AAI-520 Final Project: Multi-Agent Financial Analysis System Report

Group 5: Ajmal Jalal, Victor Hugo Germano, Michael De Leon, Darin Verduzco, and Antonio Recalde

University of San Diego  
AAI-520: Natural Language Processing and GenAI  
October 19, 2025

# Introduction

The Financial Agentic System developed for AAI-520 demonstrates a comprehensive application of autonomous agentic design principles in natural language processing and generative artificial intelligence. The system integrates six major modules, Memory Analyzer, Data Ingestion, Summarizer, Memory Management, Evaluator-Optimizer, and Orchestrator, to automate the process of financial data collection, analysis, summarization, and iterative refinement. Each component contributes to a structured pipeline that reflects the agentic AI paradigm, where modules plan, act, and reflect independently to produce higher-quality, evidence-based investment insights (Human Security, 2024). Through the combination of concurrent data ingestion, contextual summarization, persistent memory, and recursive evaluation loops, the project illustrates how modular design and feedback-driven optimization can enhance both reliability and interpretability in financial analytics systems.  
  
 Architecturally, the system implements a multi-agent collaboration pattern inspired by LangChain’s StateGraph and Evaluator-Optimizer models, utilizing OpenAI’s ChatOpenAI interface for structured LLM calls. Each module inherits from a base class, BaseWorker, ensuring consistent interface behavior through the execute() method. This approach exemplifies strong software engineering practices such as separation of concerns, dependency isolation, and reusable state management (Dey et al., 2025). Collectively, these design choices promote both transparency and adaptability, core goals in explainable and human-aligned AI research. The following sections provide a detailed technical analysis of each subsystem, including direct code references, theoretical grounding, and critical assessment of functionality.

# Memory Analyzer

The *MemoryAnalyzer* module (memory\_analyzer.py) is the system’s first logical checkpoint, assessing whether previously stored data in memory is adequate and current for addressing a user’s query. The execute() function receives two parameters, memory\_snapshot and query, and evaluates sufficiency through both content and temporal analysis. Within the method, the \_extract\_timestamp() function applies regular expressions such as r'Timestamp:\s\*(\d{4}-\d{2}-\d{2}[T\s]\d{2}:\d{2}:\d{2})' to locate time markers in stored memory strings. This temporal extraction enables comparison between the recorded timestamp and the current datetime, determining data freshness within a 24-hour threshold. If data exceeds the threshold window, the system conservatively marks it as stale (is\_data\_fresh = False) to ensure reliability when generating subsequent analyses.

A critical feature of this component lies in its use of structured large language model (LLM) reasoning via LangChain’s with\_structured\_output() method, which constrains the model to return a defined schema (SufficiencyAssessment) consisting of is\_sufficient, requires\_fresh\_data, and missing\_information. This structured design reduces ambiguity and improves explainability, enabling the model to self-assess memory adequacy using textual evidence and metadata (Human Security, 2024). When the model fails, the code includes a fallback mechanism that defaults to conservative assumptions, explicitly printing "LLM analysis failed" and returning a dictionary that forces a new data retrieval. Such defensive programming ensures operational continuity under partial LLM unavailability.

The module exemplifies intelligent decision-making consistent with agentic AI research, where systems autonomously evaluate their situational context before deciding whether to act or recall past data (OpenAI Research, 2025). By combining symbolic logic (timestamp validation) with probabilistic reasoning (LLM evaluation), the *MemoryAnalyzer* integrates two reasoning modalities, a hallmark of hybrid intelligence architectures. Its conservative defaults ensure that any uncertainty leads to data refreshment, thereby reducing the risk of relying on outdated financial information. This approach is particularly crucial in domains like finance, where temporal precision and data validity directly affect investment decisions.

Finally, the design demonstrates principles of *explainable autonomy*. Each return dictionary provides explicit rationale for its decision, listing missing information strings such as “Data is too old (48.3 hours > 24 hour threshold).” This transparency supports traceability and human verification, aligning with responsible AI standards (Dey et al., 2025). Overall, the *MemoryAnalyzer* module not only enhances reliability but also operationalizes the philosophical foundation of agentic reasoning: autonomous self-evaluation guided by verifiable logic.

# Data Ingestion

The *Ingestion* subsystem encompasses three interdependent classes: FinancialDataIngestion, NewsDataIngestion, and Ingestion. Each implements execute() methods that collectively form a concurrent data retrieval system. In FinancialDataIngestion.execute(), the Yahoo Finance API (yfinance.Ticker) is called to pull historical OHLCV data and company fundamentals. The code extracts quantitative metrics such as current\_price, avg\_volume\_30d, and volatility\_30d using pandas operations on the returned dataframe. Meanwhile, NewsDataIngestion.execute() queries both Yahoo RSS feeds and the NewsAPI REST endpoint, parsing XML and JSON responses into structured dictionaries of article metadata. Together, these workers illustrate a multi-source aggregation design optimized for heterogeneity and fault tolerance.

The orchestration occurs within Ingestion.execute(), which uses Python’s ThreadPoolExecutor to run the financial and news ingestion in parallel. By submitting futures (executor.submit(...)) and collecting results with as\_completed(future\_to\_source), the code implements a classic *fan-out/fan-in* concurrency pattern (Dey et al., 2025). This parallelization reduces latency by overlapping network I/O operations, a significant advantage when accessing APIs with variable response times. The method \_combine\_results() merges outputs into a unified bundle containing financial\_data, news\_data, errors, and an overall status derived from the \_determine\_status() function, which classifies results as "success", "partial\_success", or "error". This explicit error propagation enables the orchestrator to make informed decisions about fallback or recovery.

The data ingestion process exemplifies modern software design principles such as modularity, fault isolation, and graceful degradation. Each worker returns a self-contained JSON-like dictionary, minimizing dependencies between threads. The inclusion of exception handling blocks within try/except clauses ensures that even partial failures, such as a missing NewsAPI key, result in recoverable "partial\_success" outputs rather than total system interruption. This approach supports resilience and aligns with cloud-native architectural practices, where distributed reliability outweighs single-point performance (SCET Berkeley, 2024). The explicit timestamping (datetime.now().isoformat()) embedded in each output further enhances auditability and synchrony across agents.

Ultimately, the *Ingestion* subsystem provides the informational foundation upon which subsequent modules operate. By standardizing diverse inputs into a uniform schema, it ensures downstream compatibility with both the *Summarizer* and *Evaluator-Optimizer*. The system’s multi-source strategy and concurrent execution framework exemplify the integration of software engineering discipline into AI workflow design, a necessary evolution for production-grade agentic systems operating in volatile data environments (OpenAI Research, 2025).

# Summarizer

The *SummarizerWorker* (src/summarizer/summarizer.py) transforms structured data into coherent analytical narratives. It routes financial and textual content through the \_route\_headlines() and \_format\_context() functions before constructing a PROMPT\_TEMPLATE containing the system’s analysis goal. The routing dictionary (ROUTES) defines three major analytical paths (earnings, macro, and company). Each mapped to a list of domain-specific keywords (e.g., "eps", "revenue", "inflation", "merger"). The \_route() helper iterates through these keys, assigning each headline to a thematic category. This symbolic preprocessing step allows the model to produce semantically clustered insights and prevents topic drift in the generated summaries (Human Security, 2024).

The \_format\_context() function merges quantitative sentiment features from news\_daily dataframes with qualitative headline text from raw\_news. The logic iterates through recent sentiment metrics (for idx, row in tail.iloc[::-1].iterrows()) and appends lines such as "- 2025-10-17: count=15, sent\_mean=+0.233, decay=-0.045", producing a hybrid narrative structure that blends statistical and textual evidence. By preserving recency order, the summarizer mirrors temporal reasoning in financial analysis, prioritizing the most recent events in the contextual prompt. The final prompt, rendered by the line PROMPT\_TEMPLATE.format(goal=goal, symbol=symbol, context=context), guides the LLM to generate specific, evidence-based bullet points.

A notable design choice in this module is the deliberate exclusion of confidence metrics. The docstring explicitly states, “NOTE: Confidence metric is fully removed (no ‘confidence’ in output).” This aligns with the team’s architectural revision, transferring quality control to the *Evaluator-Optimizer* rather than embedding it in the summarization stage. Such modular delegation enhances maintainability and prevents metric redundancy. The summary\_text variable, currently stubbed as a placeholder, includes a comment indicating future replacement by an actual LLM call, exemplifying forward compatibility with generative APIs such as GPT-4o (OpenAI Research, 2025).

Overall, the *SummarizerWorker* operationalizes *prompt chaining* (SCET Berkeley, 2024), a process by which structured inputs are successively refined to generate domain-specific narratives. Its use of dynamic context assembly and keyword routing demonstrates the integration of symbolic filtering with data-driven synthesis. This dual methodology aligns with best practices in AI explainability, as every piece of generated text is traceable to both data sources and logical conditions in the code. Functionally, it bridges the gap between raw information retrieval and evaluative reasoning, serving as the system’s cognitive intermediary.

# Memory Management

The *MemoryWorker* (src/memory/memory.py) implements persistent storage for agent experiences using a JSON file (./data/agent\_memory.json). It maintains an in-memory list \_memory\_store synchronized with disk through the \_load\_memory() and \_save\_memory() functions. Upon import, \_load\_memory() attempts to read existing entries, catching both JSONDecodeError and FileNotFoundError exceptions to ensure safe initialization. The design exemplifies lightweight persistence, allowing the agent to accumulate analytical outcomes across sessions without requiring external databases (Human Security, 2024). Each record includes a UTC timestamp, text content, and an array of tags, providing both temporal and semantic traceability.

The execute() method defines four operations, add, search, get\_recent, and retrieve\_by\_symbol. When performing add, the function constructs a dictionary entry with "timestamp": datetime.now(timezone.utc).isoformat(), appends it to \_memory\_store, and writes the updated list to disk via \_save\_memory(). The search and retrieve\_by\_symbol operations implement simple yet effective keyword filtering, using Python list comprehensions such as [m for m in reversed(\_memory\_store) if query in m['text'].lower()]. This approach leverages reverse chronological ordering to prioritize recency, a key factor in time-sensitive analyses like stock research. The get\_recent operation directly slices the list, returning the latest n entries for quick access.

From a conceptual standpoint, this module models *episodic memory*, a core feature of cognitive agents that retain structured experiences for reuse in future reasoning. By using JSON serialization, the memory system remains human-readable and easily transferable across environments, promoting transparency and reproducibility (Dey et al., 2025). Its simplicity belies its importance: the orchestrator and analyzer depend on this data for sufficiency checks and context reconstruction. Unlike more complex vector databases, this implementation emphasizes interpretability and portability over algorithmic sophistication.

Finally, the *MemoryWorker* plays a critical role in enabling *lifelong learning* across analytical sessions. Each saved summary or routed note contributes to the agent’s evolving understanding of financial trends. Its deterministic behavior and lightweight footprint make it ideal for iterative prototyping and reproducible experimentation. By aligning persistent memory with symbolic tagging, the module provides a balance between structured recall and adaptable growth, principles central to developing transparent and trustworthy AI systems (OpenAI Research, 2025).

# Evaluator-Optimizer

In designing the *EvaluatorOptimizer*, the development team consulted the workflow pattern guidelines from the LangChain documentation to establish a modular, agent-based structure for iterative reasoning and optimization (LangChain, 2025). These workflow patterns provided architectural insight into managing data flow between nodes, enabling smooth execution of tasks in a stateful environment. By leveraging these guidelines, the system incorporated nodes that could interact dynamically through well-defined states, supporting recursive feedback and adaptive control flow.

The incorporation of workflow patterns from LangChain’s documentation was particularly instrumental in structuring the *generator* and *evaluator* nodes. Each node functioned as an independent yet cooperative component within the LangGraph framework, ensuring modularity and extensibility. This modular architecture enabled the EvaluatorOptimizer to maintain separation of concerns, where generation, evaluation, and optimization could evolve independently, while preserving a unified state graph for coordinated processing. The reference framework provided by LangChain streamlined the graph-building process, ensuring clarity and alignment with contemporary best practices in agent-based workflow orchestration (LangChain, 2025).

A crucial aspect of designing these agents involved the construction of highly effective prompts. Prompt engineering emerged as the determining factor for achieving quality outputs that aligned with the project’s system requirements. The team emphasized a balance between generic and specialized design principles. Ensuring that the agents remained adaptable to a wide variety of query types while maintaining a consistent structure in their responses. This approach minimized prompt drift and enhanced the reliability of subsequent evaluations, allowing for reproducible and intelligible summaries that could be meaningfully interpreted by downstream agents.

Furthermore, the emphasis on maintaining structural uniformity across agent outputs ensured that the generated results could seamlessly serve as inputs for memory agents. This interoperability facilitated knowledge continuity, reducing the need for redundant data retrieval. When applicable, the agents could generate comprehensive answers to future queries based solely on previously stored context. The deliberate design of prompt templates, grounded in the quality criteria defined in the EvaluatorOptimizer, contributed to the development of a resilient system capable of learning and improving autonomously (LangChain, 2025).

The success of this framework was evidenced in the model’s ability to generate intelligible outputs suitable for continuous optimization without external supervision. Each iteration not only refined the textual coherence of summaries but also contributed to a self-reinforcing feedback cycle that elevated the overall reasoning quality. Through the careful integration of evaluation criteria, completeness, coherence, and evidence-based reasoning, the EvaluatorOptimizer demonstrated agentic behavior resembling that of self-correcting cognitive systems.

Looking ahead, the system’s optimization potential could be further expanded by diversifying the models used in the evaluation and optimization processes. While the current proof of concept relied on a single OpenAI model for simplicity, adopting models from distinct families (e.g., Anthropic, Mistral, or Cohere) could provide complementary perspectives during evaluation, increasing robustness and reducing model-specific biases. Such hybrid model configurations could enhance generalization across domains while maintaining the transparency and interpretability demanded by modern AI governance standards.

In conclusion, the *EvaluatorOptimizer* embodies a practical implementation of recursive feedback and model self-improvement through workflow-driven design. Its integration of LangChain’s workflow patterns, coupled with deliberate prompt engineering and modular agent construction, establishes a foundation for scalable and adaptive AI evaluation. The system’s ability to translate qualitative feedback into structured, quantitative performance metrics underscores its role as both a technical and methodological milestone in agentic optimization.

# Orchestrator

The *Orchestrator* module (src/orchestrator.py) integrates all functional components into a cohesive pipeline that autonomously performs financial research and decision support. Its execute() function embodies the agentic lifecycle: retrieving past memories, assessing memory sufficiency, fetching new data when no previously optimized summary is sufficient in memory, generating summaries, optionally running summary optimization, and storing optimized results in memory. The module is accessed with a convenience function run\_investment\_analysis(symbol, instructions), which initializes the Orchestrator and calls execute() to return the optimized financial summary in Markdown format based on the inserted stock symbol and user query input as instructions.

Early in execution, it retrieves prior records with self.workers["memory"].execute("retrieve\_by\_symbol", symbol) and composes them into a formatted text snapshot using \_format\_memory\_snapshot(). This snapshot is then analyzed for sufficiency by MemoryAnalyzer.execute(), which considers both content relevance (is\_sufficient) and data freshness (requires\_fresh\_data) before deciding whether a cached optimized summary can be reused. If an optimized summary exists and satisfies the current query, it is reused in the final output, skipping the evaluator-optimizer phase. These decision nodes operationalize *conditional planning*, a hallmark of autonomous systems design (Dey et al., 2025).

When memory is insufficient or no optimized summary exists, the orchestrator invokes the Ingestion and Summarizer workers sequentially, passing symbol-specific inputs into summarizer.execute({ "symbol": symbol, "raw\_news": news\_articles, ... }). This new initial summary, along with its routed notes, is stored in memory via MemoryWorker.execute("add", note, [symbol, "summary"]), ensuring persistence before evaluation and optimization.

Afterward, if no sufficient optimized summary was found in memory, the newly generated summary proceeds to the *Evaluator-Optimizer* phase. The orchestrator creates a structured state dictionary and calls evaluator.workflow.invoke(initial\_state). This optimization loop only runs when the summary originates from the current execution rather than from cached optimized summaries. Final optimized summaries are written back into persistent storage with descriptive metadata, including timestamps and tags (summary, optimized), enabling future reuse and sufficiency checks from memory.

The orchestrator’s design exemplifies *modular integration* and *functional composition*. Each worker acts as a microservice within a local ecosystem, communicating through standardized data dictionaries. This approach maintains both scalability and fault tolerance; for instance, failures in one worker do not cascade into others due to isolated exception handling. The orchestration logic is both linear and conditional, following a deterministic route yet capable of dynamic branching based on sufficiency outcomes. Such a structure parallels cognitive control architectures in multi-agent systems, where a central executive coordinates specialized submodules to achieve complex goals (Human Security, 2024).

Ultimately, the *Orchestrator* represents the system’s intelligence backbone, where planning, action, and reflection converge. By integrating diverse modules through a cohesive control loop, it realizes the practical embodiment of agentic AI principles. The orchestrator’s design is not merely procedural; it demonstrates reasoning about process flow, dynamically adapting its strategy based on environmental inputs and system feedback - modeling the essence of self-regulated AI systems envisioned in contemporary research on autonomous decision pipelines (OpenAI Research, 2025).

# References

Dey, A., Verduzco, D., Recalde, A., & Germano, V. H. (2025). Retrieval-Augmented Generation and Agentic Workflow Design. University of San Diego.

LangChain. (2025). *Workflow agents and design patterns in LangGraph (Python)*. LangChain

Documentation. <https://docs.langchain.com/oss/python/langgraph/workflows-agents>

Human Security. (2024). Agentic AI Systems: Reasoning, Autonomy, and Decision Loops.

OpenAI Research. (2025). Structured Feedback Loops in Generative Agents.

SCET Berkeley. (2024). Prompt Chaining and Multi-Agent Collaboration in GenAI Systems.