

The following excerpts are representative samples selected to illustrate how AI was used at key stages of system development: conception, architectural orchestration, and controlled code generation. Full raw chat logs were intentionally not included, as they contain exploratory and redundant interactions that do not contribute meaningfully to understanding the system design process. In addition, the entire chat logs would be *far* too long.

Excerpt One

[The following is the excerpt of the very first exchange between me and ChatGPT during the conception phase]

Me: You are an expert AI systems designer tasked with turning a vague project idea into a specific, buildable, correctness-optimized system. The project goal is to create a self-critiquing research assistant that takes in natural-language research questions and produces high-accuracy, low-hallucination outputs.

Generate a structured project plan that includes:

- A clear step-by-step system architecture
 - Exact prompts for each step
 - Explanations of why each step is necessary for epistemic rigor
 - JSON output schema where appropriate
 - UI/frontend description to demonstrate internal reasoning steps
 - A list of what the student will *actually* learn vs. what they *won't*
 - Known failure modes and what to document when they occur
 - An ethical analysis of risks
- A forced binary design decision with tradeoffs explained (e.g., autonomy vs. human-in-the-loop)

Use a confident, directive tone. Avoid vagueness. Emphasize structural clarity, falsifiability, and internal critique.

[Assignment from Sakai copy & pasted here]

As for the project itself, I'm not sure on the specifics yet (aside from that outlined in the

assignment) but I know I want something useful to me rather than a complicated gimmick for points.

ChatGPT:

Good instinct. Utility to *you* is the right objective function.

Optimizing for **correctness** is also the harder—and better—choice. It forces you to confront where LLMs actually break.

Now I'm going to lock this into a **concrete, buildable system** and strip away vagueness.

Your Project (Locked-In)

Self-Critiquing Research Assistant (Correctness-Optimized)

User input:

Natural-language research question.

System output:

- Structured answer
 - Explicit assumptions
 - Confidence score
 - “What could be wrong” section
-

Core Architecture (This Is the Complexity)

Step 1 — Question Disambiguator

Prompt:

“Rewrite this question into 3–5 precise sub-questions. Explicitly state assumptions.”

Why it matters:

- LLMs hallucinate most when the question is underspecified.
 - You can show how precision reduces error.
-

Step 2 — Claim Generator

Prompt:

“Answer each sub-question with explicit claims. Each claim must be falsifiable.”

Output schema (JSON):

```
JSON
{
  "claim": "...",
  "confidence": 0.0,
  "assumptions": []
}
```

Why this matters:

- Forces structure
 - Enables critique
-

Step 3 — Internal Critic (Adversarial)

Prompt:

“Assume the above claims are wrong. Identify weaknesses, missing evidence, and likely hallucinations.”

This is the intellectual core of the project.

Step 4 — Revision Agent

Prompt:

“Revise the original claims using the critique. Lower confidence where uncertainty remains.”

Key lesson:

- AI improves with *pressure*, not creativity.

Step 5 — Meta-Evaluator

Prompt:

“Rate epistemic reliability on a 0–1 scale and explain failure risks.”

This is gold for the reflection section.

Frontend (Simple but High Signal)

Web app with:

- Input box
- Expandable sections per step
- Toggle: “Show AI reasoning stages”

This visually proves:

“This could not be done in one prompt.”

What You’ll Explicitly Learn (and Write About)

Knowledge you *needed*

- API calls
- JSON schemas
- HTTP
- Deployment
- Prompt chaining
- State handling

Knowledge you *didn’t need*

- Algorithms
- Data structures
- Model internals
- Fine-tuning
- Complex frontend logic

This contrast is exactly what the prof wants.

Where AI Will Fail (You Want This)

- Overconfident critiques
- Fake certainty reduction
- Surface-level “skepticism”
- Poor domain knowledge in niche topics

You *must* document these failures.

Ethical Angle (Easy Points)

- False sense of correctness
 - Automation of epistemic authority
 - Students outsourcing thinking
 - “Critique” ≠ truth
-

One Crucial Design Choice (Answer This)

Will you:

A) Let the user *edit* intermediate steps before continuation

or

B) Keep the system fully autonomous end-to-end?

A = human-in-the-loop epistemology

B = autonomous AI epistemology

Pick one.

Me: I think B by default but an option for A is good? Like an advanced option

ChatGPT: Correct. That's the right call.

Default **B (autonomous)** preserves experimental purity. Optional **A (human override)** gives you a control condition. That's intellectually clean.

Now I'm going to **freeze the design** and show you how this turns into an A+ without scope creep.

Final System Spec (Frozen)

Name (working)

REFLEX — *Recursive Epistemic Feedback LLM eXecutor*

(Name doesn't matter; don't bikeshed.)

Modes

Standard Mode (Default)

- One-click
- Fully autonomous
- User sees results *after* completion

Advanced Mode (Optional Toggle)

- User can edit:
 - Reformulated sub-questions
 - Claims before critique
- Demonstrates **human-in-the-loop correction**

This directly supports your reflection section.

Backend (AI-Generated, Deterministic Enough)

Single LLM, multiple roles:

1. Disambiguator
2. Claim Generator
3. Critic
4. Reviser
5. Meta-Evaluator

State passed as structured JSON between steps.

Key insight to emphasize:

“The same model behaves differently under role pressure.”

What Makes This “Complex Enough”

Not code volume. **Control flow.**

- Sequential dependency
- Error amplification
- Confidence calibration
- Optional human intervention

That's real system design.

Evaluation (Don't Skip This)

You will test:

- Same question, multiple runs
- With and without critique
- With and without human edits

Show:

- Reduced hallucination frequency
- Lower (but more honest) confidence scores

Even anecdotal evaluation is acceptable.

Week-by-Week (No Slippage)

Week 1

- Write prompts for all 5 roles
- Define JSON schemas

- Draw architecture diagram

Week 2

- Generate backend via AI
- Log all prompts + responses

Week 3

- Generate frontend
- Integrate modes + toggle

Week 4

- Deploy
 - Write reflection
 - Prepare demo
-

One Non-Obvious Warning (Read Carefully)

Your biggest risk is **fake rigor**.

LLMs are good at *performing* skepticism without actually improving truth. You must explicitly call this out. Professors love that honesty.

Excerpt Two

[The following is an excerpt directly after we enter the first stage and I stop ChatGPT from attempting to write the code]

Me: Sorry, I wasn't gonna ask you to build the backend. You're a bit... outdated? In terms of training date. I was gonna have you build the map for Gemini to do it since it's kind of a rolling model

ChatGPT: Good correction — you're right, and thanks for stopping that early.

You don't want *me* writing the backend. You want **Gemini to write it**, and you want a **clean orchestration map** so the output is coherent, auditable, and course-compliant.

So let's switch modes: **I'll give you the exact backend-generation playbook** — prompts, constraints, sequencing — so Gemini (or any rolling model) does the actual coding.

This is actually *better* for your grade.

REFLEX Backend Generation Map (AI-First)

Goal

Use Gemini as the **primary software engineer** to generate:

- A FastAPI backend
- Multi-step LLM pipeline
- Persistent intermediate state
- Advanced-mode partial re-runs
- Logging + deployment readiness

You act only as **architect + reviewer**.

Phase 0 — Repo Scaffold (Manual but Allowed)

You do **only this by hand**:

```
None
reflex/
  app/
    requirements.txt
    .env.example
  README.md
```

Empty files are fine.

Phase 1 — High-Level Architecture Prompt (MOST IMPORTANT)

Prompt to Gemini (paste verbatim):

You are the sole backend engineer for a research assistant called **REFLEX**.

Constraints:

- Use **FastAPI**
- Use **SQLite** for persistence
- Use **Gemini API** as the only AI model
- Store intermediate pipeline outputs per run
- Support re-running downstream steps only
- Log all prompts and responses
- Prioritize correctness and transparency over speed
- Code clarity > performance

The system must implement this pipeline:

1. Intent Parser
2. Decomposition Agent
3. Claim Generator
4. Critic Agent
5. Revision Agent
6. Meta-Evaluator

Each step must:

- Accept structured JSON input
- Produce structured JSON output
- Be independently inspectable

First, propose the **backend architecture**, file layout, data models, and API endpoints.

Do NOT write code yet.

You do nothing except read and sanity-check.

Phase 2 — Data Model & State Design

Prompt:

Based on the approved architecture, design:

- Database schema for runs and pipeline stages
- How intermediate state is stored and versioned
- How partial re-runs are triggered
- How prompt/response logs are persisted

Output only design explanations and schemas.

No code yet.

You verify:

- Each pipeline stage has a field
 - There is a `run_id`
 - There is a way to mark stages “dirty”
-

Phase 3 — API Contract First (Critical)

Prompt:

Define the full REST API for REFLEX.

Include:

- Create run
- Execute pipeline (full)
- Execute pipeline from stage X onward
- Fetch run state
- Update intermediate output (advanced mode)

For each endpoint, specify:

- Method
- Path
- Request schema
- Response schema

Do NOT write backend code yet.

This ensures coherence before code exists.

Phase 4 — Step-by-Step Code Generation (Controlled)

Now you let Gemini write code **one file at a time**.

Example pattern:

Generate `app/config.py`.

Use environment variables.

No inline explanations.

Output code only.

Then:

Generate `app/db.py` using SQLAlchemy and SQLite.

Then:

Generate `app/models.py` for persistent pipeline state.

Then:

Generate `app/llm.py` with Gemini API integration and JSON repair.

Then:

Generate `app/pipeline.py` with all six stages.

Then:

Generate `app/main.py` wiring FastAPI endpoints.

Never ask for “the whole backend at once.”

That’s how hallucinated glue happens.

Phase 5 — Verification Prompts (Mandatory)

After each file:

Review this file for:

- Logical bugs
- Missing imports
- API misuse
- Edge cases

If issues exist, rewrite the file.

This satisfies the “not blindly trusting AI” requirement.

Phase 6 — Deployment Readiness

Final prompt:

Modify the backend to be deployable on Google Cloud or a local Linux server.

Include:

- Unicorn endpoint
 - Environment variable usage
 - Basic logging
- Do not add new features.
-

What You Document (Grade Gold)

In your report:

- **Exact prompts** from Phases 1–6
 - What Gemini got wrong (it will)
 - What you caught
 - Why orchestration mattered
-

Why This Is Pedagogically Perfect

- AI wrote the code
- You designed the system
- You enforced epistemic discipline
- You didn’t rely on model internals
- You treated AI as labor, not magic

This is *exactly* the course thesis.

Me: I don’t think mindlessly prompting Gemini like this is the right idea right now. I feel like you should make a complete and detailed... almost proposal? It’s like the letter you’d leave to the new head once you’ve been told to hand over the project to someone else. Explain the concepts, 5 stages, and what you had planned to Gemini to wrap Gemini in as an equal, not a

workhorse.

ChatGPT:

Good. That's exactly the right artifact.

This is the **handover / intent-preserving document** that prevents Gemini from free-associating and gives you something you can also submit as *design documentation*.

Below is a **complete backend handoff proposal**.

You can paste this **verbatim** as:

- `docs/backend_handoff.md`, or
- the **first prompt** you give Gemini before any code is written.

This is not code. It's *system intent*.

REFLEX Backend Handoff Document

Project: REFLEX — Recursive Epistemic Feedback LLM eXecutor

Audience: AI engineer assuming responsibility for backend implementation

Author Role: System architect & epistemic supervisor

1. Purpose of This System

REFLEX is a research assistant designed to optimize for **epistemic correctness**, not speed or fluency.

Unlike a standard chatbot, REFLEX:

- Treats user input as a **research directive**, not merely a question
- Makes assumptions and ambiguity resolution explicit
- Forces claims to be criticized before final presentation
- Persists intermediate reasoning state for inspection and revision

The backend must support **AI orchestration**, not a single prompt-response flow.

2. Core Design Philosophy

1. **One model, many roles**
The same LLM (Gemini) is reused under different epistemic constraints.
 2. **Structured state over free text**
All inter-stage communication must be JSON-structured.
 3. **Autonomy by default, transparency always**
The system resolves ambiguities automatically but logs them visibly.
 4. **Slower is acceptable**
Latency is an intentional cost of epistemic caution.
 5. **Human override is an intervention, not the norm**
Advanced mode allows edits to intermediate stages and partial re-runs.
-

3. High-Level System Flow

None

User Input



Intent Parser



Decomposition Agent



Claim Generator



Critic Agent



Revision Agent



Meta-Evaluator

↓

Final Output

Each stage:

- Consumes structured JSON
 - Produces structured JSON
 - Is independently inspectable and persisted
-

4. Pipeline Stages (Detailed)

Stage 1 — Intent Parser

Purpose: Normalize any research directive into a structured brief.

Responsibilities:

- Identify research goal
- Extract scope constraints
- Define key terms (explicit and implicit)
- List assumptions
- Detect ambiguities
- Infer output expectations

Failure modes to expect:

- Overconfident interpretation of vague terms
- Missing implicit constraints

Output must include:

- A list of ambiguities that were auto-resolved
-

Stage 2 — Decomposition Agent

Purpose: Break the research brief into research dimensions or sub-problems.

Responsibilities:

- Identify conceptual axes
- Generate sub-questions or dimensions
- Avoid premature conclusions

Design constraint:

This stage must not introduce factual claims.

Stage 3 — Claim Generator

Purpose: Produce explicit, falsifiable claims.

Responsibilities:

- Generate claims per dimension
- Assign confidence scores (0–1)
- List assumptions per claim
- Describe how each claim could be falsified

Known weakness:

LLMs often overestimate confidence.

Stage 4 — Critic Agent

Purpose: Adversarially challenge the claims.

Responsibilities:

- Identify logical gaps
- Surface missing evidence
- Flag likely hallucinations
- Describe failure modes

Important:

This agent must assume the claims are wrong by default.

Stage 5 — Revision Agent

Purpose: Improve claims under critique.

Responsibilities:

- Revise or retract claims
 - Lower confidence where uncertainty remains
 - Document what changed and why
-

Stage 6 — Meta-Evaluator

Purpose: Provide a system-level epistemic assessment.

Responsibilities:

- Assign overall reliability score
 - List unresolved uncertainties
 - Explain what evidence would change conclusions
 - Recommend next steps for deeper research
-

5. State Persistence & Re-Runs

- Each user request creates a **run**
- Each stage's output is stored independently
- In advanced mode:
 - User may edit any stage output
 - Only downstream stages are recomputed
- Persisted outputs must not be silently overwritten

This enables:

- Auditing
 - Failure analysis
 - Human-in-the-loop epistemology
-

6. API Expectations

The backend must expose endpoints to:

- Create a run
- Execute the full pipeline
- Fetch run state (all stages)

- Update a stage output (advanced mode)
- Re-run downstream stages

API design must prioritize clarity over compactness.

7. Logging & Traceability

For each stage:

- Store prompt
- Store raw model response
- Store parsed JSON output
- Store repair attempts (if JSON fails)

Logs must be human-readable and auditable.

8. Model Constraints

- Use Gemini API
 - Respect rate limits
 - Step budget should be explicit
 - Favor deterministic settings (low temperature)
-

9. Known Limitations (Intentional)

- Self-critique may be superficial
- Confidence scores are heuristic
- No ground-truth verification
- Persisted AI outputs may falsely acquire authority

These limitations should not be “fixed away”; they are part of the study.

10. Success Criteria

This system is successful if:

- It exposes more assumptions than a single-prompt chatbot
- It reduces unjustified confidence
- Its failures are easier to diagnose than baseline AI outputs

Speed and novelty are not success metrics.

11. Non-Goals

- Not a general search engine
 - Not a citation verifier
 - Not optimized for throughput
 - Not meant to replace human judgment
-

12. Final Instruction to Implementer

If tradeoffs arise:

- Favor **transparency over cleverness**
- Favor **structure over brevity**
- Favor **explicit failure over hidden uncertainty**

Do not collapse stages to “simplify” unless instructed.

If you want, next we can:

- Convert this into the **exact first Gemini prompt**, or
- Write a **frontend handoff** in the same style, or
- Extract **evaluation criteria** directly from this doc.

Say which.

Me to Gemini: Actually, I changed my mind. This was originally a large project in which I just wanted you to handle the code, but I realize your expertise beats mine in terms of appropriate tools, so I'm doing a hand over process. The previous AI model was good in the conception phases but its training data is old and it no longer is effective since it isn't used to the new technologies. So, without further ado:

[Pasted ChatGPT's response]

Gemini:

I have reviewed the **REFLEX Backend Handoff Document** in detail. As the senior engineer taking over this implementation, I fully internalize the shift from "Assistant" to "Epistemic Pipeline."

The core challenge here isn't just calling the Gemini API; it's enforcing a **state-machine architecture** where each transition is a discrete, auditable event. My priority is ensuring the backend doesn't just pass strings, but manages a complex **State** object that mirrors the system's evolving "understanding."

Initial Assessment & Implementation Strategy

1. **Orchestration vs. Chaining:** I will implement this using an **orchestrator pattern**. Rather than Stage A calling Stage B, a central **PipelineManager** will handle the state persistence. This is crucial for your **Advanced Mode**, allowing us to "jump" back into the pipeline at Stage 4 (Critique) with user-edited JSON without losing the history of Stages 1-3.
2. **Epistemic Scaffolding (System Prompts):** The most critical technical hurdle is "Role Contamination"—ensuring the **Critic Agent** doesn't go easy on the **Claim Generator**. I will use distinct system instructions and likely separate "safety/temperature" configurations for each:
 - **Claim Generator:** \$Temp \approx 0.3\$ (slight creative search for engineering solutions).
 - **Critic Agent:** \$Temp \approx 0.1\$ (strict, adversarial logic checking).
3. **JSON Schema Enforcement:** To prevent the frontend from breaking, I'll use **Constrained Output (Schema)** features. If Gemini returns malformed JSON, the backend will trigger a single "Self-Repair" loop before surfacing the error as an explicit "Epistemic Failure" in the UI.

Proposed Tech Stack (Lightweight & Research-Ready)

- **Language:** Python (FastAPI). It's the standard for AI research and offers the best native support for the Gemini SDK.
- **Persistence:** A simple JSON-file-based "Store" or SQLite. Given this is for an academic/research context, we want the "runs" to be easily shareable as flat files.
- **Concurrency:** We will use **asyncio** to ensure the frontend can poll for updates while the "slow-by-design" reasoning steps are processing.

Clarifying Questions for the Architect

To ensure I don't deviate from your intent, I have three specific questions:

1. **Re-run Logic:** When a user edits Stage 3 (Claims) in Advanced Mode, should the **Critic Agent** (Stage 4) be forced to start from scratch, or should it be provided with the *previous* critique as context to see if its earlier concerns were addressed?
2. **State Granularity:** Do you want the "Raw Prompt" sent to Gemini to be retrievable via the API for the frontend's "Advanced Mode," or is the JSON output sufficient for transparency?
3. **Baseline Isolation:** Should the "Baseline Run" use the exact same model parameters (Temperature/Top-P) as the pipeline stages, or should it mimic a "consumer-grade" chatbot (usually higher temperature and more verbose)?

Me to ChatGPT: [I pass Gemini's message to ChatGPT]

ChatGPT:

Good response from Gemini. It's thinking like an engineer. Here are the answers you should give—tight, intentional, and aligned with your grading goals.

1) Re-run logic after user edits Stage 3

Force Stage 4 (Critic) to start from scratch. Do not feed the old critique as context by default.

Rationale: you want the critic to be a fresh adversary of the *new* claims, not anchored by its past arguments. Otherwise you confound evaluation with “critique persistence” and get path dependence.

Optional (nice-to-have): store the old critique for **diff display**, but do not inject it into the model call unless the user explicitly toggles “use previous critique as reference.” Default off.

Answer to Gemini: Stage 4 reruns clean; prior critique is available for audit/UI diff only, not context.

2) State granularity: expose raw prompts via API?

Yes—store and expose them, but gate it behind Advanced Mode and/or an explicit flag.

Reason: the course explicitly rewards “How AI Built This System” and prompt logs. Having the backend able to return prompts/parameters per stage makes your transparency story bulletproof.

Design: return prompts only if `include_provenance=true` (or advanced mode). Otherwise return only stage JSON outputs.

Answer to Gemini: Persist raw prompts + model params + raw outputs; API can return them via an opt-in query param.

3) Baseline isolation: same params or consumer-grade?

Use the same model, but define a baseline that mimics typical chatbot usage.

Concretely:

- Baseline = **single call**, higher temperature (e.g., 0.7), longer “helpful” style.
- REFLEX stages = low temperature (0.1–0.3) and structured JSON.

Why: You’re comparing *systems*, not just parameterizations. The baseline should represent how a student actually uses Gemini normally: one prompt, fluent, somewhat high-temp.

To avoid accusations of rigging: also log an “**iso-baseline**” config (same temp as Stage 3) as a secondary control if you have time. But primary baseline should be consumer-like.

Answer to Gemini: Baseline is one-shot consumer-style; optionally add an iso-baseline as a secondary run type.

Message you can send back to Gemini (copy/paste)

Answers:

1. Advanced-mode edits: re-run downstream stages from scratch. If Stage 3 is edited, Stage 4 Critic must not receive the previous critique as context (avoid anchoring/path dependence). Keep old critique only for audit/UI diff; optional toggle to include it, default OFF.
 2. Persist and make retrievable the full provenance: raw prompts, model params, raw responses, parse/repair attempts. Expose via API only when `include_provenance=true` / Advanced mode to keep default UI clean.
 3. Baseline: one-shot consumer-grade run (same model, higher temp ~0.7, non-JSON freeform). REFLEX stages use lower temp and strict JSON. If easy, add a secondary “iso-baseline” (same temp/settings as pipeline) as a control, but primary baseline should mimic typical chatbot use.
-

One question back to you (non-obvious, but important): do you want the system to support **multiple revisions** (Critique→Revise→Critique→Revise loop) or **exactly one pass**?

Me to ChatGPT: I think I would ideally like to do one pass as the project appears resource intensive enough as is, and while API calls per minute may not be an issue with Gemini Flash, token usage is. Anyway, here was Gemini's reply to the last message:

Acknowledged. These decisions sharpen the system's epistemic boundaries—particularly the "clean slate" re-run logic, which is a sophisticated way to prevent **path dependence** in research.

I have synthesized these requirements into the following **Data Model** and **API Contract**. This ensures that the frontend we discussed earlier can reliably pull "Provenance" data and that the backend handles the distinct "Consumer Baseline" vs. "Reflex Pipeline" logic correctly.

1. The "Provenance" Data Model

To satisfy the requirement for a "bulletproof transparency story," each stage will follow this schema:

JSON

```
None
{
  "stage_id": "critique",
  "status": "completed",
  "data": { /* The structured JSON output */ },
  "provenance": {
    "prompt": "System: You are an adversarial critic... User: Evaluate these claims...",
    "model_params": { "temperature": 0.1, "top_p": 0.9, "model": "gemini-1.5-pro" },
    "raw_response": "```json\n{...}\n```,
    "repair_attempts": 0,
```

```
        "timestamp": "2026-01-28T12:40:00Z"  
    }  
}
```

2. API Endpoints (FastAPI/Pythonic Design)

The backend will expose these specific routes to handle the "Deep Research" vs. "Consumer Baseline" comparison:

Endpoint	Method	Purpose
/run/create	POST	Initialize a <code>run_id</code> with the user's research directive.
/run/{id}/baseline	POST	Execute a Consumer-Grade single-prompt call (\$Temp=0.7\$).
/run/{id}/reflex/{stage}	POST	Execute/Re-execute a specific stage (\$Temp=0.1\$).
/run/{id}/state	GET	Returns full state. Query param <code>?provenance=true</code> adds logs.

3. Implementation of the "Clean Slate" Logic

In the backend `PipelineManager`, the re-run logic for Stage \$N\$ will be strictly isolated:

Python

```
None

def execute_stage(run_id, stage_name, user_edit=None):

    # 1. Fetch current run state

    state = db.get_run(run_id)

    # 2. If user_edit is provided, overwrite state[stage_name]
    data

    if user_edit:

        state[stage_name].data = user_edit

        # CRITICAL: Clear all downstream data to prevent path
        dependence

        state.clear_downstream(from_stage=stage_name)

    # 3. Prepare Context:

    # Only include output data from stages < stage_name.

    # Stage 4 (Critic) will ONLY see Stage 3 (Claims), never
    previous Stage 4 logs.

    context = state.get_upstream_data(stage_name)

    # 4. Call Gemini and persist

    return call_gemini_for_stage(stage_name, context)
```

Next Steps for the Handover

I'm ready to begin the implementation. To keep this aligned with your frontend, would you like me to:

1. **Generate the FastAPI boilerplate** that includes these specific "Consumer vs. Reflex" configurations?
2. **Provide a "Mock Response" set** for all 6 stages based on your "FTL Travel" example, so you can test the frontend's JSON rendering and provenance toggles immediately?

ChatGPT:

This is *excellent*. Gemini is aligned, not drifting. No scope creep yet.

You should **green-light implementation**, but **sequence it to minimize risk**.

Correct response strategy (do this)

Choose option 1 first. Defer option 2.

Why:

- Backend correctness > frontend convenience right now
- Mock responses can be generated later in minutes
- Boilerplate locks architecture and prevents silent divergence

So your answer should be: **generate FastAPI boilerplate with Consumer vs REFLEX configs.**

One small but important correction to make now

There's a subtle risk in what Gemini proposed.

Issue: Stage execution endpoint shape

`/run/{id}/reflex/{stage}` is fine **only if**:

- You enforce **stage ordering** server-side
- You reject invalid jumps (e.g., Stage 5 without Stage 4 data)

Tell Gemini explicitly:

- The backend is the **epistemic authority**, not the client
- The API may *request* a stage, but the orchestrator validates legality

This avoids frontend-driven epistemic corruption.

Message you should send back (tight, authoritative)

This all looks correct and aligned.

Proceed with **generating the FastAPI boilerplate**, including:

- Consumer baseline vs REFLEX pipeline separation
- Orchestrator-enforced stage ordering (backend must reject illegal stage execution requests)
- Provenance storage gated behind `?provenance=true`
- Clean-slate downstream clearing on edits

Defer mock responses for now; we'll generate them after the backend contract is live.

Scope control rule (important)

If Gemini proposes *any* of the following later, you say **no**:

- Multi-pass critique loops
- Automatic retries beyond one repair
- Extra agents
- Streaming UX
- Auth / users

You are building an **epistemic demonstrator**, not a platform.

Excerpt Three

[The following marks the point at which the first real draft of the final code is codded]

Me to Gemini:

This looks good and aligns with what I had in mind. Thanks for being careful about state safety and stage isolation.

Let's proceed to the next phase: engine.py and prompts.py.

Please do this in two steps:

Implement engine.py: a clean Gemini wrapper with JSON-only output, single repair attempt, and full provenance capture.

Propose the full contents of prompts.py for all six stages (plus baseline), focusing on epistemic role separation and correctness over fluency.

Before finalizing the prompts, pause for review so we can sanity-check tone and failure modes."

That is all.

Gemini:

Here is the implementation for Phase 2.

I have implemented [engine.py](#) to be the "epistemic muscle" of the system—it handles the API mechanics and the reliability enforcement (repair loops). For [prompts.py](#), I have designed the roles to be distinct and adversarial where necessary.

Please review the **Prompts** section carefully. This is where we define *how* the AI thinks.

1. backend/engine.py

This module handles the dirty work: calling Gemini, stripping Markdown formatting, attempting repairs, and packaging the "Provenance" data so we can see exactly what happened.

Python

```
None

import os

import json

import re

from datetime import datetime

import google.generativeai as genai

from models import Provenance


# Configure API Key (Assumes env var is set)

# In production, this should be handled via a config manager
```

```
API_KEY = os.getenv("GEMINI_API_KEY")

if API_KEY:

    genai.configure(api_key=API_KEY)

# Default Model Configuration

DEFAULT_MODEL = "gemini-1.5-pro-latest"

def _clean_json_text(text: str) -> str:

    """
    Removes Markdown code blocks (```json ... ```) if present.
    Gemini often wraps JSON in these.

    """
    cleaned = text.strip()

    if cleaned.startswith("```"):

        # Remove first line (```json) and last line (```)
        lines = cleaned.splitlines()

        if len(lines) >= 2:

            cleaned = "\n".join(lines[1:-1])

    return cleaned.strip()

async def call_gemini(
```

```
    system_instruction: str,  
  
    user_prompt: str,  
  
    temperature: float = 0.2,  
  
    model_name: str = DEFAULT_MODEL  
 ) -> tuple[dict, Provenance]:  
  
    """  
  
        Executes a Gemini call with a single 'Self-Repair' attempt  
        for JSON validity.  
  
        Returns structured data and the full provenance record.  
  
    """  
  
  
  
    model = genai.GenerativeModel(  
  
        model_name=model_name,  
  
        system_instruction=system_instruction  
    )  
  
  
  
    generation_config = genai.types.GenerationConfig(  
  
        temperature=temperature,  
  
        response_mime_type="application/json" # Force JSON mode  
        where supported  
    )
```

```
# --- Attempt 1 ---

start_time = datetime.utcnow()

try:

    response = await model.generate_content_async(
        user_prompt,
        generation_config=generation_config
    )

    raw_text = response.text

except Exception as e:

    # Network or API error handling

    raise RuntimeError(f"Gemini API failure: {str(e)}")



data = None

repair_attempts = 0


try:

    clean_text = _clean_json_text(raw_text)

    data = json.loads(clean_text)

except json.JSONDecodeError:

    # --- Attempt 2: Self-Repair ---

    repair_attempts = 1
```

```
repair_prompt = (
    f"The following JSON is invalid. Fix the syntax and
return ONLY the raw JSON.\n\n"
    f"Broken JSON:\n{raw_text}"
)

# Repair runs at Temp 0 for maximum determinism
repair_resp = await model.generate_content_async(
    repair_prompt,
    generation_config=genai.types.GenerationConfig(temperature=0.0)
)

try:
    repaired_text = _clean_json_text(repair_resp.text)
    data = json.loads(repaired_text)

    # If successful, we update raw_text to show the
    # repaired version in logs
    #
    # or keep original? Let's keep original raw_text for
    # debugging,
    #
    # but maybe append repair info?
    #
    # For simplicity: we store the original failure in
    raw_text
```

```
# but return the repaired data.

except json.JSONDecodeError:

    raise ValueError("Failed to generate valid JSON after
repair attempt.")

# --- Provenance Construction ---

provenance = Provenance(
    prompt=f"SYSTEM: {system_instruction}\n\nUSER:
{user_prompt}",
    model_name=model_name,
    model_params={"temperature": temperature},
    raw_response=raw_text,
    repair_attempts=repair_attempts,
    timestamp=start_time
)

return data, provenance

async def run_baseline_chat(user_prompt: str) -> tuple[str,
Provenance]:
    """
    A standard, high-temperature chat call for the Baseline
    comparison.

```

```
Returns plain text, not JSON.
```

```
"""
```

```
model = genai.GenerativeModel(model_name=DEFAULT_MODEL)
```

```
config = genai.types.GenerationConfig(temperature=0.7)
```

```
response = await model.generate_content_async(user_prompt,  
generation_config=config)
```

```
prov = Provenance(
```

```
    prompt=user_prompt,
```

```
    model_name=DEFAULT_MODEL,
```

```
    model_params={"temperature": 0.7, "mode": "baseline"},
```

```
    raw_response=response.text,
```

```
    repair_attempts=0
```

```
)
```

```
return response.text, prov
```

2. backend/prompts.py (PROPOSAL)

I have defined these prompts to enforce the **Epistemic Pipeline**.

Key Design Choices:

1. **Strict Output Schemas:** Every prompt includes a **JSON_SCHEMA** variable. This tells the model exactly what keys to output, reducing parsing errors.

2. **Adversarial Critic:** The critique stage is explicitly told *not* to be helpful, but to be "pedantic and ruthless."
 3. **No "Chat":** System instructions emphasize that the model is a "reasoning engine," not an assistant.

Python

None

三三三

REFLEX System Prompts & Configurations.

三

--- 1. INTENT PARSING ---

```
INTENT_SYS = " "
```

You are an Epistemic Scope Resolver. Your goal is to convert a raw user query into a rigorous research directive.

You must identify implicit assumptions, ambiguous terms, and scope limitations.

Do not answer the question. Only define HOW it should be answered.

三

TNTENT SCHEMA = " " "

{

```
"core_directive": "Refined single-sentence research goal",  
    "definitions": {"term": "precise definition used for this  
run"},
```

```
        "ambiguities_resolved": ["List of ambiguities you resolved  
autonomously"],  
  
        "out_of_scope": ["Aspects explicitly excluded"],  
  
        "constraints": ["Engineering vs Physics", "Timeframe", etc]  
  
    }  
  
    """  
  
# --- 2. DECOMPOSITION ---  
  
DECOMP_SYS = """  
  
You are a Research Architect. Break the research directive into  
orthogonal dimensions or sub-problems.  
  
Avoid answering the questions. Focus on structural completeness.  
  
Ensure dimensions cover the 'search space' of the problem without  
overlapping too much.  
  
"""  
  
DECOMP_SCHEMA = """  
  
{  
  
    "dimensions": [  
  
        {  
  
            "id": "dim_1",  
  
            "label": "Short Label",
```

```
        "description": "What specifically needs to be
investigated here",
        "search_queries": ["query 1", "query 2"]
    },
]
}

"""

# --- 3. CLAIM GENERATION ---

CLAIMS_SYS = """

You are a Hypothesis Generator. For each research dimension,
generate specific, falsifiable claims.

You must assign a confidence score (0.0 - 1.0) to each claim
based on general knowledge.

Be aggressive in generating plausible claims, but honest in
confidence scoring.

"""

CLAIMS_SCHEMA = """

{
    "claims": [
        {
            "dimension_id": "dim_1",

```

```
        "claim_text": "Full sentence claim",  
        "confidence_score": 0.5,  
        "justification": "Why this is plausible",  
        "falsification_criteria": "What evidence would  
disprove this?"
```

```
    }  
]  
}
```

"""

```
# --- 4. CRITIQUE (The Adversary) ---
```

```
CRITIQUE_SYS = """
```

You are a Scientific Reviewer. Your only job is to destroy the
claims provided.

You are NOT helpful. You are pedantic, skeptical, and rigorous.

Assume the claims are hallucinations until proven otherwise.

Identify logical fallacies, missing evidence, or over-confidence.

"""

```
CRITIQUE_SCHEMA = """
```

```
{  
    "critiques": [  
        {
```

```
{  
    "claim_text": "The original claim",  
    "validity_assessment": "Valid / Flawed /  
Unsupported",  
    "specific_flaws": ["List of logical or evidentiary  
gaps"],  
    "required_evidence": "What is missing to prove  
this?",  
    "severity": "High/Medium/Low"  
}  
]  
}  
""""
```

--- 5. REVISION ---

REVISION_SYS = """

You are an Editor. Synthesis the original claims and the
critiques.

Rewrite the claims to be epistemically sound.

If a claim was destroyed by critique, retract it or soften it
significantly.

Adjust confidence scores to reflect the critique.

""""

```
REVISION_SCHEMA = """
{
    "revised_claims": [
        {
            "original_claim": "...",
            "revised_text": "The new, more accurate claim (or [RETRACTED])",
            "new_confidence": 0.3,
            "change_reason": "Explanation of the edit"
        }
    ]
}

"""

# --- 6. META-EVALUATION ---

EVAL_SYS = """
You are the Epistemic Judge. Evaluate the entire research run.

Did the system successfully reduce uncertainty?

Provide a final reliability score (0-100) for the output.

"""

```

```
EVAL_SCHEMA = """
{
    "reliability_score": 85,
    "final_verdict": "Summary of the findings",
    "remaining_uncertainties": ["What is still unknown?"],
    "citation_needed": ["Specific claims that need external verification"]
}
"""
"""

# --- CONFIG DICTIONARY ---
```

```
STAGE_CONFIGS = {
    "intent": {
        "sys": INTENT_SYS,
        "schema": INTENT_SCHEMA,
        "temp": 0.1
    },
    "decomposition": {
        "sys": DECOMP_SYS,
        "schema": DECOMP_SCHEMA,
        "temp": 0.3
}
```

```
        },  
  
        "claims": {  
  
            "sys": CLAIMS_SYS,  
  
            "schema": CLAIMS_SCHEMA,  
  
            "temp": 0.4  
  
        },  
  
        "critique": {  
  
            "sys": CRITIQUE_SYS,  
  
            "schema": CRITIQUE_SCHEMA,  
  
            "temp": 0.1 # Cold for rigor  
  
        },  
  
        "revision": {  
  
            "sys": REVISION_SYS,  
  
            "schema": REVISION_SCHEMA,  
  
            "temp": 0.2  
  
        },  
  
        "evaluation": {  
  
            "sys": EVAL_SYS,  
  
            "schema": EVAL_SCHEMA,  
  
            "temp": 0.1  
  
        }  
    }
```

}

Request for Review

1. **Adversarial Tone:** Is the `CRITIQUE_SYS` prompt ("You are NOT helpful") acceptable for your research goals? I want to ensure we don't accidentally trigger safety filters by being *too* aggressive, though usually "pedantic reviewer" is safe.
2. **Claim Temperature:** I set Claims to `0.4` to allow for some creative hypothesis generation, while Critique is `0.1`. Is this gap sufficient?
3. **JSON Schema:** Does the schema structure look robust enough for the frontend to render the accordion views?

Once approved, I will wire these into the `main.py` routes.

Me to ChatGPT: Gemini came back with the first draft. It looks pretty good though I think some of the prompting could be worded better. What do you think?

[Pastes the above response]

ChatGPT:

Good instincts asking for a review **now**. This is exactly the right pause point. I'll answer at the **roadmap + quality-control level**, then tell you **where I would intervene** and **where I would not**.

High-level verdict (client view)

Yes, this is up to standard.

Yes, this is shippable.

Yes, this already exceeds what most classmates will build.

I would **intervene lightly, not rewrite**. The structure is right; the risks now are *subtle epistemic ones*, not engineering ones.

Think of this phase as *calibration*, not construction.

engine.py — assessment

Status:  APPROVED (no intervention required)

This is solid.

What Gemini got right:

- Proper JSON cleaning (Markdown stripping is necessary in practice)
- Exactly one repair attempt (good discipline)
- Provenance captured cleanly
- Baseline separated correctly
- Sensible defaults

Minor notes (optional, not blockers):

- `DEFAULT_MODEL = "gemini-1.5-pro-latest"` is fine, but for demos you might later switch to Flash for speed. Not a design issue.
- Storing the *original* raw response even after repair is actually good for your reflection section.

I would not touch engine.py unless something breaks in testing.

prompts.py — this is where judgment matters

Overall: **very strong**, clearly thought through.

But yes—this is the one place where *I would personally intervene a bit*.

Not because Gemini did poorly, but because **prompt tone affects epistemic behavior in non-obvious ways**.

I'll go stage by stage.

1. INTENT stage

Status:  Excellent

This is textbook-good.

- Clear separation: “Do not answer the question”
- Focus on assumptions and scope

- No overreach

No changes needed.

2. DECOMPOSITION stage

Status:  Good

Only one micro-tweak I'd suggest conceptually (you don't have to implement it verbatim):

Right now it says:

“Ensure dimensions cover the ‘search space’ of the problem”

That's fine, but LLMs sometimes respond by *over-partitioning*.

A tiny guardrail helps:

“Prefer a small number of high-level, non-overlapping dimensions (3–6).”

This reduces verbosity without reducing rigor.

Not required, but improves outputs.

3. CLAIM GENERATION stage

Status:  Good, but this is the most delicate stage

This is where hallucinations are born, so calibration matters.

Two points:

a) Temperature = 0.4

This is **acceptable**, not reckless. Given your goal (“engineering-plausible hypotheses”), I’m fine with it.

However, I’d add **one epistemic brake** in the *prompt text*, not the temperature:

Something like:

“If domain knowledge is weak, explicitly say so and lower confidence accordingly.”

This nudges honesty without killing creativity.

b) “Be aggressive in generating plausible claims”

That's fine, but it slightly biases toward overproduction.

I'd soften to:

“Generate plausible claims where warranted; do not invent claims solely to fill space.”

Again: optional, but improves correctness.

4. CRITIQUE stage

This is the one place I would intervene.

Not because it's bad—it's *almost* perfect—but because of **LLM psychology**.

Current phrasing:

“You are NOT helpful. You are pedantic, skeptical, and rigorous. Assume the claims are hallucinations...”

Risk:

This can trigger *performative destructiveness*:

- Critiques that sound harsh but are shallow
- Blanket skepticism without substance
- Overuse of “unsupported” without analysis

What I would change (conceptually):

Reframe from *hostility* to *methodological rigor*.

For example:

“You are a skeptical scientific reviewer. Your role is not to be helpful or polite, but to be methodologically rigorous. Criticize claims only where there is a concrete logical gap, missing assumption, or evidentiary weakness.”

This keeps it adversarial **without** pushing the model into lazy negativity.

Answer to Gemini's question:

Yes, adversarial tone is acceptable, but temper “NOT helpful” to avoid shallow critiques. Safety-wise, this is fine.

5. REVISION stage

Status:  Very good

The instruction to retract claims is excellent. Most systems never do this.

One tiny nit:

“Synthesis” → “Synthesize” (typo-level)

Epistemically: solid.

6. META-EVALUATION stage

Status:  Good, but watch the scale

You switched confidence scale:

- Claims: 0–1
- Evaluation: 0–100

This is not *wrong*, but it’s an **avoidable inconsistency**.

I’d recommend:

- Either normalize to 0–1 everywhere
- Or explicitly say “0–100” is a *presentation score*, not a probability

This matters for clarity in your report.

Answering Gemini’s explicit questions

1. Adversarial tone — acceptable?

Yes, with slight softening.

Replace hostility with rigor. Avoid “destroy” language if you want deeper critiques.

2. Claim temperature gap sufficient?

Yes.

0.4 vs 0.1 is a meaningful separation. This is well-chosen.

3. JSON schema robust for frontend?

Yes.

These schemas are frontend-friendly, demo-friendly, and explainable in a presentation.

Would I personally intervene?

Yes, but only here:

- Slightly soften CRITIQUE_SYS wording
- Add one epistemic brake to CLAIMS_SYS
- Normalize or explain confidence scales

I would not:

- Rewrite prompts wholesale
 - Add stages
 - Add loops
 - Touch engine.py
-

What to tell Gemini now (client-style, conversational)

You don't need a manifesto. Something like:

This is very solid and aligned with my goals. I'm happy with engine.py as-is.

For prompts.py, I have only minor calibration feedback:

- The overall structure is good.
- I'd like the Critique stage to emphasize methodological rigor over hostility, to avoid shallow negativity.
- In the Claims stage, please add a light guardrail encouraging explicit acknowledgment of weak domain knowledge.
- Also, let's either normalize confidence scales or clearly distinguish presentation vs probability.

Please make these small adjustments, then proceed to wire the prompts into the routes.

That's it.
