🚀 Sentinel AI: Implementation Milestones

🏁 Current State

- **Frontend:** Next.js (Authentication UI Ready)
- **Backend:** NestJS (Authentication API Ready)
- **Database:** Postgres (User Table Ready)

📍 Milestone 1: The Core Infrastructure & "Brain"

Goal: Your NestJS backend can "talk" to the Python AI service and save results to the DB. No GitHub yet.

1.1 Update Database Schema

- **Action:** Apply the `Extended Schema` (provided in context) to your `apps/api/prisma/schema.prisma`.
- **Task:** Run `npx prisma migrate dev --name add_sentinel_models`.
- **Outcome:** Your DB now has tables for `Repository`, `Review`, and `ApiKey`.

1.2 Infrastructure Setup

- **Action:** Update `docker-compose.yml` to include **Redis** (needed for Queues later) and ensure Postgres uses the `pgvector` image.
- **Task:** Run `docker-compose up -d`.

1.3 Build the AI Service (Python)

- **Action:** Create the `apps/ai-engine` FastAPI application.
- **Task:** Create a simple endpoint `POST /analyze` that accepts `{ code, language }` and returns a **mocked** JSON response (don't burn API credits yet).
- **Outcome:** You can `curl localhost:8000/analyze` and get a JSON critique back.

1.4 Internal API Connection

- **Action:** Create a `ReviewService` in NestJS.
- **Task:** Use `HttpService` (Axios) in NestJS to call the Python service.
- **Test:** Create a temporary API endpoint in NestJS (`POST /test-review`) that sends hardcoded text to Python and logs the response.

📍 Milestone 2: The GitHub "Listening" Layer

Goal: When you push code to a dummy GitHub repo, your NestJS server logs the event.

2.1 Register GitHub App

- **Action:** Go to GitHub Developer Settings -> Create New GitHub App.
- **Config:** Set Webhook URL to your machine (use `ngrok` to expose localhost: `https://<id>.ngrok.io/webhooks/github`).
- **Permissions:** Request `Pull Requests (Read/Write)` and `Contents (Read)`.

2.2 Webhook Handling (NestJS)

- **Action:** Create a controller `apps/api/src/github/github.controller.ts`.
- **Task:** Listen for `POST /webhooks/github`. Verify the webhook signature.
- **Logic:** Handle only `pull_request.opened` and `pull_request.synchronize` events.
- **Outcome:** When you open a PR in your test repo, your NestJS console prints: *"Received PR #5 from Repo X"*.

2.3 Fetching the Code

- **Action:** Use `Octokit` (GitHub SDK) in NestJS.
- **Task:** When the webhook hits, use the `pull_request.diff_url` to fetch the raw diff text of the changes.

📍 Milestone 3: The Async Processing Loop (MVP)

Goal: End-to-End flow. Push Code -> Queue -> AI Analysis -> Comment on PR.

3.1 Setup BullMQ (Job Queue)

- **Why:** LLMs are slow (20s+). GitHub Webhooks timeout in 10s. You *must* use a queue.
- **Action:** Install `@nestjs/bullmq` and `bullmq`.
- **Task:** Create a `ReviewQueue` in NestJS.

3.2 Producer / Consumer Flow

- **Producer (Webhook Controller):** When PR comes in -> Add job to Queue -> Return `200 OK` immediately.
- **Consumer (Worker Service):**
  1. Pick up job.
  2. Fetch Diff from GitHub.
  3. Send Diff to Python AI Service.
  4. Receive JSON critique.
  5. Save to Postgres `Review` table.

3.3 The Feedback Loop

- **Action:** Update the Worker to post back to GitHub.
- **Task:** Format the JSON critique into a Markdown string.
- **API Call:** Use Octokit to `createComment` on the specific PR issue.
- **Outcome: Working Prototype!** You open a PR, and 30 seconds later the bot comments.

📍 Milestone 4: The Dashboard Integration

Goal: Users can log in and see the history of reviews (since they are now in the DB).

4.1 "My Reviews" Page

- **Frontend:** Create `/dashboard/reviews` page.
- **Backend:** Create endpoint `GET /reviews` (filtered by `req.user.id`).
- **UI:** Render a list of PRs. Clicking one shows the AI summary and score.

4.2 API Key Management

- **Frontend:** Create `/settings` page.
- **Backend:** Create endpoint `POST /settings/key`.
- **Task:** Save user's OpenAI Key to the `ApiKey` table (encrypted).
- **Logic:** Update Python call to send this key header if it exists.

📍 Milestone 5: The "Enterprise" Upgrade (RAG)

Goal: Make the bot smart by understanding the whole repo, not just the diff.

5.1 Python Vector Store

- **Action:** Install `langchain`, `pgvector` in Python.
- **Task:** Create logic to chunk code files and generate embeddings.

5.2 Indexing Workflow

- **Trigger:** Listen for `installation.created` webhook (when user installs bot).
- **Queue:** Trigger `index-repo-queue`.
- **Python:** Clone repo -> Embed -> Store in Postgres.

5.3 Context-Aware Review

- **Update:** Modify Python `analyze` endpoint.
- **Logic:** Before calling LLM, query Vector DB: *"Find files related to the code in this diff"*.
- **Prompt:** Inject those related files into the System Prompt.

📝 Summary of Next Step

Start immediately with **Milestone 1.1** and **1.2**. Do not worry about GitHub yet. Just get the DB ready and the Python service running locally.