Technical Documentation – FinHelp

This document introduces and explains the core workflow used in FinHelp, along with the tech stack and engineering that have been used to create this tool.

For starters, the main component of this project uses:

1) Tavily

Tavily is an AI-powered search engine API optimized for AI agents and LLM applications. Unlike traditional search engines that return raw results, Tavily is specifically designed to provide clean, structured, and contextually relevant information that language models can easily process. FinHelp uses Tavily in two different modes: the **Model Context Protocol (MCP)** implementation for the Finance Chat feature, and the **Direct API** for the Earnings Analyzer workflow.

The Tavily MCP integration allows our finance chat assistant to seamlessly search the web when answering user queries, with the search decision being made automatically by OpenAI's function calling mechanism. The Direct API, on the other hand, provides more granular control over search parameters and extraction methods, which is essential for finding and extracting specific earnings call transcripts from financial websites. Tavily's extraction capabilities handle complex web pages, parse JavaScript-rendered content, and return clean text that can be analyzed by our AI models.

Official Website: https://www.tavily.com
Documentation: https://docs.tavily.com
GitHub: https://github.com/tavily-ai

---

2) LangGraph

LangGraph is a framework for building stateful, multi-step agent workflows with conditional logic and cycles. Developed by LangChain, it allows developers to create sophisticated AI agents that can make decisions, retry failed operations, and maintain complex state across multiple processing steps. In FinHelp, LangGraph powers the Earnings Analyzer workflow, orchestrating the entire pipeline from transcript search to final summarization.

The framework uses a **graph-based architecture** where nodes represent individual processing steps (search, extract, summarize) and edges define the flow between these steps. Conditional edges enable dynamic routing based on the current state—for example, if transcript extraction fails, the workflow can automatically retry with alternative search strategies, or if no transcript is found, it gracefully terminates with an informative error message. LangGraph's **StateGraph** maintains a typed dictionary of all workflow data, ensuring type safety and clear data flow throughout the agent's execution. This approach provides significantly more control and reliability than simple sequential processing, as the agent can adapt its behavior based on intermediate results.

Official Website: https://langchain-ai.github.io/langgraph/
Documentation: https://langchain-ai.github.io/langgraph/tutorials/
GitHub: https://github.com/langchain-ai/langgraph

---

3) Model Context Protocol (MCP)

The Model Context Protocol is an open standard developed by Anthropic for connecting AI models to external data sources and tools. MCP provides a unified interface for LLMs to interact with various services like databases, APIs, file systems, and web search engines without requiring custom integration code for each service. In FinHelp, we use MCP to connect our OpenAI-powered finance chat to Tavily's search capabilities.

MCP works through a **client-server architecture** where the MCP client (running in our FastAPI backend) communicates with MCP servers (such as Tavily's hosted MCP endpoint) using a standardised protocol. This abstraction layer means our chat assistant can call tools like "search the web" without knowing the underlying implementation details of Tavily's API. The protocol supports both stdio (standard input/output) and HTTP-based communication, with FinHelp using the **streamable HTTP** transport to connect to Tavily's remote MCP server at `https://mcp.tavily.com/mcp/`. This architecture makes our system extensible—adding new tools (like database access, document retrieval, or additional search providers) requires minimal code changes.

Official Website: https://modelcontextprotocol.io
Documentation: https://modelcontextprotocol.io/docs
GitHub: https://github.com/modelcontextprotocol
Python SDK: https://github.com/modelcontextprotocol/python-sdk

---

4) OpenAI GPT-4o

OpenAI's GPT-4o (Generative Pre-trained Transformer 4 Optimized) serves as the core reasoning engine for FinHelp, handling both natural language understanding and generation. The model is used in two primary capacities: (1) as a **conversational agent** in the Finance Chat, where it employs function calling to determine when to invoke external tools, and (2) as an **analysis engine** in the Earnings Analyzer, where it processes lengthy earnings call transcripts and produces structured summaries with specific financial insights.

FinHelp leverages GPT-4o's **function calling** capability, which allows the model to recognize when it needs external information and programmatically invoke predefined functions (like web search). This is more sophisticated than simple prompt engineering—the model outputs structured JSON indicating which function to call and with what parameters, enabling truly agentic behavior. For earnings analysis, we use carefully engineered prompts that instruct the model to extract specific data points (revenue figures, EPS, forward guidance, executive quotes) rather than generating generic summaries, ensuring that users receive actionable financial intelligence.

Official Website: https://openai.com
API Documentation: https://platform.openai.com/docs
Model Details: https://platform.openai.com/docs/models/gpt-4o

---

5) MongoDB Atlas

MongoDB Atlas is a fully managed cloud database service that provides FinHelp's data persistence layer. We use MongoDB to store user accounts, authentication credentials (with bcrypt-hashed passwords), and complete chat session histories. Atlas's free M0 tier offers 512 MB of storage, which is sufficient for thousands of chat sessions given our intelligent token management and automatic cleanup policies.

The database schema uses three main collections: **users** (authentication and profile data), **chat_sessions** (conversation messages and loaded earnings contexts), and potential future collections for analytics. Each chat session document contains the full message history, any loaded earnings contexts (ticker, quarter, year, transcript excerpt, summary), and metadata (creation/update timestamps, message count). Our implementation uses **Motor**, an async MongoDB driver for Python, ensuring non-blocking database operations that don't slow down API responses. We employ a **session management strategy** that updates the current session during active use and creates new sessions only when users explicitly start a new chat, preventing database bloat while maintaining comprehensive history.

Official Website: https://www.mongodb.com/cloud/atlas
Documentation: https://www.mongodb.com/docs/atlas/
Python Driver (Motor): https://motor.readthedocs.io/

---

1. System Architecture

FinHelp implements a modern three-tier architecture with distinct separation between presentation, application logic, and data persistence layers.

Architecture Overview:

The system follows a **client-server model** with asynchronous communication patterns. The React frontend communicates with the FastAPI backend through RESTful HTTP endpoints, using JSON for data interchange. The backend orchestrates multiple AI services and tools through well-defined interfaces, maintaining separation of concerns between different functional modules.

Component Breakdown:

Presentation Layer (Frontend):

- React-based single-page application (SPA) with component-based architecture
- State management using React hooks (useState, useEffect, useCallback)

- localStorage for client-side token persistence
- Dual-interface design: Finance Chat and Earnings Analyzer tabs
- Real-time UI updates with loading states and error handling

Application Layer (Backend):

- FastAPI framework providing async HTTP endpoints
- Modular service architecture:
  - `app.py` - API routes and endpoint definitions
  - `agent.py` - LangGraph workflow orchestration
  - `earnings.py` - Tavily search and extraction utilities
  - `finance_chat.py` - Chat logic with tool integration
  - `auth.py` - JWT authentication and password hashing
  - `database.py` - MongoDB connection management
  - `mcp_client.py` - MCP session management for Tavily

Data Layer:

- MongoDB Atlas cloud database
- Async operations using Motor driver
- Collections: users, chat_sessions
- Indexing on user_id and updated_at for query performance

External Services Integration:

- **Tavily MCP** - Connected via streamable HTTP transport for Finance Chat
- **Tavily Direct API** - HTTP REST calls for Earnings Analyzer
- **OpenAI API** - GPT-4o for summarization and conversation
- **LangGraph** - In-process workflow execution

Data Flow:

```
User Browser
    ↓ (HTTPS)
React Frontend (localhost:3000)
    ↓ (REST API / JSON)
FastAPI Backend (localhost:8000)
    ↓ ↓ ↓
    ↓ ↓ └──→ MongoDB Atlas (User data, Sessions)
    ↓ └────→ OpenAI API (GPT-4o LLM)
    └──────→ Tavily (MCP + Direct API)
                 ↓
            Web Sources (Seeking Alpha, Yahoo Finance, etc.)
```

Communication Patterns:

- **Frontend ↔ Backend**: Async fetch() calls with JSON payloads
- **Backend ↔ MongoDB**: Async Motor driver with connection pooling
- **Backend ↔ Tavily MCP**: Persistent WebSocket-like connection via streamablehttp_client
- **Backend ↔ Tavily API**: Standard HTTPS requests with SDK wrapper

2. Agent Roles and Responsibilities

FinHelp employs one AI agent with multiple tools, each with distinct responsibilities and capabilities.

Agent 1: Finance Chat Assistant

Role: General-purpose financial information assistant with dynamic tool access

Responsibilities:

- Answer finance-related questions using both internal knowledge and web search
- Apply guardrails to reject non-finance queries
- Dynamically invoke Tavily search when current information is needed
- Maintain conversation context across multiple turns
- Provide source citations for web-sourced information

Decision-Making Process: The Finance Chat agent uses OpenAI's **function calling** mechanism to determine when external tools are needed. When a user query arrives, GPT-4o analyzes the question and decides whether to:

1. Answer directly from its training data (for conceptual questions like "What is P/E ratio?")
2. Invoke the `search_finance_info` function (for current data like "What is Apple's stock price today?")

The agent is configured with a **system prompt** that defines its scope (finance-only), capabilities (web search available), and behavioral guidelines (professional, clear, cite sources). When the function is called, the agent receives search results and synthesizes them into a coherent response with citations.

Tools Available:

- `search_finance_info(query: str)` - Invokes Tavily MCP to search the web

Guardrails:

- Rejects non-finance topics (sports, entertainment, politics, cooking, etc.)
- Returns polite redirection: "I'm specialized in finance topics..."
- Validates that questions relate to: stocks, companies, markets, economics, investing, personal finance

2.1 Non-Agent Workflow

Direct Tavily integration: Earnings Analyzer (LangGraph Workflow)

Role: Specialized tavily based analyser for finding, extracting, and analysing quarterly earnings call transcripts

Responsibilities:

- Search for specific earnings transcripts by ticker, quarter, and year
- Validate transcript relevance and extract full content
- Generate detailed financial summaries with specific metrics
- Handle failures gracefully with retry mechanisms
- Provide processing transparency through step-by-step logging

Multi-Node Architecture:

Search Node:

- **Purpose:** Find the earnings transcript URL
- **Tools:** Tavily Direct API with multiple search strategies
- **Logic:**
    - Executes 2-3 different search queries with varying specificity
    - Scores results based on: URL pattern match, title relevance, source quality
    - Filters for article pages containing "transcript" and matching quarter/year
    - Returns highest-scoring URL or triggers retry
- **State Updates:** Sets `transcript_found`, `transcript_url`, `source`

Extract Node:

- **Purpose:** Retrieve full transcript content from URL
- **Tools:** Tavily extract() API
- **Logic:**
    - Calls Tavily's extraction endpoint with the discovered URL
    - Validates content length (minimum 1,000 characters)
    - Checks for presence of financial terms (revenue, earnings, guidance)
    - Falls back to raw_content field if standard extraction insufficient
- **State Updates:** Sets `transcript_content` or `error`

Summarize Node:

- **Purpose:** Generate structured financial analysis
- **Tools:** OpenAI GPT-4o
- **Logic:**
    - Sends first 15,000 characters of transcript to GPT-4o
    - Uses specialized prompt requesting specific sections:
        - Financial Performance (revenue, EPS, margins, growth rates)
        - Key Highlights (announcements, executive quotes, segment performance)
        - Forward Guidance (next quarter expectations, projections)
        - Risks & Concerns (challenges, headwinds, competitive pressures)
        - Strategic Initiatives (investments, expansion plans, operational changes)
    - Temperature set to 0.2 for consistent, factual output

    - Max tokens: 2,000 to ensure comprehensive coverage
- **State Updates:** Sets `summary` or `error`

Retry Node:

- **Purpose:** Attempt alternative search if initial search fails
- **Logic:**
    - Increments retry_count
    - Clears previous error
    - Re-executes search with potentially different parameters
    - Maximum 1 retry attempt to prevent infinite loops
- **State Updates:** Increments `retry_count`, clears `error`

Conditional Routing Logic:

```
after search_node:
  if error AND retry_count < 1: → retry_node
  elif error: → END
  elif transcript_found: → extract_node
  else: → END

after retry_node:
  [same logic as search_node]

after extract_node:
  if error OR no content: → END
  elif has content: → summarize_node

after summarize_node:
  → END (workflow complete)
```

Multi-Earnings Context Manager

Role: Enhanced chat agent with awareness of multiple loaded earnings transcripts

Responsibilities:

- Maintain context of multiple earnings calls simultaneously (e.g., AAPL Q3 2024 + TSLA Q2 2024)
- Answer comparative questions ("Which company had better revenue growth?")
- Reference specific details from any loaded transcript
- Handle earnings-specific queries while maintaining general finance chat capability

Context Management: When earnings contexts are loaded, the system prompt is augmented with:

- Summary of each loaded earnings call
- First 5,000 characters of each transcript for detailed reference
- Clear delineation between different companies/quarters

This allows the LLM to perform cross-company analysis and answer questions like "Compare their guidance" or "Which faced more headwinds?" with specific evidence from the transcripts.

---

3. LangGraph Workflow Deep Dive

State Machine Design:

LangGraph implements the Earnings Analyzer as a **finite state machine** where each node is an async Python function that receives the current state, performs operations, and returns an updated state. The state is a TypedDict that flows through the entire workflow:

```
class EarningsAgentState(TypedDict):
    # Inputs
    ticker: str
    quarter: str
    year: str

    # Messages (for user feedback)
    messages: list  # List of HumanMessage/AIMessage objects

    # Search results
    transcript_found: bool
    transcript_url: str
    transcript_title: str

    # Content
    transcript_content: str

    # Output
    summary: str

    # Error handling
    error: str
    retry_count: int
```

Node Implementation Pattern:

Each node follows a consistent pattern:

1. Extract relevant data from state
2. Log progress message (appended to state["messages"])
3. Perform async operation (search, extract, or API call)
4. Handle success/failure
5. Update state with results or errors
6. Return modified state

Graph Construction:

```
workflow = StateGraph(EarningsAgentState)

# Add nodes
```

```
workflow.add_node("search", search_node)
workflow.add_node("retry", retry_search_node)
workflow.add_node("extract", extract_node)
workflow.add_node("summarize", summarize_node)

# Define entry point
workflow.set_entry_point("search")

# Add conditional edges with routing functions
workflow.add_conditional_edges(
    "search",
    should_continue_after_search,
    {"extract": "extract", "retry": "retry", "end": END}
)

# Compile into executable graph
earnings_agent = workflow.compile()
```

Execution Flow:

When `earnings_agent.ainvoke(initial_state)` is called:

1. Graph starts at the entry point (search_node)
2. Node executes and returns updated state
3. Routing function evaluates state to determine next node
4. Process continues until reaching END
5. Final state is returned to caller

Error Handling Strategy:

- **Graceful Degradation:** Errors don't crash the workflow; they set error flags and route to END
- **Retry Logic:** Search failures trigger one retry attempt with alternative strategies
- **User Transparency:** All steps logged to state["messages"] array for debugging visibility
- **Informative Errors:** Error messages explain what failed and why (e.g., "Transcript not found for TSLA Q3 2024. It may not be published yet.")

State Persistence Across Nodes:

LangGraph's state management ensures that data from earlier nodes is available to later nodes. For example:

- `search_node` sets `transcript_url`
- `extract_node` reads `transcript_url` to know what to extract
- `summarize_node` reads `transcript_content` to know what to analyze

This eliminates the need for global variables or complex parameter passing.

---

4. Database Schema

FinHelp uses MongoDB's document-oriented structure with the following collections:

Collection: `users`

Stores user account information and authentication credentials.

```
{
  "_id": ObjectId("..."),
  "email": "user@example.com",
  "name": "John Doe",
  "password": "$2b$12$hashedpassword...",  // bcrypt hash
  "created_at": ISODate("2024-10-26T10:30:00Z")
}
```

Indexes:

- `email` (unique) - Fast user lookup during login

Fields:

- `_id`: MongoDB auto-generated unique identifier
- `email`: User's email address (unique constraint, used for login)
- `name`: Display name
- `password`: Bcrypt-hashed password (never stored in plaintext)
- `created_at`: Account creation timestamp

Collection: `chat_sessions`

Stores complete chat conversations with earnings contexts and metadata.

```
{
  "_id": ObjectId("..."),
  "user_id": "507f1f77bcf86cd799439011",
  "messages": [
    {
      "role": "user",
      "content": "What is Apple's business model?"
    },
    {
      "role": "assistant",
      "content": "Apple operates primarily through..."
    }
  ],
  "earnings_contexts": [
    {
      "ticker": "AAPL",
      "quarter": "Q3",
      "year": "2024",
      "summary": "Apple reported...",
      "transcript_content": "Full transcript text..."
    }
  ],
  "message_count": 12,
  "created_at": ISODate("2024-10-26T10:00:00Z"),
  "updated_at": ISODate("2024-10-26T10:45:00Z")
}
```

Indexes:

- `user_id` - Fast retrieval of all sessions for a user
- `updated_at` (descending) - Efficient sorting for "recent chats"

Fields:

- `user_id`: Reference to users._id (foreign key equivalent)
- `messages`: Array of chat messages with role (user/assistant) and content
- `earnings_contexts`: Array of loaded earnings call data
    - `ticker`: Stock symbol
    - `quarter`: Q1, Q2, Q3, or Q4
    - `year`: Calendar year
    - `summary`: AI-generated earnings summary
    - `transcript_content`: Truncated transcript text (first ~10K chars stored)
- `message_count`: Cached count for quick display in history list
- `created_at`: Session creation time
- `updated_at`: Last modification time (updated on every save)

Storage Optimization:

To stay within MongoDB Atlas free tier limits (512 MB), FinHelp implements:

1. **Token-based Truncation:** Messages exceeding ~8,000 tokens are truncated from the beginning, keeping recent conversation intact
2. **Session Limit:** Maximum 5 sessions per user; older sessions automatically deleted
3. **Transcript Truncation:** Only first 10,000 characters of transcripts stored in chat contexts
4. **Update-in-Place:** Active sessions are updated rather than creating duplicates (session < 1 hour old)

Calculation:

- Average session size: ~40 KB (after truncation)
- 512 MB ÷ 40 KB = ~12,800 sessions
- At 5 sessions per user = ~2,560 users supported

---

5. Agent Roles and Behaviors

Finance Chat Agent

System Prompt:

```
You are a helpful finance assistant. You ONLY answer questions related to:
```

```
- Finance, investing, trading, stocks, bonds, ETFs
- Companies, business models, corporate finance
- Economic concepts, markets, GDP, inflation
- Personal finance, budgeting, retirement planning
- Financial ratios, analysis, accounting

If someone asks about non-finance topics, politely decline.
When answering, use the search tool for current information and cite
sources.
```

Function Definition:

```
{
  "name": "search_finance_info",
  "description": "Search the web for real-time financial information...",
  "parameters": {
    "type": "object",
    "properties": {
      "query": {"type": "string", "description": "Search query"}
    }
  }
}
```

Behavior Patterns:

Pattern 1: Direct Answer (No Search)

- User: "What is a P/E ratio?"
- Agent Decision: No search needed (conceptual question)
- Response: Explains from training data

Pattern 2: Search-Augmented Answer

- User: "What is Microsoft's current revenue?"
- Agent Decision: Invokes search_finance_info("Microsoft revenue 2024 latest")
- Tavily returns: Articles with revenue data
- Agent: Synthesizes answer with citations

Pattern 3: Rejection (Non-Finance)

- User: "How do I cook pasta?"
- Agent Decision: Violates finance-only guardrail
- Response: "I'm specialized in finance topics. I can help with questions about investing, companies, markets..."

Earnings Analyzer Agent (LangGraph)

Agent Characteristics:

- **Deterministic Workflow:** Unlike conversational agents, this follows a fixed pipeline

- **Multi-Tool Orchestration:** Coordinates Tavily search, Tavily extract, and OpenAI summarization
- **State-Based Decision Making:** Uses state properties to route between nodes
- **Error Resilience:** Retry mechanisms and graceful failure handling

Node Specifications:

search_node:

- **Input:** `ticker`, `quarter`, `year`
- **Process:**
  1. Constructs search queries: `"{ticker} {quarter} {year} earnings call transcript"`
  2. Calls `search_for_any_transcript()` which uses Tavily Direct API
  3. Filters results for transcript articles matching quarter/year
  4. Scores candidates based on URL patterns and source reliability
- **Output:** `transcript_url` or `error`
- **Next Node:** `extract_node` (if found), `retry_node` (if error), or `END`

extract_node:

- **Input:** `transcript_url`
- **Process:**
  1. Calls `extract_webpage_tavily(url)` using Tavily's extract API
  2. Validates content length (>1,000 chars) and presence of financial keywords
  3. Handles JSON-wrapped responses by extracting `raw_content` field
- **Output:** `transcript_content` or `error`
- **Next Node:** `summarize_node` (if successful) or `END`

summarize_node:

- **Input:** `transcript_content`, `ticker`, `quarter`, `year`
- **Process:**
  1. Samples first 15,000 characters of transcript
  2. Constructs specialized prompt with required output structure
  3. Calls OpenAI GPT-4o with temperature=0.2 for consistency
  4. Receives structured summary covering 5 key sections
- **Output:** `summary` or `error`
- **Next Node:** `END`

retry_node:

- **Input:** Current state with error
- **Process:**
  1. Increments `retry_count`
  2. Clears `error` flag
  3. Re-executes search with potentially broader queries
- **Output:** Delegates to search_node output
- **Next Node:** Based on search results (max 1 retry)

Routing Functions:

```
def should_continue_after_search(state):
    if state["error"] and state["retry_count"] < 1:
        return "retry"
    elif state["error"]:
        return "end"
    elif state["transcript_found"]:
        return "extract"
    else:
        return "end"
```

This logic encodes the agent's decision-making: "If I failed but haven't retried yet, try again. If I found something, extract it. Otherwise, give up."

Multi-Earnings Context Agent

Role: Enhanced chat agent with simultaneous access to multiple earnings transcripts

System Prompt Construction:

When users load multiple earnings (e.g., AAPL Q3 2024 + TSLA Q2 2024), the system prompt is dynamically constructed:

```
You are an expert financial analyst discussing multiple earnings calls.

You have access to the following earnings calls:

### AAPL Q3 2024 Earnings Call
Summary: [AI-generated summary]
Transcript excerpt: [First 5,000 chars]

### TSLA Q2 2024 Earnings Call
Summary: [AI-generated summary]
Transcript excerpt: [First 5,000 chars]

Answer questions about these specific calls. You can compare, analyze
trends,
and reference specific details.
```

Capabilities:

- Cross-company comparisons: "Which had better margin expansion?"
- Temporal analysis: "How did AAPL's guidance change from Q2 to Q3?"
- Specific detail retrieval: "What did Tesla's CEO say about production targets?"
- Trend identification: "Are both companies facing similar supply chain issues?"

Context Size Management:

With multiple transcripts, token counts can grow large:

- Each transcript excerpt: ~5,000 chars = ~1,250 tokens

- Summary: ~500 tokens
- 3 loaded earnings = ~5,250 tokens in system prompt
- Leaves ~2,750 tokens for conversation history (with 8K token limit)

When token limits approach, older conversational messages are truncated, but earnings contexts are preserved as they're the primary value.

---

6. LangGraph Flow Detailed Breakdown

Graph Compilation and Execution:

LangGraph compiles the workflow into an executable graph at application startup:

```
earnings_agent = workflow.compile()
```

This compilation process:

1. Validates all nodes are connected
2. Checks routing functions return valid next-node names
3. Ensures state schema consistency across nodes
4. Creates optimized execution plan

Invocation:

```
final_state = await earnings_agent.ainvoke(initial_state)
```

The `ainvoke()` method:

- Accepts initial state dictionary
- Executes workflow asynchronously
- Returns final state after reaching END
- Supports cancellation and timeout (not currently implemented)

State Transitions Example:

```
Initial State:
{
  "ticker": "AAPL",
  "quarter": "Q3",
  "year": "2024",
  "messages": [HumanMessage(...)],
  "transcript_found": False,
  "error": "",
  ...
}

After search_node:
{
  ...
  "transcript_found": True,
  "transcript_url": "https://seekingalpha.com/article/...",
  "messages": [..., AIMessage("✅ Found transcript")],
  ...
}

After extract_node:
{
  ...
  "transcript_content": "Apple Inc Q3 2024 Earnings Call...",
  "messages": [..., AIMessage("✅ Extracted 15,247 characters")],
  ...
}

After summarize_node:
{
  ...
  "summary": "**Financial Performance**\nRevenue: $94.93B...",
  "messages": [..., AIMessage("✅ Analysis complete")],
  ...
}
```

Error State Handling:

If extraction fails:

```
After extract_node (failed):
{
  ...
  "error": "Extraction failed - content too short",
  "messages": [..., AIMessage("❌ Extraction failed")],
  ...
}

Routing decision: END (no retry for extraction failures)
```

Benefits of LangGraph Approach:

1. **Modularity:** Each node is independently testable
2. **Transparency:** State changes are explicit and traceable
3. **Flexibility:** Routing logic can be modified without changing nodes
4. **Debuggability:** Full state history available for troubleshooting
5. **Extensibility:** New nodes (e.g., "validate_transcript", "enrich_with_financials") can be added easily

Comparison to Sequential Approach:

Traditional sequential code:

```
url = search()
content = extract(url)
summary = summarize(content)
```

Problems: No error handling between steps, no retry logic, no state visibility

LangGraph approach:

- Explicit state at each step
- Conditional routing on errors
- Retry mechanisms built-in
- Full execution trace for debugging

---

7. Database Schema Design Rationale

Why MongoDB (NoSQL) vs Relational?

1. **Flexible Schema:** Chat messages vary in structure (some have sources, some don't; earnings contexts are optional)
2. **Nested Documents:** Messages array naturally fits document model vs. complex joins
3. **Rapid Iteration:** Schema changes don't require migrations
4. **JSON Native:** Perfect match for API JSON payloads
5. **Horizontal Scaling:** Future-proof if the user base grows

Schema Evolution Considerations:

Current schema supports future enhancements:

- Adding `session_title` field (user-defined names for chats)
- `tags` array for categorizing sessions (e.g., ["earnings", "AAPL"])
- `shared_with` array for collaborative features
- `metadata` object for analytics (query latency, token usage, etc.)

Query Patterns:

1. **Load Recent Sessions:**
2. `db.chat_sessions.find({"user_id": user_id}).sort("updated_at", -1).limit(5)`
   - Uses compound index on (user_id, updated_at)
   - O(log n) performance with index
3. **Load Specific Session:**
4. `db.chat_sessions.find_one({"_id": session_id, "user_id": user_id})`
   - Security: Ensures user owns the session
   - O(1) lookup on _id (default index)
5. **Delete Old Sessions:**
6. `db.chat_sessions.delete_many({"_id": {"$in": old_session_ids}})`
   - Batch deletion for efficiency
   - Triggered when user exceeds 5 sessions

Consistency Model:

- **Eventual Consistency:** Acceptable for chat history (slight delay in history updates is fine)
- **Strong Consistency:** Required for authentication (ensured by MongoDB's default write concern)
