# Architecting the Autonomous Market Research Analyst: A Comprehensive Technical Blueprint for Agentic RAG in Financial Systems

## 1. Executive Architecture Overview

The intersection of quantitative finance and generative artificial intelligence has precipitated a paradigm shift in how market research is conducted. We are transitioning from an era of passive information retrieval—where analysts manually query disparate databases—to an era of autonomous agentic systems capable of orchestrating complex, multi-step research campaigns. This report details the architectural design and technical implementation of a "Market Research Analyst Agent," a sophisticated Agentic RAG (Retrieval-Augmented Generation) system designed to autonomously research a target company, synthesize financial data from secured vector stores, integrate real-time market intelligence from the web, and author a professional-grade investment memo.

The system is architected upon a modern stack comprising **LangGraph** for stateful orchestration, **Tavily** for optimized web intelligence, and **LangChain** for cognitive tooling, with a specialized focus on the ingestion of complex financial documents via **LlamaParse**. Unlike traditional linear RAG pipelines, which suffer from fragility and hallucination when dealing with multi-modal financial data, this agent utilizes a cyclic graph topology. This allows for iterative reasoning, self-correction, and the dynamic routing of tasks between specialized sub-agents—specifically a "Web Researcher," a "Financial Quantitative Analyst," and a "Memo Synthesizer"—overseen by a "Supervisor" node.

The following analysis provides an exhaustive examination of the design decisions, trade-offs, and implementation strategies required to build this system. It addresses the unique challenges of parsing unstructured financial tables, ensuring deterministic control flow in non-deterministic LLM environments, and structuring the output to meet the rigorous standards of institutional investment committees.

## 2. The Evolution of Financial AI: From Passive Retrieval to Agentic Action

To understand the necessity of the proposed architecture, one must first appreciate the limitations of the incumbent "Naive RAG" approach in the financial domain. Financial analysis is inherently an iterative, hypothesis-driven process, not a simple question-answering task.

### 2.1 The Limitations of Linear RAG in Finance

In a traditional linear RAG system, the workflow is unidirectional: Query -> Embed -> Retrieve -> Generate. While effective for simple queries (e.g., "What is Apple's mission statement?"), this architecture collapses under the weight of complex financial questions (e.g., "Analyze the impact of rising interest rates on Tesla's debt servicing costs relative to its automotive gross margin decline in Q3 2024").

The linear model fails for three primary reasons:

1. **Lack of Planning:** It cannot decompose a complex query into constituent sub-tasks (e.g., first finding the debt levels, then finding the interest rates, then calculating the cost).
2. **No Error Recovery:** If the retrieval step fetches irrelevant documents (a common occurrence with dense 10-K filings), the generation step has no mechanism to reject the context and retry the search with modified parameters.1
3. **Static Context:** It treats all data as equal, lacking the ability to distinguish between historical data in a vector store and real-time data on the web.

### 2.2 The Agentic Paradigm: Cycles and State

The "Action-Oriented" Agentic RAG system proposed here fundamentally differs by introducing **cycles** and **state**. The agent is not a pipeline but a loop. It maintains a persistent memory state that evolves as it gathers information. It possesses the cognitive architecture to "reason" about the data it finds.

If the agent retrieves a balance sheet but cannot locate the "Long-Term Debt" figure due to parsing errors, an agentic system detects this gap. It then formulates a new plan—perhaps to search for "Total Non-Current Liabilities" instead or to consult a secondary data source like a web API. This capability—**self-correction**—is the defining characteristic of a robust Market Research Analyst Agent.2

This system creates a "Collective Intelligence Model" where specialized nodes (agents) collaborate. A Supervisor agent delegates tasks to a Researcher (who browses the web) and an Analyst (who queries the database), aggregating their outputs into a cohesive whole.3

## 3. Orchestration Core: The Strategic Choice of LangGraph

The heart of any agentic system is its orchestration layer—the framework that manages the flow of control, state persistence, and error handling. For this project, the choice lies between **LangGraph** and **CrewAI**. While both are capable frameworks, a rigorous analysis of the requirements for financial accuracy and architectural control dictates the selection of **LangGraph**.

### 3.1 Comparative Analysis: LangGraph vs. CrewAI

The decision to utilize LangGraph is driven by the need for low-level control over the agent's cognitive loops. Financial research requires precise, deterministic workflows interspersed with LLM reasoning.

| **Feature** | **LangGraph** | **CrewAI** | **Implications for Market Research Agent** |
| --- | --- | --- | --- |
| **Control Topology** | **Graph-Based (Cyclic):** Nodes and edges define a state machine. Allows for complex looping, branching, and conditional routing based on data quality.3 | **Role-Based (Hierarchical):** High-level abstraction of "Agents" and "Tasks." Focuses on sequential or hierarchical process execution.3 | LangGraph is superior for implementing "Self-Correction" loops (e.g., "If retrieval score < 0.7, rewrite query"). CrewAI's abstraction makes such low-level logic harder to implement.3 |
| **State Management** | **Explicit State Schema:** Uses a shared State object (TypedDict/Pydantic) that persists across steps. Allows granular control over what data is passed between nodes.5 | **Implicit Context:** Context is shared automatically between agents, but managing specific data structures (like extracted financial tables) is less transparent. | Financial analysis requires carrying specific structured data (e.g., a Pandas DataFrame of revenue) through the workflow. LangGraph's explicit state is essential for this.6 |
| **Flexibility** | **High:** "Low-level control over agent workflows." Can integrate custom RAG architectures and precise retrieval control.3 | **Medium:** "Abstracts away complexity." Excellent for rapid prototyping but less suited for custom, highly engineered flows.7 | The requirement for "Action-Oriented" RAG implies custom behavior (e.g., parsing a specific 10-K section). LangGraph offers the necessary granularity.7 |
| **Error Handling** | **Robust:** Supports "human-in-the-loop" checkpoints, retries, and fallback logic at the node level.7 | **Agent-Driven:** Relies on the agent to "figure it out," which can lead to loops or hallucinations without strict guardrails. | In finance, we cannot afford infinite loops or unverified data. LangGraph's control flow allows for strict termination conditions.8 |

### 3.2 LangGraph Architecture for Finance

LangGraph models the application as a **StateGraph**. The "State" is the single source of truth, updated by each node (agent) in the graph.

#### 3.2.1 The State Schema

For a Market Research Analyst, the state must be rigorous. It is not merely a list of messages but a structured repository of the research campaign.

Python

from typing import TypedDict, List, Dict, Any  
from langchain\_core.messages import BaseMessage  
  
class AnalystState(TypedDict):  
 company\_name: str  
 user\_query: str  
 # The 'Memory' of the investigation  
 financial\_filings: List] # Metadata of retrieved 10-Ks  
 extracted\_metrics: Dict[str, float] # e.g., {'revenue\_2023': 100000}  
 web\_research\_summary: str # Synthesized news  
 draft\_sections: Dict[str, str] # {'executive\_summary': '...', 'risks': '...'}  
 # Control Flow State  
 iteration\_count: int # To prevent infinite loops  
 quality\_check\_pass: bool # Flag from the Reviewer node  
 messages: List # Chat history for LLM context

This schema ensures that when the "Financial Analyst" node finishes its job, it places the data into extracted\_metrics, which the "Writer" node can then access deterministically. This prevents the "Telephone Game" effect often seen in purely prompt-based agent chains.5

#### 3.2.2 The Supervisor-Worker Pattern

We employ a **Supervisor** architecture (also known as the Router pattern). The Supervisor is an LLM node that analyzes the user's request and routes execution to the appropriate sub-agents. It does not do the work; it manages the experts.6

* **Supervisor Node:** Receives "Analyze Google." Determines it needs both *Web Research* and *Financial Docs*. Routes to both.
* **Web Researcher Node:** Specialized in Tavily search.
* **Financial Analyst Node:** Specialized in Vector DB retrieval.
* **Writer Node:** specialized in synthesis.

This separation of concerns is critical. The prompt for the "Financial Analyst" can be hyper-specialized (e.g., "You are a forensic accountant...") without polluting the context of the "Web Researcher" (who needs a prompt like "You are a market journalist...").10

## 4. The Knowledge Foundation: Vector Database & Financial Data Pipelines

The "Market Research Analyst Agent" is only as intelligent as the data it can access. The requirement to "pull financial reports from a vector DB" introduces significant complexity. Financial documents (10-Ks, 10-Qs, Earnings Transcripts) are among the most difficult unstructured data sources to index effectively due to their density and reliance on tabular data.

### 4.1 The Tabular Data Conundrum

Standard RAG pipelines utilizing simple OCR (like pypdf or Tesseract) fail catastrophically on financial tables. A PDF balance sheet is a visual grid. When converted to a text stream by a standard parser, the spatial relationship between the header (e.g., "2023") and the value (e.g., "$14,000") is often destroyed, appearing as a jumbled sequence of numbers.

If the agent retrieves a chunk of text that looks like Revenue 100 120 140 Cost 80 90 100, the LLM cannot reliably determine which number belongs to which year. This leads to hallucination—the cardinal sin of financial analysis.11

### 4.2 Solution: LlamaParse for Visual-Semantic Parsing

To solve this, we integrate **LlamaParse**, a state-of-the-art document parsing solution from LlamaIndex. LlamaParse does not just read text; it uses a vision-language model to "see" the document structure. It identifies tables, understands row/column alignment, and converts them into **Markdown** representation.12

* **Mechanism:** LlamaParse renders the PDF page, identifies the table boundaries, and reconstructs the table into a pipe-delimited Markdown format.
* **Result:** The LLM sees | Year | Revenue | \n | 2023 | $140 |.
* **Benefit:** This preserves the semantic integrity of the data. The agent can now accurately perform mathematical reasoning on the rows and columns because the structure is explicit in the context window.14

### 4.3 Indexing Strategy: The Parent-Child (Multi-Vector) Retriever

Indexing financial documents requires a sophisticated strategy. Indexing the entire 10-K as a single chunk is impossible (context limits). Indexing it as small 500-token chunks loses the broader context (e.g., "Which subsidiary does this revenue table belong to?").

We employ the **Parent-Child** or **Multi-Vector Retriever** strategy.16

1. **Parent Document:** The full 10-K is split into large "Parent" chunks (e.g., entire sections like "Management's Discussion and Analysis").
2. **Child Documents (Summaries):** We use an LLM to generate concise summaries of these parent chunks. For tables, we generate a text summary: "Table 4: Consolidated Revenue breakdown by region for 2021-2023, showing 15% growth in APAC."
3. **Embedding & Storage:** We embed the *Child Summaries* and store them in the Vector DB (e.g., ChromaDB).
4. **Retrieval Logic:** When the agent asks "How did APAC perform?", the system matches the *Summary* embedding.
5. **Context Injection:** Crucially, the system does *not* return the summary to the agent. It uses the link to fetch the **Parent Document** (the raw Markdown table) and feeds *that* to the agent.

This ensures the retrieval is highly sensitive (semantic search on summaries) while the generation is highly accurate (reasoning on raw data).16

### 4.4 ChromaDB Implementation and Metadata Filtering

The vector database (ChromaDB) must be architected with a strict schema to support **Metadata Filtering**. Financial queries are time-bound. An agent asking for "2023 Revenue" must not retrieve a document from 2019.18

**Schema Strategy:**

* **Collection:** financial\_reports
* **Metadata:**
  + ticker: (e.g., "TSLA")
  + year: (e.g., 2023)
  + quarter: (e.g., "Q3")
  + doc\_type: (e.g., "10-K", "10-Q", "Earnings\_Transcript")
  + section: (e.g., "Risk\_Factors", "Financial\_Statements")

Self-Querying Retriever:

We utilize LangChain's Self-Querying Retriever. The agent does not just send a raw string query. It uses an LLM to translate the user's natural language into a structured query filter.

* *User Query:* "What were the risks mentioned in Tesla's 2023 10-K?"
* *Translated Filter:*  
  JSON  
  {  
   "query": "risks market competition",  
   "filter": {  
   "$and":  
   }  
  }

This precise filtering is non-negotiable for an "Analyst" agent. Without it, the RAG system is merely a keyword search engine; with it, it becomes a structured database query engine.19

## 5. The Sensory Layer: Integrating Tavily for Real-Time Intelligence

While the Vector DB provides historical truth, a Market Research Analyst must be aware of the *now*. Financial markets move on news, rumors, and macro-economic shifts that are not yet in the 10-K. The **Tavily API** serves as the agent's connection to live web data.

### 5.1 Tavily: The Search Engine for Agents

Tavily distinguishes itself from traditional search APIs (like Google or Bing) by optimizing for **RAG ingestion**. Standard search engines return HTML pages full of noise (ads, navigation bars, scripts). Tavily acts as a pre-processor, performing search, scraping, and content extraction in a single transaction.21

**Key Features for the Analyst Agent:**

* **search\_depth="advanced":** This parameter is critical. It triggers a deeper crawl, aggregating data from multiple sources to construct a comprehensive answer. For an investment memo, surface-level snippets are insufficient; the agent needs deep dives into articles.23
* **include\_raw\_content=True:** We ingest the raw text content. This allows the Agent's own LLM (likely GPT-4 or Claude 3.5) to perform the summarization and extraction, rather than relying on Tavily's internal summarizer. This preserves nuance—for example, the specific phrasing of a CEO's quote in a press release.21
* **Domain Filtering:** Using the topic="finance" parameter (if available) or appending domain restrictions (e.g., site:bloomberg.com OR site:reuters.com OR site:wsj.com) ensures the agent sources data from reputable financial journalism, ignoring blogs or forums which might induce hallucination or bias.

### 5.2 The Web Research Node

In the LangGraph topology, the **Web Researcher** node utilizes the Tavily tool dynamically.

* **Iterative Search Pattern:**
  1. **Initial Query:** "Tesla recent news" (Too broad).
  2. **Refinement:** The agent analyzes the initial results. "I see news about a recall. I need more details on the financial impact."
  3. **Secondary Query:** "Tesla recall Q3 2024 financial provision estimate."
  4. **Synthesis:** The agent combines these findings into a "Market Context" object in the state.

This iterative process—searching, reading, and searching again—mimics human research behavior and is enabled by the cyclic graph structure.24

## 6. Cognitive Architecture: Designing the Agentic Workflow

Having defined the orchestration (LangGraph), data (LlamaParse/Chroma), and web (Tavily) layers, we now assemble them into the cognitive architecture of the agent. This is the "Brain" of the system.

### 6.1 The Graph Topology

The system uses a **Hierarchical State Graph**.

**Nodes:**

1. **Supervisor:** The entry point. Parses the user input (e.g., "Research NVIDIA") and creates a research plan.
2. **Web\_Researcher:** Executes Tavily searches.
3. **Financial\_Analyst:** Queries the Vector DB.
4. **Memo\_Writer:** Synthesizes data into the final report.
5. **Quality\_Reviewer:** A "Critic" node that evaluates the draft.

**Edges (The Control Flow):**

* **Start -> Supervisor:** Initialize state.
* **Supervisor -> (Parallel) Web\_Researcher & Financial\_Analyst:** Decompose the task. The Supervisor determines that "Research" implies both looking at the past (DB) and the present (Web).6
* **Web\_Researcher -> Join:** Update state with market news.
* **Financial\_Analyst -> Join:** Update state with financial metrics.
* **Join -> Memo\_Writer:** All data is now in the State. The Writer generates the draft.
* **Memo\_Writer -> Quality\_Reviewer:** The draft is submitted for inspection.
* **Quality\_Reviewer -> Conditional Edge:**
  + *Condition A (Pass):* If the draft meets quality standards (citations present, risks addressed), transition to **END**.
  + *Condition B (Fail):* If the draft is lacking (e.g., "Missing P/E ratio"), transition back to **Supervisor** with feedback: "Retrieve P/E ratio for 2023."

This cycle ensures that the agent does not produce incomplete work. It forces the system to perform "Deep Research" by looping until the criteria are met.1

### 6.2 The Supervisor's Logic (The Router)

The Supervisor is implemented using an LLM with structured output (Function Calling). It outputs a RoutingDecision object.

Python

class RoutingDecision(BaseModel):  
 next\_steps: List[Literal["web\_researcher", "financial\_analyst", "memo\_writer"]]  
 reasoning: str

If the user asks "What is the current stock price?", the Supervisor sees no need for the Vector DB and routes *only* to the Web\_Researcher. If the user asks "Compare 2021 Revenue to 2023," it routes *only* to Financial\_Analyst. This efficiency prevents wasted API calls and reduces latency.25

## 7. The Analyst's Toolkit: Tool Engineering & Prompt Strategy

The specialized agents ("Workers") require specific tools and distinct personas.

### 7.1 The Financial Analyst Agent

* **Persona:** "You are a skeptical, data-driven forensic accountant. You trust only the data found in the provided documents. You cite every claim."
* **Tools:**
  + vector\_store\_retriever: Access to the ChromaDB.
  + python\_repl: A Python shell.
* **The Math Problem:** LLMs are notoriously bad at arithmetic. They often hallucinate calculations (e.g., "100 / 3 = 33").
* **The Solution:** We give the Financial Analyst a **Python REPL** tool. When it needs to calculate the "Debt-to-Equity Ratio," it does not calculate it in its "head" (the model weights). Instead, it writes a Python script:  
  Python  
  debt = 5000000  
  equity = 12000000  
  ratio = debt / equity  
  print(ratio)  
    
  The agent executes this tool, gets the precise result (0.4166...), and uses that in the report. This ensures 100% mathematical accuracy, a requirement for any credible financial agent.26

### 7.2 The Web Researcher Agent

* **Persona:** "You are a market investigator. You look for trends, rumors, and macro-factors. You prioritize reputable sources like Bloomberg, Reuters, and WSJ."
* **Tools:** tavily\_search.
* **Strategy:** This agent utilizes a "Tree of Thoughts" approach. It searches, summarizes the results, and then asks "Is this enough?" If not, it generates a new search query based on the gaps in the previous search.24

## 8. The Output Engine: Synthesizing the Investment Memo

The ultimate deliverable is the **Investment Memo**. The quality of this document determines the utility of the entire system. We structure this output based on the "gold standard" templates used by top Venture Capital firms like Sequoia and Andreessen Horowitz (a16z).27

### 8.1 The Memo Structure

The Memo\_Writer node is prompted to populate the following schema:

| **Section** | **Content Source** | **Purpose** |
| --- | --- | --- |
| **1. Executive Summary** | Synthesis of all data | High-level "Buy/Sell/Hold" thesis. |
| **2. Company Overview** | Web Research (Tavily) | Description of business model and product mix. |
| **3. Market Analysis** | Web Research (Tavily) | TAM/SAM calculations, competitor landscape. |
| **4. Financial Performance** | Vector DB (LlamaParse) | **Table:** Revenue, Net Income, EBITDA, CAGR. Analysis of trends. |
| **5. Strategic Moat** | Synthesis | Analysis of competitive advantage (Network effects, IP, etc.). |
| **6. Risks & Mitigations** | Vector DB (Risk Factors) + Web (News) | "Pre-Mortem" analysis. Regulatory, Operational, and Market risks. |
| **7. Conclusion** | Synthesis | Final recommendation. |

### 8.2 Citation and Grounding

A critical requirement is **citations**. The System Prompt for the Writer enforces a strict citation format: . Because the `AnalystState` carries the metadata from the retrieval steps, the Writer has access to the source provenance. It can append to the "Revenue" figure. This turns the memo from a creative writing exercise into a **defensible financial document**.29

## 9. Operational Resilience & Observability

Deploying an autonomous agent into production requires robust operational scaffolding.

### 9.1 Handling Hallucinations and Loops

* **Infinite Loops:** In a cyclic graph, an agent might get stuck trying to find data that doesn't exist. We implement a max\_iterations counter in the AnalystState. If the loop count exceeds 5, the Supervisor forces a transition to the Writer with a "Best Effort" flag, noting the missing data rather than stalling forever.8
* **Hallucination Guardrails:** The **Quality\_Reviewer** node serves as a halluncination check. It can be equipped with a separate LLM call that takes the generated claim and performs a "Fact Check" search. If the fact check fails, the memo is rejected.30

### 9.2 Observability with LangSmith

We utilize **LangSmith** to trace the execution. LangSmith visualizes the LangGraph nodes, showing exactly what input entered the "Financial Analyst" and what documents were retrieved. This is essential for debugging. If the agent reports the wrong revenue, we can look at the LangSmith trace to see if the error was in the *Retrieval* (wrong doc) or the *Reasoning* (wrong math).31

## 10. Conclusion

The construction of a "Market Research Analyst Agent" represents a significant leap forward in the application of Generative AI to finance. By moving beyond linear RAG and embracing the **Agentic** architecture provided by **LangGraph**, we create a system that can plan, self-correct, and reason.

The integration of **LlamaParse** solves the critical "unstructured table" problem, unlocking the vast wealth of data trapped in PDF 10-Ks. The use of **Tavily** ensures the agent is not hermetically sealed but is alive to the pulse of the market. Finally, the **Supervisor-Worker** topology allows for specialized prompting, ensuring that the "Creative Writer" and the "Forensic Accountant" do not interfere with each other's distinct cognitive tasks.

This architecture delivers on the promise of the "Action-Oriented" RAG: a system that does not just retrieve information, but acts upon it to produce high-value, synthesized knowledge.

# 11. Implementation Guide: Building the Market Research Analyst Agent

This section provides a detailed, conceptual implementation guide for the architecture described above. It breaks down the code structure, the specific prompt engineering strategies, and the configuration of the tools.

## 11.1 The Environment Setup

The foundation of the agent relies on a specific set of libraries. The Python environment must be configured to support LangGraph's stateful operations and the specific integrations for Tavily and Chroma.

Python

# Core dependencies for the Agentic Stack  
pip install langgraph langchain langchain-openai langchain-community  
pip install tavily-python chromadb llama-index llama-parse  
pip install pandas numpy # For the Python REPL tool

## 11.2 The State Definition (The "Memory")

As discussed in the architecture section, the State is the backbone of the agent. It is passed from node to node, accumulating the results of the research. We define this using TypedDict to ensure type safety within the graph.

Python

from typing import TypedDict, List, Annotated  
import operator  
from langchain\_core.messages import BaseMessage  
  
class AgentState(TypedDict):  
 company: str  
 ticker: str  
   
 # Research Data Accumulators  
 financial\_context: Annotated[List[str], operator.add] # Append-only list of financial findings  
 market\_context: Annotated[List[str], operator.add] # Append-only list of web findings  
   
 # The Iterative Draft  
 memo\_sections: dict  
   
 # Conversation History for the LLM  
 messages: Annotated, operator.add]  
   
 # Control Flags  
 research\_iterations: int  
 is\_data\_sufficient: bool

**Insight:** The use of Annotated[List, operator.add] is a LangGraph feature that ensures that when a node returns {"market\_context": ["News item A"]}, it *appends* to the existing list rather than overwriting it. This allows the Web Researcher to run multiple times, building a rich corpus of knowledge.5

## 11.3 Node 1: The Web Researcher (Tavily Integration)

The Web Researcher is responsible for the "Market Context" and "Company Overview." We configure the Tavily tool to be aggressive in its search depth.

Python

from langchain\_community.tools.tavily\_search import TavilySearchResults  
from langchain\_core.prompts import ChatPromptTemplate  
from langchain\_openai import ChatOpenAI  
  
# Initialize Tavily Tool  
tavily\_tool = TavilySearchResults(  
 max\_results=5,  
 search\_depth="advanced", # Crucial for deep research  
 include\_raw\_content=True # We want the full text for analysis  
)  
  
# The Node Function  
def web\_research\_node(state: AgentState):  
 query = f"{state['company']} market analysis recent news competitors"  
   
 # Execute Tool  
 results = tavily\_tool.invoke(query)  
   
 # Process Results (Synthesis)  
 # In a real impl, we would use an LLM here to summarize the raw content  
 summary = f"Found {len(results)} articles. Key themes:..."   
   
 return {"market\_context": [summary]}

**Strategic Insight:** The web\_research\_node shouldn't just dump raw JSON into the state. It should ideally perform a "Synthesis" step using an LLM to compress the 5 search results into a concise summary. This preserves the context window for the final writer.

## 11.4 Node 2: The Financial Analyst (LlamaParse & Chroma)

This is the most complex node. It interfaces with the Vector DB.

### 11.4.1 Ingestion Pipeline (Pre-computation)

Before the agent runs, we must ingest the 10-Ks.

Python

from llama\_parse import LlamaParse  
from langchain\_community.vectorstores import Chroma  
from langchain\_openai import OpenAIEmbeddings  
  
# 1. Parse PDF to Markdown  
parser = LlamaParse(result\_type="markdown")  
documents = parser.load\_data("./TSLA\_2023\_10K.pdf")  
  
# 2. Chunking (Markdown-Aware)  
# We use a splitter that respects table boundaries  
from langchain\_text\_splitters import MarkdownHeaderTextSplitter  
splitter = MarkdownHeaderTextSplitter(headers\_to\_split\_on=[("#", "Header 1"), ("##", "Header 2")])  
chunks = splitter.split\_text(documents.text)  
  
# 3. Indexing  
vectorstore = Chroma.from\_documents(  
 documents=chunks,  
 embedding=OpenAIEmbeddings(),  
 collection\_name="financial\_reports"  
)  
retriever = vectorstore.as\_retriever()

### 11.4.2 The Analyst Node Logic

The node uses the retriever to answer specific financial questions.

Python

def financial\_analyst\_node(state: AgentState):  
 # The agent decides what to look for based on the current gaps  
 questions =} revenue in 2023?",  
 f"What are the risk factors for {state['company']}?"  
 ]  
   
 findings =  
 for q in questions:  
 docs = retriever.invoke(q)  
 findings.append(f"Question: {q}\nAnswer: {docs.page\_content}")  
   
 return {"financial\_context": findings}

**Insight:** In a production system, this node would use **Tool Calling**. The LLM would generate the list of questions dynamically based on the memo\_sections that are currently empty. If the "Risk" section is empty, it generates risk-related queries.

## 11.5 Node 3: The Writer (Synthesis)

The Writer node takes the accumulated state and generates the Markdown output.

Python

def writer\_node(state: AgentState):  
 # Combine contexts  
 context = "\n".join(state['financial\_context'] + state['market\_context'])  
   
 prompt = ChatPromptTemplate.from\_template(  
 """You are an Investment Analyst. Write an investment memo for {company}.  
 Use the following retrieved context:  
 {context}  
   
 Format: Markdown.  
 Sections: Executive Summary, Financials, Risks.  
 """  
 )  
   
 chain = prompt | ChatOpenAI(model="gpt-4-turbo")  
 memo = chain.invoke({"company": state['company'], "context": context})  
   
 return {"memo\_sections": {"full\_draft": memo.content}}

## 11.6 The Graph Assembly

Finally, we wire the nodes together.

Python

from langgraph.graph import StateGraph, END  
  
workflow = StateGraph(AgentState)  
  
# Add Nodes  
workflow.add\_node("researcher", web\_research\_node)  
workflow.add\_node("analyst", financial\_analyst\_node)  
workflow.add\_node("writer", writer\_node)  
  
# Add Entry Point  
workflow.set\_entry\_point("researcher")  
  
# Add Edges (Linear flow for simplicity in this snippet, but can be cyclic)  
workflow.add\_edge("researcher", "analyst")  
workflow.add\_edge("analyst", "writer")  
workflow.add\_edge("writer", END)  
  
# Compile  
app = workflow.compile()

**Execution:**

Python

result = app.invoke({"company": "Tesla", "ticker": "TSLA"})  
print(result['memo\_sections']['full\_draft'])

This implementation guide translates the high-level architecture into executable logic. It demonstrates how **LangGraph** manages the flow, how **Tavily** feeds the web context, and how **LlamaParse/Chroma** provide the financial grounding.

## 12. Conclusion

The construction of a **Market Research Analyst Agent** is a sophisticated engineering challenge that pushes the boundaries of current Generative AI capabilities. It requires moving beyond the "chatbot" paradigm to the "agentic" paradigm.

The architecture proposed in this report—anchored by **LangGraph** for orchestration, **Tavily** for web intelligence, and **LlamaParse** for financial data ingestion—provides a robust foundation for this task. It addresses the core failure modes of standard RAG: the inability to parse tables, the lack of iterative planning, and the absence of self-correction.

By implementing this system, organizations can dramatically accelerate their investment research workflows. An agent that never sleeps, instantly digests 100-page 10-Ks, and continuously monitors the web for market signals represents a profound competitive advantage. The era of the AI Analyst is not just coming; with this architecture, it is buildable today.

#### Works cited

1. Conversational Agentic RAG: LangGraph, Memory, and Multi-Turn Logic, accessed November 27, 2025, <https://www.youtube.com/watch?v=Fc0qYL6Divg>
2. Improving RAG using LangGraph and LangChain | by Mehul Gupta - Medium, accessed November 27, 2025, <https://medium.com/data-science-in-your-pocket/improving-rag-using-langgraph-and-langchain-bb195bfe4b44>
3. LangGraph vs CrewAI: Let's Learn About the Differences - ZenML Blog, accessed November 27, 2025, <https://www.zenml.io/blog/langgraph-vs-crewai>
4. Building Multi-Agent Systems with LangGraph — A Comprehensive Guide | by S Sankar | Nov, 2025, accessed November 27, 2025, <https://medium.com/@AIBites/building-multi-agent-systems-with-langgraph-a-comprehensive-guide-c20ba96ab3ba>
5. Agentic RAG Using LangGraph: Build an Autonomous Customer Support Agent - LanceDB, accessed November 27, 2025, <https://lancedb.com/blog/agentic-rag-using-langgraph-building-a-simple-customer-support-autonomous-agent/>
6. langchain-ai/langgraph-supervisor-py - GitHub, accessed November 27, 2025, <https://github.com/langchain-ai/langgraph-supervisor-py>
7. Choosing Between CrewAI and LangGraph (Or Why You Might Use Both) - Medium, accessed November 27, 2025, <https://medium.com/@mayadakhatib/choosing-between-crewai-and-langgraph-or-why-you-might-use-both-c6cb8cd2a312>
8. Build a Text Classification Router in LangGraph | by Rjnclarke - Medium, accessed November 27, 2025, <https://medium.com/@rjnclarke/build-a-text-classification-router-in-langgraph-04ab77df4677>
9. Building Multi-Agent Systems with LangGraph-Supervisor - DEV Community, accessed November 27, 2025, <https://dev.to/sreeni5018/building-multi-agent-systems-with-langgraph-supervisor-138i>
10. 5-Getting Started With Agentic RAG With Detailed Implementation Using LangGraph, accessed November 27, 2025, <https://www.youtube.com/watch?v=Chl-cRcwVpA>
11. RAG for Pdf with tables : r/LangChain - Reddit, accessed November 27, 2025, <https://www.reddit.com/r/LangChain/comments/18xp9xi/rag_for_pdf_with_tables/>
12. Best PDF Extractor for RAG: LlamaParse vs Unstructured vs Vectorize - Chitika, accessed November 27, 2025, <https://www.chitika.com/best-pdf-extractor-rag-comparison/>
13. Parsing PDFs(text, image and tables) for RAG based applications using LlamaParse (LlamaIndex). | by Salujav | Medium, accessed November 27, 2025, <https://medium.com/@salujav4/parsing-pdfs-text-image-and-tables-for-rag-based-applications-using-llamaparse-llamaindex-0f4c5ed50fb7>
14. The RAG Engineer's Guide to Document Parsing : r/LangChain - Reddit, accessed November 27, 2025, <https://www.reddit.com/r/LangChain/comments/1ef12q6/the_rag_engineers_guide_to_document_parsing/>
15. Parsing PDFs with LlamaParse: a how-to guide - LlamaIndex, accessed November 27, 2025, <https://www.llamaindex.ai/blog/pdf-parsing-llamaparse>
16. RAG: Multi Vector Retriever - Kaggle, accessed November 27, 2025, <https://www.kaggle.com/code/marcinrutecki/rag-multi-vector-retriever>
17. Is the Langchain Multi-Vector Retriever Worth It? | by Andrew Batutin | Medium, accessed November 27, 2025, <https://medium.com/@abatutin/is-the-langchain-multi-vector-retriever-worth-it-0c8565e8a8fd>
18. Metadata Filtering - Chroma Docs, accessed November 27, 2025, <https://docs.trychroma.com/docs/querying-collections/metadata-filtering>
19. How to filter documents based on a list of metadata in LangChain's Chroma VectorStore?, accessed November 27, 2025, <https://stackoverflow.com/questions/78489329/how-to-filter-documents-based-on-a-list-of-metadata-in-langchains-chroma-vector>
20. Filtering through metadata : r/LangChain - Reddit, accessed November 27, 2025, <https://www.reddit.com/r/LangChain/comments/1hhqhkj/filtering_through_metadata/>
21. langchain-tavily - PyPI, accessed November 27, 2025, <https://pypi.org/project/langchain-tavily/>
22. tavily-ai/tavily-python: A python wrapper for Tavily search API - GitHub, accessed November 27, 2025, <https://github.com/tavily-ai/tavily-python>
23. LangChain - Tavily Docs, accessed November 27, 2025, <https://docs.tavily.com/documentation/integrations/langchain>
24. How to Build an Advanced AI Agent with Search (LangGraph, Python, Bright Data & More), accessed November 27, 2025, <https://www.youtube.com/watch?v=cUC-hyjpNxk>
25. LangGraph 101: Let's Build A Deep Research Agent | Towards Data Science, accessed November 27, 2025, <https://towardsdatascience.com/langgraph-101-lets-build-a-deep-research-agent/>
26. Build an AI Stock Analyst with LangGraph & MCP | Generative AI - Medium, accessed November 27, 2025, <https://medium.com/generative-ai/langgraph-react-agents-unlocked-fastmcp-custom-tool-integration-guide-6f082558cd0b>
27. Investment Memos: Tips, Templates, and How to Write One - Visible.vc, accessed November 27, 2025, <https://visible.vc/blog/investment-memo/>
28. The Investment Memo - Turning Startup Dreams into Conviction, accessed November 27, 2025, <https://taghash.io/blog/the-investment-memo-turning-startup-dreams-into-conviction/>
29. AI Investment Memo Generation Agent | Automate Deal Memos - V7 Go, accessed November 27, 2025, <https://www.v7labs.com/agents/ai-investment-memo-generation-agent>
30. Build a 🧠 Reflective Agentic RAG Workflow using LangGraph, Typesense, Tavily, Ollama and Cohere, accessed November 27, 2025, <https://nayakpplaban.medium.com/build-a-reflective-agentic-rag-workflow-using-langgraph-typesense-tavily-ollama-and-cohere-c9a7b0aca667>
31. Building an Accurate, Tested RAG Pipeline for PDF Document Search with LangChain, accessed November 27, 2025, <https://medium.com/@bpmcgough/building-an-accurate-tested-rag-pipeline-for-pdf-document-search-with-langchain-01032abbe5e1>
32. LangChain, accessed November 27, 2025, <https://www.langchain.com/>