SmartTrip — Robust Recommendation Engine Guide (≤3 pages)

Audience: first full-stack project (student-friendly)
Goal: evolve your current rules + scoring engine into a more robust, measurable, and scalable recommender.

---

1) Where you are today (baseline)

Your current engine (`backend/app.py` → `POST /api/recommendations`) is a **two-tier rules + weighted scoring** system:

- **Tier 1 (Primary):** hard filters + scoring (0–100)
- **Tier 2 (Relaxed):** expanded search with penalties to avoid "0 results"
- **Explainability:** returns `match_details`

This is a great MVP because it's understandable, testable, and fast to iterate.

Main limitation: it is mostly **static** (same weights for everyone) and you don't yet have a measurement loop (tracking → evaluation → improvement).

---

2) What "more robust" means (target outcomes)

A robust recommender has these properties:

- **Measurable:** you can quantify if changes improved results (offline + online)
- **Personalized:** learns from user behavior (clicks, saves, bookings)
- **Resilient:** still works with sparse or missing user input
- **Scalable:** performs well as trips/users grow
- **Explainable:** can still justify results ("why recommended?")
- **Safe:** prevents bad outputs (cancelled trips, wrong dates, unfair bias)

---

3) Roadmap (phased, practical)

Phase 0 — Make the current algorithm measurable (1–2 days)

Why: you can't improve what you can't measure.

- **Add request/response logging** (structured JSON logs) for `/api/recommendations`
  - store: timestamp, preferences, result IDs, scores, whether relaxed triggered
- **Add basic metrics**
  - response time
  - % requests that trigger relaxed tier
  - average top score
  - "no results" rate
- **Create a reproducible evaluation dataset**
  - 30–50 "persona searches" (like your tests) saved as JSON

App changes needed:

- Backend: log to file or DB table (recommended) `recommendation_requests`

Phase 1 — Collect user feedback signals (3–7 days)

Goal: capture what users *actually* like.

Add tracking events from frontend:

- `view_results` (search submitted)
- `impression` (trip card shown)
- `click_trip` (trip opened)
- `save_trip` / `contact_whatsapp` / `start_booking` (stronger intent)

Minimum DB tables (Postgres) to add:

- `users` (even anonymous IDs are ok at first)
- `events` :
    - `id, user_id, session_id, event_type, trip_id, timestamp, metadata(json)`

Frontend changes needed:

- Add a **session id** (cookie/localStorage)
- On results page, send events to backend: `POST /api/events`

Backend changes needed:

- New endpoint `POST /api/events`
- Store events in DB

Why this matters:

- With events you can later learn: "users who click X also like Y".

Phase 2 — Improve ranking quality without ML (1–2 weeks)

Before training models, make the rule engine smarter.

2.1 Feature engineering (better signals) Add more ranking features beyond current weights:

- popularity: click-through rate by trip
- freshness: new trips
- diversity: avoid top 10 all from same country/type
- price-value: normalize price by duration or category
- seasonality: match month preference to historical success

2.2 Better normalization Right now scores are "sum then clamp". Improve stability:

- compute a normalized score based on active criteria
- keep separate components: `filter_score` , `preference_score` , `business_score`

2.3 Better handling of missing inputs If user didn't pick themes/difficulty/budget, don't penalize theme mismatch.

- rule: only apply penalties when user explicitly stated that preference

2.4 Use a proper search layer (optional) If you want faster filtering and text search:

- add Postgres full-text search or Elasticsearch later

App changes needed:

- Backend: refactor scoring into testable functions (module like `backend/recommender/`)
- Tests: add scenarios for diversity and missing inputs

Phase 3 — Personalization (ML-lite) (2–4 weeks)

Start with simple personalization that works well with small data.

3.1 User profile aggregation Build a lightweight profile per user from events:

- preferred continents/countries (based on clicks)
- preferred themes/types
- typical budget/duration

Store a derived table:

- `user_profiles (user_id, top_countries, top_themes, price_range, updated_at)`

3.2 Collaborative filtering (basic) Use "users who clicked A also clicked B":

- item-item similarity based on co-clicks
- blend it with your existing score:
  - final_rank = 0.7 * rules_score + 0.3 * personalized_boost

3.3 Cold start strategy For new users with no history:

- use "popular trips this month" + your rule-based filters

App changes needed:

- Background job to compute similarities / profiles daily
  - simplest: a scheduled script (cron) on Render
  - later: Celery/RQ worker + Redis

Phase 4 — True Learning-to-Rank (4–8+ weeks)

Only do this after you have enough data.

4.1 Define your objective Pick one primary metric:

- booking rate, or
- "qualified click" rate (click + time on page), or
- save/contact rate

4.2 Train a model

- start with logistic regression / XGBoost on engineered features
- later: neural ranking, embeddings

4.3 A/B testing Add experiment flags:

- 50% users see model A, 50% see model B
- measure uplift

App changes needed:

- Experiments table / feature flags
- Model versioning (store model id used for each response)

---

4) Technical upgrades you'll likely need

4.1 Data layer

- Prefer Postgres in dev too (so behavior matches production).
- Add indexes on event tables: `(user_id, timestamp)` , `(trip_id, timestamp)` .

4.2 Backend architecture

Refactor current engine into modules:

- `backend/recommender/filters.py`
- `backend/recommender/scoring.py`
- `backend/recommender/relaxed.py`
- `backend/recommender/personalization.py`

Benefits:

- easier unit tests
- easier to plug in ML later

4.3 Performance and reliability

- Add Redis caching for:
  - `/api/locations` , `/api/tags` , `/api/trip-types`
  - popular recommendations (same query repeated)
- Ensure DB connection pooling is configured (you already have pool settings).

4.4 Explainability (keep trust)

Even with personalization/ML, return "why":

- "Matches your interests: Wildlife + Photography"
- "Similar users booked this trip"
- "High urgency: last places"

---

5) Suggested "final" hybrid ranking formula

A strong real-world approach is a **hybrid**:

- **Hard filters:** availability, dates, geography constraints
- **Rules score:** your current weighted scoring
- **Personalization boost:** user profile + co-click similarity
- **Business constraints:** guaranteed/last places, departing soon
- **Diversity re-rank:** spread top results across countries/types

Conceptually:

```
final_score = (
  0.60 * rules_score
+ 0.25 * personalization_score
+ 0.15 * business_score
)
then apply diversity re-ranking
```

---

6) Concrete “next 7 tasks” checklist

1. Add `POST /api/events` and store events
2. Add session/user identifiers in frontend
3. Add basic dashboards (even simple CSV export) for metrics
4. Refactor scoring into `backend/recommender/` modules
5. Add diversity re-ranking (avoid duplicates)
6. Build daily job to compute `user_profiles`
7. Blend personalization boost into final ranking

---

7) What not to do yet (common mistakes)

- Don’t jump to deep learning before collecting events.
- Don’t optimize performance before you measure bottlenecks.
- Don’t remove explainability; users trust transparent recommendations.

End of roadmap.