SmartTrip Recommendation Engine — Short Summary (Student Version)

Goal: explain (in ~3 pages) how the recommendation engine works and what to look at in the code.

---

1) What the recommendation engine does

When a user searches for a trip, the backend returns a **ranked list of trips**.

- It **filters** trips that are clearly not suitable.
- It then **scores** the remaining trips (0–100) so the best matches appear first.
- If the strict search returns too few results, it runs a **relaxed search** to still show options.

Main endpoint: `POST /api/recommendations` (in `backend/app.py`).

---

2) Inputs (what the frontend sends)

The frontend sends a JSON object of preferences. Key fields:

- **Geography**
  - `selected_countries` : list of country IDs
  - `selected_continents` : list of continent names
- **Trip type (style)**
  - `preferred_type_id` : **hard filter** (exact match)
- **Themes (interests)**
  - `preferred_theme_ids` : **soft scoring** (up to 3)
- **Constraints**
  - `min_duration` , `max_duration`
  - `budget`
  - `difficulty`
  - `year` , `month` (hard time filter)

Frontend call location:

- `src/app/search/results/page.tsx` sends a `fetch()` request to `POST /api/recommendations` .

---

3) Step-by-step algorithm (high level)

Step A — Validate and sanitize input

The backend checks types/ranges and sanitizes strings to reduce errors and prevent abusive inputs.

Why: it protects the API and prevents crashes from bad input.

Step B — Primary search (strict filtering)

The backend builds a database query and applies **hard filters** first.

Hard filters (primary tier):

- **Geography**: selected countries and/or continents (implemented with OR/UNION logic)
- **Trip type**: exact match if `preferred_type_id` is provided
- **Date**: only future trips; optionally strict year/month filtering
- **Availability**: exclude cancelled; exclude full trips (spots_left > 0)
- **Difficulty**: allow ±1 level (if user chose difficulty)
- **Budget**: allow up to 30% over budget (if user provided budget)

Output of this step: a list of **candidate trips**.

Step C — Score each candidate (0–100)

Each candidate starts with a base score, then earns bonuses (or penalties) based on match quality.

Finally the score is clamped to 0–100.

Step D — Sort and select top results

Trips are sorted by:

1. **Score descending** (best match first)
2. **Start date ascending** (sooner trips first)

Top 10 are selected.

Step E — Relaxed search (only if needed)

If primary results are fewer than 6, the backend runs a relaxed search to fill up to 10.

Relaxed tier changes:

- Expands geography (often to continent-level)
- Expands date window (around the selected month/year)
- Loosens difficulty tolerance (±2)
- Loosens budget tolerance (up to 50% over)
- Stops filtering by trip type (but adds penalties for different types)
- Applies a base penalty to mark them as “expanded results”

Relaxed results are marked with `is_relaxed: true`.

---

4) Scoring model (what “match_score” means)

Scoring is a **weighted points system**.

Base idea:

- Passing hard filters earns **BASE_SCORE**.
- Then the trip earns extra points (or penalties) for how well it matches.

Weights used in this project

These live in `backend/app.py` in `SCORING_WEIGHTS`.

- **Base**
  - `BASE_SCORE`: +25
- **Themes (interests)**
  - `THEME_FULL`: +25 (2+ theme matches)

    - `THEME_PARTIAL` : +12 (1 theme match)
    - `THEME_PENALTY` : -15 (no theme match, when user selected themes)
- **Difficulty**
    - `DIFFICULTY_PERFECT` : +15 (exact match)
- **Duration**
    - `DURATION_IDEAL` : +12 (within range)
    - `DURATION_GOOD` : +8 (close)
- **Budget**
    - `BUDGET_PERFECT` : +12 (within budget)
    - `BUDGET_GOOD` : +8 (within 110%)
    - `BUDGET_ACCEPTABLE` : +5 (within 120%)
- **Status / urgency**
    - `STATUS_GUARANTEED` : +7
    - `STATUS_LAST_PLACES` : +15
    - `DEPARTING_SOON` : +7 (within 30 days)
- **Geography**
    - `GEO_DIRECT_COUNTRY` : +15
    - `GEO_CONTINENT` : +5

Color thresholds

- **Turquoise:** score ≥ 70 (strong match)
- **Orange:** score ≥ 50 (partial match)
- **Red:** score < 50 (weak match)

These thresholds are returned to the frontend so it can color-code the UI.

---

5) Example (simple)

User selects:

- Asia continent
- Themes: Cultural + Food
- Duration: 7–10 days
- Budget: 10,000
- Difficulty: 2

A trip is:

- In Asia
- Has Cultural theme only
- Duration 9 days
- Price 9,500
- Difficulty 2
- Status Guaranteed

Approx score:

- Base +25
- Continent +5
- Theme partial +12

- Difficulty perfect +15
- Duration ideal +12
- Budget perfect +12
- Guaranteed +7 = 88 (excellent match)

The backend also returns `match_details` strings to explain why.

---

6) Where to read the code (fast path)

- **Entry point (recommendations):** `backend/app.py` → `POST /api/recommendations`
- **Weights & thresholds:** `backend/app.py` → `SCORING_WEIGHTS` , `SCORE_THRESHOLDS` , `RecommendationConfig`
- **Database models:** `backend/models.py`
- **Frontend request:** `src/app/search/results/page.tsx` (builds JSON and calls backend)

---

7) Why this design is used (engineering logic)

- **Hard filters** prevent nonsense results (e.g., cancelled trips, too expensive).
- **Scoring** allows "best available matches" without requiring perfection.
- **Two-tier (relaxed)** prevents "0 results" which frustrates users.
- **Explainability** (`match_details`) makes the system transparent and easier to trust.

---

8) What you can tune safely

In `backend/app.py` you can adjust:

- `SCORING_WEIGHTS` : to make certain criteria more important.
- `SCORE_THRESHOLDS` : to make the UI more/less strict.
- `RecommendationConfig.*` : to tighten/loosen filtering.

Rule of thumb:

- If users complain results are too strict: relax filters or reduce penalties.
- If users complain results feel random: increase weights for the most meaningful criteria.

End of short summary.