Project Documentation - Multi-Step Reasoning Agent with Self-Checking

1) What the project is solving (and constraints)

This project implements the required 3-phase "Planner → Executor → Verifier" reasoning loop that solves structured word problems, retries on failure, and returns a **single JSON** response with answer, status, reasoning_visible_to_user, and debugging metadata (plan, checks, retries).

A key constraint from the assignment is that long chain-of-thought must not be shown directly to the user (only a short explanation), while still keeping internal logs for debugging.

2) Why the prompts are designed this way

A) Planner prompt (PLANNER_PROMPT)

Goal: produce a short, numbered plan without solving.

Why this design:

- **Separation of concerns:** The planner's job is only to outline steps (parse → extract → compute → sanity checks → format). This matches the assignment requirement that the planner outputs a concise plan.
- **Reduces hallucinated math early:** If the planner starts "solving," it tends to introduce wrong intermediate values that the executor then anchors on. Keeping it "plan only" helps the executor remain grounded in the question.
- **Keeps metadata useful but compact:** Since metadata.plan is mainly for debugging/evaluation, a concise plan is easier to inspect than a verbose pseudo-solution.

B) Planner-with-feedback prompt (PLANNER_WITH_FEEDBACK_PROMPT)

Goal: when verification fails, generate a *new* plan that explicitly adapts to verifier feedback.

Why this design:

- **Closed-loop self-correction:** The verifier's feedback becomes the "training signal" for the next attempt, satisfying the retry requirement when verification fails.
- **Explicit variables {question} and {feedback}:** You fixed a common reliability issue: earlier versions often forgot to actually inject these values, causing the planner to ignore what went wrong. Using .format(question=..., feedback=...) ensures the replanning step is correctly conditioned.

C) Executor prompt (EXECUTOR_PROMPT)

Goal: execute the plan and produce a solution text that includes intermediate reasoning for internal use, plus a deterministic final answer marker.

Why this design:

- **"FINAL ANSWER:" marker enables deterministic extraction:** Your agent uses a regex (FINAL ANSWER:\s*(.*)) so that the final answer is not dependent on another model call. This reduces "answer drift" from the formatter.

- **Hard rules for time arithmetic (minutes-from-midnight):** Time-difference questions are a classic failure mode (borrowing errors). By mandating conversion to minutes first, you constrain the executor to a safer procedure (and you also repeat it in the verifier prompt to cross-check).
- **Controlled output contract:** The executor is free to reason, but the system still gets one canonical "final answer" string for downstream formatting.

This aligns with the assignment's expectation that the executor produces intermediate calculations/deductions, but the user shouldn't see raw chain-of-thought.

D) Verifier prompt (VERIFIER_PROMPT)

Goal: independently re-solve and validate constraints, then return strict JSON:
{ passed: boolean, feedback: "...", corrected_answer: "..." | null }

Why this design:

- **Independent re-solve is the strongest self-check:** It meets the requirement of "re-solve independently and compare" and/or validate constraints.
- **Strict JSON output enables programmatic gating:** The agent loop can reliably decide whether to retry or accept based on passed, instead of fuzzy text interpretation.
- **corrected_answer is a practical recovery hook:** If the executor's final marker is missing or malformed, the verifier can still provide the correct short answer as a fallback.

E) Final formatter prompt (FINAL_FORMATTER_PROMPT)

Goal: produce user-safe output without revealing step-by-step reasoning.

Why this design:

- **Explicit "FORBIDDEN" rules** ("don't show Step 1", "don't show raw calculations") are direct guardrails for the user-facing reasoning_visible_to_user. This implements the assignment's "no chain-of-thought to the user" constraint.
- **JSON-only output:** avoids UI parsing ambiguity in the Streamlit app.

F) Failure summarizer prompt (FAILURE_SUMMARIZER_PROMPT)

Goal: if retries are exhausted, explain failure in a user-friendly way.

Why this design:

- **Prevents confusing raw verifier logs:** Users get a clean explanation, while engineers still have checks_log in metadata.
- **Matches the required failure behavior:** mark status as failed and explain why.

3) Common pitfalls + how my code addresses them

1) Relying on the formatter to "extract" the answer

What went wrong: If the formatter is asked to "extract the final answer," some models will paraphrase, "improve," or even change it-especially when the executor text is long.

My mitigation:

- **Regex extraction first** (FINAL ANSWER:) is the most deterministic path.
- **Verifier corrected answer as fallback** if regex fails.
- **Formatter answer only as a last fallback**.

2) Verifier returning invalid JSON

What went wrong: Even in JSON mode, smaller/local models may wrap JSON in code fences or add commentary.

My mitigation:

- _clean_json() strips ```json fences.
- If parsing fails, I mark verification as failed (Verifier JSON Error) which triggers retry.

3) Prompt variables not being injected during retries

What went wrong: The replanning prompt originally risked treating {question} / {feedback} literally, so the model never got the actual context of the failure.

My mitigation:

- _plan() uses .format(question=..., feedback=...) so the replanner always sees the real question + last failure feedback.

4) Time arithmetic mistakes persisting across attempts

What went wrong: Many models repeatedly make the same borrowing mistakes if you only say "double-check time."

My mitigation:

- I hard-coded a *procedure* (minutes-from-midnight) in both executor and verifier prompts, increasing the odds that at least one pass computes correctly.

5) Ambiguous logic questions

What went wrong: Questions like "John runs faster than Bob… who is slowest?" can be solved, but some ambiguous problems can't be verified cleanly if the model chooses an interpretation.

Current limitation: The verifier can still pass a solution that's coherent, but I don't yet have an explicit "ambiguous" status or a mechanism to ask a clarifying question. This shows up most on "tricky" tests (which the assignment explicitly expects).

4) What I would improve with more time

A) Stronger determinism with tool-based arithmetic

- Add a small **Python math/time evaluator** for structured extraction (numbers, times, intervals). The executor can still explain reasoning, but calculations become code-checked.
- For time-slot scheduling problems, compute overlaps and durations programmatically; let the LLM only handle parsing and explanation.

B) Better verifier: multi-check strategy

Instead of one verifier pass:

- **Check 1:** independent re-solve (current approach).
- **Check 2:** constraint validation (non-negative counts, totals match, duration fits slot, etc.).
- **Check 3:** unit consistency (minutes vs hours, apples vs dollars).
  Then aggregate into a single passed with richer feedback.

C) Add few-shot examples inside prompts (or expand tests)

The assignment encourages examples either in prompt or tests.

I already added a small test suite; I'd expand it with:

- time edge cases (crossing midnight, same start/end),
- slots exactly equal to meeting length,
- multi-step arithmetic with negatives and parentheses,
- ambiguous questions where the correct behavior is "cannot determine."

D) Handle ambiguity explicitly

Introduce:

- status: "failed" with a specific message like "Ambiguous question: multiple valid interpretations," **or**
- an "ask clarifying question" mode (if the interface allows).

E) Improve answer extraction reliability further

- Enforce executor output structure more strictly (e.g., require a final JSON block from executor) and validate it with a schema checker.
- Add a post-processor that normalizes whitespace and strips accidental trailing punctuation.

F) Observability + evaluation metrics

- Track pass rate per category (math/time/logic/slots).
- Save verifier feedback by category to see systematic failure patterns.
- Add a "diff" view when verifier disagrees with executor to speed debugging.