Name:Wu Sitong Student ID:1155253147

Agentic AI for Business and FinTech (FTEC5660) HW1

1. Task Objective

The goal of this homework is to build a receipt agent that takes multiple receipt images plus one user query, and returns a single numeric answer. The system only needs to support two queries:

- **Query 1:** Total amount actually paid across all receipts (after discounts)
- **Query 2:** Total amount that would have been paid without discounts

It also must reject irrelevant / out-of-domain questions.

2. Approach & Architecture

I built the solution as a simple but reliable map-reduce style pipeline. Instead of asking the model to read all receipts at once (which in my testing sometimes misses a receipt or mixes up totals), I process receipts one by one and only use Python to combine results.

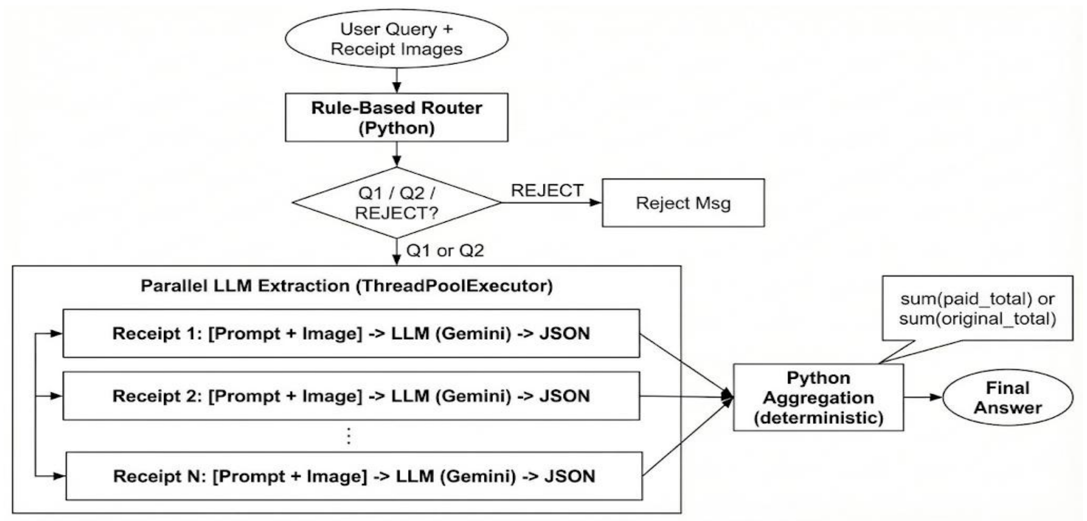
Workflow:

- Router (Python): classify the question into Q1 (total paid after discounts), Q2 (total before discounts), or REJECT for unrelated questions.

- Map phase (LLM extraction, parallel): for each receipt image, the model extracts two numbers in a fixed JSON schema: paid_total and original_total. These per-receipt calls run concurrently using ThreadPoolExecutor.

- Reduce phase (Python, deterministic): sum the relevant field across receipts and return a single float (no LLM arithmetic).

I also implemented a small interactive CLI with a session cache: totals are extracted once and stored, so repeated questions can be answered instantly without re-running extraction.Figure 1: Pipeline Architecture


User Query + Receipt Images
Rule-Based Router (Python)
Q1 / Q2 / REJECT?
REJECT
Reject Msg
Q1 or Q2
Parallel LLM Extraction (ThreadPoolExecutor)
Receipt 1: [Prompt + Image] -> LLM (Gemini) -> JSON
Receipt 2: [Prompt + Image] -> LLM (Gemini) -> JSON
Receipt N: [Prompt + Image] -> LLM (Gemini) -> JSON
sum(paid_total) or sum(original_total)
Python Aggregation (deterministic)
Final Answer

3. Methodological Details

The first step is intent routing. I used a simple rule-based router with conservative keyword patterns: if the query mentions "without/before discount(s)", route to Q2; if it clearly asks about total spent/paid, route to Q1; otherwise return REJECT. The main reason for this approach is that rule-based routing is fast and predictable, and it directly addresses the requirement to reject irrelevant queries. Here's the basic logic:

```python
def route_intent(question: str) -> str:
    q = question.strip().lower()

    # Q2: without/before discount
    if (("without" in q or "before" in q) and
        ("discount" in q or "discounts" in q)):
        return "Q2"

    # Q1: total spent/paid
    if (("spend" in q or "paid" in q or "pay" in q) and "total" in q):
        return "Q1"

    return "REJECT"
```

For the extraction part, each receipt image gets sent to the model with a prompt that forces JSON output in a fixed schema: {"paid_total": <float>, "original_total": <float>}. The paid_total is the final amount actually paid after discounts, and original_total is what it would have been before discounts. I included a concrete example in the prompt showing how to infer original_total by adding back discount line items when the receipt doesn't explicitly show this number. The model I'm using is Gemini 2.5 Flash accessed through AIHubMix API (via LangChain's langchain_openai package), with the API key loaded from an environment variable.

Since receipts are independent, I process them all concurrently using Python's ThreadPoolExecutor with max_workers=5. Each receipt gets one extraction task, and results are collected after everything completes. This cuts down latency a lot compared to processing receipts one by one, especially when handling 7+ receipts.

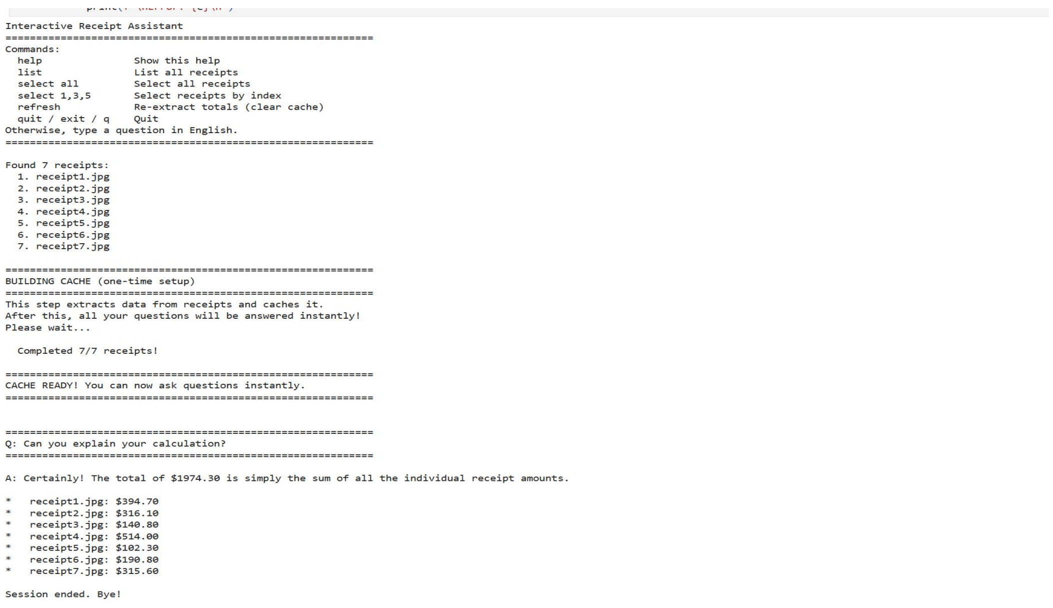
For robustness, I added some basic error handling because the model sometimes returns extra text like code fences or small formatting issues. My approach is to extract the first {...} block from the model output using regex and parse that. If parsing fails, I retry once with a simplified prompt that focuses on lines like TOTAL / SUBTOTAL / DISCOUNT. I consider this a lightweight repair mechanism rather than a heavy reflection system, and it works well enough for this assignment's scope.

After extraction, the final answer is computed deterministically in Python: if intent is Q1, sum all paid_total values; if Q2, sum all original_total values; then round to 2 decimals. The key point here is that the cross-receipt arithmetic happens in code, not in natural language, so I can guarantee correctness. The model is mainly used for per-receipt numeric extraction and inference from the image content.

4. Performance & Caching (Evidence)

I also implemented a small interactive CLI to make manual testing easier. The main engineering point here is the session cache. On the first run, the system performs a one-time extraction for the selected receipts ("BUILDING CACHE"). After that, repeated questions reuse the cached per-receipt totals, so the system can respond without re-calling the model.

CLI Output Demonstration:



```
Interactive Receipt Assistant
============================================================
Commands:
  help              Show this help
  list              List all receipts
  select all        Select all receipts
  select 1,3,5      Select receipts by index
  refresh           Re-extract totals (clear cache)
  quit / exit / q   Quit
Otherwise, type a question in English.
============================================================

Found 7 receipts:
  1. receipt1.jpg
  2. receipt2.jpg
  3. receipt3.jpg
  4. receipt4.jpg
  5. receipt5.jpg
  6. receipt6.jpg
  7. receipt7.jpg

============================================================
BUILDING CACHE (one-time setup)
============================================================
This step extracts data from receipts and caches it.
After this, all your questions will be answered instantly!
Please wait...

  Completed 7/7 receipts!

============================================================
CACHE READY! You can now ask questions instantly.
============================================================


============================================================
Q: Can you explain your calculation?
============================================================

A: Certainly! The total of $1974.30 is simply the sum of all the individual receipt amounts.

*   receipt1.jpg: $394.70
*   receipt2.jpg: $316.10
*   receipt3.jpg: $140.80
*   receipt4.jpg: $514.00
*   receipt5.jpg: $102.30
*   receipt6.jpg: $190.80
*   receipt7.jpg: $315.60

Session ended. Bye!
```

Observations:

1. The cache is built once per receipt set (initial extraction is the slow step).
2. After caching, repeated queries return quickly since no re-extraction is needed.
3. The CLI can display a receipt-level breakdown, which helps verify the final sum.
4. The CLI also supports simple follow-up questions (e.g., asking for an explanation), using the cached breakdown.

5. Alignment with Assignment Requirements

Requirement	How my system handles it
Q1: total spent after discounts	Route to Q1 and return sum(paid_total) across all receipts
Q2: total without discounts	Route to Q2 and return sum(original_total) across all receipts
Reject irrelevant queries	Router returns REJECT for queries outside Q1/Q2 domain

Requirement	How my system handles it
Output format	Always returns a single float answer (or a short reject message)
Multimodal input	Uses Gemini 2.5 Flash to process receipt images with text prompts
Code quality	Modular functions, error handling, parallel processing

6. Limitations & Reflection

This approach works well for the task, but there are still some limitations worth mentioning. Different stores print totals in different places, and some receipts may not explicitly show an "original total" or itemized discounts. In those cases, original_total depends on the model's inference from available discount cues. Some promotions like buy-one-get-one or multi-item bundles can also be hard to interpret consistently, especially when the receipt doesn't clearly separate per-item discounts from total savings.

Image quality is another issue. Blurred, rotated, or low-resolution images can cause extraction errors that propagate to the final sum. The current system doesn't have automatic image preprocessing like rotation correction or contrast enhancement, so it relies on the images being reasonably clear.

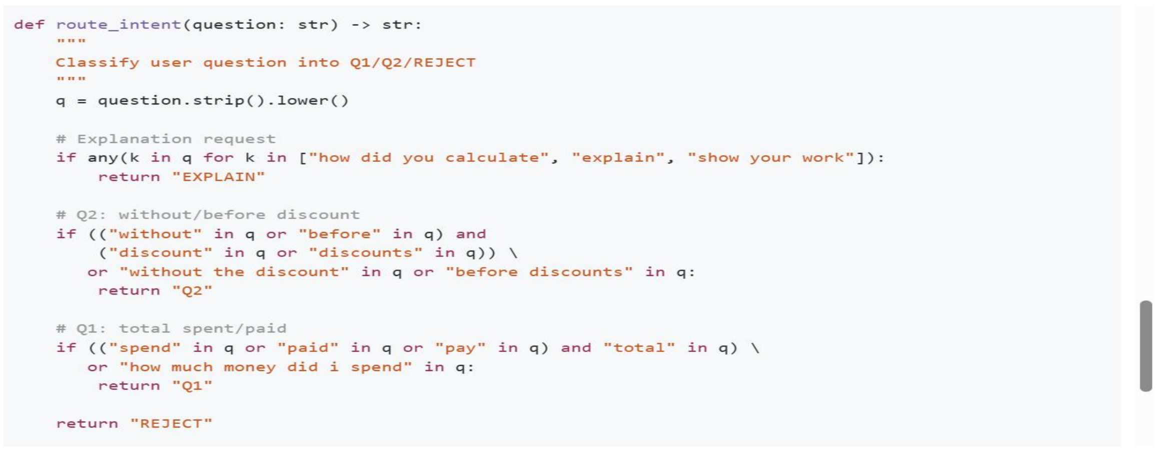
One thing my implementation doesn't have is explicit consistency checks. Unlike some approaches that verify original_total = paid_total + discounts mathematically, my system directly trusts the model's extraction of both values based on the prompt instructions. This was a design trade-off for simplicity and speed, but it means inconsistent receipts might not be caught automatically.

If I had more time, I would add confidence scoring so the model returns a confidence level per receipt and flags low-confidence extractions for manual review. I'd also add basic consistency checks like verifying original_total ≥ paid_total and triggering a second verification prompt only for suspicious receipts. Image preprocessing and multi-model validation would be nice additions too.

7. Code Implementation Examples (Optional)

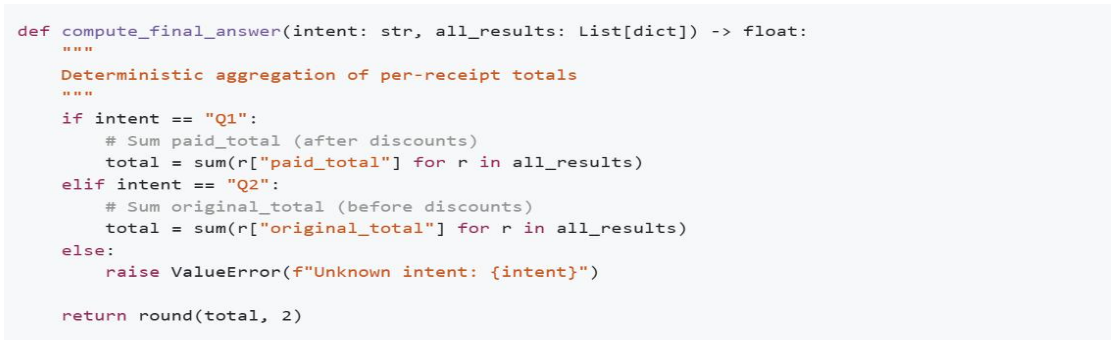
To keep the report short, I only include the two most important snippets that directly relate to the grading criteria: intent routing (Q1/Q2/REJECT) and deterministic aggregation in Python. The remaining components (LLM extraction, parallel execution, and session cache) follow standard patterns and are implemented in the notebook.

Intent Router (Q1 / Q2 / REJECT):



```python
def route_intent(question: str) -> str:
    """
    Classify user question into Q1/Q2/REJECT
    """
    q = question.strip().lower()

    # Explanation request
    if any(k in q for k in ["how did you calculate", "explain", "show your work"]):
        return "EXPLAIN"

    # Q2: without/before discount
    if (("without" in q or "before" in q) and
        ("discount" in q or "discounts" in q)) \
       or "without the discount" in q or "before discounts" in q:
        return "Q2"

    # Q1: total spent/paid
    if (("spend" in q or "paid" in q or "pay" in q) and "total" in q) \
       or "how much money did i spend" in q:
        return "Q1"

    return "REJECT"
```

Deterministic Aggregation (Python, not LLM):



```python
def compute_final_answer(intent: str, all_results: List[dict]) -> float:
    """
    Deterministic aggregation of per-receipt totals
    """
    if intent == "Q1":
        # Sum paid_total (after discounts)
        total = sum(r["paid_total"] for r in all_results)
    elif intent == "Q2":
        # Sum original_total (before discounts)
        total = sum(r["original_total"] for r in all_results)
    else:
        raise ValueError(f"Unknown intent: {intent}")

    return round(total, 2)
```

8. Conclusion

Overall, my receipt agent is designed to be simple, fast, and reasonably robust for this homework. The key idea is to use a strict rule-based router (Q1/Q2/REJECT), extract per-receipt totals in a fixed JSON schema, run extraction in parallel to reduce latency, and compute the final cross-receipt sum deterministically in Python (instead of relying on the LLM to do multi-receipt arithmetic).

As a quick sanity check using the provided local test_query function (±$2 tolerance), my system matched the expected answers:

- Q1 (total paid): expected 1,974.30 → output 1,974.30 (PASS)
- Q2 (before discounts): expected 2,348.20 → output 2,348.20 (PASS)

There are still limitations due to receipt layout differences and image quality, especially for inferring original_total when it is not explicitly printed. If I had more time, I would add lightweight validation checks (e.g., original_total ≥ paid_total) and selectively re-run extraction on suspicious receipts.