Project spec for “SysRev”

Esther Kim

HCI 584

General description of the project

Sketch → enhanced. SysRev is a lightweight assistant for the initial stages of a systematic review. The app ingests a text export from Clarivate EndNote containing article metadata (title, authors, abstract, year, journal, DOI), structures it into a clean dataset, and, using ChatGPT, assigns each record a relevance score from **1–10** based on user-supplied inclusion criteria (e.g., “intervention study using a wearable device; outcome = physical activity”). The end result is a filterable table and a downloadable CSV that accelerate human screening while keeping final decisions with the researcher.

The first version prioritizes reliability and transparency: explicit input formats, an auditable scoring column, and optional short AI notes. The scoring is heuristic, not definitive, intended to triage at scale and surface likely matches quickly.

External mechanisms. Major packages/APIs include: pandas for tabular processing; rispy for RIS parsing (backup: tab-delimited TSV parsing with pandas); openai (Chat Completions) for relevance scoring; python-dotenv for key management. Optional UI via Streamlit (for a fast web MVP). Logging goes to simple CSV/JSONL artifacts for audit.

GUI vs. CLI. Ideally, a single-page web GUI (file upload → criteria textbox → progress → filter panel → table → download). Minimally, a CLI run (python revlens.py --in export.ris --criteria "..." --out results.csv) that echoes progress and writes a CSV.

Remote control/API. A simple FastAPI endpoint could accept a file + criteria and return a JSON/CSV payload for batch scoring. This would allow remote or scripted control (e.g., from a lab notebook or pipeline). Version 1 focuses on local CLI/Streamlit; the REST API is a stretch goal…

Task Vignettes (User activity flow)

1) Upload & preview

User opens the app and drops a Clarivate EndNote export (.ris or tab-delimited .txt/.tsv) into the upload area, then clicks Parse. The app validates the format, extracts the expected fields, and shows a 10-row preview with columns: Title, Authors, Abstract, Year, Journal, DOI. Duplicates (same DOI, or same normalized Title+Year) are flagged and merged.

Details/ideas for later:

- Accept .csv with recognized headers; warn on missing abstracts.
- Normalize author lists to a consistent Last, F.; Last, F. format.
- Provide a dedup report (kept vs. dropped rows) for audit.
- Handle odd EndNote tag variants (e.g., PY vs. Y1 for year, JO vs. T2 for journal).

2) Enter inclusion criteria

A modal titled "What are you looking for?" offers a large text box. The user types: "Intervention studies that use wearable devices and report physical activity outcomes."

3) Auto-relevance scoring

The user clicks "Score Articles". The app iterates through abstracts, feeding each along with the criteria to the ChatGPT. A progress bar and status line show counts, elapsed time, and retries on transient errors. Each returned result includes a relevance_score (1–10) and a short ai_notes string (less or equals to 20 words). Results are appended to the in-memory table and written to disk incrementally.

Details/ideas for later:
- Enforce a strict JSON schema and perform light repair if the response isn't valid JSON.
- Skip items without abstracts; mark them score=None and surface in a warning panel.

4) Filter, inspect, and export

With scoring complete, a Filter panel shows checkboxes for scores 1–10 (multi-select). The user checks 9 and 10. A Columns section lets them toggle which fields to display (Title, Authors, Year, Journal, DOI, Abstract, Score, Notes). They click "Apply" and the table updates. They then click "Download CSV" to save either the full results or the filtered view.

Details/ideas for later:
- "Visible-only export" vs. "All columns."
- Simple charts (bar of score distribution) for a snapshot.
- Save/restore session state for later review.

Technical flow

<u>Overall data flow (words + blocks)</u>

1. File Ingestor → accepts .ris/.tsv/.txt (and optionally .csv).
2. Parser/Normalizer → maps fields to canonical schema: title:str, authors:list[str] (also a authors_str:str), abstract:str, year:int|None, journal:str, doi:str|None; trims whitespace; lowercases DOI; converts year.
3. Deduper → collapses duplicates by DOI (or normalized Title+Year); keeps provenance notes.
4. Criteria Manager → stores the free-text criteria + advanced flags; stamps criteria_used on each row.
5. ChatGPT Scoring Worker → per record: render prompt → call Chat Completions

(temperature=0) → parse JSON → clamp score to 1–10 → attach relevance_score, ai_notes, model, prompt_version, scored_at.
6. Cache & Logger → key on hash(abstract+criteria+model); write JSONL of raw results and a rolling CSV.
7. Filter/View → client-side filtering by selected score buckets; column visibility toggles.
8. Exporter → CSV writer (UTF-8, quoting abstracts). Optional filtered export.

Data types moving between blocks

- Ingested rows: list[dict] → pandas.DataFrame with canonical columns.
- Criteria: simple str + options dict.
- ChatGPT response: JSON object {relevance_score:int, notes:str, tags:list[str]}.
- Cache key: str
- Exports: .csv and optional .jsonl for raw responses.

User interaction touchpoints

- Input: file upload (GUI) or --in flag (CLI); criteria text area or --criteria flag.
- Output: progress bar/console logs; table view; CSV download; warnings panel.

Minimal implementation constraints

- If GUI is deferred, the CLI must: (a) parse file, (b) score all abstracts, (c) emit CSV with new relevance_score and ai_notes columns.
- Provide a --max-rows flag for quick tests runs?

Unknowns/risks?

- EndNote export variability (RIS tags and tab-delimited headers).
- ChatGPT cost/latency on very large batches; need batching and resume.