docs/finrobot-enhancement-plan.md

118 119 120 openai-rate-limiting-architecture.md
https://github.com/amaurybrisou/mosychlos/blob/dde6b2005935822034e19a8bb344e7b7f5278f13/docs/openai-rate-limiting-architecture.md