Error Detection API - Technical Assignment

Overview

You are tasked with building a `/detect-error` API that is a critical component of an AI-powered educational platform providing real-time feedback to students.
The API should:

- Take two inputs:
    1. **Question Image**
    2. **Student's Handwritten Attempt**
- Analyze the student's work and identify **step-level errors**
- Return a predefined **JSON response object** (see schema below)

Business Context

Students submit handwritten mathematical solutions by drawing on a digital canvas. The system needs to:

1. Keep getting the latest solution image of the canvas
2. Analyze the mathematical steps for errors
3. Provide helpful feedback including error descriptions, corrections, and hints
4. Store results for analytics and caching

Guidelines

- Example question and student work inputs here
- Performance is evaluated via latency SLOs - report p50/p90/p95 (see Performance section).
- Ideal Case Requirement: p95 ≤ 10 s end-to-end at ~5 concurrent requests for images ≤2 MB.

What We'll Evaluate

- **ML Rigor:** sensible problem framing, baseline → improvement, eval quality (not just prompts)
- **Engineering Quality:** clean API, reliability, observability, basic performance under concurrency
- **System Architecture:** clear components, data flow, scaling, trade-offs
- **Cost & Latency Awareness:** measure, report, and justify trade-offs
- **Reproducibility & Clarity:** one-command eval; clear README; minimal, working demo

- **Product Sense:** useful outputs and failure handling (partial results > silent failure)

Core Requirements

1. Modeling & Data

- **Approach:** you may use OCR→LLM/VLM, direct VLM reasoning, or hybrids. Explain your choice.
- **Dataset (small but real):**
  - **Option A (curate/synthesize):** ≥10 problems, **≥60 step lines** total with step-level labels on Math problems. Include ≥3 noisy/edge cases.
  - **Option B (programmatic labels):** generate synthetic steps with known errors (scripts included) and add **3~5 real samples**.
- **Baseline + at least one improvement:** show an ablation (before/after) isolating the impact.
- **Metrics (test split):** step-level scoring scheme and appropriate accuracy/performance metrics for error detection

2. API Specification

- **Endpoint:** `POST /detect-error` (HTTP; CLI mirror acceptable)

Request Payload

"bounding_box" here represents the coordinates of the area where the edits were done in the respective turn

```
{
    "question_url": "https://example.com/question_image.png",
    "solution_url": "https://example.com/solution_image.png",
    "bounding_box": {
        "minX": 316,
        "maxX": 635,
        "minY": 48.140625,
        "maxY": 79.140625
    },
    "user_id": "optional_user_identifier",
    "session_id": "optional_session_identifier",
    "question_id": "optional_question_identifier"
}
```

Response Format

```
{
    "job_id": "unique_job_identifier",
    "y": 150.5,   //Float
    "error": "Incorrect application of quadratic formula",    //String
    "correction": "The discriminant should be b² - 4ac, not b² + 4ac",
    "hint": "Remember: discriminant = b² - 4ac for quadratic equations",
    "solution_complete": false,     //Bool
    "contains_diagram": true,       //Bool
    "question_has_diagram": true,   //Bool
    "solution_has_diagram": false,  //Bool
    "llm_used": true,               //Bool
    "solution_lines": [...],        //Optional
    "llm_ocr_lines": [...]          //Optional
}
```

- **Engineering must-haves:**
    - Input validation; structured error responses
    - **Concurrency:** handle **≥5 concurrent** requests without crashing (use timeouts; partial results allowed)
    - **Observability:** structured logs + counters (requests, errors)
    - **Persistence (light):** save requests + responses (JSON files, SQLite, etc). This enables auditing.
    - **Security:** accept an API key via header (no hard-coding)
- **Optional (bonus, not required):** streaming responses, caching, circuit breakers, rate-limits, UI.

3. Performance, Cost & Robustness

- **Latency:** report **p50/p90/p95** end-to-end latency on the test set (your hardware noted).
- **Cost:** estimate **cost per 100 requests** (brief math + model pricing/version).
- **Throughput:** short load script (e.g., 1 minute @ 5 RPS) → report success rate & error mix.
- **Robustness:** report accuracy deltas on your noisy/edge subset.

4. System Architecture (Required)

Provide a concise **architecture proposal** (diagram + notes) that covers:

- **Components & Flow:** API layer, preprocessing/OCR (if any), reasoner (LLM/VLM), post-processor, persistence, observability, and the **eval harness path**.
- **Request Lifecycle:** show timeouts, retries, backoff, and **backpressure** under concurrency.

- **Scalability:** statelessness of API, horizontal scaling plan, where to cache/batch, and how you'd separate **sync path** (user request) vs **async jobs** (heavy OCR or re-evals).
- **Reliability & Security:** failure modes, circuit breakers, idempotency, secrets management, PII handling/signed URLs.
- **Performance & Cost Controls:** token/vision budget, image downscaling, caching, prompt/program structure.
- **Trade-offs:** 2–3 alternatives you considered and why you chose this design.

Format: include a flowchart in your repo and a few bullets of trade-offs. Keep it **≤1 page** (diagram + notes).

5. Eval Harness (single command)

- One command (e.g., `make eval` or `bash run_eval.sh`) that:
    1. Loads a **frozen test set**
    2. Runs **baseline** and **improved** variants
    3. Prints a **metrics table** (Key performance metric(s) + p50/p90/p95 + cost + robustness)
    4. Exports per-case results to JSON/CSV

Pin model versions where possible, set a seed and note any non-determinism.

Deliverables

1. **Code** + minimal **demo** (`/detect-error` or CLI)
2. **Data artifact** (or generator script) + brief labeling/creation notes
3. **Eval harness** (single command) + metrics table (baseline vs. improved)
4. **Architecture.md (≤1 page):** Mermaid diagram + trade-offs, scaling & reliability notes
5. **Report.md (≤1 page):** assumptions, design choices, results, ablation, failure modes, next steps
6. **AI-assist log:** which tools/models, key prompts, what was generated vs. authored
7. **README:** setup, **how to run** demo & eval; include `.env.sample` for keys

Submission: GitHub repo or ZIP. We will: `make setup` → `make eval` → run demo.

Scoring Rubric (100)

- **ML Quality & Metrics** (baseline, improvement, ablation, test results) — **30**
- **Engineering & Reliability** (API quality, validation, persistence, concurrency, observability) — **30**
- **System Architecture** (diagram, flow, trade-offs, scalability, reliability) — **10**
- **Performance & Cost** (p50/p90/p95, cost/100, basic throughput results) — **15**
- **Clarity & Repro** (README, harness, report) — **10**
- **Product Sense** (useful outputs, failure handling) — **5**

Checklist before submission:

- eval reproducible
- metrics table present
- demo works on ≥3 sample problems
- data/generator included
- architecture diagram provided

Questions for Clarification

Feel free to ask questions about:

- Specific API requirements
- External service configurations
- Performance expectations
- Technology stack preferences

Good luck! We're excited to see your implementation approach and technical decisions.