Research and Planning Report

94-815 Agent Based Modelling and Agentic Technology

Team 8 - Fanxing Bu, Ivan Wiryadi

Reflection Essay

Tool Selection Trade-offs

To prototype our agent-based financial portal effectively within the constraints of time, budget, and scope, we carefully selected tools across six technical pillars: OCR, language reasoning, agent orchestration, memory/persistence, financial data access, and frontend interface. Our guiding principles were: **maximize learning**, **enable modular iteration**, and **stay within a manageable stack**.

OCR and Information Extraction

We chose **Mistral VLM** over traditional OCR + KIE pipelines (like Tesseract or PaddleOCR) due to its ability to directly extract structured entities from receipts in a single forward pass. This decision reduced our engineering complexity and increased robustness to unstructured layouts.

> **Trade-off**: While we sacrificed control over individual text spans or fine-grained bounding boxes, the one-shot multimodal interface gave us speed and resilience—ideal for a proof-of-concept.

Language Model Reasoning

We adopted a hybrid LLM strategy by combining **GPT-4** and **Mistral**, leveraging their respective strengths in different task contexts. **GPT-4** provides superior reasoning and summarization capabilities, making it ideal for high-stakes tasks such as multi-agent coordination and natural language financial reporting. While GPT-4 incurs usage costs, it remains reasonably priced for targeted use.

In contrast, **Mistral** is free and well-suited for routine, lightweight tasks such as classification, metadata extraction, or fallback interactions where perfect output is not critical. By selectively invoking different LLM APIs based on task complexity, we were able to **balance performance with cost efficiency** in a flexible and modular way.

> **Trade-off**: While GPT-4 delivers superior reasoning quality at a moderate cost, Mistral provides a cost-free alternative with more limited capabilities. Our selective application of each model according to task complexity allowed us to effectively balance performance with budget considerations.

Agent Orchestration Framework

We selected **LangChain** for its mature abstractions around tool use, memory, and prompt composition. Its `AgentExecutor`, `ChatPromptTemplate`, and memory modules gave us a solid foundation to implement multi-agent workflows quickly.

> **Trade-off**: LangChain's high-level design accelerated development but sometimes obscured internals, making low-level debugging or runtime planning adjustments less flexible than Autogen.

Memory and Persistence

We used **SQLite** combined with **LangChain memory modules** to balance structured data storage with short-term context sharing across agents. This pairing allowed us to build personalized, memory-aware agents without managing a full database server.

> **Trade-off**: SQLite met our needs for logging and retrieval, but limited advanced query capabilities (e.g., fuzzy matching, temporal joins) that would have been useful for more intelligent planning.

Financial Data and News Integration

Our system used the **Yahoo Finance API** for market data and relied on **LLM browsing capabilities** (e.g., GPT-4 with browsing) for financial news summarization. This two-tiered method ensured both factual correctness and semantic richness.

> **Trade-off**: While we gained versatility by combining structured APIs with real-time LLM summarization, we acknowledged the limitations of hallucinations in LLM outputs and the lack of deterministic retrieval.

Frontend Interface and User Experience

For the frontend, we opted for a **Streamlit-based UI**. Streamlit offered a low-code yet highly expressive Python-native interface, allowing us to build interactive dashboards with buttons, charts, and real-time chat-like interaction—without needing to manage JavaScript or complex routing.

> **Trade-off**: While Streamlit limited our control over layout customization and animations compared to React-based frontends, it drastically reduced development time and enabled our team to focus on agent logic instead of frontend plumbing.

We also explored **Streamlit components** to render OCR-ed receipt images, show tabled financial summaries, and trigger agent workflows (e.g., "extract receipt", "generate monthly report", "query investment plan") with a single click.

This choice aligned with our goal of **rapid prototyping**, allowing all team members—including those without frontend experience—to contribute to user-facing features.

> **Reflection**: In a production system, we would consider migrating to a more scalable framework (e.g., React + Flask API backend), but for this academic setting, Streamlit gave us the perfect balance of speed and clarity.

Summary

Each tool choice reflected a balance between **capability**, **simplicity**, and **learning value**. While some advanced features (e.g., vector search, full orchestration with Autogen or CrewAI) were out of scope, our stack enabled us to build a functioning multi-agent system that demonstrated key agentic patterns and supported real user-facing use cases.

Ethical Considerations

There are several ethical considerations that immediately come to mind (although there could be more):

Automation Bias

Users may develop excessive trust in AI-generated financial insights the agent. Therefore, the system should implement contextual disclaimers and encourage verification with qualified professionals.

Data Privacy

Financial data reveals sensitive personal information about spending habits, income, and merchant relationships. Sticklet introduces multiple exposure points through:

- Receipt processing via Mistral's OCR
- Queries handled by OpenAI models
- Transaction storage in unsecured SQLite database

Each data processing should be properly vetted and provide sufficient controls.

Security Vulnerabilities

The RAG pattern implemented in Sticklet creates potential attack vectors, particularly through the `SQLQueryTool` which could be vulnerable to SQL injection without proper input sanitization. Validation must be performed.

Moreover, there are inherent risks with Language Models based input and output, such as recent cases in Jailbreaking, harmful outputs, etc. Sufficient guardrails and controls should be put in place.

Accuracy and Responsibility

While the Agent aims to improve data quality through self-reflection and human-reflection, there still can be mistakes. Moreover, the nature of Language Models output may not be necessarily true (hallucinating). Care must be taken into account to make sure the user is aware of such risks.

Lesson Learnt

Our experience building Sticklet—a multi-agent, AI-powered financial portal—revealed several important lessons about aligning product thinking with system architecture and implementation.

1. Product clarity is the foundation of good architecture

We began by clearly defining our business scenario: intelligent receipt management and financial insight generation. This helped us articulate concrete user needs—structured data extraction, monthly summaries, and Q&A interfaces—which in turn guided our technical direction. Starting with a problem-oriented perspective rather than a model-centric mindset helped us stay focused on user value rather than model capabilities.

2. Design technical architecture based on product scope

Once our product goals were established, we identified the minimum technical requirements to support them—OCR, reasoning, memory, and user interaction. This led to a focused architecture using Mistral for OCR, GPT-4 and Mistral for language reasoning, LangChain for agent orchestration, SQLite for persistence, and Streamlit for frontend development. This stack offered a pragmatic balance between simplicity and capability, enabling rapid prototyping without overengineering.

3. Build the minimum viable system before scaling

We implemented a minimal functional pipeline—OCR to Coordinator Agent to SQL and LLM to UI—as early as possible. The Coordinator Agent, implemented using LangChain, handled task orchestration, memory lookup, and tool invocation. This architecture allowed us to validate the system workflow quickly and begin iterative development.

4. Scale through modularity and gradual extension

Once the core system was validated, we incrementally introduced new components to extend functionality. These included multi-model dispatching logic (e.g., routing complex reasoning to GPT-4), memory-aware interaction, self-reflection patterns, and specialized agents such as Reader Agent, Reporter Agent, and Market Agent. Our modular design enabled us to add features without destabilizing the overall system.

5. Plan for scalability and extensibility early

We found that early attention to system modularity and separation of concerns greatly improved long-term extensibility. Designing agents with reusable prompts, generic tool interfaces, and clear responsibilities allowed us to incorporate new tools, data sources, and workflows with minimal friction.

6. API and framework choices are strategic

Tool and API selection had a significant impact on development speed, system flexibility, and cost control. LangChain facilitated fast development through built-in memory and agent abstractions, although its high-level design occasionally limited customizability. GPT-4 provided high-quality outputs for complex tasks, while Mistral offered a zero-cost alternative suitable for simpler operations. These decisions, while often made early, directly affected system performance, maintainability, and experimentation flexibility.

Conclusion

In summary, the development of Sticklet demonstrated that building effective multi-agent systems requires more than assembling tools and models. A successful architecture must be grounded in product clarity, designed for modularity, and implemented with strategic consideration of trade-offs. Scalability, extensibility, and iterative validation were essential principles that enabled us to deliver a system that was both functional and adaptable to future needs.