ChatGPT

Agentic Orchestration for LLMs in Financial Recommendations and Advice

Academic Research Approaches (2023–2025)

FinRobot (2024) – Open-Source Multi-Agent Financial AI Platform

Summary: *FinRobot* is an open-source platform that exemplifies agentic orchestration in finance [1] [2]. It organizes multiple specialized **Financial AI agents** to tackle complex tasks, using *Financial Chain-of-Thought (CoT)* prompting to break down problems into logical steps [2]. The architecture is layered: a **Perception** module ingests multimodal financial data, a **Brain** (LLM) module reasons with CoT and decides actions, and an **Action** module executes tools (e.g. trading or reporting) [3]. A central *Smart Scheduler* (Director Agent) orchestrates task assignment to the best-suited agent or model, enabling use of diverse LLMs for different sub-tasks [4]. This design improves reasoning and accuracy by decomposing analysis and matching each part with an optimal agent or model.

Implementation Guidance: To apply FinRobot's approach in a new project:
1. **Layered Architecture:** Divide your system into layers. Include an **LLM-powered reasoning agent** (the "Brain") and subordinate tool-using agents. Implement a *perception layer* to feed relevant data (market feeds, documents) to the LLM, and an *action layer* for the LLM to invoke tools or APIs (e.g. trading functions, database queries) [3].
2. **Chain-of-Thought Prompting:** Design prompts or agent logic that break down financial questions into steps. For example, prompt the LLM agent to **think step-by-step** through portfolio analysis or risk assessment. This CoT strategy helps the agent handle complex reasoning, akin to how FinRobot's agents dissect problems into logical sub-tasks [2].
3. **Specialized Agents & Tools:** Create specialized sub-agents for distinct tasks (e.g. *Market Forecasting Agent*, *Document Analysis Agent*, *Trade Strategy Agent*). Assign each agent specific tools or data sources (APIs for financial data, document parsers, calculators). Each agent should have a clear goal and domain expertise to improve accuracy in its niche [2].
4. **Orchestration & Scheduling:** Implement a **controller or scheduler agent** that monitors tasks and delegates subtasks to the appropriate specialized agent. FinRobot's Director Agent plays this role, ensuring each query is routed to the best agent and LLM model for that job [4]. In practice, this means setting up a workflow where the primary agent parses the user's query and then dynamically invokes sub-agents in sequence or in parallel as needed.
5. **Model Selection for Tasks:** If different LLMs or models excel at different tasks, integrate a mechanism to select models per task (FinRobot's *multi-source integration* picks suitable models for each financial task [5]). For example, use a larger finance-tuned model for quantitative analysis, and a smaller one for simple queries to save costs.
6. **Iteration and Memory:** Incorporate memory or state tracking so agents learn from prior steps. FinRobot's design allows agents to maintain context and results between modules (e.g. passing interim calculations from Brain to Action). Ensure your agents log their reasoning ("show your work") for transparency and debugging.

By following this blueprint, you create an orchestrated agent system that can perceive financial data, reason through complex analyses, and act (e.g. recommend products or execute trades) in a coordinated, autonomous manner. This improves reasoning depth and ensures each sub-problem is handled by an expert agent, leading to more personalized and accurate financial recommendations.

FinAgent (2024) – Multimodal Trading Agent with Reflection

Summary: *FinAgent* is a multimodal LLM-based trading agent that demonstrates how agentic orchestration improves decision-making in finance. It combines text, numerical data, and even charts (visual data) to inform trading decisions [6]. Internally, FinAgent is composed of modules including a **Market Intelligence module** (aggregating news, prices, etc.), a dual-level **Reflection module** (which has low-level agents analyzing market reactions to data, and high-level agents reflecting on the agent's own past trading decisions), and a tool-augmented **Decision module** [7] [8]. The *reflection agents* enable the system to adapt to market changes and learn from historical outcomes, improving reasoning and trustworthiness of its advice [9]. FinAgent also integrates domain knowledge (e.g. known trading strategies and expert rules) into the decision process [10]. In experiments it significantly outperformed prior methods, boosting trading profit metrics by ~36% on average [11]. This shows that orchestrating multiple reasoning processes (data analysis, reflection, memory retrieval) leads to better financial recommendations and actions.

Implementation Guidance: Key steps to implement a FinAgent-like system:
1. **Multimodal Data Integration:** Equip your agent with the ability to handle multiple data types. Set up sub-agents or tool pipelines for each modality – for example, one component fetches and interprets numeric market data (time series), another parses textual news or reports, and another handles visual inputs like price trend charts [6]. Ensure the primary agent can orchestrate these and combine their insights (e.g. correlate news sentiment with price movements).
2. **Reflection Mechanisms:** Implement a two-tier reasoning loop. Introduce a *low-level reflection agent* that, after initial analysis, reviews how market data correlates with outcomes (e.g. "Given the news and price change, did our prediction align with actual market move?"). Then include a *high-level reflection agent* that periodically assesses the agent's own past decisions – identifying errors or successes in previous recommendations [12] [13]. For instance, after a week of portfolio suggestions, this agent can analyze which suggestions performed poorly and why. Incorporate these reflections back into the decision policy to continuously improve advice quality.
3. **Memory and Learning:** Add a memory store for historical data and agent experiences. Use a vector database or structured memory to let the agent retrieve past situations similar to the current query. FinAgent's diversified memory retrieval system lets it **learn from historical data**, which you can emulate by storing prior user portfolios, market conditions, and outcomes [9]. When giving new advice, the agent can recall analogies from memory (e.g. "This situation resembles last month's interest rate hike scenario").
4. **Tool Augmentation:** Provide the agent with tools for calculations and data retrieval. For example, integrate APIs for real-time stock quotes or macroeconomic data and allow the agent to call these tools during its reasoning. FinAgent leveraged tools in its decision module (it references *Toolformer*-like abilities) [14] – your agent should be able to invoke functions such as a pricing model, risk calculator, or external ML model to refine its output.
5. **Domain Knowledge & Rules:** Encode domain-specific guidelines into the agent's process. FinAgent was explicitly **rooted in sound financial principles**, incorporating expert trading rules [10]. Similarly, you might hard-code or fine-tune the agent on financial best practices (e.g. diversification rules, regulatory constraints). During generation, have the agent check its recommendations against these rules (for example, a compliance sub-agent can veto advice that violates risk limits).

6. **Evaluation and Tuning:** Simulate the agent's performance on historical scenarios. FinAgent's creators evaluated it on multiple datasets and saw improved profits [11]. In your project, backtest the agent's advice: feed it past market data and see if its recommendations would have been profitable or appropriate. Use this feedback to adjust the agent modules (tweak prompts, retrain decision models, refine reflection criteria).

By integrating rich data sources and adding self-reflective agents, you create a finance advisor that not only analyzes more holistically (news + numbers + context) but also **learns from its mistakes and successes**, leading to more reasoned, trustworthy, and personalized recommendations over time [9].

ElliottAgents (2025) – Multi-Agent Stock Forecasting with Tech Analysis

Summary: *ElliottAgents* is a research prototype that marries traditional financial analysis techniques with LLM-based agents for stock trend forecasting [15]. It uses a **multi-agent architecture (7 agents)** orchestrated in a pipeline to analyze stock data through the lens of the Elliott Wave Principle (EWP, a technical analysis framework) [16] [17]. Key agents include: a *Coordinator* agent (primary orchestrator), a *Data Engineer* (fetches historical prices), an *Elliott Wave Analyst* (identifies technical patterns using an EWP tool), a *Backtester* (validates patterns with reinforcement learning on historical data), a *Technical Analysis Expert* (chooses the most plausible wave pattern scenario), an *Investment Advisor* (synthesizes insights into actionable buy/sell advice with context from a retrieval tool), and a *Report Writer* (produces a user-friendly report) [18] [19]. The agents communicate sequentially, each contributing a piece of the analysis which the next builds upon, under the supervision of the Coordinator. By combining *classical analysis* (EWP patterns) with *modern AI* (LLMs for pattern recognition, RAG for context, DRL for learning), this orchestrated system improved interpretability and reliability of market predictions [20] [21]. Experiments showed the multi-agent approach effectively recognized complex wave patterns and adapted to changing market conditions, yielding profitable forecasts in medium/long-term tests [22]. This showcases how agentic orchestration can enhance recommendation quality (more nuanced trading signals) while keeping reasoning transparent for end-users (through the report and "show work" at each agent stage).

Implementation Guidance: Lessons for building a similar orchestrated forecasting/advisory system:
1. **Task Decomposition:** Break the overall problem (e.g. "Recommend an investment strategy for stock X") into a sequence of sub-tasks handled by dedicated agents. ElliottAgents did this by following a logical finance workflow: data gathering → pattern recognition → strategy formulation → reporting [17] [23]. In your project, identify the stages of analysis. For instance, stages could be: *(a)* gather relevant financial data (pricing, fundamentals), *(b)* apply analysis models (technical indicators or fundamental valuation), *(c)* evaluate the results or run simulations, *(d)* formulate advice, and *(e)* explain the recommendation. Create one agent (or LLM prompt) per stage.
2. **Define Specialized Roles and Tools:** Implement each stage as an agent with a clear role and equip it with needed tools or models. In ElliottAgents, the *Elliott Wave Analyst* had a custom tool to detect wave patterns, and the *Backtester* agent employed a deep reinforcement learning module to test those patterns [24]. Similarly, you might use a *Fundamental Analysis Agent* with access to company financials and a valuation model, or a *Risk Analysis Agent* that uses a Monte Carlo simulator. Provide each agent with a prompt that outlines its specific duties and input/output format (the paper shows an example Investment Advisor prompt guiding it to produce comprehensive strategy with risk highlights [25] [26]).
3. **Agent Orchestration (Coordinator):** Implement a top-level orchestrator that passes data and control flow through the agent pipeline. The Coordinator should initialize the process (e.g. by feeding the user's query and context into the first agent), then route the output of each agent to the next. In code, this could

be a loop or state machine that triggers Agent1, takes its result and feeds Agent2, and so on. Ensure the data passed along includes both the raw data and any intermediate insights (ElliottAgents, for example, forwarded identified chart patterns and backtest outcomes into the Investment Advisor's context) [23]. This chaining ensures each agent builds upon prior results, improving the final outcome's coherence.

4. **Continuous Learning:** To improve personalization and adaptability, incorporate a feedback loop. ElliottAgents included a reinforcement learning *continuous learning agent* to refine pattern validation over time [27] [28]. In your case, after delivering advice, monitor real outcomes (did the stock actually go up? Did the advice align with the client's preferences?). Use these outcomes to fine-tune agents. For example, if the risk analysis agent consistently underestimates volatility, adjust its parameters or training data. Over time, this continuous learning will make recommendations more tailored and accurate.

5. **Combine Knowledge Sources (Hybrid AI):** One strength of ElliottAgents was combining human domain knowledge (EWP rules) with AI. You can similarly blend **retrieval-augmented generation** for context (e.g. retrieve related historical cases or pertinent news) with algorithmic analysis. Implement a *knowledge retrieval agent* or integrate a vector database query at the stage where strategic context is needed (ElliottAgents' Investment Advisor pulled in extra context via a RAG tool) [29] [30]. This could fetch relevant market analogies or expert explanations to enhance the advice's depth and justify it, thereby improving user trust.

6. **User-Friendly Output:** Lastly, dedicate a component to translating the multi-agent process into a clear recommendation for the end user. The *Report Writer* agent in ElliottAgents takes all aggregated insights to produce a concise report [19]. You should implement a similar final step: format the output with explanations (e.g. "Our recommendation is to **Buy** stock X because A, B, C..."), include supporting data (charts or key metrics), and note any assumptions or risks (as ElliottAgents explicitly did via the Investment Advisor prompt guiding to highlight risks [26]). This ensures the sophisticated reasoning remains transparent and actionable to the client receiving the advice.

By orchestrating a team of narrow-focused agents in a logical workflow, you can mirror ElliottAgents' success: the resulting system can leverage multiple analysis techniques and data sources, yielding recommendations that are **more reliable (backed by validation), interpretable (each step can be reviewed), and tailored to complex financial decision processes** [20] [21].

Industry Implementations & Case Studies

Broadridge *BondGPT / BondGPT+* (2023–2025) – Agent-Orchestrated Bond Trading Assistant

Summary: BondGPT is a real-world enterprise application that showcases agentic orchestration for financial product recommendation and advisory. Developed by Broadridge Financial Solutions for bond trading desks, it uses an LLM (OpenAI GPT models) as a controller orchestrating **multiple specialized AI agents** to fetch and analyze data across various sources [31]. For example, when a trader asks a question (like "What's the best bond to buy given X criteria?"), BondGPT's orchestrator LLM delegates subtasks to different agents: one might retrieve market data from internal databases, another queries analytical models (e.g. pricing models, risk models), and another agent fetches relevant news or research. All these results are then aggregated by the LLM to form a final answer. This design produces *timely, accurate* responses by automatically pulling in up-to-date info from multiple datasets and models simultaneously [32]. **Personalization** is a key focus: BondGPT+ (the enterprise version) integrates the client's proprietary data and uses user-specific profiles (like the trader's role or portfolio holdings) to tailor answers [33] [34]. It also introduced governance features: a "**Show Your Work**" transparency option that reveals the step-by-

step reasoning and data sources used (enhancing trust), a *multi-agent adversarial checker* (agents cross-verify each other's results for higher accuracy), and an *AI compliance agent* that ensures answers adhere to compliance rules [35]. By 2025, Broadridge secured a patent for this LLM-orchestration method, underscoring its novelty. In practice, BondGPT improved traders' workflows by saving time (automating data gathering) and providing more comprehensive pre-trade analysis than a single model could [33]. This resulted in better-informed product recommendations and trade ideas, all delivered with personalized insight and compliance in mind.

Implementation Guidance: To implement a BondGPT-like orchestration in your project:
1. **Central Orchestrator LLM:** Use a high-quality LLM as the "brain" that interprets user queries and plans the solution. The orchestrator should break the query into sub-tasks and invoke various **ML agents or tools** to fulfill them [31]. For instance, upon a question about bonds, the LLM might decide: (a) get current market quotes, (b) analyze credit risk metrics, (c) retrieve relevant news. You can implement this by defining a prompt format where the LLM outputs directives or tool-calls (following a format like OpenAI function calling or a custom schema) which your system executes.
2. **Multi-Agent Integration:** Integrate multiple data sources and analytics by creating specialized agents or tool APIs for each. In a finance context, set up agents for **data retrieval** (market data, yield curves, financial statements), **analytical modeling** (e.g. an agent that runs a pricing model or Monte Carlo simulation), and **external knowledge** (search or document retrieval for research reports). Ensure each agent can communicate results back to the orchestrator. For implementation, this could mean writing functions or microservices for each capability and allowing the LLM to call them. BondGPT's patent notes using *multiple datasets and analytical models in parallel* [36] – mimic this by designing your system to fetch data concurrently where possible (for speed) and then have the LLM wait for and merge results.
3. **Personalization Module:** Incorporate user-specific context to tailor advice. Maintain a **user profile** (e.g. role, permissions, preferences, portfolio holdings) and allow the orchestrator or a dedicated agent to filter and adjust outputs based on it [34]. For example, if the user's role is "risk officer," the answer might emphasize compliance and risk; if a portfolio manager, the answer can reference their current holdings or strategy. Implement role-based prompts or retrieval: the orchestrator can inject phrases like "(User is a conservative investor)" into its planning context, and a profile agent can enforce data-security rules (only retrieve data the user is allowed to see). This ensures recommendations are not one-size-fits-all but aligned with the user's needs and constraints.
4. **Transparency and Verification:** Include a mechanism for the system to explain and verify its reasoning. Borrow BondGPT's "Show your work" concept by having the LLM produce a reasoning trace or list of sources used [37]. Technically, you can prompt the LLM to output an **explanation** alongside the answer or log its intermediate rationale in a hidden field. Additionally, implement an **adversarial agent** or at least a sanity-check step: for critical calculations, have a secondary method or agent re-compute the result. For example, if the main agent recommends a bond, have another agent independently fetch that bond's attributes and confirm key figures (yield, rating) match the recommendation. If there's a discrepancy, the orchestrator can resolve it (perhaps asking the LLM to reconsider or correct based on the second opinion). This cross-verification boosts accuracy and reliability of the advice [37].
5. **Compliance and Constraints:** In financial advising, adherence to regulations is crucial. Program a *compliance agent* or rule-checker that runs every recommendation through a set of rules [38]. Implementation could be as simple as a checklist the final answer must pass (e.g. no prohibited phrases, required disclosures are present) or as complex as using another LLM fine-tuned on compliance to review the advice. For instance, if a user asks for advice that ventures into regulated territory ("Should I invest in…?"), the compliance layer might ensure the response includes a disclaimer (not personalized financial advice) or avoids outright directives if that's against policy. By integrating this step in the agent chain, you

safeguard the output, which is especially important in enterprise settings.
6. **Efficiency and Scalability:** Use parallelism and resource management for speed. BondGPT was prized for providing *timely* answers by simultaneous data retrieval [31]. Architect your orchestrator to launch data agents in parallel threads or async tasks, rather than serially, and then aggregate responses. Also, plan for scaling: as data sources or models grow, the orchestrator may become a bottleneck. You can scale horizontally (multiple orchestrator instances) or throttle tasks. Consider using event-driven orchestration frameworks (like Ray or Celery in Python) to manage multiple agent calls gracefully.

By combining these elements, you'll build a system where an LLM intelligently coordinates many moving parts – much like BondGPT – resulting in rich, custom-tailored financial recommendations. This approach directly improves **recommendation quality** (by drawing on comprehensive data/models), **reasoning transparency** (via step-by-step justifications), and **personalization** (through user-aware adjustments), which were all critical innovations in BondGPT's success [33] [35].

Amazon Bedrock Multi-Agent Financial Assistant (2025) – Specialized Agents for Portfolio, Data, and Reporting

Summary: Amazon's Bedrock platform introduced a multi-agent collaboration framework, showcased through a financial advisor assistant that uses **multiple specialized agents working in concert** [39]. In their 2025 example, a three-agent architecture was built to handle investment research tasks: a **Portfolio Agent** (creating customized investment portfolios based on user criteria), a **Data/Research Agent** (gathering financial data, economic indicators, and analyzing documents like Fed meeting minutes), and a **Communication Agent** (generating reports and stakeholder emails) – all coordinated by a primary **Supervisor agent** [40] [41]. The primary agent parses the user's request (e.g. "Generate an investment report focusing on tech stocks under \$5B market cap with high growth") and either routes it to the appropriate single sub-agent or breaks it into sub-tasks for each, then aggregates their outputs [42] [43]. This hierarchical orchestration (sometimes in *router mode* for straightforward queries, and *supervisor mode* for complex ones) allows **parallel processing** of complex workflows, like simultaneously checking compliance, running risk analysis, and doing industry research [44] [45]. The benefits observed were more **in-depth analysis** (each agent dives deep into its specialty), better coverage of necessary checks (risk, compliance) with auditability, and easier scalability (one can update or add an agent without overhauling the whole system) [45] [46]. In short, the Bedrock financial assistant demonstrates that splitting a financial advisory task among expert agents leads to more robust and accountable recommendations, which would be hard to achieve with a single monolithic LLM approach [44].

Implementation Guidance: If you want to implement a similar multi-agent financial assistant:
1. **Identify Key Subdomains:** Partition the financial advisory task into a few core domains, each handled by an agent. In Amazon's example, they used three: *Portfolio construction*, *Data analysis*, and *Report generation*. Depending on your use case, you might choose agents like: **Product Recommender** (e.g. suggests specific financial products like funds or insurance based on user profile), **Analyst Agent** (performs quantitative analysis or retrieves data), **Compliance/Risk Agent** (checks recommendations against rules or risk tolerance), and **User Interaction Agent** (formats outputs, asks clarification questions). Clearly delineate what each agent is responsible for and what tools or data it can use.
2. **Primary Orchestrator (Supervisor):** Implement a top-level agent that receives user input and delegates tasks. Using a *Supervisor design* means this agent will dynamically decide how to break down the query [47]. Start by writing logic (or an LLM system prompt) for the orchestrator to do the following: interpret the user's intent and parameters (e.g. the user wants a portfolio of N companies in sector Y), then assign subtasks. If

the query is simple ("Give me latest price of X"), the orchestrator might just pick the Data agent to respond (router mode) [48]. If it's complex, it might say: "PortfolioAgent, create a list of companies; DataAgent, get financial metrics for them; CommunicationAgent, draft a report with these findings." In practice, you can implement this by having the primary agent call each sub-agent's function and passing along needed info, or by using an orchestration library that handles dependencies. Ensure the primary agent **waits for sub-agents' results**, then composes a final answer.

3. **Parallel and Hierarchical Execution:** Design the system so that independent tasks can run in parallel. For example, if one sub-agent is pulling market data and another is analyzing a PDF, they can operate simultaneously to save time, with the supervisor synchronizing at completion [44] [49]. Use asynchronous programming or background tasks to achieve this. Also, be prepared to have agents call on each other if needed (hierarchies). For instance, the Portfolio Agent, after picking stocks, might internally invoke the Data Agent for each stock's indicators – this is a form of sub-agent chaining. Amazon's Bedrock framework supports such hierarchical collaboration, and you can implement it by letting agents themselves be able to trigger other agents or by routing all through the supervisor as intermediary.

4. **Knowledge Base Integration:** In financial advice, a lot of context (financial reports, market research) might reside in documents. A practical step is to set up a **shared knowledge base** (vector store or database of docs) that certain agents can query. In the Bedrock solution, the Data assistant agent was linked to a Bedrock Knowledge Base with ingested financial PDFs [50] [51]. Implement something similar: ingest relevant documents (prospectuses, economic reports, etc.) into a vector store; give your data analysis agent access to a retrieval function that the orchestrator can call when needed (e.g. "pull the latest quarterly report for company X"). This ensures your agents provide up-to-date, factual information rather than relying on the LLM's parametric memory alone.

5. **Maintain Audit Trails:** One advantage of multi-agent setups is clearer auditability – each agent's outputs can be logged. Emulate this by having each agent produce a traceable record of their actions and findings. For example, when the Risk/Compliance agent checks a portfolio, it should output which rules were tested and any flags raised. The supervisor agent can compile these logs. This way, the final advice can be accompanied by an "audit trail" (for instance, "ComplianceAgent verified diversification and concentration limits: OK"). This is particularly useful in finance for accountability and building user trust. It also aids debugging: if a mistake is made, you can pinpoint which agent's output led to it.

6. **Modular Growth:** Design your agents to be loosely coupled and easily extendable. The Bedrock example emphasizes how new agents or data sources can be added without breaking the system [46] [52]. In practice, this means following good software modularity: each agent should have a clear interface (inputs/outputs) and not maintain hidden global state that others rely on. The orchestrator can be configured via some metadata about available agents. For instance, you could have a configuration file where you list all agent services and their roles; adding a new *"Tax Optimization Agent"* could then be as simple as adding it to the config and updating the orchestrator's logic to use it when relevant. This modular approach ensures your financial assistant can evolve (say new regulations require a new check – just plug in a new agent for that) without a total rewrite.

By following these steps, you can recreate Amazon's multi-agent financial advisor paradigm: a system where **each agent handles a slice of the problem in depth, and a supervising agent stitches together a comprehensive answer**. Such a system will excel in complex reasoning (thanks to task focus and parallelism) and can offer **personalized, high-quality advice** – for example, customized portfolios with thorough analysis and proper compliance checks – delivered in a maintainable and scalable way [44] [45].

Agentic Finance Assistant by Relari (2024) – Supervisor and Sub-Agents Framework Comparison

Summary: Relari.ai demonstrated an *Agentic Finance Assistant* as a case study to compare agent orchestration frameworks (LangGraph, CrewAI, OpenAI's Swarm) [53]. In their December 2024 blog, they built the same financial QA assistant using each framework, highlighting how the agentic design itself boosts reasoning. The assistant uses a **Supervisor Agent** at the top to interpret user questions and delegate tasks to three **sub-agents**: a *Financial Data agent* (with API tools for stock prices, financial statements, etc.), a *Web Research agent* (scraping web for news or company info), and an *Output Summarizing agent* (compiling and explaining the results) [54]. For example, if a user asks "Compare the financial health of Airline A vs its competitors," the supervisor might assign the data agent to fetch financial metrics, the web agent to gather recent news or web info, and then the summarizer to produce a comparison report. This flexible orchestration leads to comprehensive answers: each sub-agent focuses on its aspect, and the final answer benefits from both hard data and contextual knowledge. The blog noted improvements in the **reasoning structure** – the agent can decide which path to take (which sub-agent to invoke in what order) dynamically, rather than a fixed script, which made it adept at complex or ambiguous queries. They also emphasized how frameworks make it easier to manage **state and memory** between agents, handle tool outputs reliably, and integrate a human-in-the-loop if needed [55] [56]. Ultimately, the Agentic Finance Assistant illustrates that using a *supervisor + specialized experts* model yields more accurate and **context-rich financial advice** (since the agent pulls from both real-time data APIs and web info) and is more maintainable (each agent can be developed or improved independently, and the orchestration logic can be tuned or monitored framework-wise).

Implementation Guidance: To build a similar multi-agent finance QA system:
1. **Supervisor-Orchestrator Pattern:** Follow the supervisor architecture as done in this case. Implement a central agent that serves as the **dispatcher** – it should take the user query and decide which sub-agents need to contribute [54]. You can hard-code some decision logic (e.g. if query contains "website" or "news" then involve the Web agent; if it's about "financial metrics" use the Data agent; in all cases use the Summary agent at end). For more flexibility, you could allow the LLM itself (as the supervisor) to determine the plan by analyzing the question (some frameworks allow the LLM to choose actions). Start simple: perhaps always call all sub-agents and have the summary agent filter irrelevant info. Then iterate to make it smarter in invoking only what's needed.
2. **Sub-Agent Roles and Tooling:** Create the three sub-agents with distinct roles as in Relari's design: **Data Agent** (focused on quantitative financial data), **Web Research Agent** (for unstructured info from the internet), **Summarizer Agent** (for output synthesis). Equip each with appropriate tools: e.g., Data Agent gets functions like `get_stock_price(ticker)` or `query_financials(company)` tied to real financial APIs [57] [58]. Web Agent gets a web search or scraping capability (could use an API like SerpAPI or a custom crawler). The Summarizer might not need external tools but should have a template or style guide for presenting results. Give each agent a persona/prompt: e.g. "You are a Financial Data agent, you have access to X API, your goal is to retrieve and return data relevant to the query" – this keeps them focused [59] [60].
3. **Tool Integration and Testing:** When building each agent, ensure their tool outputs are in a format the others can consume. For instance, Data Agent could output a JSON with key metrics. Test each tool standalone (Relari's blog shows examples using a `@tool` decorator to wrap functions for agents [57] [58]). Once verified, integrate them so the agent can call them during its reasoning (most agent frameworks handle this if you register the tools). Make sure to handle exceptions – e.g., if an API call fails or returns nothing, decide how the agent should respond (maybe output an error message that the supervisor can detect and possibly trigger the Web agent as a fallback). Robust tool use will improve your system's

reliability in answering detailed financial questions.
4. **Memory and State Sharing:** Use the framework or a custom approach to maintain context between agents. For example, the Web Agent might find an important insight ("Company X is facing a lawsuit"), and the Summarizer should mention it alongside the Data Agent's numbers. This means the Supervisor should collect outputs from both and pass them into the Summarizer's input context. Implement a simple state object or use the framework's memory: after each sub-agent finishes, store its result in a dictionary (e.g. `state['data_result'] = ...`). When calling the Summarizer agent, include in its prompt both the original question and an aggregated summary of those results (or even the raw results). This way the summarization step has the full picture. Many frameworks (like LangGraph or CrewAI mentioned by Relari) provide abstractions for sharing memory/state – leverage those to avoid reinventing the wheel [61] [62].
5. **Human Override (Optional):** In some financial advisory contexts, you might want a human advisor to review before finalizing. Relari's comparison notes human-in-the-loop as a consideration [55]. Design your system such that after the Summarizer agent drafts the response, there is an opportunity for review. This could be as simple as routing the draft to a UI for approval, or as complex as having a "Moderator Agent" that simulates a human check using a different model. Depending on your use case (say, internal tool for advisors vs. fully automated client chatbot), you can include this step to catch any issues the agents might have missed (especially important if the stakes are high or compliance requires human sign-off).
6. **Framework Selection:** Relari's blog evaluated different frameworks for such an agentic system, and found trade-offs in complexity vs. flexibility [63] [64]. If you implement from scratch, you'll handle the orchestration logic yourself. Alternatively, consider using an **agent framework** to simplify, such as:
– *LangChain/LangGraph:* which allows you to define agents and tools and handles calling logic (LangGraph uses a graph execution model, which can explicitly model the flow between Supervisor and sub-agents [65] [66]).
– *Microsoft AutoGen or DSPy:* which are designed for multi-agent dialogues and might help structure the conversation between your agents.
– *OpenAI Functions/Swarm:* where you define each sub-agent function and let the LLM choose to call them (OpenAI's Swarm approach was to embed the routine in the prompt rather than code [67]).
Evaluate which fits your needs and your team's expertise. The framework can handle low-level details (like formatting, looping until task done, error catching), letting you focus on finance-specific logic. Whichever route, modularize your code so you can swap frameworks if needed, and gradually enhance the orchestration (the blog suggests initially all frameworks produced similar quality outputs, but differences emerge in how easily one can debug or extend agents – so design with clarity).

By implementing a supervised multi-agent setup with clear-cut agent roles and using available frameworks, you'll replicate the Relari assistant's strengths. Your system will be able to **gather real-time data and deep web info**, then synthesize it into coherent financial advice. This orchestration boosts reasoning (complex queries get broken down and answered piecewise) and yields personalized, well-rounded recommendations – because each sub-agent contributes its expertise (numbers, context, explanation), the final answer is both data-driven and narratively rich, as proven by the Agentic Finance Assistant's results.

Local LLM Financial Advisor (Sebastian Logsdon, 2023) – Meta-Agent Orchestration with Real-Time Data

Summary: An industry practitioner's blog by S. Logsdon (2023) details building a *Financial Advisor Agent* using local LLMs with an **agentic orchestration** approach. In this design, a lightweight local LLM (like a Llama 2 model) serves as the core reasoning engine, while specialized helper agents fetch information from various sources, coordinated by a central **Meta-Agent** [68]. Concretely, the system has a **Finance Data**

Agent (pulls real-time market data via an API, e.g. Alpha Vantage for stock prices and forex rates), a **Web Scraper Agent** (scrapes latest financial news or Google search results to gauge market sentiment), and a **PDF Knowledge Base Agent** (embeds and retrieves company-specific documents or emails from an internal knowledge base for insights) [69] [70] . These agents run asynchronously, feeding their findings into a shared "work pad." The Meta-Agent (the orchestrator) then synthesizes all inputs into the final response, ensuring that the answer cites up-to-the-minute data, relevant news context, and any pertinent internal info [71] [72] . This orchestrated approach markedly improves the **personalization and relevance** of financial advice: the agent can tailor responses to the user's context (by leveraging internal company data via the PDF agent) and reason with current market conditions (via live data and news). The blog also highlights *strengths* (privacy, customizability of local models) and *limitations* (smaller LLMs may need augmentation for complex reasoning) of using a local agentic setup [73] [74] . The result is a cost-effective, secure AI advisor that autonomously handles data gathering and analysis, providing businesses with actionable insights that are both timely and context-aware.

Implementation Guidance: To create a similar agentic financial advisor with local models:
1. **Set Up Data Agents:** Implement modular agents for each data source: - **Market Data Agent:** Use financial APIs (e.g. Alpha Vantage, Yahoo Finance) to fetch real-time quotes, indices, or economic indicators [69] . This agent should accept queries like "get me current price of X" or "fetch last quarter's GDP" and return structured data. - **Web News Agent:** Use a search API or web scraping tool to gather recent news headlines or articles related to the query (for example, if asked about a company's outlook, fetch recent news on that company) [70] . This agent might perform a search and then scrape snippets from top results. Ensure you include some parsing to avoid irrelevant content. - **Document QA Agent:** Create an internal knowledge retriever. Utilize a library like HuggingFace Transformers to embed text from internal PDFs or knowledge base, store embeddings in a vector database (FAISS as mentioned) [75] . This agent, given a user query or a specific ticker/company, retrieves relevant passages (e.g. "the latest annual report says…") that the LLM can use. This adds personalization if those docs are user or organization-specific. Each agent runs independently; test them by inputting sample requests and verifying outputs (e.g., does Market agent correctly parse API JSON? Does the PDF agent return meaningful sentences?).
2. **Meta-Agent Orchestration:** Develop a Meta-Agent that serves as the conductor. This is essentially your LLM orchestrator – since you're using a local LLM, you might incorporate the orchestration logic in your application code or within the LLM prompt itself. A straightforward approach: have your application asynchronously call all relevant data agents for a given query, then feed the collected results into the LLM's prompt. The blog's approach had the Meta-Agent gather outputs into a "work-pad" which is then included in the final prompt to the LLM [68] . Concretely, you could structure a prompt like: "You are a financial advisor. The user asks: {question}. Here is data: [Market data: …; News: …; Internal info: …]. Using this, answer the question with detailed reasoning." The LLM then produces a synthesized answer. The meta-agent code should also handle errors (if an agent fails, maybe note "market data unavailable" so the LLM can still respond gracefully).
3. **Asynchronous Execution:** To improve responsiveness, run agent calls in parallel. In Python, for example, use `asyncio` or multi-threading to have the Market Data Agent, Web Agent, and PDF Agent work concurrently [76] [77] . This way, the slowest source (perhaps the web scrape) doesn't hold up faster ones; the Meta-Agent assembles results as they come in. The blog specifically used `asyncio` for agent collaboration and noted the importance of this for efficiency [77] . Implement safeguards: e.g., a timeout for each agent so one stuck source doesn't block the whole answer.
4. **Tool and Model Integration:** Keep the local LLM lightweight for cost but augment it with tools (which you are doing through agents). If using a smaller LLM (say 7B or 13B parameters), recognize its limits in complex reasoning. The blog suggests mitigation strategies like offloading heavy reasoning to bigger

models via APIs for certain queries or using retrieval augmentation to compensate for knowledge gaps [74] [78]. You could, for instance, have a fallback where if the local LLM is not confident or the question is very complex ("explain the macroeconomic impact on our portfolio"), your system routes that to a larger cloud LLM. However, if privacy is paramount, another strategy is to fine-tune your local model on domain data or allow it to iterate with self-reflection (though that's advanced). Start with the retrieval approach as it's simpler: ensure the LLM prompt always has the necessary facts from your agents so it doesn't need to "know" them internally.

5. **Consolidation & Post-Processing:** Once the LLM produces an answer, you might do some post-processing. For example, format the answer, or enforce that certain data points from the agents are included. If the LLM overlooked something important that an agent returned (say a critical news headline), you could detect that and append a follow-up prompt to have it consider it. A simpler method is to design the prompt to explicitly instruct the LLM to use *all* provided information ("Make sure to reference relevant data from the input"). Additionally, log which sources were used for transparency – you can even have the LLM cite the sources by name if that's useful to the user ("According to internal report X, …"). This increases user trust as they see the answer is grounded in real data, not just the model's opinion.

6. **Testing and Iteration:** Test the full pipeline with various scenarios (market up, market down, incomplete data, etc.). Evaluate the quality of reasoning and personalization. For instance, if you input "Should we be worried about Company Y's stock?", check that the answer includes both current stock performance (from Market agent) and recent news (from Web agent), as well as any internal exposure your company has to Company Y (from PDF agent). If something is missing, adjust agent prompts or the meta-agent logic. The blog's conclusion was that such an agent *"automates complex tasks, integrates real-time data, and unlocks internal information"* effectively [79]. Use that as a checklist – your solution should demonstrate improved reasoning (complex tasks handled via multiple steps), personalization (answers specific to user's context), and recommendation quality (grounded in latest data and comprehensive analysis). Continue to refine by expanding the knowledge base, upgrading the local model if needed, and possibly adding new sub-agents (the blog hints at future additions like a Knowledge Graph agent or even a Vision agent for charts [80] [81]). This modular growth will further enhance your advisor's capabilities.

By orchestrating local tools and data sources around an LLM, you achieve a powerful synergy: the system remains private and customizable, yet it's **augmented with live data and specialized knowledge**, leading to more informed and personalized financial advice than an LLM alone could provide. As shown, even a smaller local LLM can punch above its weight when backed by an effective agentic framework that supplies it with the right information and context [82] [83].

Sources: Recent academic papers and industry reports were used to compile the above findings and recommendations. Key references include Seabra et al. (2024) on multi-agent orchestration for multi-source QA, Yang et al. (2024) "FinRobot" open-source framework [2] [3], Zhang et al. (2024) "FinAgent" multimodal trading agent [6] [11], Wawer & Chudziak (2025) "ElliottAgents" stock forecasting with LLM agents [20] [22], Broadridge's BondGPT patent press release (2025) [36] [35], an AWS blog on Bedrock multi-agent financial assistants (2025) [44] [45], Relari's comparison of agent frameworks with a finance assistant example (2024) [54], and a Medium tutorial on building a local-LLM-powered financial advisor agent (2023) [68] [83]. These sources demonstrate the state-of-the-art techniques and practical steps for enhancing LLM-based financial recommendation systems through agentic orchestration, leading to superior reasoning, personalization, and recommendation quality.

---

1 2 3 4 5 GitHub - AI4Finance-Foundation/FinRobot: FinRobot: An Open-Source AI Agent Platform for Financial Analysis using LLMs
https://github.com/AI4Finance-Foundation/FinRobot

6 9 10 11 [2402.18485] A Multimodal Foundation Agent for Financial Trading: Tool-Augmented, Diversified, and Generalist
https://arxiv.org/abs/2402.18485

7 8 12 13 14 [2402.18485] A Multimodal Foundation Agent for Financial Trading: Tool-Augmented, Diversified, and Generalist
https://ar5iv.org/html/2402.18485v3

15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 Integrating Traditional Technical Analysis with AI: A Multi-Agent LLM-Based Approach to Stock Market Forecasting
https://arxiv.org/html/2506.16813v1

31 32 33 34 35 36 37 38 Broadridge Announces New Patent on Large Language Model Orchestration of Machine Learning Agents
https://www.prnewswire.com/news-releases/broadridge-announces-new-patent-on-large-language-model-orchestration-of-machine-learning-agents-302454711.html

39 40 41 42 43 44 45 46 47 48 49 50 51 52 Build a gen AI–powered financial assistant with Amazon Bedrock multi-agent collaboration | Artificial Intelligence
https://aws.amazon.com/blogs/machine-learning/build-a-gen-ai-powered-financial-assistant-with-amazon-bedrock-multi-agent-collaboration/

53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 Choosing the Right AI Agent Framework: LangGraph vs CrewAI vs OpenAI Swarm | Relari | Relari Blog
https://blog.relari.ai/choosing-the-right-ai-agent-framework-langgraph-vs-crewai-vs-openai-swarm-56f7931b4249

68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 Unleashing the Power of Local LLMs: Building a Financial Advisor Agent | by Sebastian Logsdon | Medium
https://medium.com/@seblogsdon/unleashing-the-power-of-local-llms-building-a-financial-advisor-agent-a26294ea209b