**Course:** Generative AI in Software Development (Midterm Project)

**Instructor:** Hafsa Javeed

**Student / SAP ID:** Areeba Usman 49039

**Project Title:** AI Debugger — a mini AI-powered coding assistant that detects, explains, and suggests fixes for bugs in small code snippets and error traces.

**1. One‑Page Proposal (Scope & Design)**

**Problem & Motivation.** Students and junior developers spend a lot of time diagnosing runtime errors and logic bugs. Existing LLMs can explain code and suggest fixes, but a focused assistant that ingests (a) the failing code (b) the runtime error/stack trace and (c) context (expected behavior) can speed up debugging and teach debugging techniques.

**Project Scope (what this assistant will do).**

* Accept a code snippet (single file or function) and an error message/stack trace.
* Identify likely root cause(s) and classify the bug (syntax, type, runtime, logic, off-by-one, resource leak, etc.).
* Provide a clear, step‑by‑step explanation aimed at a beginner (short bullets + one detailed paragraph).
* Suggest one or more concrete fixes: short code patch (diff) and minimal test or reproduce steps.
* Optionally produce a one-line commit message and unit-test suggestion.
* **Out of scope:** full-project automatic refactoring, running untrusted code inside the assistant, or replacement for full IDE debuggers.

**Why it is useful.** Shortens the debug cycle, helps students learn debugging patterns, and integrates with existing code (lab assignments) for demonstrable, reproducible help.

**Design snapshot.** The prototype will be a simple web-based front-end + Python backend (Flask/FastAPI) that calls an LLM API (OpenAI). The backend will contain small prompt templates and a lightweight rule-based pre-check (like simple static checks) to improve accuracy and reduce tokens.

**Success criteria (for prototype).**

* The assistant can take a short Python or JavaScript snippet and an error and return a plausible root cause and at least one patch.
* Demo uses *one* real buggy assignment from the student's lab work and shows before/after code + explanation.

**2. Design & Architecture**

**High level components**

1. **Frontend** — small web form (paste code + error + language + reproduce steps). Shows explanation and suggested patch.
2. **Backend** — REST API that: pre-processes input, assembles prompt, calls LLM, post-processes LLM output into structured JSON (classification, explanation, patch).
3. **LLM Layer** — chosen API (OpenAI) with a carefully crafted system prompt + few-shot examples.
4. **Test harness** — a folder of 4–6 small buggy examples (from your past labs) used for evaluation and demo.

**Data flow**: Frontend → Backend (/debug) → LLM → Backend postprocess → Frontend.

**Failure modes & mitigation**: hallucinated fixes — mitigate by asking LLM to mark confidence level and include minimal tests; rate limits — include throttling and caching.

**3. Inputs & Outputs**

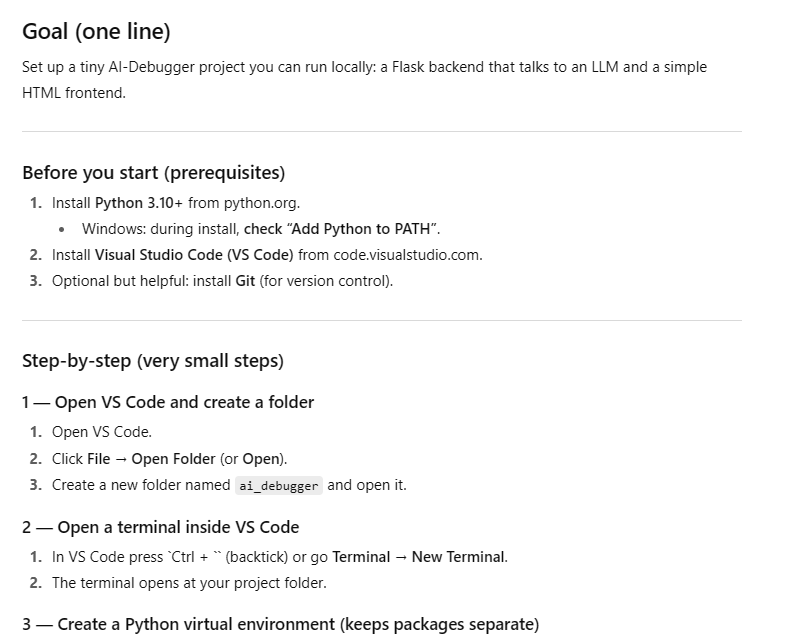
**Inputs**: language (python/js), code snippet or file, error message/stack trace, optional "expected behavior" text.

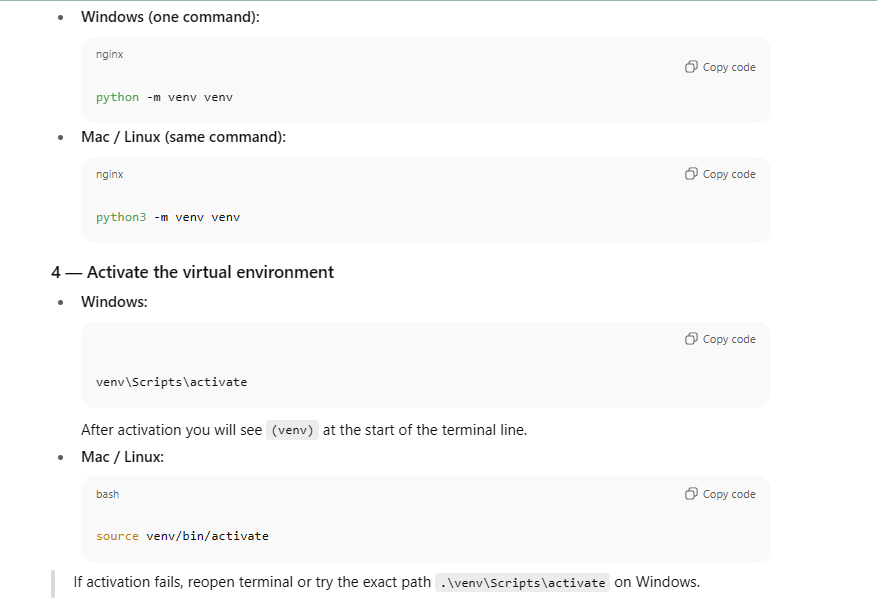
**Outputs**: JSON object with keys: classification, root\_cause, explanation, suggested\_patch (diff or replacement), confidence (low/med/high), test\_suggestion.

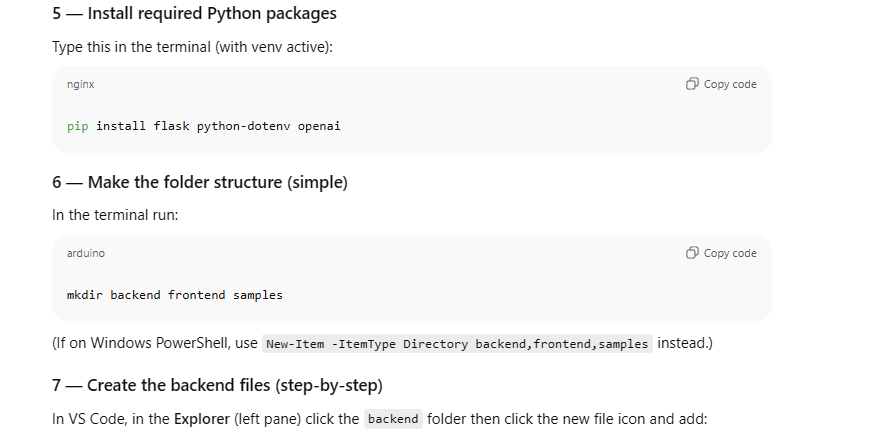
**4. Chosen LLM API**

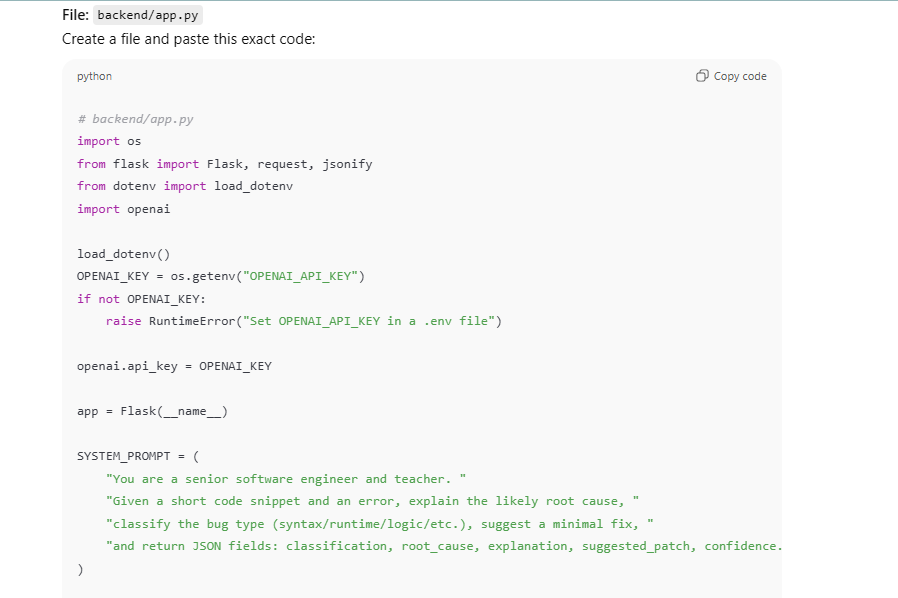
**Recommendation:** OpenAI Chat Completion endpoint (GPT family). Rationale: easy to integrate, good code understanding, stable SDKs. (If you prefer Anthropic or Ollama instead, the same architecture applies.)

**5. Implementation:**



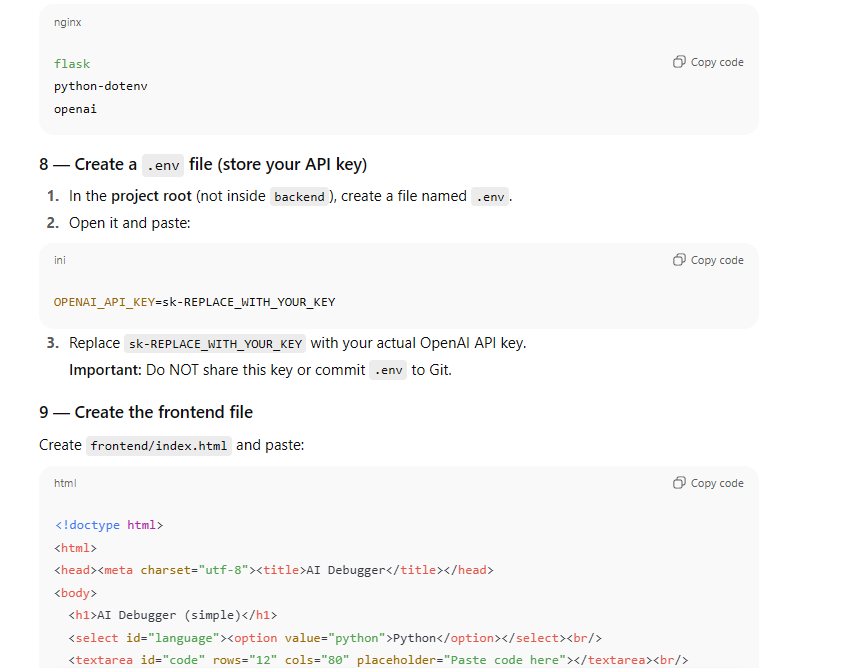
****

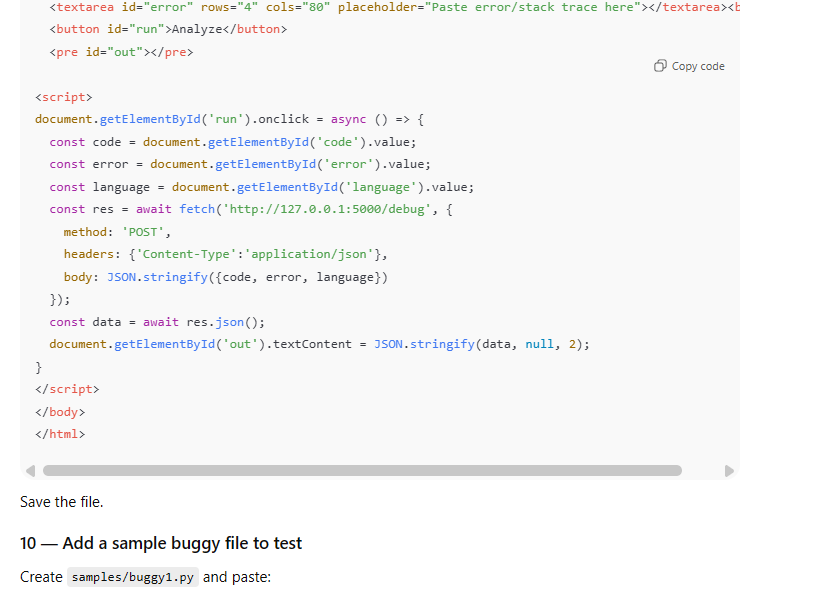
****

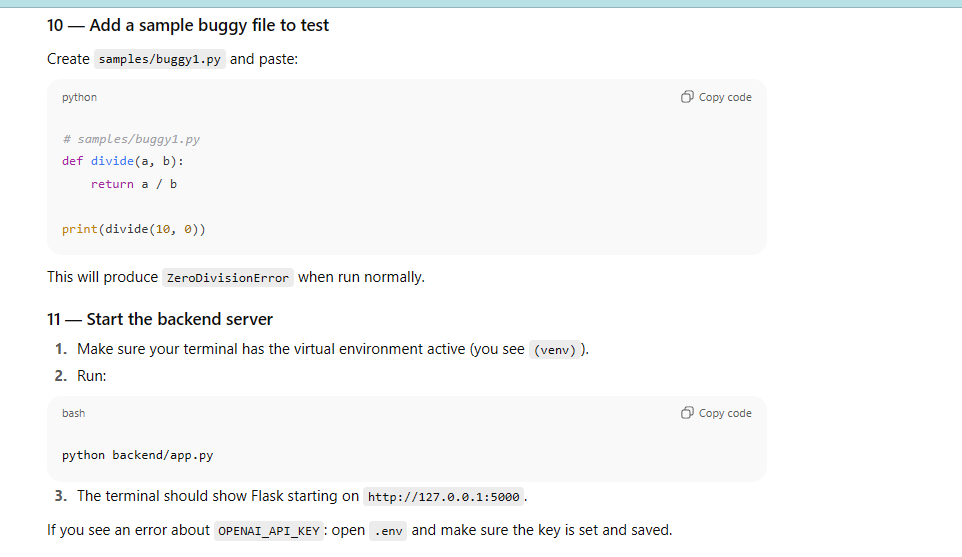
****

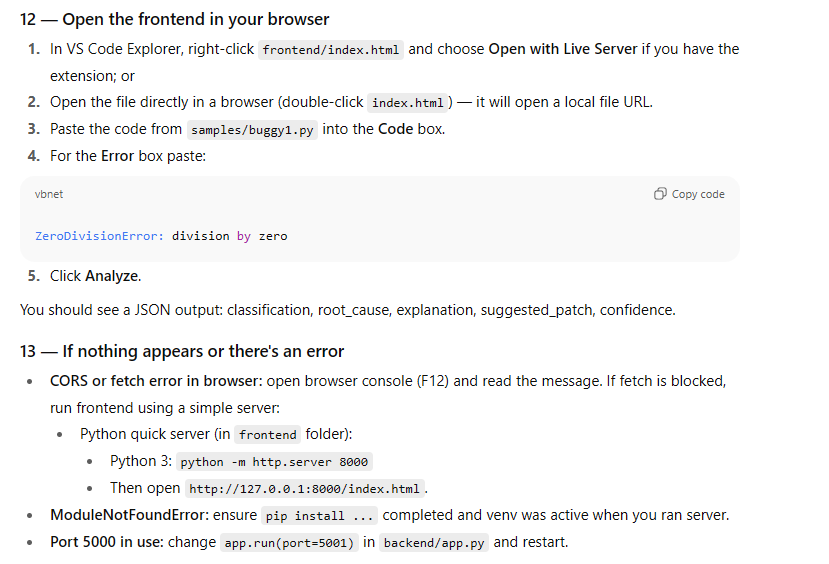
****

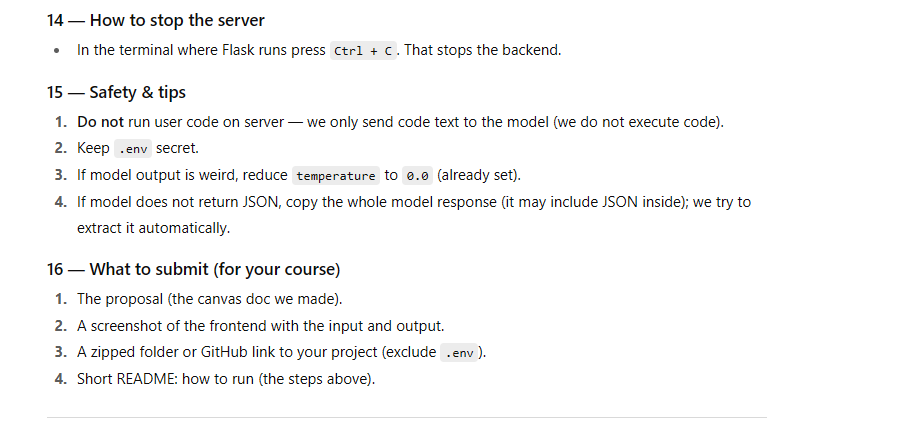
****

****

****

****

****

****