# Strategic Architecture for Intelligent Financial Analysis: A Comprehensive Guide to Building Agentic Chatbots

## 1. Introduction: The Paradigm Shift in Financial Analytics

The intersection of quantitative finance and generative artificial intelligence has precipitated a fundamental transformation in how market participants interact with data. Historically, financial analysis was a bifurcated discipline: quantitative analysts operated within the rigid syntax of Python scripts, SQL queries, and Bloomberg terminals, while fundamental analysts synthesized qualitative insights from lengthy 10-K filings and earnings call transcripts. The emergence of Large Language Models (LLMs) has bridged this divide, enabling the creation of conversational interfaces capable of performing both rigorous data manipulation and nuanced textual reasoning. This report outlines a comprehensive strategy for constructing a financial analysis chatbot, utilizing a robust open-source technology stack comprising **LangChain** for orchestration, **Streamlit** for the user interface, and **yfinance** for real-time market data ingestion.

The objective of this architectural blueprint is to move beyond simple question-answering systems toward **Agentic Workflows**. Unlike passive chatbots that merely retrieve pre-indexed information, an agentic system acts as a reasoning engine. It observes the user's intent, formulates a multi-step execution plan—potentially involving data scraping, statistical modeling, and visualization—and iteratively corrects its course based on intermediate observations.1 This capability is particularly vital in finance, where queries often require complex dependency chains, such as "Compare the volatility of the technology sector against the S&P 500 over the last fiscal quarter and visualize the correlation."

This report provides an exhaustive technical analysis of the required components, examining the theoretical underpinnings of the reactive programming model in Streamlit, the security implications of executing arbitrary Python code via LLMs, and the architectural patterns for integrating retrieval-augmented generation (RAG) with real-time web search. By synthesizing insights from diverse technical documentation and expert tutorials, this document serves as a definitive guide for developers seeking to engineer production-grade financial AI assistants.

## 2. The Core Technology Stack: Selection and Synergy

The selection of the technology stack is the foundational decision that dictates the scalability, maintainability, and capability of the financial agent. The triad of LangChain, Streamlit, and yfinance represents a strategic alignment of tools optimized for rapid prototyping, data-centric interactions, and community support.

### 2.1 LangChain: The Cognitive Orchestrator

LangChain serves as the cognitive backbone of the application. In the context of financial analysis, it is not merely a wrapper for API calls to OpenAI or Anthropic; it is the framework that enables **chains of thought**. Financial queries are rarely zero-shot tasks. They require context retention, tool selection, and structured output parsing.

LangChain facilitates this through several key abstractions:

* **Prompt Templates:** Financial analysis requires precise instruction tuning. System prompts must enforce strict constraints (e.g., "Do not hallucinate stock prices," "Always cite the fiscal year") to ensure reliability.2
* **Agents and Toolkits:** The core innovation relevant to this domain is the Agent, which uses the LLM to decide *what* to do. The create\_pandas\_dataframe\_agent is specifically optimized to treat a Pandas DataFrame as a queryable database, translating natural language into Python manipulation commands.5
* **Memory Management:** Financial conversations are stateful. A user asking "What is the P/E ratio of Apple?" followed by "How does that compare to Microsoft?" requires the system to retain the context of the previous entity and metric. LangChain's memory modules (e.g., ConversationBufferMemory) manage this state, injecting relevant history into the prompt window for each subsequent turn.1

### 2.2 Streamlit: The Reactive Interface

Streamlit revolutionizes the frontend development for data applications by employing a **reactive programming model**. Unlike traditional web frameworks (Flask, Django) that require defining explicit routes and handling HTTP requests, Streamlit scripts run top-to-bottom every time an interaction occurs. This architectural choice significantly lowers the barrier to entry for Python developers but introduces specific challenges regarding state persistence.7

For a chatbot, the reactive model implies that every time a user inputs a message, the entire script re-executes. Without careful management of the **Session State**, the chat history would vanish with each update. This report will detail the implementation of st.session\_state to persist the conversation across reruns, ensuring a seamless user experience indistinguishable from standard messaging platforms.6 Furthermore, Streamlit's native support for data visualization (via st.pyplot, st.plotly\_chart) makes it the ideal candidate for rendering the graphical outputs generated by the financial agent.10

### 2.3 yfinance: The Data Ingestion Layer

Data is the lifeblood of any financial application. **yfinance** offers a Pythonic interface to scrape public financial data from Yahoo Finance. While not an official API, its ubiquity and ease of use make it the standard for open-source financial analysis.

The library provides three critical categories of data essential for a holistic analysis:

1. **Market Data:** Historical price and volume data (Open, High, Low, Close, Volume) accessible via the history() method. This is the raw material for technical analysis (e.g., calculating Moving Averages, RSI, Bollinger Bands).7
2. **Fundamental Data:** Balance sheets, cash flow statements, and income statements. This allows the agent to perform fundamental analysis, calculating metrics like Debt-to-Equity ratios or Free Cash Flow yield.
3. **Qualitative Data:** Ticker metadata, sector information, and news streams. This provides the contextual grounding necessary for the LLM to "understand" the company beyond the numbers.7

The integration of yfinance with **Pandas** is the critical link. The data retrieved is returned as Pandas DataFrames, which are the native language of data science and the precise format required by LangChain's DataFrame Agent.5

### Table 1: Technology Stack Synergy and Responsibilities

| **Component** | **Primary Role** | **Key Artifacts** | **Integration Point** |
| --- | --- | --- | --- |
| **LangChain** | Reasoning Engine | Agents, Chains, Memory, Tools | Orchestrates LLM logic and binds tools to the model. |
| **Streamlit** | User Interface | st.chat\_message, st.session\_state | Visualizes the agent's output and captures user input. |
| **yfinance** | Data Provider | yf.Ticker, DataFrame | Fetches raw data which is fed into the LangChain agent. |
| **Pandas** | Data Structure | DataFrames, Series | Serves as the intermediate format for analysis and plotting. |
| **Matplotlib/Plotly** | Visualization | Figures, Charts | Generated by the agent, rendered by Streamlit. |

## 3. Data Engineering: Structuring Financial Intelligence

Before an AI agent can reason about the market, the underlying data pipelines must be robustly engineered. The quality of the agent's insights is directly proportional to the structure and cleanliness of the data ingestion layer.

### 3.1 Advanced Usage of the yfinance API

While basic tutorials often demonstrate fetching the last closing price, a robust financial chatbot requires deep interaction with the yfinance object model. The Ticker object is the entry point for all company-specific data.

* **Historical Data Granularity:** The history() method supports various intervals (1m, 1h, 1d, 1wk, 1mo). An intelligent agent must be capable of selecting the appropriate granularity based on the user's query intent. A request for "intraday volatility" requires 1m or 5m data, whereas "long-term trend analysis" necessitates 1wk or 1mo data to avoid noise.7
* **Handling Multi-Ticker Queries:** Comparative analysis is a staple of finance. The agent must handle lists of tickers (e.g., ``) and restructure the resulting MultiIndex DataFrame into a format suitable for the LLM. Flattening column headers (e.g., converting ('Close', 'AAPL') to AAPL\_Close) is often a necessary preprocessing step to reduce ambiguity for the Pandas Agent.4
* **Fundamental Data Retrieval:** Accessing ticker.info returns a large dictionary. Passing this entire dictionary to the LLM's context window is inefficient and costly. A "Fetch Fundamentals" tool should be engineered to extract specific keys (e.g., forwardPE, pegRatio, marketCap) rather than dumping raw JSON. This optimization—**Context Pruning**—is critical for maintaining performance and reducing token costs.7

### 3.2 The Pivot to Pandas: Data Normalization

The LangChain create\_pandas\_dataframe\_agent relies on the predictable structure of Pandas DataFrames. Therefore, all data fetched from yfinance must be normalized.

1. **Date Indexing:** yfinance returns data with a DateTime index. The agent must be prompted to understand that the "Date" is the index, not a column, or the index should be reset (df.reset\_index()) to make the date accessible as a standard column for querying (e.g., "Filter for rows where Date is greater than 2023-01-01").5
2. **Missing Data Handling:** Financial data is often sparse (e.g., holidays, trading halts). The pipeline must include a cleaning step (e.g., df.fillna(method='ffill')) before passing the dataframe to the agent. If the agent encounters NaN values during a calculation, the Python REPL may throw an error, breaking the conversation flow.11
3. **Schema Injection:** To assist the LLM, the prompt should explicitly include the dataframe's schema (columns and dtypes). LangChain does this automatically, but for large datasets, providing a summarized schema (first 5 rows + column descriptions) in the system prompt enhances the model's understanding of the available data fields.12

## 4. The Cognitive Core: Architecting the Agent

The transition from a script that *calculates* to an agent that *reasons* involves the implementation of the ReAct (Reasoning + Acting) pattern. This section details the construction of the LangChain agent, focusing on tool binding, prompt engineering, and the security mechanics of code execution.

### 4.1 The Pandas DataFrame Agent Architecture

The create\_pandas\_dataframe\_agent is a specialized implementation designed to bridge the gap between natural language and tabular data. It functions by treating the Python interpreter as a tool.

* **Mechanism:** When a user asks, "What is the correlation between Volume and Close price?", the agent does not retrieve a pre-calculated answer. Instead, it generates a Python script: df['Volume'].corr(df['Close']). It executes this script in a virtual REPL, captures the standard output, and then synthesizes the result into a sentence.2
* **Zero-Shot vs. OpenAI Functions:**
  + **Zero-Shot ReAct:** This agent type uses a text-based thought loop ("Thought: I need to calculate correlation. Action: python\_repl\_ast..."). While flexible, it is prone to parsing errors if the LLM's output deviates from the expected format.1
  + **OpenAI Functions (Tool Calling):** This is the superior architecture for modern financial agents. It leverages the fine-tuned capability of models like GPT-3.5-Turbo and GPT-4 to output structured JSON arguments for function calls. This drastically reduces "hallucinated" tool calls and ensures valid code generation. The agent\_type="tool-calling" or OPENAI\_FUNCTIONS parameter should be strictly used for production stability.4

### 4.2 Security: The Sandboxing Imperative

Allowing an LLM to generate and execute Python code (allow\_dangerous\_code=True) introduces a massive security vulnerability: **Remote Code Execution (RCE)**. A malicious user could theoretically prompt the agent to execute import os; os.system('rm -rf /') or exfiltrate environment variables containing API keys.5

**Mitigation Strategies:**

1. **Dockerized Execution:** The standard PythonREPLTool runs in the host process. A secure architecture replaces this with a **DockerSandbox**. The agent sends code to a containerized Python environment (e.g., via a Docker API or a library like e2b). The code executes in isolation, and only the text output (stdout) is returned to the host. If the container is compromised, the host machine remains secure.1
2. **Read-Only Filesystem:** Where Docker is not feasible (e.g., certain cloud deployments), the execution environment should be restricted to a read-only filesystem, preventing the agent from modifying or deleting files.
3. **Library Whitelisting:** The execution environment should strictly limit available imports. A financial agent may only need pandas, numpy, and matplotlib. Importing os, sys, or subprocess should be blocked at the AST (Abstract Syntax Tree) level before execution.13

### 4.3 Prompt Engineering for Financial Context

The default system prompt for the Pandas Agent is generic. To create a "Financial Analyst" persona, the prompt must be overridden with domain-specific instructions.

* **System Prompt Customization:**
  + *Role Definition:* "You are a senior quantitative analyst. You rely on data, not assumptions."
  + *Output constraints:* "Always format currency values with two decimal places. When mentioning dates, use the format YYYY-MM-DD."
  + *Error Handling:* "If the dataframe is empty or the calculation fails, explicitly state that data is unavailable rather than guessing."
* **Suffix/Prefix Injection:** LangChain allows appending instructions to the prompt. For a yfinance agent, a useful suffix is: "The dataframe df contains historical stock data. The index is the Date. Columns are Open, High, Low, Close, Volume. Do not assume other columns exist without checking df.columns first".12

## 5. Visualization Strategy: The "Blind Agent" Problem

A major challenge in text-based LLM agents is visualization. When an analyst asks for a "trend line," they expect a chart, not a text description of a chart. However, LLMs output text, and standard Python plotting libraries (matplotlib) render to a backend that is often invisible in a stateless web context.

### 5.1 The Challenge of Headless Plotting

When the agent executes df.plot() inside the REPL, the plot object is created in memory. In a local Jupyter notebook, this displays automatically. In a hosted Streamlit app, this object is lost unless explicitly captured and passed to the frontend.14 The agent might report "I have plotted the chart" while the user sees nothing.

### 5.2 Architectural Solutions for Plotting

There are three primary strategies to solve this, ranked by complexity and robustness:

#### Strategy A: The "Save and Retrieve" Pattern (Beginner)

The agent is instructed to save the plot to a file named plot.png.

* **Prompt Instruction:** "If asked to plot, save the figure to 'plot.png' and reply 'Plot saved'."
* **Streamlit Logic:** The frontend code checks for the existence of plot.png after every agent interaction. If found, it renders it using st.image("plot.png").
* **Limitation:** This is not thread-safe. In a multi-user environment, User A's plot could overwrite User B's plot. It requires unique file naming (e.g., using UUIDs in the session state) to be viable.15

#### Strategy B: The "Return Object" Pattern (Intermediate)

We define a custom tool that wraps the plotting logic and returns the Figure object itself, rather than printing to stdout.

* **Tool Definition:** A function generate\_plot(df, x, y, type) is created. It uses matplotlib.pyplot.subplots() to create a figure, plots the data, and returns the fig object.
* **Integration:** The agent calls this tool. The AgentExecutor must be configured to return intermediate steps or specific tool outputs.
* **Frontend Logic:** Streamlit detects that the tool output is a matplotlib Figure and renders it using st.pyplot(fig).10

#### Strategy C: The Python REPL Capture (Advanced)

This involves customizing the PythonREPL tool to capture not just stdout but also any "display" artifacts.

* **Implementation:** Overriding the run method of the tool to check plt.get\_fignums(). If a figure exists after code execution, it is extracted, passed to the Streamlit context, and then the figure is cleared from memory to prevent artifacts in subsequent queries.17
* **Streamlit Synergy:** This method allows the agent to use standard pandas plotting syntax (df.plot()) without needing to know about the frontend implementation. It provides the most natural "analyst" experience.

### Table 2: Visualization Strategy Comparison

| **Strategy** | **Complexity** | **Thread Safety** | **Agent Freedom** | **Recommended For** |
| --- | --- | --- | --- | --- |
| **Save to File** | Low | Low (requires UUIDs) | High (Standard Code) | Prototyping |
| **Custom Tool** | Medium | High | Low (Restricted Syntax) | Production (Strict Control) |
| **REPL Capture** | High | Medium | High (Standard Code) | Advanced Conversational UI |

## 6. The User Interface: Mastering Streamlit's Reactive Loop

Streamlit's simplicity belies the complexity of its execution model. Building a persistent chat interface requires a deep understanding of **Session State** and **Callbacks**.

### 6.1 The Mechanics of Persistence

In Streamlit, the variable messages = defined at the top of a script will be reset to `` every time the user clicks "Send". To create a memory of the conversation, we must use st.session\_state.

* **Initialization Pattern:**  
  Python  
  if "messages" not in st.session\_state:  
   st.session\_state.messages =  
    
  This ensures the list exists only once and persists across reruns.1
* **The Rerun Cycle:**
  1. **Script Start:** Load imports, setup st.session\_state.
  2. **Render History:** Loop through st.session\_state.messages and use st.chat\_message(msg["role"]).write(msg["content"]) to display the conversation so far.6
  3. **Capture Input:** prompt = st.chat\_input() halts execution and waits for the user.
  4. **Interaction:** When the user types and hits Enter, the script *reruns*. The input is now captured in prompt.
  5. **Processing:** The script appends the user prompt to session state, displays it, calls the LangChain agent, and then appends/displays the agent's response.
  6. **Completion:** The script finishes. The UI is updated.

### 6.2 Managing Latency with UX Elements

Financial API calls and LLM inference are slow. A frozen screen frustrates users.

* **st.spinner:** Wrapping the agent execution in with st.spinner("Analyzing market data..."): provides essential visual feedback.
* **Streaming Responses:** Advanced implementations use StreamlitCallbackHandler. This LangChain utility hooks into the agent's execution loop and streams the "Thought" and "Action" tokens directly to the Streamlit UI in real-time. This is crucial for keeping the user engaged while the agent performs long calculations or web searches.8

## 7. Integrating External Knowledge: RAG and Web Search

Quantitative data from yfinance is only half the picture. Financial analysis requires qualitative context—news, earnings reports, and macro trends. This requires two distinct retrieval mechanisms: **Web Search** (for real-time news) and **RAG** (for document analysis).

### 7.1 Web Search with Tavily

For real-time context (e.g., "Why did Apple stock drop today?"), the agent needs internet access. **Tavily** is a search API specifically built for AI agents. Unlike Google or DuckDuckGo, which return HTML that requires heavy parsing, Tavily returns clean, LLM-ready context.21

* **Tavily vs. DuckDuckGo:**
  + **DuckDuckGo (DDG):** Free, but returns generic web snippets. Often requires extra scraping steps to get the full content, which slows down the agent.
  + **Tavily:** Optimized for RAG. It aggregates results, filters for "news" or "finance" topics, and provides "answer" fields that synthesize the findings, reducing the cognitive load on the LLM.22
  + **Integration:** The TavilySearchResults tool is added to the agent's toolkit. When the user asks a current events question, the agent (via OpenAI Functions) recognizes it cannot answer from its training data or yfinance history and autonomously invokes the Tavily tool.24

### 7.2 Document Analysis (RAG) for 10-K Filings

To analyze specific documents (e.g., a PDF of an annual report), we implement a **Retrieval-Augmented Generation** pipeline.

1. **Ingestion:** Users upload a PDF via st.file\_uploader.
2. **Parsing & Chunking:** PyPDFLoader extracts text. Financial documents have complex structures (columns, tables). Standard text splitters often break these tables, rendering them unintelligible to the LLM. Using a **RecursiveCharacterTextSplitter** with large overlap is essential to preserve the context of financial narratives.25
3. **Vectorization:** The text chunks are embedded using OpenAI's text-embedding-3-small (or HuggingFace models for local privacy) and stored in a vector database like **ChromaDB**. Chroma is preferred for this tutorial architecture because it can run locally within the application container without needing a separate server.27
4. **Retrieval:** A separate "Document Expert" tool is created. When the user asks about the PDF, the agent queries the vector store, retrieves the top $k$ relevant chunks, and synthesizes an answer.26

## 8. Advanced Orchestration: Moving to LangGraph

As the bot grows in complexity—handling visualization, search, data analysis, and document retrieval—a linear "Chain" or simple "Agent" becomes brittle. It creates a "spaghetti code" of tools where the LLM gets confused about which tool to use.

**LangGraph** solves this by modeling the agent as a state machine.28

* **Nodes as Specialists:** Instead of one massive agent, we define specialized nodes:
  + Researcher: Uses Tavily to find news.
  + DataAnalyst: Uses yfinance and Pandas to crunch numbers.
  + Chartist: Uses Matplotlib to generate visuals.
* **The Supervisor Pattern:** A "Supervisor" LLM node routes the user's query to the correct specialist. If the DataAnalyst finds missing data, it can loop back to the Researcher to find it. This **Cyclic Graph** capability is impossible in standard chains but essential for robust financial workflows where the first attempt at analysis often requires refinement.9

## 9. Deployment Strategy: Streamlit Community Cloud

Deploying the application makes it accessible to stakeholders. Streamlit Community Cloud offers a seamless deployment experience for GitHub repositories, but it imposes specific constraints.1

### 9.1 Dependency Management

The requirements.txt file is the blueprint for the build environment. It must explicitly pin versions to avoid conflict, especially between langchain, pandas, and numpy.

* *Critical:* yfinance, streamlit, langchain, langchain-experimental (for the pandas agent), openai, tiktoken, matplotlib, plotly, tabulate (for dataframe printing).1

### 9.2 Secrets Management

API keys (OpenAI, Tavily) must never be committed to GitHub.

* **Local Development:** Keys are stored in .streamlit/secrets.toml.
* **Cloud Deployment:** Keys are entered into the Streamlit Community Cloud dashboard's "Secrets" area. The application code accesses them uniformly via st.secrets (e.g., os.environ = st.secrets).32

### 9.3 Resource Constraints and Optimization

The Free Tier typically provides limited RAM (often ~1GB).

* **Implication:** Loading a massive vector store or a large historical dataset into memory will crash the app.
* **Optimization:** Use @st.cache\_data decorators on data fetching functions. This tells Streamlit to cache the result of yfinance calls. If the user changes a filter but the data remains the same, the app retrieves it from the cache rather than hitting the API or consuming memory to rebuild the object. This is vital for performance and stability on the cloud tier.34

## 10. Conclusion

The architecture of a modern financial analysis chatbot extends far beyond simple text generation. It is a sophisticated orchestration of **reactive UI design**, **data engineering**, **vector search**, and **agentic reasoning**. By leveraging **LangChain** for the cognitive architecture, **yfinance** for data democratization, and **Streamlit** for the interactive layer, developers can build powerful tools that rival professional financial terminals.

However, success lies in the details: the rigorous sandboxing of Python execution, the careful management of conversational state, the strategic use of structured tool calling over generic prompting, and the seamless integration of multi-modal outputs like dynamic charts. As the ecosystem evolves towards **LangGraph** and multi-agent systems, these chatbots will cease to be mere assistants and will become autonomous analysts capable of generating deep, actionable market intelligence. The blueprint provided here serves as the robust foundation for this future.

#### Works cited

1. LLM-powered Chatbots for Financial Queries - Packt, accessed on January 15, 2026, <https://www.packtpub.com/en-us/learning/how-to-tutorials/llm-powered-chatbots-for-financial-queries>
2. Unleashing Power of AI Analytics with LangChain and Azure ..., accessed on January 15, 2026, <https://techcommunity.microsoft.com/blog/educatordeveloperblog/democratizing-data-unleashing-power-of-ai-analytics-with-langchain-and-azure-ope/4061915>
3. Agents - Docs by LangChain, accessed on January 15, 2026, <https://docs.langchain.com/oss/python/langchain/agents>
4. Introduction to Custom and Built-in Python LangChain Agents, accessed on January 15, 2026, <https://wellsr.com/python/working-with-python-langchain-agents/>
5. LangChain: Pandas DataFrame Agent Guide - Kaggle, accessed on January 15, 2026, <https://www.kaggle.com/code/ksmooi/langchain-pandas-dataframe-agent-guide>
6. Chat Application using Streamlit and Text Bison | by Biju Kunjummen, accessed on January 15, 2026, <https://bijukunjummen.medium.com/chat-application-using-streamlit-and-text-bison-05024f939827>
7. Stock Financial Analysis with Streamlit and YFinance API - alphasec, accessed on January 15, 2026, <https://alphasec.io/stock-financial-analysis-with-streamlit-and-yfinance-api/>
8. Build a basic LLM chat app - Streamlit Docs, accessed on January 15, 2026, <https://docs.streamlit.io/develop/tutorials/chat-and-llm-apps/build-conversational-apps>
9. Part 1 — Building your first AI chatbot using LangGraph and Streamlit, accessed on January 15, 2026, <https://medium.com/@ratneshyadav_26063/building-your-first-ai-chatbot-using-langgraph-and-streamlit-4b2a9f61ed87>
10. Showing Plotly Graphs generated by CSV agent by Langchain ..., accessed on January 15, 2026, <https://discuss.streamlit.io/t/showing-plotly-graphs-generated-by-csv-agent-by-langchain-inside-streamlit-ui/47296>
11. Pandas Dataframe - Docs by LangChain, accessed on January 15, 2026, <https://docs.langchain.com/oss/python/integrations/tools/pandas>
12. Revolutionizing Data Interaction: Harnessing LangChain's Pandas ..., accessed on January 15, 2026, <https://medium.com/@alsinanpk/unleashing-the-power-of-data-a-new-era-of-interaction-with-data-using-langchains-pandas-agent-4826c92d3bef>
13. Building a Secure Sandbox for LangChain's ... - Medium, accessed on January 15, 2026, <https://medium.com/@yeqing_40735/building-a-secure-sandbox-for-langchains-create-pandas-dataframe-agent-using-docker-61c347aaec22>
14. Unable to display plots in a chat with CSV app - Using Streamlit, accessed on January 15, 2026, <https://discuss.streamlit.io/t/unable-to-display-plots-in-a-chat-with-csv-app/50866>
15. LangChain: Data Analysis with REPL-Tool and LLM - Kaggle, accessed on January 15, 2026, <https://www.kaggle.com/code/ksmooi/langchain-data-analysis-with-repl-tool-and-llm>
16. how to save plots/graphs when using PythonREPLTool in langchain ..., accessed on January 15, 2026, <https://github.com/langchain-ai/langchain/issues/16067>
17. Display PythonREPL graphics with Streamlit, accessed on January 15, 2026, <https://discuss.streamlit.io/t/display-pythonrepl-graphics-with-streamlit/73093>
18. Chat with and visualise your data - Part II, accessed on January 15, 2026, <https://giasemidis.github.io/2023/09/30/chatgpt-chatbot-and-plot-own-data.html>
19. Building a basic LLM chat app with Streamlit chat element functions ..., accessed on January 15, 2026, <https://medium.com/@onnyunhui/building-a-basic-llm-chat-app-with-streamlit-chat-element-functions-using-only-google-colab-70ab2ce05142>
20. LangChain Streamlit, accessed on January 15, 2026, <https://blog.langchain.com/langchain-streamlit/>
21. 5 Tavily Alternatives for Better Pricing, Performance, and Extraction ..., accessed on January 15, 2026, <https://www.firecrawl.dev/blog/tavily-alternatives>
22. Tavily Search - Docs by LangChain, accessed on January 15, 2026, <https://docs.langchain.com/oss/python/integrations/tools/tavily_search>
23. tavily-ai/langchain-tavily - GitHub, accessed on January 15, 2026, <https://github.com/tavily-ai/langchain-tavily>
24. How to Add Real-Time Web Search to Your LLM Using Tavily, accessed on January 15, 2026, <https://www.freecodecamp.org/news/how-to-add-real-time-web-search-to-your-llm-using-tavily/>
25. RAG Project: Build an AI Onboarding Chatbot with Streamlit ..., accessed on January 15, 2026, <https://www.youtube.com/watch?v=WUUujm1MRQg>
26. DocuMentor: Build a RAG Chatbot with Ollama, Chroma & Streamlit, accessed on January 15, 2026, <https://akanshasaxena.com/post/documentor-rag-app/>
27. Modular Design with LangChain, Streamlit and ChromaDB, accessed on January 15, 2026, <https://dev.to/zlash65/refactoring-rag-pdfbot-modular-design-with-langchain-streamlit-and-chromadb-41fn>
28. Quickstart - Docs by LangChain, accessed on January 15, 2026, <https://docs.langchain.com/oss/python/langgraph/quickstart>
29. Smarter Model Tuning: An AI Agent with LangGraph + Streamlit That ..., accessed on January 15, 2026, <https://towardsdatascience.com/smarter-model-tuning-an-ai-agent-with-langgraph-streamlit-that-boosts-ml-performance/>
30. How to Deploy Your App to Streamlit Community Cloud - YouTube, accessed on January 15, 2026, <https://www.youtube.com/watch?v=HKoOBiAaHGg>
31. Build DeepSeek Financial Analyst with Streamlit and Langchain, accessed on January 15, 2026, <https://www.youtube.com/watch?v=yJ9RyNleS3U>
32. stenio123/langchain-streamlit-chatbot - GitHub, accessed on January 15, 2026, <https://github.com/stenio123/langchain-streamlit-chatbot>
33. Project Tutorial: Build a Web Interface for Your Chatbot ... - Dataquest, accessed on January 15, 2026, <https://www.dataquest.io/blog/build-a-web-interface-for-your-chatbot-with-streamlit/>
34. Creating AI assistant for data analysis using LangChain and Streamlit, accessed on January 15, 2026, <https://medium.com/@sawantmitali22/creating-ai-assistant-for-data-analysis-using-langchain-and-streamlit-ec8986043b6b>
35. Streamlit free tier max users - Community Cloud, accessed on January 15, 2026, <https://discuss.streamlit.io/t/streamlit-free-tier-max-users/23474>