EPPS 6354 Information Management

Final Report

OutfitDB

Personalized Outfit Recommendation System

13-table relational schema with three-axis ML personalization

Tzu-Yuan Chen

Spring 2026

Project page: https://chentzuyuan.github.io/outfitdb.html

Live demo: https://outfitdb.onrender.com

1. Project Overview

1.1 System Introduction

OutfitDB is a personalized outfit recommendation system. It stores the user's wardrobe data, actual wear logs, and preference ratings on three independent axes (temperature, aesthetic, occasion) in a 13-table relational database. Three XGBoost classifiers are trained on top of this database, and their outputs are combined multiplicatively to produce daily Top-K outfit recommendations.

Unlike typical fashion apps that rely on dress-code templates or fixed style rules, OutfitDB's recommendations evolve with the user's ratings and wear feedback — the system does not tell the user what looks good, but learns what the user thinks looks good.

1.2 Product Positioning

Form	Purpose
Desktop App	Primary product — packaged with PyInstaller for macOS / Windows
Web trial	Hosted on Render as a public demo / marketing entry point

Core property: **local-first** — each user has an independent SQLite file, and data never leaves the user's machine.

1.3 Version Status

- Current release: v0.3.1 (May 2026)
- Architecture: 13 tables, 3 XGBoost classifiers, 4-stage multiplicative scoring
- Scale: ~2,000 lines of Python (backend); ~3,000 lines of vanilla JS / HTML / CSS (frontend)

2. Technology Stack

2.1 Full Stack Listing

Layer	Technology	Purpose
Backend	FastAPI (Python 3.12)	HTTP routing, async support, Pydantic validation
ORM	SQLAlchemy 2.x	Object-relational mapping; avoids hand-written SQL
Database	SQLite	Embedded relational DB; one file per user
Frontend	Vanilla HTML / CSS / JS	No build step (no React)
Charts	Chart.js 4.4 (CDN)	Stats dashboard + per-item modal
ML	XGBoost	Gradient-boosted trees; three independent classifiers
Migrations	create_all + idempotent ALTER	No Alembic
Hosting (web)	Render.com	uvicorn + persistent disk for SQLite
Packaging	PyInstaller	Cross-platform .dmg / .exe bundling
Style sharing	.odstyle JSON	Trained model exported as a portable file

2.2 Evolution of Technology Choices

OutfitDB's tech stack was not planned in full at the outset; it took shape through iterative adjustment during development. The following records the actual evolution at each layer.

2.2.1 Backend: Shiny → FastAPI

The first deployment used R Shiny. The original public URL is preserved at the original Shiny version (the project was named ClosetMind at that stage; this version is kept to illustrate the prototype phase, in which several functions could not run reliably on Shiny).

Three concrete pain points emerged in production:

- File upload was awkward — the lifecycle of multi-photo validation, resizing, on-disk storage, and linking to a DB row falls outside Shiny's design space.
- Cross-page state management was difficult — Shiny's reactive model is not well suited to a multi-page workflow such as upload → tag → wear → train → recommend.
- ML inference blocked the R session — XGBoost calls would tie up R workers.

The backend was rewritten in FastAPI + uvicorn and deployed on Render. The fundamental issue is that Shiny is a dashboard framework while OutfitDB is a multi-page web application — they are different categories of tool, not different difficulty levels of the same task.

2.2.2 Database: PostgreSQL → SQLite

The first version of the database was built on PostgreSQL. The schema was designed against my personal wardrobe; additional clothing items were later added manually to broaden the demo dataset, and some clothing photographs were generated with DALL-E because the photographic environment was constrained.

During deployment, PostgreSQL's requirement for a separate DB server conflicted with the local-first product positioning. The data was therefore exported from PostgreSQL and migrated to SQLite, with each user mapped to an independent file. SQLite's file-based nature aligns with local-first design, the storage capacity is more than sufficient for an individual wardrobe of about 50 items, deployment is simple, and the user can back up or migrate by copying the file directly.

2.2.3 ML Framework: XGBoost

XGBoost was chosen for the ML layer from the beginning and was not changed. Reasoning:

- Training data is small tabular data (typically fewer than 5,000 ratings per user).
- Gradient boosting outperforms neural networks at this scale (Grinsztajn et al., NeurIPS 2022).
- Inference latency is low and the model is interpretable (feature importance, SHAP).

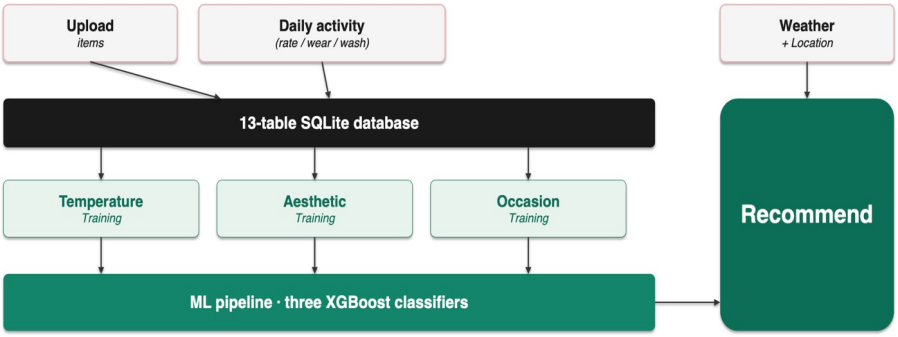
2.2.4 Frontend: HTML/CSS → Vanilla JS

The frontend was initially written in HTML and CSS only. As the feature set grew (dynamic lists, real-time updates, AJAX with the backend, recommendation-page interactivity), pure HTML and CSS were no longer sufficient. After research, vanilla JavaScript was added rather than React.

The reasons for choosing vanilla JS over React: the application is small (5–10 pages), no React-level overhead is required; the build-step complexity (Vite / webpack / npm dependencies) is avoided; initial load is faster; and the learning curve is gentler — there is no need to learn JSX, build tooling, or state management up front.

3. Overall Architecture

OutfitDB's data flow consists of three writers feeding a central database, three classifiers reading from the database, and one final recommendation engine. The architecture is summarized below.



3.1 Three Writers

- **Upload** — one-time wardrobe registration (uploading clothing photos and attributes).
- **Daily activity** — continuous writes (rating, marking worn, marking washed).
- **Weather + Location** — real-time fetch, not persisted to the DB; fed directly into the recommendation engine.

3.2 Three Training Pipelines

Training data read from the database feeds three classifiers respectively:

- Stage 1 — temperature classifier
- Stage 2 — aesthetic classifier
- Stage 3 — occasion classifier

4. Database Schema (13 tables, 5 clusters)

The schema is designed in 3NF, with deliberate denormalizations only at three documented locations (see Chapter 5). Below, each of the five clusters is introduced table by table.

4.1 Cluster A — Wardrobe Identity (3 tables)

This cluster stores the static identity data describing what the user owns.

users

One SQLite file corresponds to one user, but the schema retains user_id as a forward-looking design choice, leaving room for future N:M style sharing.

Key fields: `id, display_name, lat, lon, temp_offset, training_complete, has_thermal_insoles, created_at`

items

One row per clothing item, with 30+ fields grouped as follows:

- **Identification**: id, name, image_path
- **Structure**: category, layer_role, thickness, can_wear_alone
- **Visual**: colors (JSON list), pattern, pattern_complexity, is_multicolor
- **Material**: material (primary fabric), composition (blend ratios as a JSON list)
- **Category-specific**: sleeve, collar, top_length (top), pants_fit, pants_length (bottom)
- **Tags & operations**: style_tags, is_active (soft-delete), wears_per_wash

item_tags

An open key-value metadata table, designed to handle attributes that are not anticipated up front (e.g., brand, purchase_date, gift_from). UniqueConstraint(item_id, key, value) prevents duplicates. The design philosophy: known attributes go in items (closed vocabulary); future, less predictable attributes go in item_tags (open vocabulary).

4.2 Cluster B — Lifecycle Partitioning (2 tables)

This cluster embodies the principle of partitioning by access frequency (see 5.1). The fields of a single clothing item are split into two 1:1-related smaller tables based on write frequency.

item_states
Hot-write region — every wear / wash event triggers an UPDATE.

Fields: `state (Enum: clean / worn / in_laundry / unavailable), worn_count, wear_count_since_wash, last_worn`

Row size is approximately 50 bytes, avoiding the rewriting of large rows that contain JSON arrays on every wear event and reducing write amplification.

item_stats
Aggregate cache region — recomputed lazily after rating events; not on the hot path.

Fields: `coverage_count (number of outfits an item appears in), total_ratings, average_rating, last_worn_date, days_since_worn`

These values are derivable from raw data; caching them avoids re-running the aggregation join on every closet page load (see 5.3).

4.3 Cluster C — Outfits & Contexts (3 tables)

This cluster records the outfit combinations and daily contexts that mediate ML training data and recommendation outputs.

outfits
One outfit, composed of multiple clothing items.

Fields: `warmth_score, coverage_curve, aesthetic_curve, optimal_layer_count, overkill_layers, aesthetic_stop_layers, underfit_flag`

outfit_items
M2M junction table linking outfit ↔ item — the textbook resolution of an N-to-M relation.
Fields: outfit_id, item_id, position.

daily_contexts
Daily wearing context, kept as a separate table (not embedded in outfits) to avoid the transitive dependency outfit → context → date → weather.

Fields: `date, temperature, weather, humidity, wind, occasion, calendar_event`

4.4 Cluster D — Three Preference Axes (3 tables)

This cluster corresponds to the three ML classifiers; each classifier has its own dedicated rating table (see 5.2).

ratings

Stage 2 aesthetic ratings — integer values -1 / 0 / 1 / 2, corresponding to dislike / meh / like / love.

UniqueConstraint(user_id, outfit_id) ensures a single user has at most one rating per outfit. The additional ideal_temp_zone field collects a temperature-calibration signal at the same time.

temperature_ratings

Stage 1 temperature ratings — a JSON list of zones the outfit is suitable for, e.g., ["cool", "mild"]. A single outfit may suit multiple temperature ranges.

occasion_ratings

Stage 3 occasion ratings — a JSON list of events the outfit is suitable for, e.g., ["casual", "work"]. Multiple selections are likewise allowed.

4.5 Cluster E — History & ML Provenance (2 tables)

This cluster records what actually happened, and is the core of the feedback loop and model-version tracking.

outfit_logs

A side-by-side table of actual wears versus system recommendations.

Fields: `recommended_outfit_id, final_worn_outfit_id, user_action (accepted / modified / rejected / skipped), scheduled_for, is_scheduled, logged_at`

This table is the source of OutfitDB's learning capability — without it, the system could only produce one-shot recommendations and would be unable to improve from the user's choices and rejections.

model_runs

An audit trail of every model training run.

Fields: `model_path, training_samples, feature_version, val_auc, val_logloss, trained_at, is_active`

The feature_version field is the gate for .odstyle compatibility on import — model packs trained against an incompatible feature pipeline are refused.

5. Three Key Design Decisions

The schema follows 3NF for the most part, with three deliberate design choices made elsewhere. Each is a documented decision driven by access patterns or performance, not an oversight.

5.1 Decision 1 — Hot/Cold Vertical Partitioning

Problem: Putting every item field into a single table means every wear event rewrites the full ~1 KB row to disk, including columns such as colors and composition that did not change in 95% of cases.

Solution: Vertical partitioning — items (cold, edited monthly) / item_states (hot, edited daily) / item_stats (cache).

Normalization addresses logical dependencies; partitioning addresses physical access — they operate at different layers of the design. This is the rationale behind Cluster B.

5.2 Decision 2 — One Rating Table per Classifier

Problem: Storing all rating axes in a single polymorphic super-table is awkward because the axes have different signal shapes (int versus JSON list); merging them introduces large numbers of NULL columns and forces every ML SELECT to filter on rating_kind.

Solution: Three concrete tables, one per classifier.

Concrete tables yield clean type contracts, no NULL pollution, and ML SELECTs do not require kind-filtering at all. Polymorphism inside one table is debt; concrete tables are easier to reason about.

5.3 Decision 3 — Two Deliberate Denormalizations

The schema contains two intentional denormalizations, both motivated by performance:

Mirror the JSON

- `items.composition` is a JSON list (blend ratios).
- `items.material` separately stores the highest-percentage fabric as an indexed VARCHAR.

Reasoning: every closet-page render filters by primary material; parsing JSON on every row is too slow. The mirror is set automatically from composition on insert and cannot drift.

Cache the Join

- `item_stats.average_rating` and `coverage_count` could in principle be computed by joining ratings × outfit_items.

Reasoning: that join would run on every page load and degrade badly; caching shifts the cost to write time via lazy updates on rating events.

Apart from these two cases, the rest of the schema is BCNF-compliant.

6. ML Pipeline

The ML layer of OutfitDB consists of three independent XGBoost classifiers, integrated via multiplicative scoring, and continuously refined through a feedback loop that learns the user's personal style.

6.1 Three Independent Classifiers

Stage	Name	Input features	Training source
1	Temperature classifier	Item features + current temp zone	temperature_ratings
2	Aesthetic classifier	Item features + outfit structure	ratings
3	Occasion classifier	Item features + current occasion	occasion_ratings

6.2 Multiplicative Scoring

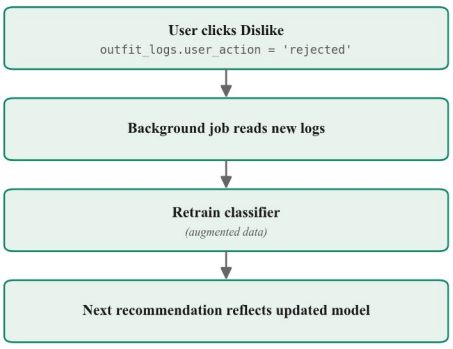
At recommendation time, each candidate outfit's total score is:

```
total = ctx_fit × stage1_pass × stage3_pass × aesthetic_blend
```

Multiplication (rather than addition) enforces that every dimension passes — if any single classifier is uncertain, the total score is pulled down. An outfit that is "aesthetically great but freezing" cannot survive this rule.

6.3 Feedback Loop — the Model Tracks Personal Style

This is the core of the system. A complete cycle of one feedback event is shown below:



The outfit_logs table is the heart of this loop: it captures not only what the system recommended, but also what the user actually chose or rejected. As usage accumulates, the model gradually

internalizes the user's personal sense of style — under the same casual rock dress code, two different users will receive different outfits because each model has learned each individual's preferences.

This is the central distinction between OutfitDB and a static dress-code template: the system does not tell the user what looks good — it learns what the user thinks looks good.

7. User Interface

7.1 Closet Page

See https://outfitdb.onrender.com/closet.

Grid view: each clothing item appears as a card (image + name + key attributes + state badge). Four filters are available — category, color, material, state — and six sort modes — recently added, name, most worn, least worn, highest rated, most recommended. The material filter matches blend partners in composition (e.g., filtering by cashmere will surface a blend that contains 15% cashmere).

Clicking any item card opens a stats modal that visualizes the item in detail:

- Four KPI tiles: total worn / in outfits / avg rating / days since worn
- A wear timeline (line chart over the past 13 weeks; Y-axis capped at 7)
- Often paired with — a horizontal bar chart of the top 5 co-occurring items
- Composition donut — shown when the item has a fabric blend

In addition, Laundry mode supports multi-select for batch state changes (worn / dirty / unavailable).

7.2 Stats Dashboard

See https://outfitdb.onrender.com/stats.

A wardrobe-wide statistics page with five KPI tiles (items / outfits / aesthetic ratings / temp ratings / occ ratings) and six charts: items by category (doughnut), top 10 colors / materials (horizontal bar), aesthetic rating distribution, most-worn and most-recommended top 10 (vertical bar). The page also exposes an .odstyle export button.

7.3 Training UIs

Three independent pages for the three ML axes:

- https://outfitdb.onrender.com/training/temperature — multi-select zone (cold / cool / mild / warm / hot)
- https://outfitdb.onrender.com/training/aesthetic — int slider (-1 / 0 / 1 / 2)

- https://outfitdb.onrender.com/training/occasion — multi-select event (casual / work / formal / sport / home)

Each axis must accumulate at least 6 batches of ratings before the recommendation feature is unlocked — a guard against the cold-start problem.

7.4 Recommendation Page

See https://outfitdb.onrender.com/recommend.

The user sets the daily context (occasion + auto-fetched weather) and clicks Run to retrieve the Top-K outfits. Each card displays six scores: three in bold (aesthetic match, weather fit, occasion fit) — the multiplicative ML axes — and three in light gray (warmth check, freshness, variety) — soft signals. Four sort modes are offered: composite, weather first, aesthetic first, occasion first; sorting is performed client-side without re-querying the server.

7.5 Settings Page

See https://outfitdb.onrender.com/settings.

Provides personal preferences (temperature units °C / °F, presence of thermal innerwear), a list of imported .odstyle packs, and the application update feature: the page displays the current version and a manual "Check for updates" button (no automatic polling). When a newer version is available, two download buttons are surfaced — incremental update and full installer.

7.6 Internationalization (i18n)

The entire UI implements bilingual support via data-i18n attributes; the Chinese / English translation dictionary lives in app/static/js/i18n.js. The language toggle in the top-right corner switches the active language instantly without a page reload. All UI strings (buttons, hints, form labels, recommendation score names) have corresponding translations in both languages. The toggle also updates dynamically populated dropdown options and modal titles.

8. Deployment Architecture

8.1 Web Demo (Render)

- **Platform**: Render.com
- **Persistence**: Render disk volume → SQLite files persist across deployments
- **Cold start**: ~50 seconds on the free tier
- **CI/CD**: git push origin main → Render auto-deploy

Public demo URL: https://outfitdb.onrender.com

8.2 Desktop Bundle (PyInstaller)

- **Targets**: macOS .dmg / Windows .exe
- **Contents**: full Python runtime + uvicorn + all dependencies (single executable)
- **Data location**: ~/Library/Application Support/OutfitDB/profiles/<name>/ (macOS)
- **Update check**: manual — the "Check for updates" button on the Settings page fetches latest.json from a public GitHub raw URL

8.3 Update Manifest Format

Schema of latest.json (introduced in v0.3.1):

```
{
  "version": "0.3.2",
  "url": "https://github.com/.../OutfitDB-0.3.2.dmg",
  "update_url": "https://.../OutfitDB-0.3.2-update.zip",
  "full_url":   "https://.../OutfitDB-0.3.2-full.dmg",
  "notes": "Adds X / fixes Y"
}
```

Both update_url (incremental update) and full_url (full installer) are optional; url serves as a fallback for backwards compatibility with older manifests.

9. Future Roadmap

9.1 Schema Cleanup (Paying Down Technical Debt)

Several known items of technical debt are scheduled for removal in subsequent releases:

- **JSON arrays → M2M tables:** items.colors and items.style_tags are currently stored as JSON lists (violating 1NF) because, during early ML development, schema changes were frequent and JSON allowed migrations to be skipped. Now that the feature pipeline is locked at FEATURE_VERSION=1, these can be refactored back into `item_colors(item_id, color)` and `item_style_tags(item_id, tag)`.
- **Remove last_worn duplication:** item_states.last_worn and item_stats.last_worn_date are written from different code paths and could in principle drift. Refactor: keep item_states.last_worn as the source of truth and compute item_stats.last_worn_date and days_since_worn at read time via Python @property.
- **Remove composition / material dual-write:** the mirror in Decision 3 is currently maintained at the application layer; refactoring material into a Python @property derived from composition removes the drift risk entirely.
- **Items table inheritance:** with five categories, the NULL cost for category-specific columns is acceptable; the trigger for refactoring into table inheritance (items_base + items_top + items_bottom + ...) is set at more than ten categories.

9.2 Completing Style Sharing

Style sharing (.odstyle) is OutfitDB's signature feature. The export and import APIs are already in place, but the UI layer is incomplete:

- **Import UI**: only the API endpoint exists today; a frontend interface for users to upload .odstyle files is needed.
- **Multi-model switching**: users should be able to switch between their own model and friends' models via the UI.
- **Visual indication of imported models**: recommendation cards should label outputs with "recommended using X's model".
- **End-to-end demo scenario**: two users exchanging .odstyle files and producing recommendations for one another.

The narrative value of this feature is to convert a dress code from a vague phrase into a shareable trained model: a party host trains a personal "casual rock" model, exports it, and sends it to all guests, who each end up wearing different clothes (drawn from their own wardrobes) but converge to the host's defined style.

9.3 Long-Term Goal — Social Platform

OutfitDB's long-term goal is to extend the .odstyle sharing mechanism into a public gallery:

- Users can publish their own .odstyle files to a public platform.
- Other users can browse and preview others' aesthetic settings.
- Try someone else's model on one's own wardrobe and observe how recommendations change.

This direction differs fundamentally from mainstream fashion apps — OutfitDB does not sell clothes; it makes aesthetics themselves the unit of exchange. The platform allows users to look at their own wardrobes through someone else's eyes, turning aesthetic taste into something measurable and shareable.

10. AI Disclosure

AI assistance was used during the development of OutfitDB (Claude / Anthropic):

- **Code**: authored by the author with Claude Code as a pair programmer.
- **Documentation, slides, and this report**: drafted with Claude's assistance; all schema design decisions, ML architecture choices, and final content were made by the author.
- **AI Disclosure**: published on the project's home page and release notes.

Translation Note

This English version was translated from the original Chinese manuscript (期末報告.docx) using AI assistance (Claude / Anthropic), and was subsequently reviewed and revised by the author. Any translation errors that remain are the author's responsibility.

GitHub source: https://github.com/chentzuyuan/OutfitDB
Live demo: https://outfitdb.onrender.com

— end of report —