# 2026 The International Mathematical Modeling Challenge (IMMC)

## Smart City Electric Scooter Planning

Batuhan Kal
Can Büyükuğurgör
Emre Yavuz
Güçlü Ada Dönmez

# Summary

A city plans to launch an electric scooter sharing system with 500 scooters distributed across five distinct zones (A–E). The municipality's objectives are simultaneously to increase user satisfaction, maximize the number of daily rides, and minimize operational costs arising from nightly transportation and charging. The challenge is that scooter demand is not uniform: zones differ in passenger density, student share, tourism intensity, and public transport accessibility, and scooters also migrate between zones during the day.

We developed a single integrated approach that answers Tasks 1–4 together: instead of proposing an initial allocation first and only later thinking about imbalance and costs, we directly **optimize** the allocation using a one-year simulation model. This ensures that the recommended starting distribution is already compatible with realistic daily drift, rebalancing effort, and charging expenses.

A key modeling difficulty is that the problem statement gives qualitative descriptions such as "low", "medium", "high" and "very high". Rather than assigning arbitrary numbers, we built a principled conversion into numerical values using **percentiles**. Concretely, we interpreted:

$$\text{low} \to 12.5\%, \quad \text{medium} \to 37.5\%, \quad \text{high} \to 62.5\%, \quad \text{very high} \to 87.5\%.$$

These percentile anchors were computed from large multi-region datasets for two inputs that we personally modeled: **tourism** and **student share**. This gave us realistic scales and prevented extreme outliers from dominating the calibration.

Our daily simulation uses three linked components: (1) **Demand and capacity:** scooters can complete at most 6 trips/day, and total usage time is limited, so supply scarcity matters; (2) **Migration:** 35% of scooters change zones during the day, with movement possible between any pair of zones; (3) **Nightly rebalancing and charging:** scooters are fully charged every night, but energy cost is proportional to distance traveled; additionally, a tunable fraction of displaced scooters is repositioned at night to restore the planned distribution.

To search for the best policy, we used a **genetic algorithm (GA)**. The GA optimizes:

- 8 seasonal allocations (4 seasons $\times$ weekday/weekend) where each allocation assigns 500 scooters across zones A–E,
- a rebalancing rate $r \in [0, 1]$ controlling how aggressively the system restores the planned distribution at night.

Every candidate solution is evaluated by a **full-year simulation**, producing total annual trips and annual operating cost. We then score solutions by a balanced objective that rewards trips while penalizing costs, reflecting the municipality's dual target.

Finally, we tested two what-if scenarios: a festival raising passenger density in Zone C by 70%, and a public transport strike in Zone B doubling scooter demand. The optimized policy adapts naturally by reallocating scooters toward the affected zones and adjusting how strongly the nightly rebalancing should compensate drift.

Overall, our model produces allocations that are stable across months, realistic under drift, and consistent with both usage maximization and cost minimization. Most importantly, every coefficient and modeling choice has a clear interpretation and can be updated later if the municipality acquires richer local data (for example, actual scooter GPS logs after deployment).

# Contents

# Letter to the Decision Makers of the IMMC

**Dear Directors and Esteemed Members of the IMMC,**

Thank you for trusting us with the design of a realistic and mathematically consistent planning model for a smart city electric scooter system. The municipality's goals - **(i)** maximizing daily rides and user satisfaction while **(ii)** minimizing transport and charging costs - are naturally conflicting, so the core of our approach is to treat this as a single integrated optimization problem instead of four separate tasks.

We emphasize one principle throughout our solution: **the first-day distribution (Task 1) cannot be chosen independently of daily imbalance (Task 2) and costs (Task 3).** A distribution that looks "good" on paper may collapse after daily scooter migration, producing shortages in high-demand zones and excessive nightly transport cost. For that reason, we developed a simulation-based optimization framework that produces allocations which remain efficient and cost-aware over time.

To avoid arbitrary numbers when translating subjective labels such as "low" and "very high", we used a percentile-based numerical scaling. This let us ground our values in real data patterns for the two inputs we explicitly modeled: student ratios and tourism. We then embedded these numerized factors into an annual simulation that accounts for drift (35% migration), nightly repositioning, and proportional charging cost.

Finally, we used a genetic algorithm to explore thousands of competing allocation strategies efficiently. The GA does not assume a single perfect rule; instead, it learns which seasonal weekday/weekend allocations and which nightly return strength best achieve the municipality's objectives over a full year.

We believe our solution is both mathematically rigorous and practically interpretable: every model component corresponds to a real operational mechanism (demand, capacity, drift, repositioning, charging). At the same time, the framework remains flexible: if new local data becomes available, it can be plugged in without changing the core logic.

Sincerely,

# 1 Assumptions

Our objective was to keep the model realistic while still simple enough to implement and explain clearly. The assumptions below are not arbitrary: each one is either explicitly stated in the problem text, or it is a practical interpretation needed to connect the given information into a working model.

## 1.1 Operational Assumptions

1. **Each scooter can perform up to 6 trips per day (optimal conditions).** This value represents the maximum possible daily usage of a scooter. As a result, the total number of trips that can occur in a zone is limited by the number of scooters placed in that zone, multiplied by 6.

2. **Daily migration occurs: 35% of scooters end the day in a different zone.** The statement does *not* say each zone loses 35%; it says *overall* about 35% of scooters relocate. Therefore, our model uses a global mixing mechanism rather than forcing 35% outflow from each zone individually.

3. **Nightly costs:** Each scooter is fully charged every night, but the *energy cost is proportional to the amount of charge added.* It costs 8 TL when fully charged, and other charging costs are calculated proportionally to this.

4. **Natural self-discharge overnight is negligible compared to daily usage energy."**. Lithium-ion batteries have a very low self-discharge rate compared to other secondary battery types; they are reported to lose approximately 1–3 % of charge per month when idle, which supports modeling a negligible overnight energy loss prior to nightly charging.

## 1.2 Connectivity and Spatial Assumptions

1. **All zones are mutually reachable in one step.** The city is divided into 5 zones A–E. We assume any scooter can move from any zone to any other zone within one day's travel pattern. In other words, the daily migration network is *fully connected.*

2. **Intra-zone distribution is approximately uniform.** Within each zone, scooters are assumed spread evenly across the zone's usable area. This means "more scooters in a zone" translates into "easier to find a scooter" at a typical location inside that zone. We have to make this assumption because we have no information about the size or shape of the zones.

## 1.3 Demand Composition Assumptions (students and tourists)

The problem gives four drivers: passenger density, student share, tourism intensity, and public transport access. A key modeling decision is: **tourists and students are not extra demand added on top of passenger density.** They are subsets of the passengers already included in passenger density. So, we model passenger density as the total mobility activity in the zone, and we

model students/tourists as *composition effects* that change how likely a passenger is to choose an e-scooter.

1. **Students have higher scooter propensity.** Students use scooters **30% more** than average passengers. This does not mean they travel more overall; it means that, given a trip need, they are more likely to choose a scooter. We encode this with a multiplier $m_{\text{stu}} = 1.30$.

2. **Tourists have lower scooter propensity.** Tourists use scooters **0.65 times** the average passenger, since tourists may prefer walking, guided transport, or might be unfamiliar with apps/roads. We encode this with $m_{\text{tour}} = 0.65$.

3. **Student activity is seasonal.** Even if student population share is stable in a zone, commuting-related student scooter usage is lower during long school breaks. Therefore, we apply a season factor that reduces the student effect in summer.

4. **Tourism mix is seasonal.** Tourism is treated as a seasonal contributor to the passenger mix: the fraction of passengers who are tourists increases in high-tourism seasons and decreases in low-tourism seasons. Importantly, this affects scooter usage only through the tourist multiplier $m_{\text{tour}}$, not by directly adding demand.

# 2 Numerizing Qualitative Levels Using Percentiles

The problem statement describes several zone attributes qualitatively: *low, medium, high, very high.* A common mistake in modeling is to assign these levels arbitrary values like 1–4. This makes the final outputs feel ungrounded because the spacing is chosen by the modeler, not by data.

Our solution is based on a simple philosophy:

*If we do not know the exact value for a city zone, we can still assign it a statistically plausible rank.*

## 2.1 Percentile anchors for four qualitative labels

We used the following mapping:

$$\text{low} \to 12.5\%, \quad \text{medium} \to 37.5\%, \quad \text{high} \to 62.5\%, \quad \text{very high} \to 87.5\%.$$

This choice is deliberate:

- It avoids extremes (0% and 100%), which are often outliers.
- It represents a smooth step through the distribution (lower quartile-ish, below median, above median, upper tail).
- It is symmetric around the median (50%), which makes comparisons fair.

When needed, we also reference 0% and 100% as minimum/maximum bounds.

Percentiles let us interpret "high" as "higher than most places" without claiming a precise number. This matches the information level in the problem: we are given relative categories, not measurements. In addition, percentiles are robust against extreme values and skewed distributions, which is especially important for tourism where a few regions can receive vastly more visitors than the typical case.

# 3 Solving Tasks 1–4 Together

At first glance, the IMMC tasks are written as if they should be solved one by one:

1. decide an initial distribution (Task 1),
2. predict end-of-day imbalance (Task 2),
3. calculate costs (Task 3),
4. then run optimization (Task 4).

However, in a real scooter-sharing system, these four steps are not independent. The **initial distribution is only meaningful if it is judged together with its daily drift and its operational costs**. For this reason, we deliberately treated Tasks 1–4 as **one single optimization problem**.

## 3.1 Our key idea: an allocation is not a one-time guess, it is a policy

Instead of distributing the scooters for the first day and leaving them as they drift to reduce transportation costs, we approached the municipality's plan as a **repeatable operating policy** that must perform well *every day* to get the necessary satisfaction from the users. In our model, a policy has two components:

- **Baseline distributions** for each season and day type (weekday / weekend),
- **A nightly rebalancing strength** $r \in [0, 1]$ that restores the baseline after daily drift.

This matches the real-world logic of shared mobility: operators do not permanently "solve Day 1," they operate a system where scooters move daily, and the system must be corrected continuously.

## 3.2 Why optimizing Task 1 automatically solves Tasks 2 and 3

The initial distribution in Task 1 might look like a simple "allocate based on demand" problem, but the statement already includes dynamics that reshape the outcome:

- during the day, scooters move between zones (Task 2),
- at night, scooters are transported and charged (Task 3),
- the municipality simultaneously wants **high usage** and **low operational cost**.

This means that even if a Day 1 distribution looks perfect on paper, it may produce:

- large shortages after migration,
- expensive nightly transport due to large drift,
- underuse in some zones (too many scooters compared to local usage),
- reduced user satisfaction.

Therefore, we cannot judge Task 1 in isolation. A "good" initial distribution is not the one that matches demand only on Day 1, but the one that stays efficient after daily movement *and* remains cost-feasible to operate.

### 3.3 How our framework works in one loop

We built one integrated framework that evaluates a policy over an entire year:

1. At the start of each day, the city begins with a certain distribution across zones.
2. During the day, scooters migrate and end the day in a new distribution (Task 2).
3. At night, the municipality rebalances scooters toward the baseline using strength $r$, and all scooters are charged (Task 3).
4. We repeat this process for all days of the year and measure two outputs:
    - total yearly trips (system usage / user satisfaction proxy),
    - total yearly cost (transport + charging).

Because the distribution interacts with drift and cost *every single day*, the only fair way to compare two candidate solutions is to simulate them under the same yearly conditions.

This unified framing makes our solution consistent and realistic: the same model that decides the "best Day 1 allocation" also guarantees that it remains efficient, balanced, and affordable across daily operations.

## 4 Ranking Regions and Extracting Percentile Values

This section explains how we computed the percentile anchors that support our numerization. We include detailed explanations only for:

- tourism intensity,
- student share,

because the other categories were handled by our teammates using separate datasets.

## 4.1 Tourism intensity percentiles

### 4.1.1 What we mean by "tourism intensity"

Tourism is represented by monthly accommodation activity (e.g., nights spent or arrivals) across many regions. Tourism data is strongly seasonal and highly skewed: a few regions dominate tourist volume.

Therefore we do not use averages. Instead we extract percentile values for each month.

### 4.1.2 Monthly percentile table

We computed the, 12.5th, 37.5th, 62.5th, and 87.5th percentiles for each month of 2024 over 793 regions:

| Percentile | M01 | M02 | M03 | M04 | M05 | M06 | M07 | M08 | M09 | M10 | M11 | M12 | TOTAL |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $p_{12.5}$ | 14268.0 | 16047.0 | 18955.0 | 25088.0 | 31539.0 | 40095.0 | 58398.0 | 55644.0 | 34025.0 | 29003.0 | 17607.0 | 16872.0 | 401872.375 |
| $p_{37.5}$ | 67412.0 | 77304.0 | 99861.0 | 123344.0 | 175326.0 | 227524.0 | 326249.0 | 318426.0 | 210570.0 | 138430.0 | 91492.0 | 81993.0 | 2180510.0 |
| $p_{62.5}$ | 244187.0 | 276405.0 | 328961.0 | 378306.0 | 510443.0 | 643431.0 | 886628.0 | 902701.0 | 579989.0 | 433617.0 | 290631.0 | 290334.0 | 6176728.0 |
| $p_{87.5}$ | 915744.0 | 1060456.0 | 1301046.0 | 1480225.0 | 2114958.0 | 2467103.0 | 3358671.0 | 3771915.0 | 2365495.0 | 1784252.0 | 1035663.0 | 1126539.0 | 24165835.875 |

Table 1: Selected monthly tourism distribution percentiles ($p_{12.5}, p_{37.5}, p_{62.5}, p_{87.5}$) across all regions.

To ensure our "tourism intensity" numbers are both realistic and reproducible, we extracted them from **Eurostat regional tourism accommodation statistics**. Specifically, we used the monthly indicator "*Nights spent at tourist accommodation establishments*" at the regional level (*NUTS 2*), which Eurostat reports in a standardized way for all regions in the dataset (Eurostat, 2025a). We downloaded the 2024 monthly records programmatically using Eurostat's official Dissemination API (Eurostat, 2025b).

In preprocessing, we filtered the data to keep only valid region observations for 2024 and then computed the empirical percentiles $\{12.5, 37.5, 62.5, 87.5\}$ *within each month* across all available regions (793 regions in our filtered sample). These percentiles were computed using a short Python pipeline (Appendix: `tourism_monthly_percentiles_eurostat_api.py`), which: (i) downloads the monthly regional tourism series, (ii) groups rows by month, and (iii) applies `numpy.quantile` to obtain the four percentile anchors for that month.

We prefer percentiles over averages because Eurostat tourism activity is highly right-skewed: a small set of regions account for very large volumes, so percentile anchors provide a stable, robust representation of "typical" low/medium/high/very high tourism magnitudes (Eurostat, 2025a).

These values serve two purposes:

1. They let us convert qualitative labels (low/medium/high/very high tourism) into realistic numeric magnitudes.

2. They let us measure **seasonality** objectively: summer months have dramatically higher percentiles than winter months.

#### 4.1.3 How we turn monthly data into seasonal multipliers

Since our final city policy uses 4 seasons, we compress monthly tourism into seasonal values by averaging the *median level* (50th percentile) of the months in each season:

- Winter: Dec–Jan–Feb
- Spring: Mar–Apr–May
- Summer: Jun–Jul–Aug
- Autumn: Sep–Oct–Nov

This produces a **season tourism multiplier** that is higher in summer and lower in winter. We then use that multiplier only for *passenger composition*, not as a direct demand boost.

### 4.2 Student share percentiles

#### 4.2.1 Definition

We define student share as:

$$s \;=\; \frac{\text{high school students} + \text{university students}}{\text{total population}}.$$

#### 4.2.2 How we computed student-share percentiles

The student-share anchors are based on a large multi-district dataset we compiled from official administrative statistics. The denominator (total population) was taken from the Address Based Population Registration System (ABPRS), published by the Turkish Statistical Institute (TurkStat, 2025). The numerator (high school student counts) was taken from the Ministry of National Education's annual *National Education Statistics: Formal Education* tables (Ministry of National Education, 2024).

For each district, we computed:

$$s \;=\; \frac{\text{HS students} + \text{Uni students}}{\text{total population}},$$

where the university component is treated as an *estimate* in districts where enrollment is not reported consistently in a single standardized table. The exact construction and cleaning steps are documented in our Python workflow (Appendix: `student_share_percentiles.py`), which merges district-level population and enrollment records and then computes $p_{12.5}, p_{37.5}, p_{62.5}, p_{87.5}$ using empirical quantiles.

| Percentile | Total student share (HS+Uni) |
|---|---|
| $p_{12.5}$ | 0.055527 |
| $p_{37.5}$ | 0.075871 |
| $p_{62.5}$ | 0.095667 |
| $p_{87.5}$ | 0.113784 |

Table 2: Selected percentiles of the estimated total student share in the population (high school + university).

## 4.3 Traffic impact percentiles

### 4.3.1 Definition

Traffic, in this scenario, can be considered to increase alongside the passenger density, as more active passengers would result in more traffic. This would create more of a need for scooters in places with high passenger density, or in other words, high traffic. The traffic values can be considered as another way to represent the impact of passenger density.

We define traffic impact as:

$$T\% = \frac{\text{time needed to go to a destination with traffic} - \text{time needed to go to a destination without traffic}}{\text{time needed to go to a destination without traffic}} * 100$$

For example, a 200 value of T will suggest that a 10 minute drive without traffic would take 30 minutes with traffic, a 20 minute increase.

From a large dataset we computed about traffic ratios worldwide (INRIX, 2024; TomTom, n.d.) :

| Percentile | Time loss due to traffic (% of original time), T |
|---|---|
| $p_{12.5}$ | 5.0012 |
| $p_{37.5}$ | 25.0089 |
| $p_{62.5}$ | 45.7209 |
| $p_{87.5}$ | 100.0045 |

So, when the problem says “passenger density: very high”, interpreting it as the values close to the 87.5th percentile of traffic is a logical way to be able to see the impact of passenger density when considering scooter distribution or the general travels that are made, rather then instinctual values.

## 4.4 Public Transport Accessibility

Access to public transport determines whether residents rely on scooters to bridge gaps in the network. Here we used the OECD urban accessibility index for 44 countries, which reports the percentage of people within a 15–minute walking distance of transit stops(Organisation for Economic

Co-operation and Development [OECD], 2023). Studies show that when walk times exceed roughly 10–15 minutes, people are more likely to turn to micromobility options.(Lin et al., 2022) In our dataset, the median city had about 90% of the population within 15 minutes of transit, while the bottom quartile had less than 70%. We mapped these statistics into four levels: *Very High* accessibility (top quartile, typically 87.5%), *High* (62.5%), *Medium* (37.5%), and *Low* (12.5%). To illustrate how cities are distributed across percentiles, Table 3 lists the boundaries of eight equal-sized percentile bins (12.5% increments) for the MN_LE15 dataset. The values indicate the percentage of the population within a 15–minute walk of transit; ranges with higher values correspond to better transit accessibility which increases the potential for scooters to serve as first/last-mile connectors.

Table 3: Percentile ranges of public transport accessibility (MN_LE15 dataset)

| **Percentile Range** | **Accessibility (% within 15 minutes)** |
|---|---|
| 12.5% | 53.3 |
| 37.5% | 82.0 |
| 62.5% | 94.0 |
| 87.5% | 97.9 |

These quantified factors feed into our subsequent optimization and simulation models. This index forms the basis for initial scooter allocation, rebalancing operations, and sensitivity analysis.

# 5 Simulation Model

Our solution is built around a **daily simulation** because the scooter system is inherently dynamic. Even if the municipality chooses a reasonable starting allocation, scooters will *drift* during the day as riders take trips and park in different places. Therefore, what matters is not only the initial placement, but also how the system behaves **after many repeated days** under: (i) capacity limits, (ii) changing passenger composition, (iii) daily migration, and (iv) nightly rebalancing.

## 5.1 Notation

We divide the city into five zones, indexed by

$$i \in \{A, B, C, D, E\}.$$

For each day $t$:

- $S_i(t)$ is the number of scooters in zone $i$ at the **start of the day**.
- $E_i(t)$ is the number of scooters in zone $i$ at the **end of the day** *after drift/migration.*
- $R_i(t)$ is the number of scooters in zone $i$ **after nightly rebalancing** (this becomes the next day's start: $S_i(t+1) = R_i(t)$).

The municipality's decision is a **policy**, not a single guess:

- A planned baseline allocation $\mathbf{B}^{(k,d)}$ for each **season** $k$ and **day type** $d$,
- A rebalancing strength $r \in [0, 1]$ that controls how strongly the city restores the baseline every night.

## 5.2 Step 1: Decide the season/day-type and select the baseline plan

Each day $t$ is labeled by:

- season $k \in \{\text{Winter, Spring, Summer, Autumn}\}$,
- day type $d \in \{\text{weekday, weekend}\}$.

So the baseline plan used for that day is simply:

$$\mathbf{B}(t) = \mathbf{B}^{(k,d)}.$$

This baseline is *what the municipality aims to maintain over time.* It is not enforced for free every morning; restoring it requires nightly rebalancing and costs.

## 5.3 Step 2: Compute *where demand should go*

A core modeling choice we made is the following:

**Passenger density already includes everyone. Students and tourists are not added on top; they only change the *propensity* of passengers to use scooters.**

This matters because otherwise we would double-count the same people.

### 5.3.1 (a) Base passenger activity in each zone

From the qualitative IMMC inputs (low / medium / high / very high), we constructed a **normalized passenger activity score**

$$P_i \in [0, 1],$$

which represents how much short-distance movement exists in zone $i$.

### 5.3.2 (b) Public-transport accessibility helps scooters (first/last-mile effect)

We also have a normalized public-transport accessibility score for each zone:

$$T_i \in [0, 1],$$

built from the 15-minute accessibility percentiles.

In dense cities, shared micromobility commonly acts as a **first/last-mile complement** to public transport (e.g., reaching metro/bus stops quickly, or finishing the last segment after transit). This is a well-known interaction in multimodal planning and is explicitly emphasized in micromobility–transit integration guidance and empirical studies [**??**]. Therefore, in our model, better PT access *increases* scooter attractiveness rather than decreasing it.

### 5.3.3 (c) Passenger composition: students and tourists as propensity multipliers

For each zone $i$ we define:

- $s_i$ : student share of passengers in zone $i$ (from percentile data).
- $u_i(t)$ : tourist share of passengers in zone $i$ on day $t$ (seasonal).

We apply the problem-specific behavioral multipliers:

$$m_{\text{stu}} = 1.30, \qquad m_{\text{tour}} = 0.65.$$

To capture the fact that student activity changes with time (weekend vs weekday, and summer vs school months), we use a student activity factor:

$$\eta_{\text{stu}}(t) = \eta_{\text{stu}}^{(\text{weekend/weekday})} \cdot \eta_{\text{stu}}^{(\text{season})}.$$

In our implementation we use $\eta_{\text{stu}}^{(\text{weekend})} = 0.90$, and $\eta_{\text{stu}}^{(\text{season})} = 0.55$ in summer (Jun–Aug), 0.90 in autumn, and 1.00 otherwise.

Then the **effective local-passenger multiplier** in zone $i$ is:

$$M_i^{\text{local}}(t) = (1 - s_i) + s_i \, m_{\text{stu}} \, \eta_{\text{stu}}(t).$$

Tourists are a fraction of the same passenger pool, so the combined propensity multiplier becomes a weighted average:

$$M_i(t) = (1 - u_i(t)) \cdot M_i^{\text{local}}(t) + u_i(t) \cdot m_{\text{tour}}.$$

This is not an arbitrary formula; it is exactly the expected scooter propensity of a mixed population.

In practice, $u_i(t)$ is computed from (i) the zone's qualitative tourism rank and (ii) a seasonal tourism intensity factor derived from the monthly tourism percentile dataset (and we apply a mild weekend scaling to reflect higher weekend tourist activity). By construction, $u_i(t)$ is always kept as a valid share: $0 \le u_i(t) \le 1$.

#### 5.3.4 (d) Passenger-based attractiveness score

We convert passenger activity into a scooter-attractiveness score by multiplying:

$$A_i^{\text{pass}}(t) = P_i \cdot M_i(t).$$

We then normalize this into a probability distribution:

$$p_i^{\text{pass}}(t) = \frac{A_i^{\text{pass}}(t)}{\sum_j A_j^{\text{pass}}(t)}.$$

#### 5.3.5 (e) Traffic congestion as an additional modifier

We incorporate a traffic impact score $\tau_i \in [0, 1]$ derived from the qualitative traffic level (low/medium/high/very high) using the percentile anchors in Section 4. Higher congestion can make short scooter trips relatively more attractive compared to modes slowed by traffic, so we apply it only as a mild demand modifier, not as a separate demand source.

#### 5.3.6 (f) Final demand probability used in the simulation

Rather than adding new demand components, we treat public transport and traffic as *modifiers* on where the passenger-driven demand concentrates. In our implementation we use:

$$\beta_{\text{PT}} = 0.20, \qquad \beta_{\text{traf}} = 0.15.$$

We define a combined intensity:

$$I_i(t) = p_i^{\text{pass}}(t) \cdot (1 + \beta_{\text{PT}} T_i) \cdot (1 + \beta_{\text{traf}} \tau_i),$$

and normalize it to obtain the final zone demand shares:

$$\boxed{\pi_i^{\text{demand}}(t) = \frac{I_i(t)}{\sum_j I_j(t)}.}$$

Combining the above steps into one expression, the final demand function is:

$$\boxed{\pi_i^{\text{demand}}(t) = \frac{\left[\dfrac{P_i\Big((1-u_i(t))\Big((1-s_i)+s_i\, m_{\text{stu}}\eta_{\text{stu}}(t)\Big)+u_i(t)m_{\text{tour}}\Big)}{\sum\limits_j P_j\Big((1-u_j(t))\Big((1-s_j)+s_j\, m_{\text{stu}}\eta_{\text{stu}}(t)\Big)+u_j(t)m_{\text{tour}}\Big)}\right]\cdot(1+\beta_{\text{PT}}T_i)\cdot(1+\beta_{\text{traf}}\tau_i)}{\sum\limits_k\left[\dfrac{P_k\Big((1-u_k(t))\Big((1-s_k)+s_k\, m_{\text{stu}}\eta_{\text{stu}}(t)\Big)+u_k(t)m_{\text{tour}}\Big)}{\sum\limits_j P_j\Big((1-u_j(t))\Big((1-s_j)+s_j\, m_{\text{stu}}\eta_{\text{stu}}(t)\Big)+u_j(t)m_{\text{tour}}\Big)}\right]\cdot(1+\beta_{\text{PT}}T_k)\cdot(1+\beta_{\text{traf}}\tau_k)}.}$$

## 5.4 Step 3: Convert probabilities into daily demand and realized trips

We set a systemwide daily demand budget $Q_{\text{tot}}$ (in our implementation, $Q_{\text{tot}} = 500 * 6 = 3000$), and define the expected demand in zone $i$ as:

$$\mu_i(t) = Q_{\text{tot}} \cdot \pi_i^{\text{demand}}(t).$$

### 5.4.1 Day-to-day randomness

Real cities are never identical day-to-day: weather, events, and random commuter variation create noise. So instead of forcing demand to equal $\mu_i(t)$ deterministically, we draw:

$$Q_i^{\text{dem}}(t) \sim \max\big(0,\ \text{round}\left(\mathcal{N}(\mu_i(t), (\sigma\mu_i(t))^2)\right)\big),$$

with $\sigma = 0.10$ in our implementation. We then apply a small adjustment so that the final $\sum_i Q_i^{\text{dem}}(t) = Q_{\text{tot}}$ exactly.

### 5.4.2 Capacity limit

If a zone has too few scooters, it cannot serve unlimited demand. With the IMMC constraint that each scooter can serve at most 6 rides/day, the daily ride capacity is:

$$C_i(t) = 6S_i(t).$$

So realized trips are:

$$\boxed{Q_i(t) = \min\left(Q_i^{\text{dem}}(t),\, C_i(t)\right).}$$

## 5.5 Step 4: End-of-day migration

The statement "35% of scooters change zones" is implemented as a **fully-connected drift model**: a fraction $m = 0.35$ of the entire fleet is redistributed across zones based on where scooters end up.

### 5.5.1 Where scooters end up: trip-driven + tourism-driven pull

Trips create natural end locations. But tourism-heavy areas also attract parking drift (coastal, leisure, and attraction zones), which is consistent with city-scale evidence from tourism-driven scooter usage [**?**]. We therefore mix two destination signals:

$$\text{TripShare}_i(t) = \frac{Q_i(t)}{\sum_j Q_j(t)}, \qquad \text{TourShare}_i(t) = \frac{u_i(t)}{\sum_j u_j(t)}.$$

With a small tourism drift factor $\varepsilon$, the destination probability is:

$$\pi_i^{\text{dest}}(t) = (1-\varepsilon)\,\text{TripShare}_i(t) + \varepsilon\,\text{TourShare}_i(t).$$

In our implementation, we use $\varepsilon = 0.15$.

### 5.5.2 End-of-day distribution

Let $N = 500$ be the total fleet size. Then:

$$\boxed{E_i(t) = (1-m)S_i(t) + m \cdot N \cdot \pi_i^{\text{dest}}(t).}$$

Interpretation:

- 65% of scooters remain where they started in net effect,
- 35% are redistributed toward zones that attracted more ride endings (and slightly toward tourism pull).

## 5.6 Step 5: Nightly rebalancing back toward the baseline ($r$)

The municipality can move scooters at night to restore its planned baseline. We model this using a single, optimizable parameter:

$$r \in [0, 1].$$

- $r = 0$ means no effort: scooters stay where they ended the day,
- $r = 1$ means full restoration to the baseline plan,
- intermediate $r$ means partial correction.

We update the rebalanced distribution as:

$$\boxed{R_i(t) = (1-r)E_i(t) + rB_i(t).}$$

Finally, the next day begins with:

$$S_i(t+1) = R_i(t).$$

# 6 Cost Model: Transport and Charging

Each night, the municipality pays two different costs:

1. **Rebalancing transport cost** (physically moving scooters between zones),
2. **Charging cost** (energy needed to refill battery usage of that day).

## 6.1 Night transport cost

Nightly rebalancing changes the fleet from the drifted state $E(t)$ to the corrected state $R(t)$. The minimum number of scooters that must be moved is captured by the standard flow identity:

$$\boxed{M(t) = \frac{1}{2} \sum_i |R_i(t) - E_i(t)|.}$$

The $\frac{1}{2}$ appears because every moved scooter simultaneously *decreases* one zone and *increases* another; without halving, we would double-count the same movement.

The IMMC statement assigns 12 TL per moved scooter, therefore:

$$\boxed{\text{Cost}_{\text{move}}(t) = 12 \cdot M(t).}$$

This modeling approach is consistent with classic rebalancing logic used in shared-mobility systems, where operators shift vehicles to reduce imbalance [**?**].

## 6.2 Charging cost

We assume scooters are charged to full every night, but the *paid energy* is proportional to how much distance the fleet traveled that day.

### 6.2.1 Distance per trip from given "optimal use" information

The IMMC provides:

- each scooter is used 18 minutes/day (optimal),
- each scooter can do up to 6 trips/day.

This implies an average trip duration of:

$$\tau_{\text{trip}} = \frac{18}{6} = 3 \text{ minutes.}$$

Using an average urban scooter speed $v$ (we use $v = 15.4$ km/h in code), the distance per trip is:

$$d_{\text{trip}} = v \cdot \frac{3}{60}.$$

This is consistent with empirical e-scooter trial reports where typical trip distances and durations fall in the few-kilometer and ∼10–15 minute range [**?**].

#### 6.2.2 Total daily kilometers and charging price

The total number of realized trips that day is $\sum_i Q_i(t)$, so the total distance is:

$$D_{\text{km}}(t) = d_{\text{trip}} \sum_i Q_i(t).$$

A full 25 km of charge corresponds to 8 TL, so the cost per kilometer is $\frac{8}{25}$ and:

$$\boxed{\text{Cost}_{\text{charge}}(t) = \frac{8}{25} D_{\text{km}}(t).}$$

### 6.3 Total cost

Daily total cost is:

$$\boxed{\text{Cost}(t) = \text{Cost}_{\text{move}}(t) + \text{Cost}_{\text{charge}}(t),} \qquad \text{and} \qquad \boxed{\text{Cost}_{\text{year}} = \sum_{t=1}^{365} \text{Cost}(t).}$$

**Summary of why this simulation is realistic.** This model intentionally stays simple enough to be explainable, but it captures the two main realities of dockless scooters:

- scooters drift daily toward where trips end (and tourism pull can slightly reshape where trips end),
- restoring balance has a direct transport cost, while usage creates a proportional energy cost.

Most importantly, the municipality's decision is evaluated as a *year-long policy*, which is exactly why we can solve Tasks 1–4 with one optimization framework.

## 7 Genetic Algorithm Optimization

Once the simulation model is fixed, the remaining question becomes:

*Which seasonal allocations and which nightly rebalancing strength produce the best overall outcome over a year?*

This is exactly why we use a **genetic algorithm (GA)**. The search space is extremely large: we are not choosing just one distribution, but an entire **seasonal operating policy**, and the system is **non-linear** (capacity caps, stochastic demand, daily migration, and non-trivial costs).

Instead of trying every possibility, the GA efficiently searches for high-performing policies by repeatedly:

- proposing candidate policies,
- simulating them for a full year,
- keeping and improving the best ones.

## 7.1 What the GA optimizes

Each candidate solution (one GA individual) represents one complete operational strategy for the municipality:

1. **8 seasonal baseline distributions** (our planned targets):
   - Winter: weekday and weekend allocation
   - Spring: weekday and weekend allocation
   - Summer: weekday and weekend allocation
   - Autumn: weekday and weekend allocation

   Each allocation is a 5-tuple

   $$\mathbf{B}^{(k,d)} = (B_A, B_B, B_C, B_D, B_E)$$

   such that

   $$B_A + B_B + B_C + B_D + B_E = 500.$$

2. **One nightly rebalancing strength**:

   $$r \in [0, 1].$$

   Intuitively:

- the distributions control **where service is placed**,
- the parameter $r$ controls **how aggressively we pay to keep the system close to that plan**.

This is the minimum set of meaningful decisions that also automatically answers Tasks 1–4:

- Task 1: Day 1 allocation is simply the baseline plan of the relevant season/day-type,
- Task 2: end-of-day imbalance emerges from migration in the simulation,
- Task 3: costs are computed daily from rebalancing + charging,
- Task 4: the GA optimizes the entire policy directly.

## 7.2 How each candidate policy is evaluated

The GA does not guess whether a policy is good. It **tests it**.

For each candidate policy, we run the simulation for all days of the year and record two main outcomes:

- **Total annual trips:** how much usage the city generates in one year,

$$Q_{\text{year}} = \sum_{t=1}^{365} \sum_{i \in \{A,B,C,D,E\}} Q_i(t),$$

  where $Q_i(t)$ is the realized number of scooter trips in zone $i$ on day $t$.

- **Total annual cost:** how much the city pays,

$$C_{\text{year}} = \sum_{t=1}^{365} \Big( \text{Cost}_{\text{move}}(t) + \text{Cost}_{\text{charge}}(t) \Big).$$

This is important: we are **not** optimizing a one-day snapshot. We are optimizing a policy that stays strong under seasonal change, weekend/weekday shifts, and daily drift.

## 7.3 How we combine "maximize trips" and "minimize cost" without arbitrary constants

A classic challenge in optimization is that "trips" and "cost" are in different units. If we combine them directly, one term may dominate purely because of scale.

To avoid introducing an arbitrary conversion constant, we use a simple and transparent strategy:

- normalize trips into a $[0, 1]$ score,
- normalize cost into a $[0, 1]$ score,
- then combine them with clear priorities.

**Trip normalization.** Each scooter can do at most 6 trips/day, and there are 500 scooters. So the theoretical maximum daily trips is:

$$Q_{\text{max,day}} = 500 \times 6 = 3000.$$

Hence the yearly upper bound is:

$$Q_{\text{max,year}} = 365 \times 3000.$$

We define:

$$\text{TripScore} = \frac{Q_{\text{year}}}{Q_{\text{max,year}}}.$$

**Cost normalization.** We define an upper bound for cost by combining:

- **maximum possible nightly moving cost** (moving the entire fleet),
- **maximum possible charging cost** (if scooters are used near their daily capacity).

Then:

$$\text{CostScore} = 1 - \frac{C_{\text{year}}}{C_{\text{max,year}}}.$$

So higher cost lowers the score automatically, on the same $[0, 1]$ scale.

**Operational stability (smoothness).** In practice, we also want an allocation policy that is not "chaotic": if the plan completely reshuffles scooters between seasons, it may be operationally fragile and sensitive to noise.

So we include a **stability preference**:

- policies that change smoothly between seasons score higher,
- policies that jump dramatically score lower.

We represent that idea as a third score in $[0, 1]$, called `SmoothScore`.

**Final fitness score.** The GA maximizes:

$$\text{Fitness} = \text{TripScore} + \text{CostScore} + w_s \cdot \text{SmoothScore},$$

where $w_s$ is a small weight that encourages realistic, stable policies without overpowering the main objectives. In our runs we used $w_s = 0.3$ so stability matters, but the primary goals remain:

- high usage,
- low cost.

## 7.4 Why the GA is appropriate here

A direct brute-force search is impossible:

- each season/day-type distribution must sum to 500,
- we have 8 distributions,
- and a continuous decision variable $r$.

Moreover, the simulation contains non-linear effects:

- trip production is capped by scooter capacity,
- daily migration reshapes supply automatically,
- costs depend on how much correction is needed each night,

- stochastic demand adds variability.

Genetic algorithms are designed specifically for this kind of large, discrete + continuous, non-linear search problem.

## 7.5 Reproducibility Notes: Fixed Constants and Clarifications

To avoid ambiguity, here we explicitly list the numerical values used in our **simulation** and **genetic algorithm (GA)** implementation.

- **Student seasonality factor $\eta_{\mathbf{stu}}(t)$:** We reduce student-driven commuting in the summer and slightly on weekends:

$$\eta_{\text{stu}}^{(\text{summer})} = 0.55, \qquad \eta_{\text{stu}}^{(\text{weekend})} = 0.90,$$

  and $\eta_{\text{stu}} = 1.0$ otherwise.

- **Tourism drift factor $\varepsilon$:** In the daily destination mixing step, we combine trip-driven drift with a tourism-based pull using:

$$\varepsilon = 0.15.$$

- **Cost penalty weight:** We do *not* subtract raw cost directly from raw trips (their magnitudes differ too much). Instead, we normalize both yearly trips and yearly costs into $[0, 1]$ and use:

$$\text{Fitness} = Q_{\text{norm}} - 0.50\, C_{\text{norm}}.$$

  (So the effective cost weight is 0.50 *after normalization*, making the objective scale-stable.)

### 7.5.1 Fixed parameters used in the simulation and GA

| Symbol / Parameter | Meaning | Value Used |
|---|---|---|
| $N$ | total scooters | 500 |
| $m$ | daily migration fraction | 0.35 |
| 12 | nightly move cost per scooter (TL) | 12 |
| 6 | max trips per scooter per day | 6 |
| $v$ | average scooter speed (km/h) | 15.4 |
| $\tau_{\text{trip}}$ | average trip time (min) | 3.0 |
| 25 | full battery range (km) | 25 |
| 8 | full recharge cost (TL) | 8 |
| $m_{\text{stu}}$ | student propensity multiplier | 1.30 |
| $m_{\text{tour}}$ | tourist propensity multiplier | 0.65 |
| $\eta_{\text{stu}}^{(\text{summer})}$ | student summer factor (Jun–Aug) | 0.55 |
| $\eta_{\text{stu}}^{(\text{weekend})}$ | student weekend factor | 0.90 |
| $\eta_{\text{tour}}^{(\text{weekend})}$ | tourist weekend factor | 1.05 |
| $\beta_{\text{PT}}$ | PT accessibility demand weight | 0.20 |
| $\beta_{\text{traf}}$ | traffic time-loss demand weight | 0.15 |
| $\varepsilon$ | tourism pull weight in drift | 0.15 |
| Population size | GA population size | 120 |
| Generations | GA generations | 100 |
| Crossover rate | GA crossover probability | 0.75 |
| Mutation rate | GA mutation probability | 0.25 |
| Elitism fraction | best fraction kept each generation | 0.15 |

Table 4: Key numerical values used for reproducibility (simulation + GA).

## 7.6 GA-Optimized Seasonal Policy (Final Output)

We modeled the municipality's decision as a **year-long policy optimization problem**, rather than a one-shot Day 1 guess. This directly solves Tasks 1–4 together: the Day 1 allocation is meaningful only if it is already optimized for (i) daily drift, (ii) nightly repositioning cost, and (iii) long-term service quality.

Using our simulation (capacity limits, passenger composition, drift, and rebalancing) and a GA search over 8 seasonal baselines plus a single rebalancing strength, we found an effective policy achieving:

- **1,075,361 trips/year** (realized demand under capacity limits),
- **680,987 TL/year** total operational cost (movement + charging),

- a high optimal **rebalancing strength** $r = 0.9235$, meaning that strong nightly repositioning is economically justified under the given **35% daily migration**.

Operationally, the optimized baselines allocate the largest share of scooters to Zone A throughout the year, while still maintaining meaningful reserves in Zones B and D. The weekend allocations shift slightly depending on season, reflecting that the passenger mix (students/tourists) and travel behavior changes between weekday commuting and weekend mobility.

### 7.6.1 Optimized annual performance

| Metric | Best GA Solution |
|---|---|
| Total scooters | 500 |
| Optimized rebalancing strength $r$ | 0.9235 |
| Annual realized trips | 1,075,361 |
| Total annual cost (TL) | 680,987.30 |
| Average daily cost (TL/day) | 1,865.72 |
| Cost per trip (TL/trip) | 0.63 |

Table 5: Performance summary of the best GA policy (1-year simulation).

### 7.6.2 Best 8 seasonal baseline distributions

| Season | Day type | A | B | C | D | E |
|---|---|---|---|---|---|---|
| Winter | Weekday | 272 | 93 | 12 | 102 | 21 |
| Winter | Weekend | 278 | 83 | 10 | 105 | 24 |
| Spring | Weekday | 275 | 89 | 13 | 101 | 22 |
| Spring | Weekend | 281 | 98 | 17 | 84 | 20 |
| Summer | Weekday | 285 | 85 | 10 | 101 | 19 |
| Summer | Weekend | 287 | 81 | 13 | 99 | 20 |
| Autumn | Weekday | 282 | 90 | 12 | 102 | 14 |
| Autumn | Weekend | 279 | 78 | 19 | 94 | 30 |

Table 6: GA-optimized seasonal baseline allocations (the policy targets these distributions and uses nightly rebalancing to maintain them).

### 7.6.3 Nightly repositioning result for Day 1 (2024-01-01)

In the simulation, the system begins with the given winter weekday allocation. Before service starts on Day 1, we apply the optimized rebalancing strength $r = 0.9235$ toward the Winter-Weekday target.

| **Distribution (Day 1)** | **A** | **B** | **C** | **D** | **E** |
|---|---|---|---|---|---|
| Initial given allocation | 272 | 93 | 12 | 102 | 21 |
| After nightly repositioning (end of Day 1) | 178 | 86 | 83 | 109 | 44 |

Table 7: Day 1 distribution after applying the optimized rebalancing policy.

This end-of-day distribution of scooters make sense, considering C is a very touristical location and the flow of scooters to zone C therefore must be very high. We can indeed see this as the majority of the scooters shift from other zones to zone C.

## 7.7 Why the results are plausible

Two observations explain why the allocations appear stable across seasons:

1. **Passenger density dominates baseline allocation.** Passenger density already includes all mobility, including tourists and students. So a zone that is structurally busy remains busy across seasons.

2. **Seasonal effects act through composition, not through "extra demand."** Since tourism does not add demand independently, it only shifts which fraction of passengers behave like tourists. This creates *moderate* shifts rather than extreme seasonal restructuring.

3. **Winter**: Both student regions of B and E show higher baseline distributions compared to their summer distributions due to the academic year. Seemingly the outliers are regions B and C as we would expect them to increase during the weekends independently due to increasing demand. As there are 81 percent more trips on weekends compared to weekdays. (Shah et al., 2021) However considering that all regions are dependent on each other as we are limited to 500 scooters this increase in demand leads to regions like A and D with higher demand conditions to require more scooters than the other regions leading to a decrease in scooter counts in those regions.

4. **Spring**: Region C has higher distribution than its winter distribution because of the tourism intensity. In addition, region A has slightly more distribution compared to its winter distribution due to its very high passenger density. For weekdays to weekends, the primary decrementation for scooters occurs in region D and slightly in region E. This can be linked to a lack of strong factors, making it unable to compete with the increased demand of other regions with very high student, passenger and tourist densities.

5. **Summer**: During the summer we see a decrease in the student regions of B and D compared to the other seasons of the academic calendar. As a result the student activity is not based on academic purposes but instead categorized under the young population usage of scooters. Even though students make up a large portion of scooter users only 9 percent of scooter destinations were academic grounds. (Shah et al., 2021) Therefore the reduction of demand

due to the summer break won't be as drastic as it may seem with scooter usage percentages of students, which the data supports as summer is the least fluctuating season for weekday-weekend comparisons.

6. **Autumn**: Region B shows a decrease in the distribution from weekdays towards weekends because region B has a high student density which is a factor that increases the weekday distribution. However, it must be stated that due to September not being fully incorporated into the academic year the student coefficient for autumn was multiplied by the percentage of school days in the season.

The rebalancing rate $r \approx 0.92$ is also meaningful: it suggests that the best strategy is neither "do nothing" nor "reset perfectly every night," but a fairly strong correction that keeps the system close to its plan while avoiding the full cost of perfect restoration.

## 7.8 Additional Scenario Analysis (January Weekday)

As per Task 5, we ran two extra stress-test scenarios on a **single January weekday**. Instead of changing the entire model, we only modified the relevant variables for each event and re-optimized the **Day-start baseline distribution** (the distribution the day begins with). Then we ran the daily simulation once to evaluate trips, drift, and costs under the new conditions.

### 7.8.1 Scenario 1: Festival in Zone C (+70% passenger density)

A large festival in Zone C increases passenger density by **70%**. In the simulation, this was modeled by multiplying Zone C's passenger-activity input by 1.70. After optimization, the best Day-start baseline distribution became:

$$(A, B, C, D, E) = (155,\ 110,\ 108,\ 106,\ 21).$$

This makes sense relative to the official baseline results: Zone C becomes temporarily much more active, so the optimizer shifts scooters away from the lower-need zone(s) and allocates a noticeably larger share to C, while still keeping strong coverage in the main high-demand zones. This drastic jump in the number of scooters in Zone C shows that the optimization surface is non-linear or threshold-based, likely because Zone C moved from "not worth the cost to serve" to "high priority".

### 7.8.2 Scenario 2: Public transport strike in Zone B (2× scooter demand)

A public transport strike in Zone B doubles scooter demand in that zone. We represented this by applying a 2.0 multiplier to Zone B's demand score (capturing that scooters become the substitute mode when PT disappears). The optimized Day-start baseline distribution became:

$$(A, B, C, D, E) = (137,\ 195,\ 56,\ 94,\ 18).$$

This result is also consistent with the official distributions: Zone B becomes the dominant demand hotspot, so the optimizer concentrates scooters heavily in B (almost 40% of the fleet), while reducing

allocations in the lower-priority areas. This is exactly the behavior expected in a strike scenario: the city must temporarily prioritize the affected region to prevent severe shortages and lost trips.

# 8 Conclusions and Recommendations

## 8.1 Model Strengths

- It respects capacity limits (6 trips/day per scooter) instead of turning demand directly into trips.
- It captures realistic imbalance formation through drift (35% migration).
- It models the main economic tradeoff: more rebalancing raises satisfaction but increases cost.
- It avoids double-counting tourists: they are handled through passenger composition.
- It naturally produces seasonal distributions rather than forcing month-to-month instability.

## 8.2 Limitations in the Model

- Real migration is not perfectly captured by a simple mixing rule; GPS data would enable a richer OD-matrix model.
- We do not explicitly model scooter breakdowns, weather disruptions, or partial availability.
- We treat zones as abstract and do not model street-level geometry.

## 8.3 Planned Improvements

- Add a richer OD-flow model (migration depends on zone pair characteristics).
- Use different trip duration distributions instead of constant 3 minutes/trip.
- Incorporate weather or school-calendar special dates for sharper seasonal student effects.
- Replace scalarized objective with a Pareto front analysis and choose a knee-point solution explicitly.

## 8.4 Conclusion

We built a practical simulation-based optimization framework for distributing 500 scooters across five city zones while accounting for demand differences, daily drift, and operational costs. Instead of treating the tasks separately, we solved Tasks 1–4 jointly by optimizing a year-long policy consisting of seasonal weekday/weekend allocations and a nightly rebalancing strength.

We modeled the municipality's decision as a **year-long policy optimization problem**, rather than a one-shot Day 1 guess. This directly solves Tasks 1–4 together: the Day 1 allocation is

meaningful only if it is already optimized for (i) day time drift, (ii) nightly repositioning cost, and (iii) long-term service quality.

Our key methodological choice was to numerize qualitative categories using percentile anchors rather than arbitrary scales. For tourism and student share, we extracted real percentile values from large multi-region datasets and used these to interpret the categories “low”, “medium”, “high” and “very high” in a defensible way.

By combining a clear daily simulation with a genetic algorithm search, we produced allocations that maximize annual trips while controlling cost. Scenario tests further demonstrated that the same policy structure remains valid under major demand shifts (festival and strike). Overall, our model provides an interpretable and adaptable blueprint for smart city scooter planning.

Operationally, the optimized baselines allocate the largest share of scooters to Zone A throughout the year, while still maintaining meaningful reserves in Zones B and D. The weekend allocations shift slightly depending on season, reflecting that the passenger mix (students/tourists) and travel behavior changes between weekday commuting and weekend mobility.

# References


- Alshayeb, H., & Stevanovic, A. (2025). Impacts of e-scooters on transit ridership in Pittsburgh, PA: A direct and catchment area analysis. *Journal of Traffic and Transportation Engineering (English Edition)*. https://doi.org/10.1016/j.jtte.2025.01.004
- Bureau of Transportation Statistics. (n.d.). *Travel time index*. U.S. Department of Transportation. https://www.bts.gov/content/travel-time-index
- Colorado Department of Transportation. (n.d.). *Methodologies: CDOT OLOS use case data explorer*. https://experience.arcgis.com/experience/9489c136714345bfb17fb27c49b98629/page/Methodologies
- Dibaj, S., Hosseinzadeh, A., Mladenović, M. N., & Kluger, R. (2021). Where have shared e-scooters taken us so far? A review of mobility patterns, usage frequency, and personas. *Sustainability, 13*(21), 11792. https://doi.org/10.3390/su132111792
- Eurostat. (2025a). *Tourism statistics at regional level*. Statistics Explained. Retrieved January 20, 2026, from `https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Tourism_statistics_at_regional_level`
- Eurostat. (2025b). *Eurostat web services: Dissemination API*. Retrieved January 20, 2026, from `https://ec.europa.eu/eurostat/web/main/help/web-services`
- Federal Highway Administration. (n.d.). *Chapter 4: Mobility and access - Policy*. U.S. Department of Transportation. https://www.fhwa.dot.gov/policy/23cpr/chap4.cfm
- Federal Highway Administration Operations. (2014). *Incorporating travel-time reliability into the congestion management process (CMP): A primer*. https://ops.fhwa.dot.gov/publications/fhwahop14034/ch2.htm
- Hosseinzadeh, A., Algomaiah, M., Kluger, R., & Li, Z. (2021). E-scooters and sustainability: Investigating the relationship between the density of E-scooter trips and characteristics of sustainable urban development. *Sustainable Cities and Society, 66*, 102624. https://doi.org/10.1016/j.scs.2020.102624
- INRIX. (2012). *INRIX® national traffic scorecard*. `https://cityclerk.lacity.org/onlinedocs/2012/12-0303_pc_3-1-10.pdf`
- INRIX. (2024). *Global traffic scorecard: INRIX global traffic rankings*. https://inrix.com/scorecard/
- Journal of King Saud University - Science. (n.d.). *Landscape geometry-based percolation of traffic in several populous cities around the world*. https://jksus.org/landscape-geometry-based-percolation-of-traffic-in-several-populous-cities-around-the-world/
- Litman, T. (2023). *Traffic speed versus roadway capacity*. Victoria Transport Policy Institute. https://www.vtpi.org/tsrc.pdf

- MAP AZ Dashboard. (n.d.). *Congestion trends*. University of Arizona. https://mapazdashboard.arizona.edu/infrastructure/congestion-trends
- Metropolitan Washington Council of Governments. (2016). *Congestion report 1st quarter 2016*. `https://www.mwcog.org/assets/1/28/NCR_Congestion_Report_2016Q1.pdf`
- Ministry of National Education. (2024). *National Education Statistics: Formal Education* (summary tables for the 2023/24 school year). Republic of Türkiye, MoNE Strategy Development Presidency. Retrieved January 20, 2026, from `https://sgm.meb.gov.tr/`
- Montgomery Planning. (2018). *Transportation terminology*. https://montgomeryplanning.org/wp-content/uploads/2018/03/Forest-Glen-Transportation-Terminology.pdf
- National Association of City Transportation Officials. (n.d.). *Literature review on vehicle travel speeds and pedestrian injuries*. `https://nacto.org/wp-content/uploads/literature_review_on_vehicle_travel_speeds_leaf.pdf`
- Olmos, L. E., Čolak, H. S., Shafiei, S., Saberi, M., & González, M. C. (2018). Macroscopic dynamics and the collapse of urban traffic. *Proceedings of the National Academy of Sciences (PNAS)*, *115*(46). https://www.pnas.org/doi/10.1073/pnas.1800474115
- Organisation for Economic Co-operation and Development. (2023). *Access to public transport* [Data set]. OECD Data Explorer. https://data-explorer.oecd.org/vis?...
- PubMed Central. (2021). *Red light camera interventions for reducing traffic violations and traffic crashes: A systematic review*. https://pmc.ncbi.nlm.nih.gov/articles/PMC8356316/
- Qucit. (n.d.). *E-scooter share vs. bike share: Understanding the challenges and choosing the right shared mobility model*. https://qucit.com/en/news/e-scooter-share-vs-bike-share-understanding-the-challenges-and-choosing-the-right-shared-mobility-model
- Reason Foundation. (n.d.). *Detailed methodology*. https://reason.org/wp-content/uploads/files/appendixb.pdf
- Shah, N. R., Guo, Y., & Cherry, C. R. (2021). *Why do people take e-scooter trips? Insights on temporal and spatial usage patterns*. SSRN. https://doi.org/10.2139/ssrn.3988137
- Texas A&M Transportation Institute. (n.d.). *Area-wide real-time traffic control*. https://library.ctr.utexas.edu/hostedpdfs/tti/1245-4.pdf
- The Geography of Transport Systems. (n.d.). *Levels of service for road transportation*. https://transportgeography.org/contents/methods/transport-technical-economic-performance-indicators/levels-of-service-road-transportation/
- TomTom. (n.d.-a). *About the TomTom traffic index*. https://www.tomtom.com/traffic-index/about/

- TomTom. (n.d.-b). *TomTom traffic index: Live traffic statistics and historical data.* https://www.tomtom.com/traffic-index/
- TomTom. (n.d.-c). *Istanbul traffic report: TomTom traffic index.* https://www.tomtom.com/traffic-index/istanbul-traffic/
- Town of Babylon. (n.d.). *Level of service definitions.* `https://www.townofbabylonny.gov/DocumentCenter/View/10208/Appendix-C_Level-of-Service-Definitions`
- Turkish Statistical Institute (TurkStat). (2025). *Address Based Population Registration System Results, 2024* (press release / official tables). Retrieved January 20, 2026, from `https://data.tuik.gov.tr/`
- Virginia Transportation Research Council. (n.d.). *Modeling travel time reliability for non-interstate national highway system routes.* https://vtrc.virginia.gov/media/vtrc/vtrc-pdf/vtrc-pdf/26-R23.pdf
- Yang, H., & Ma, X. (2024). Spatiotemporal evolution of shared e-scooter usage patterns: A case study of Chicago. *PLOS ONE, 19*(5), e0302388. https://doi.org/10.1371/journal.pone.0302388
- Zukowski, D. (2024, July 11). *Report: E-scooters and e-bikes reduce traffic congestion, commuting time.* Smart Cities Dive. https://www.smartcitiesdive.com/news/archive-acc-report-e-scooters-and-e-bikes-reduce-traffic-congestion-commuting-time/754570/

.

# A Code Listings

Student percentages percentiles code:

```
from __future__ import annotations

import json
import math
import random
import time
from dataclasses import dataclass
from pathlib import Path
from typing import Any, Dict, List, Optional, Tuple

import pandas as pd
import requests
from tqdm import tqdm

N_FINAL = 200
N_CANDIDATES = 1000
YEAR_WORLDPOP = 2020
SEED = 42

POP_BINS = 5

WORLDPOP_RUNASYNC = False
WORLDPOP_TIMEOUT_S = 30

SLEEP_BETWEEN_CALLS = 0.05

OUT_DIR = Path("immc_district_students")
OUT_DIR.mkdir(exist_ok=True)

OUT_CSV = OUT_DIR / "district_student_share_ranked.csv"
OUT_PERCENTILES = OUT_DIR / "percentiles.txt"
CACHE_DIR = OUT_DIR / "cache"
CACHE_DIR.mkdir(exist_ok=True)

random.seed(SEED)

GEOB_ADM2_META_URL =
    "https://www.geoboundaries.org/api/current/gbOpen/ALL/ADM2/"
WORLDPOP_STATS_URL = "https://api.worldpop.org/v1/services/stats"
```

```
WORLDPOP_TASK_URL = "https://api.worldpop.org/v1/tasks/{}"
WB_INDICATOR_SEARCH = "https://api.worldbank.org/v2/indicator"
WB_WLD_SERIES = "https://api.worldbank.org/v2/country/WLD/indicator/{}"

@dataclass
class DistrictFeature:
    iso3: str
    name: str
    boundary_id: str
    geometry: Dict[str, Any]

def get_json(url: str, params: Optional[Dict[str, Any]] = None, retries:
    int = 3) -> Any:
    last_err = None
    for i in range(retries):
        try:
            r = requests.get(url, params=params, timeout=60)
            r.raise_for_status()
            return r.json()
        except Exception as e:
            last_err = e
            time.sleep(0.8 * (i + 1))
    raise RuntimeError(f"GET␣failed:␣{url}␣params={params}␣err={last_err}")

def wb_find_indicator_code(preferred_names: List[str]) -> Optional[str]:
    search_terms = [
        "Gross␣enrolment␣ratio,␣upper␣secondary,␣both␣sexes",
        "Gross␣enrollment␣ratio,␣upper␣secondary,␣both␣sexes",
        "School␣enrollment,␣upper␣secondary"
    ]

    for term in search_terms:
        data = get_json(
            WB_INDICATOR_SEARCH,
            params={"format": "json", "per_page": 20000, "search": term},
        )
        if isinstance(data, list) and len(data) >= 2 and data[1]:
            for ind in data[1]:
                nm = (ind.get("name") or "").strip()
                code = (ind.get("id") or "").strip()
                if nm in preferred_names:
                    return code
            for ind in data[1]:
                nm = (ind.get("name") or "").strip().lower()
```

```
                code = (ind.get("id") or "").strip()
                if "upper␣secondary" in nm and ("enrol" in nm or "enroll"
                    in nm):
                    return code
    return None

def wb_world_latest_value(indicator_code: str) -> Tuple[Optional[float],
    Optional[str]]:
    data = get_json(
        WB_WLD_SERIES.format(indicator_code),
        params={"format": "json", "per_page": 20000},
    )
    if not (isinstance(data, list) and len(data) >= 2 and data[1]):
        return None, None

    rows = [r for r in data[1] if r.get("value") is not None]
    if not rows:
        return None, None

    rows.sort(key=lambda x: int(x.get("date", "0")), reverse=True)
    return float(rows[0]["value"]), str(rows[0]["date"])

def load_world_enrollment_ratios() -> Dict[str, Any]:
    uni_code = "SE.TER.ENRR"

    hs_preferred_names = [
        "Gross␣enrolment␣ratio,␣upper␣secondary,␣both␣sexes␣(%)",
        "Gross␣enrollment␣ratio,␣upper␣secondary,␣both␣sexes␣(%)",
    ]
    hs_code = wb_find_indicator_code(hs_preferred_names)

    uni_val, uni_year = wb_world_latest_value(uni_code)

    hs_val, hs_year = (None, None)
    if hs_code:
        hs_val, hs_year = wb_world_latest_value(hs_code)

    if hs_val is None:
        fallback_code = "SE.SEC.ENRR"
        hs_val, hs_year = wb_world_latest_value(fallback_code)
        hs_code = fallback_code

    hs_frac = None if hs_val is None else min(max(hs_val / 100.0, 0.0), 1.0)
```

```
    uni_frac = None if uni_val is None else min(max(uni_val / 100.0, 0.0),
        1.0)

    return {
        "hs_indicator_code": hs_code,
        "hs_value_percent": hs_val,
        "hs_value_year": hs_year,
        "hs_fraction": hs_frac,
        "uni_indicator_code": uni_code,
        "uni_value_percent": uni_val,
        "uni_value_year": uni_year,
        "uni_fraction": uni_frac,
    }

def load_geoboundaries_adm2_metadata() -> List[Dict[str, Any]]:
    meta = get_json(GEOB_ADM2_META_URL)
    if not isinstance(meta, list):
        raise RuntimeError("Unexpected␣geoBoundaries␣ADM2␣meta␣response.")
    return meta

def pick_country_sets(meta_list: List[Dict[str, Any]], k: int = 40) ->
    List[Dict[str, Any]]:
    candidates = []
    for m in meta_list:
        iso = m.get("boundaryISO")
        if not iso or iso == "ALL":
            continue
        if not (m.get("gjDownloadURL") or
            m.get("simplifiedGeometryGeoJSON") or
            m.get("staticDownloadLink")):
            continue
        candidates.append(m)

    random.shuffle(candidates)
    return candidates[:k]

def download_country_adm2_geojson(meta: Dict[str, Any]) -> Dict[str, Any]:
    iso = meta["boundaryISO"]
    cache_path = CACHE_DIR / f"geob_{iso}_ADM2.geojson"

    if cache_path.exists():
        return json.loads(cache_path.read_text(encoding="utf-8"))

    url = meta.get("gjDownloadURL") or meta.get("staticDownloadLink")
```

```
    if not url:
        raise RuntimeError(f"No␣geojson␣download␣URL␣found␣for␣{iso}")

    r = requests.get(url, timeout=120)
    r.raise_for_status()
    geo = r.json()

    cache_path.write_text(json.dumps(geo), encoding="utf-8")
    return geo

def worldpop_stats(dataset: str, year: int, geometry: Dict[str, Any]) ->
    Dict[str, Any]:
    feature_collection = {
        "type": "FeatureCollection",
        "features": [{"type": "Feature", "properties": {}, "geometry":
            geometry}],
    }
    geojson_str = json.dumps(feature_collection, separators=(",", ":"))

    params = {
        "dataset": dataset,
        "year": str(year),
        "geojson": geojson_str,
        "runasync": "false" if not WORLDPOP_RUNASYNC else "true",
    }

    resp = get_json(WORLDPOP_STATS_URL, params=params)

    if resp.get("status") == "finished" and not resp.get("error", False):
        return resp["data"]

    if resp.get("status") in ("created", "queued") and resp.get("taskid"):
        taskid = resp["taskid"]
        start = time.time()
        while True:
            task = get_json(WORLDPOP_TASK_URL.format(taskid))
            if task.get("status") == "finished" and not task.get("error",
                False):
                return task["data"]
            if task.get("error", False):
                raise RuntimeError(f"WorldPop␣task␣error:␣
                    {task.get('error_message')}")
            if time.time() - start > 180:
                raise TimeoutError("WorldPop␣task␣timeout␣(3␣minutes).")
```

```
            time.sleep(1.0)

    raise RuntimeError(f"WorldPop␣unexpected␣response:␣{resp}")

def extract_total_population(data: Dict[str, Any]) -> Optional[float]:
    return data.get("total_population")

def extract_age_bins(data: Dict[str, Any]) -> Dict[str, float]:
    out = {}
    pyramid = data.get("agesexpyramid", [])
    if not pyramid:
        return out

    for row in pyramid:
        age = row.get("age")
        if not age:
            continue
        male = float(row.get("male", 0.0) or 0.0)
        female = float(row.get("female", 0.0) or 0.0)
        out[age] = male + female
    return out

def build_candidate_pool(meta_list: List[Dict[str, Any]], n_candidates:
    int) -> List[DistrictFeature]:
    country_sets = pick_country_sets(meta_list, k=60)

    pool: List[DistrictFeature] = []

    for m in tqdm(country_sets, desc="Downloading␣ADM2␣boundaries"):
        iso = m.get("boundaryISO", "UNK")
        try:
            geo = download_country_adm2_geojson(m)
        except Exception:
            continue

        feats = geo.get("features", [])
        if not feats:
            continue

        random.shuffle(feats)
        take = max(3, min(20, len(feats) // 50 + 3))
        for f in feats[:take]:
            geom = f.get("geometry")
            props = f.get("properties", {}) or {}
```

```
            name = (
                props.get("shapeName")
                or props.get("ADM2_NAME")
                or props.get("name")
                or props.get("NAME_2")
                or props.get("shapeGroup")
                or "UNKNOWN_ADM2"
            )

            boundary_id = (
                props.get("shapeID")
                or props.get("shapeID_1")
                or props.get("boundaryID")
                or props.get("gbID")
                or f"{iso}_{name}"
            )

            if not geom:
                continue

            pool.append(DistrictFeature(iso3=iso, name=str(name),
                boundary_id=str(boundary_id), geometry=geom))

            if len(pool) >= n_candidates:
                return pool

    return pool

def stratified_sample_by_population(
    candidates: List[DistrictFeature],
    year: int,
    n_final: int,
    bins: int,
) -> List[Tuple[DistrictFeature, float]]:
    rows: List[Tuple[DistrictFeature, float]] = []

    for feat in tqdm(candidates, desc="WorldPop total pop (candidates)"):
        cache_path = CACHE_DIR / f"pop_{feat.iso3}_{feat.boundary_id}.json"
        if cache_path.exists():
            pop =
                json.loads(cache_path.read_text(encoding="utf-8")).get("total_population")
        else:
            try:
```

```
                data = worldpop_stats("wpgppop", year, feat.geometry)
                pop = extract_total_population(data)
                cache_path.write_text(json.dumps({"total_population":
                    pop}), encoding="utf-8")
                time.sleep(SLEEP_BETWEEN_CALLS)
            except Exception:
                continue

        if pop is None or pop <= 0:
            continue

        rows.append((feat, float(pop)))

    if len(rows) < n_final:
        return rows[:n_final]

    pops = [p for _, p in rows]
    lo = math.log10(min(pops))
    hi = math.log10(max(pops))
    if hi - lo < 1e-6:
        random.shuffle(rows)
        return rows[:n_final]

    edges = [lo + (hi - lo) * i / bins for i in range(bins + 1)]

    binned: List[List[Tuple[DistrictFeature, float]]] = [[] for _ in
        range(bins)]
    for feat, pop in rows:
        x = math.log10(pop)
        idx = min(bins - 1, max(0, int((x - lo) / (hi - lo) * bins)))
        binned[idx].append((feat, pop))

    per_bin = max(1, n_final // bins)
    chosen: List[Tuple[DistrictFeature, float]] = []
    for b in binned:
        random.shuffle(b)
        chosen.extend(b[:per_bin])

    if len(chosen) < n_final:
        remaining = []
        chosen_ids = {c[0].boundary_id for c in chosen}
        for feat, pop in rows:
            if feat.boundary_id not in chosen_ids:
                remaining.append((feat, pop))
```

```
        random.shuffle(remaining)
        chosen.extend(remaining[: (n_final - len(chosen))])

    return chosen[:n_final]

def compute_student_share_for_districts(
    selected: List[Tuple[DistrictFeature, float]],
    year: int,
    hs_frac: float,
    uni_frac: float,
) -> pd.DataFrame:
    out_rows = []

    for feat, pop_total in tqdm(selected, desc="WorldPop␣age/sex␣
        (selected)"):
        cache_path = CACHE_DIR / f"ages_{feat.iso3}_{feat.boundary_id}.json"
        if cache_path.exists():
            ages = json.loads(cache_path.read_text(encoding="utf-8"))
        else:
            try:
                data = worldpop_stats("wpgpas", year, feat.geometry)
                ages = extract_age_bins(data)
                cache_path.write_text(json.dumps(ages), encoding="utf-8")
                time.sleep(SLEEP_BETWEEN_CALLS)
            except Exception:
                continue

        pop_15_20 = float(ages.get("15␣to␣20", 0.0) or 0.0)
        pop_20_25 = float(ages.get("20␣to␣25", 0.0) or 0.0)

        share_15_19 = pop_15_20 / pop_total if pop_total > 0 else 0.0
        share_20_24 = pop_20_25 / pop_total if pop_total > 0 else 0.0

        hs_student_share = hs_frac * share_15_19
        uni_student_share = uni_frac * share_20_24
        total_student_share = hs_student_share + uni_student_share

        out_rows.append(
            {
                "iso3": feat.iso3,
                "district_name": feat.name,
                "district_id": feat.boundary_id,
                "population_total_est": pop_total,
                "share_15_19_est": share_15_19,
```

```
                "share_20_24_est": share_20_24,
                "hs_student_share_est": hs_student_share,
                "uni_student_share_est": uni_student_share,
                "total_student_share_est": total_student_share,
            }
        )

    df = pd.DataFrame(out_rows)
    if df.empty:
        return df

    df = df.sort_values("total_student_share_est",
        ascending=True).reset_index(drop=True)

    mn = df["total_student_share_est"].min()
    mx = df["total_student_share_est"].max()
    if mx > mn:
        df["relative_usage_index_0_1"] = (df["total_student_share_est"] -
            mn) / (mx - mn)
    else:
        df["relative_usage_index_0_1"] = 0.0

    return df

def save_percentiles(df: pd.DataFrame, path: Path) -> None:
    qs = [0.0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1]
    qvals = df["total_student_share_est"].quantile(qs).to_dict()

    lines = ["Percentiles␣for␣total_student_share_est␣(HS+Uni):"]
    for q in qs:
        lines.append(f"␣␣{int(q*100):>3d}th␣percentile:␣{qvals[q]:.6f}")
    path.write_text("\n".join(lines), encoding="utf-8")

def main():
    print("1)␣Loading␣global␣enrollment␣ratios␣(World␣Bank,␣WLD)...")
    enr = load_world_enrollment_ratios()

    hs_frac = enr["hs_fraction"]
    uni_frac = enr["uni_fraction"]

    print("\nWorld␣enrollment␣ratios␣used:")
    print(f"␣␣HS␣indicator:␣␣{enr['hs_indicator_code']}␣␣
        value={enr['hs_value_percent']}␣␣year={enr['hs_value_year']}")
```

```
print(f"  UNI indicator: {enr['uni_indicator_code']}
    value={enr['uni_value_percent']} year={enr['uni_value_year']}")
print(f"  HS fraction used (capped 0..1):   {hs_frac}")
print(f"  UNI fraction used (capped 0..1): {uni_frac}")

if hs_frac is None or uni_frac is None:
    raise RuntimeError("Could not load enrollment ratios from World
        Bank.")

print("\n2) Loading geoBoundaries ADM2 metadata...")
meta_list = load_geoboundaries_adm2_metadata()
print(f"  ADM2 country-sets found: {len(meta_list)}")

print("\n3) Building candidate district pool...")
candidates = build_candidate_pool(meta_list, N_CANDIDATES)
print(f"  Candidate districts collected: {len(candidates)}")

if len(candidates) < N_FINAL:
    raise RuntimeError("Not enough district candidates to continue.")

print("\n4) Stratified sampling by population (WorldPop wpgppop)...")
selected = stratified_sample_by_population(candidates, YEAR_WORLDPOP,
    N_FINAL, POP_BINS)
print(f"  Selected districts: {len(selected)}")

print("\n5) Fetching age/sex structure and computing student shares
    (WorldPop wpgpas)...")
df = compute_student_share_for_districts(selected, YEAR_WORLDPOP,
    hs_frac, uni_frac)

if df.empty:
    raise RuntimeError("No results computed. Possibly API failures or
        blocked downloads.")

print("\n6) Saving ranked CSV + percentile report...")
df.to_csv(OUT_CSV, index=False, encoding="utf-8")
save_percentiles(df, OUT_PERCENTILES)

print("\nDONE    ")
print(f"Ranked output CSV: {OUT_CSV}")
print(f"Percentiles file:   {OUT_PERCENTILES}")
print("\nPreview (top 5 lowest student share):")
print(df.head(5)[["iso3", "district_name", "population_total_est",
    "total_student_share_est", "relative_usage_index_0_1"]])
```

```
    print("\nPreview␣(top␣5␣highest␣student␣share):")
    print(df.tail(5)[["iso3", "district_name", "population_total_est",
        "total_student_share_est", "relative_usage_index_0_1"]])

if __name__ == "__main__":
    main()
```

Monthly tourism percentiles code:

```
import argparse
import gzip
import re
from pathlib import Path

import numpy as np
import pandas as pd
import requests


# Dataset: Nights spent at tourist accommodation establishments
# by month and NUTS 2 region (from 2020 onwards)
DATASET = "tour_occ_nin2m"

BASE_URL = (
    "https://ec.europa.eu/eurostat/api/dissemination/sdmx/2.1/data/"
    f"{DATASET}?format=TSV&compressed=true"
)

PERCENTILES = [0, 12.5, 25, 37.5, 50, 62.5, 75, 87.5, 100]


def download(url: str, out_path: Path, timeout: int = 120) -> None:
    out_path.parent.mkdir(parents=True, exist_ok=True)
    if out_path.exists() and out_path.stat().st_size > 1000:
        return

    with requests.get(url, stream=True, timeout=timeout) as r:
        r.raise_for_status()
        with open(out_path, "wb") as f:
            for chunk in r.iter_content(chunk_size=1024 * 1024):
                if chunk:
                    f.write(chunk)


def sniff_first_line_gz(path: Path) -> str:
```

```
    with gzip.open(path, "rb") as f:
        line = f.readline()
    return line.decode("utf-8-sig", errors="replace").strip("\n\r")


def parse_numeric_cell(x: str) -> float:
    """
␣␣␣␣Eurostat␣values␣can␣be␣like:
␣␣␣␣␣␣␣"1234",␣"1234 e",␣":"␣,␣": c"
␣␣␣␣We␣keep␣the␣first␣numeric␣token;␣missing␣->␣NaN.
␣␣␣␣"""
    if x is None:
        return np.nan
    s = str(x).strip()
    if not s or s.startswith(":"):
        return np.nan
    m = re.match(r"^-?\d+(?:\.\d+)?", s)
    if not m:
        return np.nan
    try:
        return float(m.group(0))
    except Exception:
        return np.nan


def choose_preferred_value(series: pd.Series, preferred: list[str]):
    opts = sorted(series.dropna().astype(str).unique().tolist())
    for p in preferred:
        if p in opts:
            return p
    if len(opts) == 1:
        return opts[0]
    return None


def month_sort_key(m: str) -> int:
    """
␣␣␣␣month␣codes␣may␣look␣like:
␣␣␣␣␣␣␣"M01".."M12"
␣␣␣␣␣␣␣"01".."12"
␣␣␣␣␣␣␣"Jan"␣etc.␣(unlikely)
␣␣␣␣We'll␣robustly␣parse␣digits.
␣␣␣␣"""
    s = str(m).strip()
```

```
    digits = re.findall(r"\d+", s)
    if digits:
        val = int(digits[0])
        if 1 <= val <= 12:
            return val
    return 999


def load_tsv_gz(path: Path) -> pd.DataFrame:
    first_line = sniff_first_line_gz(path)
    if "\t" not in first_line:
        raise RuntimeError(
            "Downloaded file is not a TSV (maybe an API error payload).\n"
            f"First line preview:\n{first_line[:200]}"
        )

    raw = pd.read_csv(path, sep="\t", compression="gzip", dtype=str,
        low_memory=False)
    raw.columns = [str(c).strip().lstrip("\ufeff") for c in raw.columns]

    first_col_name = raw.columns[0]
    left = first_col_name.split("\\")[0]  #
        "freq,c_resid,nace_r2,month,unit,geo"
    dim_names = [d.strip().lower() for d in left.split(",")]

    key_series = raw.iloc[:, 0].fillna("").astype(str)
    parts = key_series.str.split(",", expand=True)
    if parts.shape[1] != len(dim_names):
        raise RuntimeError(
            f"Dimension parsing mismatch.\nHeader dims: {dim_names}\nParsed
                dims: {parts.shape[1]}"
        )

    parts.columns = dim_names
    df = pd.concat([parts, raw.iloc[:, 1:]], axis=1)

    # Strip spaces in all column names
    df.columns = [str(c).strip() for c in df.columns]
    return df


def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--year", type=int, required=True, help="Example: 2024")
```

```
ap.add_argument("--outdir", default="out_tourism", help="Output folder")
args = ap.parse_args()

outdir = Path(args.outdir)
outdir.mkdir(parents=True, exist_ok=True)

cache_dir = Path("cache_eurostat")
cache_dir.mkdir(exist_ok=True)
gz_path = cache_dir / f"{DATASET}.tsv.gz"

print(f"[download] {BASE_URL}")
download(BASE_URL, gz_path)

print(f"[load] {gz_path}")
df = load_tsv_gz(gz_path)

# --- Slice the dataset to a single consistent meaning ---
# unit=NR (nights), c_resid=TOTAL (dom+for), nace_r2=I551-I553 (hotels
    etc.)
if "unit" in df.columns:
    unit_pick = choose_preferred_value(df["unit"], ["NR"])
    if unit_pick:
        df = df[df["unit"] == unit_pick]
        print(f"[slice] unit={unit_pick}")

if "c_resid" in df.columns:
    cres_pick = choose_preferred_value(df["c_resid"], ["TOTAL"])
    if cres_pick:
        df = df[df["c_resid"] == cres_pick]
        print(f"[slice] c_resid={cres_pick}")

if "nace_r2" in df.columns:
    nace_pick = choose_preferred_value(df["nace_r2"], ["I551-I553",
        "TOTAL"])
    if nace_pick:
        df = df[df["nace_r2"] == nace_pick]
        print(f"[slice] nace_r2={nace_pick}")

# --- Validate expected structure ---
needed = ["month", "geo"]
for c in needed:
    if c not in df.columns:
        raise RuntimeError(f"Dataset does not contain required column:
            {c}")
```

```
year_col = str(args.year)
if year_col not in df.columns:
    raise RuntimeError(
        f"Year␣column␣'{year_col}'␣not␣found.\n"
        f"Available␣year␣columns:␣{[c␣for␣c␣in␣df.columns␣if␣
            re.fullmatch(r'\\d{4}',␣c)]}"
    )

# Convert that year's column to numeric
df[year_col] = df[year_col].map(parse_numeric_cell)
df = df.dropna(subset=[year_col])

# List months
months = sorted(df["month"].dropna().unique().tolist(),
    key=month_sort_key)
if not months:
    raise RuntimeError("No␣months␣found␣in␣'month'␣column.")

rankings_dir = outdir / f"rankings_{args.year}"
rankings_dir.mkdir(parents=True, exist_ok=True)

qs = [p / 100.0 for p in PERCENTILES]
pct_rows = []

for m in months:
    dm = df[df["month"] == m][["geo", year_col]].copy()
    if dm.empty:
        continue

    dm = dm.rename(columns={year_col: "nights"})
    dm["nights"] = pd.to_numeric(dm["nights"], errors="coerce")
    dm = dm.dropna(subset=["nights"])

    if dm.empty:
        continue

    # Aggregate in case duplicates exist
    dm = dm.groupby("geo", as_index=False)["nights"].sum()

    # Rank low->high
    dm = dm.sort_values("nights", ascending=True).reset_index(drop=True)
    dm.insert(0, "rank_low_to_high", np.arange(1, len(dm) + 1))
```

```
        # Save per-month ranking
        safe_month = str(m).replace("/", "_").replace("␣", "")
        rank_path = rankings_dir / f"ranking_{args.year}_{safe_month}.csv"
        dm.to_csv(rank_path, index=False, encoding="utf-8")
        print(f"[ok]␣{args.year}␣{m}␣->␣{rank_path.name}")

        vals = dm["nights"].to_numpy(dtype=float)
        quants = np.nanquantile(vals, qs)

        row = {
            "year": args.year,
            "month": m,
            "n_regions": int(np.isfinite(vals).sum())
        }
        for p, v in zip(PERCENTILES, quants):
            row[f"p{str(p).replace('.',␣'_')}"] = float(v)
        pct_rows.append(row)

    pct_df = pd.DataFrame(pct_rows)

    pct_path = outdir / f"percentiles_by_month_{args.year}.csv"
    pct_df.to_csv(pct_path, index=False, encoding="utf-8")

    print("\n===␣DONE␣   ␣===")
    print(f"Rankings␣folder:␣{rankings_dir}")
    print(f"Percentiles␣CSV:␣{pct_path}")


if __name__ == "__main__":
    main()
```

Genetic algorithm code:

```
#!/usr/bin/env python3

from __future__ import annotations

import math
import random
import statistics
import threading
import time
from dataclasses import dataclass
from datetime import date, timedelta
from typing import Dict, List, Tuple
```

```
try:
    import tkinter as tk
    from tkinter import ttk, messagebox
except Exception:
    tk = None

TOTAL_SCOOTERS = 500
MAX_TRIPS_PER_SCOOTER_PER_DAY = 6
TOTAL_DAILY_DEMAND = TOTAL_SCOOTERS * MAX_TRIPS_PER_SCOOTER_PER_DAY

MOVE_COST_PER_SCOOTER = 12.0

FULL_RANGE_KM = 25.0
FULL_CHARGE_COST_TL = 8.0

AVG_SPEED_KMH = 15.4
AVG_TRIP_MINUTES = 18.0 / 6.0

STUDENT_PROPENSITY_MULTIPLIER = 1.30
TOURIST_PROPENSITY_MULTIPLIER = 0.65

WEEKEND_STUDENT_ACTIVITY_MULT = 0.90
WEEKEND_TOURISM_ACTIVITY_MULT = 1.10

STUDENT_SEASON_ACTIVITY = {
    "Winter": 1.00,
    "Spring": 1.00,
    "Summer": 0.55,
    "Autumn": 0.90,
}

DAILY_DRIFT_FRACTION = 0.35
EPS_TOUR_DRIFT = 0.15

ZONES = ["A", "B", "C", "D", "E"]

NEIGHBORS = {z: [k for k in ZONES if k != z] for z in ZONES}

TOURISM_MONTHLY = {
    "M01": (14268.0, 67412.0, 244187.0, 915744.0),
    "M02": (16047.0, 77304.0, 276405.0, 1060456.0),
    "M03": (18955.0, 99861.0, 328961.0, 1301046.0),
    "M04": (25088.0, 123344.0, 378306.0, 1480225.0),
    "M05": (31539.0, 175326.0, 510443.0, 2114958.0),
```

```
    "M06": (40095.0, 227524.0, 643431.0, 2467103.0),
    "M07": (58398.0, 326249.0, 886628.0, 3358671.0),
    "M08": (55644.0, 318426.0, 902701.0, 3771915.0),
    "M09": (34025.0, 210570.0, 579989.0, 2365495.0),
    "M10": (29003.0, 138430.0, 433617.0, 1784252.0),
    "M11": (17607.0, 91492.0, 290631.0, 1035663.0),
    "M12": (16872.0, 81993.0, 290334.0, 1126539.0),
}

TOURISM_MONTHLY_P50 = {
    "M01": 127559.0,
    "M02": 139955.0,
    "M03": 177776.0,
    "M04": 211020.0,
    "M05": 298541.0,
    "M06": 380704.0,
    "M07": 549296.0,
    "M08": 562832.0,
    "M09": 345798.0,
    "M10": 235559.0,
    "M11": 154990.0,
    "M12": 145031.0,
}

STUDENT_SHARE_LEVELS = {
    "low": 0.055527,
    "mid": 0.075871,
    "high": 0.095667,
    "vhigh": 0.113784,
}

PT_5MIN_LEVELS = {
    "weak": 53.2625,
    "mid": 81.9875,
    "good": 94.0125,
    "vgood": 97.9,
}

PASSENGER_LEVELS = {
    "low": 12.5,
    "mid": 37.5,
    "high": 62.5,
    "vhigh": 87.5,
}
```

```
TOURISM_LEVELS = {
    "low": 12.5,
    "mid": 37.5,
    "high": 62.5,
    "vhigh": 87.5,
}

ZONE_CATS = {
    "A": {"pass": "vhigh", "stud": "mid", "tour": "low", "pt": "vgood",
        "traf": "vhigh"},
    "B": {"pass": "mid", "stud": "high", "tour": "low", "pt": "mid",
        "traf": "mid"},
    "C": {"pass": "low", "stud": "low", "tour": "vhigh", "pt": "weak",
        "traf": "low"},
    "D": {"pass": "mid", "stud": "mid", "tour": "mid", "pt": "good",
        "traf": "mid"},
    "E": {"pass": "low", "stud": "high", "tour": "low", "pt": "weak",
        "traf": "low"},
}

TRAFFIC_LEVELS = {
    "low": 12.5,
    "mid": 37.5,
    "high": 62.5,
    "vhigh": 87.5,
}

TRAFFIC_TIMELOSS_ANCHORS = {
    "low": 0.050012,
    "mid": 0.250089,
    "high": 0.457209,
    "vhigh": 1.000045,
}

BETA_PT = 0.20
BETA_TRAFFIC = 0.15

DEMAND_SIGMA = 0.10

SEASONS = ["Winter", "Spring", "Summer", "Autumn"]
SEASON_MONTHS = {
    "Winter": ["M12", "M01", "M02"],
    "Spring": ["M03", "M04", "M05"],
```

```
    "Summer": ["M06", "M07", "M08"],
    "Autumn": ["M09", "M10", "M11"],
}

def clamp(x: float, lo: float, hi: float) -> float:
    return max(lo, min(hi, x))

def sigmoid(x: float) -> float:
    if x >= 0:
        z = math.exp(-x)
        return 1.0 / (1.0 + z)
    z = math.exp(x)
    return z / (1.0 + z)

def softmax(vec: List[float]) -> List[float]:
    m = max(vec)
    exps = [math.exp(v - m) for v in vec]
    s = sum(exps)
    if s <= 0:
        return [1.0 / len(vec)] * len(vec)
    return [e / s for e in exps]

def round_to_total(weights: List[float], total: int) -> List[int]:
    if total <= 0:
        return [0] * len(weights)
    s = sum(weights)
    if s <= 1e-12:
        base = total // len(weights)
        out = [base] * len(weights)
        for i in range(total - base * len(weights)):
            out[i] += 1
        return out

    scaled = [w / s * total for w in weights]
    flo = [int(math.floor(x)) for x in scaled]
    rem = total - sum(flo)
    fracs = sorted([(scaled[i] - flo[i], i) for i in range(len(weights))],
        reverse=True)
    for k in range(rem):
        flo[fracs[k % len(fracs)][1]] += 1
    return flo

def season_from_month(m: int) -> str:
    if m in (12, 1, 2):
```

```
        return "Winter"
    if m in (3, 4, 5):
        return "Spring"
    if m in (6, 7, 8):
        return "Summer"
    return "Autumn"

def is_weekend(dt: date) -> bool:
    return dt.weekday() >= 5

def seasonal_tourism_intensity_factor() -> Dict[str, float]:
    season_p50: Dict[str, float] = {}
    for sname, months in SEASON_MONTHS.items():
        season_p50[sname] = sum(TOURISM_MONTHLY_P50[m] for m in months) /
            len(months)
    mx = max(season_p50.values())
    if mx <= 1e-12:
        return {s: 0.0 for s in SEASONS}
    return {s: season_p50[s] / mx for s in SEASONS}

SEASON_TOUR_INTENSITY = seasonal_tourism_intensity_factor()

def _tourism_rank(z: str) -> float:
    return TOURISM_LEVELS[ZONE_CATS[z]["tour"]] / 100.0

def tourism_fraction_by_zone(season: str, daytype: str) -> Dict[str, float]:
    base = {z: _tourism_rank(z) for z in ZONES}
    s = sum(base.values())
    if s <= 1e-12:
        rel = {z: 1.0 / len(ZONES) for z in ZONES}
    else:
        rel = {z: base[z] / s for z in ZONES}

    inten = SEASON_TOUR_INTENSITY[season]
    wk = WEEKEND_TOURISM_ACTIVITY_MULT if daytype == "WE" else 1.0

    frac = {z: clamp(rel[z] * inten * wk, 0.0, 1.0) for z in ZONES}
    return frac

def zone_feature_scores(
    season: str, daytype: str
) -> Tuple[Dict[str, float], Dict[str, float], Dict[str, float], Dict[str,
    float]]:
    Pn = {z: PASSENGER_LEVELS[ZONE_CATS[z]["pass"]] for z in ZONES}
```

```
    S = {z: STUDENT_SHARE_LEVELS[ZONE_CATS[z]["stud"]] for z in ZONES}

    PT_raw = {z: PT_5MIN_LEVELS[ZONE_CATS[z]["pt"]] for z in ZONES}
    _pt_min = min(PT_raw.values())
    _pt_max = max(PT_raw.values())
    PTnorm = {z: 0.0 if _pt_max == _pt_min else (PT_raw[z] - _pt_min) /
        (_pt_max - _pt_min) for z in ZONES}

    traffic_raw = {z: TRAFFIC_TIMELOSS_ANCHORS[ZONE_CATS[z]["traf"]] for z
        in ZONES}
    _traf_min = min(traffic_raw.values())
    _traf_max = max(traffic_raw.values())
    TrafficScore = {z: 0.0 if _traf_max == _traf_min else (traffic_raw[z] -
        _traf_min) / (_traf_max - _traf_min) for z in ZONES}

    stu_day_mult = WEEKEND_STUDENT_ACTIVITY_MULT if daytype == "WE" else 1.0
    stu_season_mult = STUDENT_SEASON_ACTIVITY[season]

    Tfrac = tourism_fraction_by_zone(season, daytype)

    PassEff: Dict[str, float] = {}
    for z in ZONES:
        local_frac = 1.0 - Tfrac[z]
        student_effect = (S[z] * STUDENT_PROPENSITY_MULTIPLIER *
            stu_day_mult * stu_season_mult)
        local_multiplier = (1.0 - S[z]) + student_effect

        tourist_multiplier = TOURIST_PROPENSITY_MULTIPLIER

        PassEff[z] = Pn[z] * (local_frac * local_multiplier + Tfrac[z] *
            tourist_multiplier)

    TourAttr = {z: _tourism_rank(z) for z in ZONES}

    return PassEff, PTnorm, TourAttr, TrafficScore

def _to_share(d: Dict[str, float]) -> Dict[str, float]:
    s = sum(d.values())
    if s <= 1e-12:
        return {z: 1.0 / len(ZONES) for z in ZONES}
    return {z: d[z] / s for z in ZONES}

def demand_probabilities(season: str, daytype: str) -> Dict[str, float]:
    PassEff, PTnorm, _, TrafficScore = zone_feature_scores(season, daytype)
```

```
    p_pass = _to_share(PassEff)

    intensity = {
        z: p_pass[z]
        * (1.0 + BETA_PT * PTnorm[z])
        * (1.0 + BETA_TRAFFIC * TrafficScore[z])
        for z in ZONES
    }
    shares = _to_share(intensity)

    eps = 1e-12
    shares = {z: max(shares[z], eps) for z in ZONES}
    s = sum(shares.values())
    return {z: shares[z] / s for z in ZONES}

def rebalance_partial(scooters: Dict[str, int], target: Dict[str, int], r:
    float) -> Tuple[Dict[str, int], int]:
    r = clamp(r, 0.0, 1.0)
    s_new_float = {z: scooters[z] + r * (target[z] - scooters[z]) for z in
        ZONES}
    new_counts = round_to_total([s_new_float[z] for z in ZONES],
        TOTAL_SCOOTERS)
    s_new = {z: new_counts[i] for i, z in enumerate(ZONES)}
    moved = int(0.5 * sum(abs(s_new[z] - scooters[z]) for z in ZONES))
    return s_new, moved

def generate_daily_demand(rng: random.Random, probs: Dict[str, float],
    total: int) -> Dict[str, int]:
    raw = []
    for z in ZONES:
        p = probs[z]
        mu = total * p
        sd = max(1.0, DEMAND_SIGMA * mu)
        raw.append(max(0.0, rng.gauss(mu, sd)))

    s = sum(raw)
    if s <= 1e-12:
        return {z: total // len(ZONES) for z in ZONES}
    scaled = [x / s * total for x in raw]
    ints = round_to_total(scaled, total)
    return {z: ints[i] for i, z in enumerate(ZONES)}

def realized_trips(demand: Dict[str, int], scooters: Dict[str, int]) ->
```

```
    Dict[str, int]:
    out = {}
    for z in ZONES:
        cap = MAX_TRIPS_PER_SCOOTER_PER_DAY * scooters[z]
        out[z] = min(demand[z], cap)
    return out

def charging_cost_km(trips: Dict[str, int]) -> float:
    total_trips = sum(trips.values())
    ride_min = total_trips * AVG_TRIP_MINUTES
    km_today = AVG_SPEED_KMH * (ride_min / 60.0)
    return FULL_CHARGE_COST_TL * (km_today / FULL_RANGE_KM)

def drift_step(
    rng: random.Random,
    scooters: Dict[str, int],
    trips: Dict[str, int],
    season: str,
    daytype: str
) -> Dict[str, int]:
    m = DAILY_DRIFT_FRACTION
    if m <= 0:
        return scooters.copy()

    total_trips = sum(trips.values())
    if total_trips <= 0:
        return scooters.copy()

    trip_share = {z: trips[z] / total_trips for z in ZONES}

    u = tourism_fraction_by_zone(season, daytype)
    u_sum = sum(u.values())
    if u_sum <= 1e-12:
        tour_share = {z: 1.0 / len(ZONES) for z in ZONES}
    else:
        tour_share = {z: u[z] / u_sum for z in ZONES}

    pi_dest = {
        z: (1.0 - EPS_TOUR_DRIFT) * trip_share[z] + EPS_TOUR_DRIFT *
           tour_share[z]
        for z in ZONES
    }

    end_float = {
```

```
        z: (1.0 - m) * scooters[z] + m * TOTAL_SCOOTERS * pi_dest[z]
        for z in ZONES
    }

    end_counts = round_to_total([end_float[z] for z in ZONES],
        TOTAL_SCOOTERS)
    out = {z: end_counts[i] for i, z in enumerate(ZONES)}

    total_now = sum(out.values())
    if total_now != TOTAL_SCOOTERS:
        diff = TOTAL_SCOOTERS - total_now
        mx = max(ZONES, key=lambda z: out[z])
        out[mx] += diff

    return out

def _fmt_dist(d: Dict[str, int]) -> str:
    return "␣".join(f"{z}={d[z]}" for z in ZONES)

def simulate_year(
    rng_seed: int,
    plans: Dict[Tuple[str, str], Dict[str, int]],
    r_return: float,
    log_lines: List[str] | None = None,
    sim_name: str = "",
) -> Tuple[int, float]:
    rng = random.Random(rng_seed)
    cur = date(2024, 1, 1)
    end = date(2024, 12, 31)

    scooters = {z: TOTAL_SCOOTERS // len(ZONES) for z in ZONES}
    for i in range(TOTAL_SCOOTERS - sum(scooters.values())):
        scooters[ZONES[i]] += 1

    total_trips_year = 0
    total_cost_year = 0.0

    if log_lines is not None:
        header = f"===␣Simulation␣seed={rng_seed}␣{sim_name}␣===".strip()
        log_lines.append(header)
        log_lines.append(
            "date\tseason\tdaytype\tstart_before\tstart_after\tend_of_day\treturn_moved\ttrip
        )
```

```
    while cur <= end:
        season = season_from_month(cur.month)
        daytype = "WE" if is_weekend(cur) else "WD"
        target = plans[(season, daytype)]

        start_before = scooters.copy()

        scooters_after, moved = rebalance_partial(scooters, target,
            r_return)
        move_cost = MOVE_COST_PER_SCOOTER * moved

        probs = demand_probabilities(season, daytype)
        demand = generate_daily_demand(rng, probs, TOTAL_DAILY_DEMAND)
        trips = realized_trips(demand, scooters_after)
        trips_day = sum(trips.values())

        charge_cost = charging_cost_km(trips)

        total_trips_year += trips_day
        total_cost_year += move_cost + charge_cost

        end_of_day = drift_step(rng, scooters_after, trips, season, daytype)

        if log_lines is not None:
            log_lines.append(
                f"{cur.isoformat()}\t{season}\t{daytype}\t{_fmt_dist(start_before)}\t{_fmt_di
            )

        scooters = end_of_day
        cur += timedelta(days=1)

    if log_lines is not None:
        log_lines.append("")

    return total_trips_year, total_cost_year

@dataclass
class GAParams:
    pop_size: int = 70
    generations: int = 120
    elite: int = 2
    tournament_k: int = 3
    crossover_rate: float = 0.85
    mutation_rate: float = 0.12
```

```
    mutation_sigma: float = 0.30
    sims_per_individual: int = 5
    stall_limit: int = 18
    reseed_frac: float = 0.20

def decode_individual(genome: List[float]) -> Tuple[Dict[Tuple[str, str],
    Dict[str, int]], float]:
    assert len(genome) == 8 * 5 + 1
    r_return = sigmoid(genome[-1])

    plans: Dict[Tuple[str, str], Dict[str, int]] = {}
    idx = 0
    for season in SEASONS:
        for daytype in ("WD", "WE"):
            block = genome[idx : idx + 5]
            idx += 5
            probs = softmax(block)
            counts = round_to_total(probs, TOTAL_SCOOTERS)
            plans[(season, daytype)] = {ZONES[i]: counts[i] for i in
                range(5)}

    return plans, r_return

def smoothness_penalty(plans: Dict[Tuple[str, str], Dict[str, int]]) ->
    float:
    def l1(p: Dict[str, int], q: Dict[str, int]) -> float:
        return sum(abs(p[z] - q[z]) for z in ZONES)

    pen = 0.0
    for dt in ("WD", "WE"):
        for a, b in zip(SEASONS[:-1], SEASONS[1:]):
            pen += 0.5 * l1(plans[(a, dt)], plans[(b, dt)]) / TOTAL_SCOOTERS
    for s in SEASONS:
        pen += 0.25 * (0.5 * l1(plans[(s, "WD")], plans[(s, "WE")]) /
            TOTAL_SCOOTERS)
    return pen / (len(SEASONS) - 1 + len(SEASONS))

def evaluate_individual(genome: List[float], seeds: List[int]) ->
    Tuple[float, float, float, float]:
    plans, r_return = decode_individual(genome)
    trips_list, cost_list = [], []
    for s in seeds:
        t, c = simulate_year(s, plans, r_return)
        trips_list.append(t)
```

```
        cost_list.append(c)
    avg_trips = statistics.mean(trips_list)
    avg_cost = statistics.mean(cost_list)
    pen = smoothness_penalty(plans)
    return avg_trips, avg_cost, pen, r_return

def tournament_select(pop: List[List[float]], fitness: List[float], k: int,
    rng: random.Random) -> List[float]:
    best_idx = None
    best_f = -1e100
    for _ in range(k):
        i = rng.randrange(len(pop))
        if fitness[i] > best_f:
            best_f = fitness[i]
            best_idx = i
    assert best_idx is not None
    return pop[best_idx][:]

def crossover_blend(a: List[float], b: List[float], rng: random.Random,
    alpha: float = 0.35) -> Tuple[List[float], List[float]]:
    out1, out2 = a[:], b[:]
    for i in range(len(a)):
        lo = min(a[i], b[i])
        hi = max(a[i], b[i])
        span = hi - lo
        low = lo - alpha * span
        high = hi + alpha * span
        out1[i] = rng.uniform(low, high)
        out2[i] = rng.uniform(low, high)
    return out1, out2

def mutate(g: List[float], rate: float, sigma: float, rng: random.Random)
    -> None:
    for i in range(len(g)):
        if rng.random() < rate:
            g[i] += rng.gauss(0.0, sigma)

_DAYS_2024 = 366
MAX_TRIPS_YEAR = _DAYS_2024 * TOTAL_DAILY_DEMAND

_FULL_CHARGE_DAY = FULL_CHARGE_COST_TL * ((AVG_SPEED_KMH *
    ((TOTAL_DAILY_DEMAND * AVG_TRIP_MINUTES) / 60.0)) / FULL_RANGE_KM)
MAX_MOVE_DAY = MOVE_COST_PER_SCOOTER * TOTAL_SCOOTERS
MAX_COST_YEAR = _DAYS_2024 * (_FULL_CHARGE_DAY + MAX_MOVE_DAY)
```

```
def global_fitness(avg_trips: float, avg_cost: float, pen: float,
    smooth_weight: float = 0.12) -> float:
    t = clamp(avg_trips / MAX_TRIPS_YEAR, 0.0, 1.0)
    c = 1.0 - clamp(avg_cost / MAX_COST_YEAR, 0.0, 1.0)
    s = 1.0 - clamp(pen, 0.0, 1.0)
    return t + c + smooth_weight * s

def run_ga(params: GAParams, status_cb=None) -> Tuple[List[float],
    Dict[str, float]]:
    rng = random.Random(int(time.time() * 1000) % (2**32 - 1))
    genome_len = 8 * 5 + 1
    pop = [[rng.gauss(0, 1) for _ in range(genome_len)] for _ in
        range(params.pop_size)]

    best_genome: List[float] | None = None
    best_eval: Dict[str, float] | None = None
    best_score = -1e100
    stall = 0
    cur_sigma = params.mutation_sigma

    seeds = [1001 + 97 * i for i in range(params.sims_per_individual)]

    smooth_weight = 0.12

    for gen in range(params.generations):
        evals = [evaluate_individual(ind, seeds) for ind in pop]

        fit = [global_fitness(e[0], e[1], e[2],
            smooth_weight=smooth_weight) for e in evals]
        fit = [f + 1e-12 * rng.random() for f in fit]

        idx_gen_best = max(range(len(pop)), key=lambda i: fit[i])
        gen_best = {
            "fitness": float(fit[idx_gen_best]),
            "avg_trips": float(evals[idx_gen_best][0]),
            "avg_cost": float(evals[idx_gen_best][1]),
            "penalty": float(evals[idx_gen_best][2]),
            "r_return": float(evals[idx_gen_best][3]),
            "gen": float(gen),
        }

        if gen_best["fitness"] > best_score + 1e-15:
            best_score = gen_best["fitness"]
```

```
            best_genome = pop[idx_gen_best][:]
            best_eval = gen_best
            stall = 0
            cur_sigma = params.mutation_sigma
        else:
            stall += 1

        if status_cb and best_eval is not None:
            status_cb(gen, gen_best, best_eval)

        elite_indices = sorted(range(len(pop)), key=lambda i: fit[i],
            reverse=True)[: params.elite]

        if stall >= params.stall_limit:
            stall = 0
            cur_sigma = min(cur_sigma * 1.35, 1.25)
            reseed_n = int(round(params.reseed_frac * params.pop_size))
            candidates = [i for i in range(params.pop_size) if i not in
                elite_indices]
            rng.shuffle(candidates)
            for j in candidates[:reseed_n]:
                pop[j] = [rng.gauss(0, 1) for _ in range(genome_len)]

        new_pop = [pop[i][:] for i in elite_indices]

        while len(new_pop) < params.pop_size:
            p1 = tournament_select(pop, fit, params.tournament_k, rng)
            p2 = tournament_select(pop, fit, params.tournament_k, rng)

            if rng.random() < params.crossover_rate:
                c1, c2 = crossover_blend(p1, p2, rng)
            else:
                c1, c2 = p1, p2

            mutate(c1, params.mutation_rate, cur_sigma, rng)
            mutate(c2, params.mutation_rate, cur_sigma, rng)

            new_pop.append(c1)
            if len(new_pop) < params.pop_size:
                new_pop.append(c2)

        pop = new_pop

    assert best_genome is not None and best_eval is not None
```

```
    return best_genome, best_eval

def format_plans(plans: Dict[Tuple[str, str], Dict[str, int]], r_return:
    float, avg_trips: float, avg_cost: float) -> str:
    lines = []
    lines.append("IMMC GA Best 8 Seasonal Distributions")
    lines.append("=" * 34)
    lines.append(f"Total scooters: {TOTAL_SCOOTERS}")
    lines.append(f"Best avg trips/year: {int(round(avg_trips))}")
    lines.append(f"Best avg cost/year: {avg_cost:.2f}")
    lines.append(f"Optimized return (rebalancing) rate r: {r_return:.4f}")
    lines.append("")
    for season in SEASONS:
        lines.append(f"{season}:")
        for dt in ("WD", "WE"):
            label = "Weekday" if dt == "WD" else "Weekend"
            p = plans[(season, dt)]
            lines.append("  " + f"{label:<8}: " + " ".join([f"{z}={p[z]}"
                for z in ZONES]))
        lines.append("")
    return "\n".join(lines)

def save_output(text: str, filename: str) -> None:
    with open(filename, "w", encoding="utf-8") as f:
        f.write(text)

def run_ui():
    if tk is None:
        raise RuntimeError("Tkinter not available.")

    root = tk.Tk()
    root.title("IMMC GA Scooter Planner")
    root.geometry("980x720")

    frame = ttk.Frame(root, padding=10)
    frame.pack(fill="both", expand=True)

    title = ttk.Label(frame, text="IMMC GA Scooter Planner", font=("Segoe
        UI", 14, "bold"))
    title.pack(anchor="w")

    desc = ttk.Label(
        frame,
        text="Optimization running. Check output for details.",
```

```
    justify="left",
)
desc.pack(anchor="w", pady=(0, 10))

params_box = ttk.LabelFrame(frame, text="GA␣Parameters")
params_box.pack(fill="x")

def add_spin(row, label, default, from_, to_, step=1):
    ttk.Label(params_box, text=label).grid(row=row, column=0,
        sticky="w", padx=5, pady=4)
    var = tk.IntVar(value=default)
    sp = ttk.Spinbox(params_box, from_=from_, to=to_, increment=step,
        textvariable=var, width=10)
    sp.grid(row=row, column=1, sticky="w", padx=5, pady=4)
    return var

pop_var = add_spin(0, "Population␣size", 70, 20, 250, 5)
gen_var = add_spin(1, "Generations", 120, 10, 400, 10)
sims_var = add_spin(2, "Simulations␣per␣individual", 5, 1, 15, 1)

status = ttk.Label(frame, text="Ready.")
status.pack(anchor="w", pady=(10, 0))

out_text = tk.Text(frame, height=24, wrap="word")
out_text.pack(fill="both", expand=True, pady=10)

def update_status(gen, gen_best, best_so_far):
    status.configure(
        text=(
            f"Gen␣{gen+1}/{int(gen_var.get())}␣|␣"
            f"GenBest␣score={gen_best['fitness']:.4f}␣
                trips={int(gen_best['avg_trips'])}␣
                cost={gen_best['avg_cost']:.0f}␣
                r={gen_best['r_return']:.3f}␣||␣"
            f"BestSoFar␣score={best_so_far['fitness']:.4f}␣
                trips={int(best_so_far['avg_trips'])}␣
                cost={best_so_far['avg_cost']:.0f}␣
                r={best_so_far['r_return']:.3f}"
        )
    )

def worker():
    try:
        params = GAParams(
```

```
                pop_size=int(pop_var.get()),
                generations=int(gen_var.get()),
                sims_per_individual=int(sims_var.get()),
            )

            best_genome, best_eval = run_ga(
                params,
                status_cb=lambda g, gb, bs: root.after(0, update_status, g,
                    gb, bs),
            )
            plans, r_return = decode_individual(best_genome)
            txt = format_plans(plans, r_return, best_eval["avg_trips"],
                best_eval["avg_cost"])
            save_output(txt, "best_8_season_distributions.txt")

            log_lines: List[str] = []
            log_seeds = [1001 + 97 * i for i in
                range(params.sims_per_individual)]
            for sd in log_seeds:
                simulate_year(sd, plans, r_return, log_lines=log_lines,
                    sim_name="(best)")
            save_output("\n".join(log_lines), "simulation_daily_log.txt")

            def finish():
                out_text.delete("1.0", "end")
                out_text.insert("1.0", txt + "\n\nSaved␣to␣
                    best_8_season_distributions.txt\nSaved␣to␣
                    simulation_daily_log.txt")
                status.configure(text="Done.␣Saved␣
                    best_8_season_distributions.txt")

            root.after(0, finish)
        except Exception as e:
            root.after(0, lambda: messagebox.showerror("Error", str(e)))

    def on_run():
        status.configure(text="Running␣GA...")
        out_text.delete("1.0", "end")
        t = threading.Thread(target=worker, daemon=True)
        t.start()

    btn = ttk.Button(frame, text="Run␣GA", command=on_run)
    btn.pack(anchor="w")
```

```
    root.mainloop()

if __name__ == "__main__":
    run_ui()
```

Genetic algorithm code for the extra scenarios:

```
"""
IMMC␣Scooter␣GA␣   ␣One-Day␣Scenario␣Optimizer␣(January␣Weekday)␣[FIXED␣
    NORMAL␣SIM]

NORMAL␣SIM␣RULE␣(what␣you␣asked␣for):
␣␣␣␣-␣Day␣starts␣with␣the␣given␣baseline␣distribution␣(GA␣solution).
␣␣␣␣-␣The␣day␣evolves␣via␣trips␣+␣drift.
␣␣␣␣-␣Rebalancing␣happens␣AFTER␣the␣day␣(night␣repositioning),␣not␣before.

Scenario␣1␣(Festival␣in␣C):
␣␣␣␣Zone␣C␣passenger␣density␣increases␣by␣+70%␣␣->␣multiplier␣1.70␣on␣
    passenger␣activity␣in␣C.

Scenario␣2␣(Public␣transport␣strike␣in␣B):
␣␣␣␣Zone␣B␣scooter␣demand␣doubles␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣->␣multiplier␣2.00␣on␣final␣
    demand␣weight␣in␣B.

Runs␣a␣*single-day*␣simulation␣and␣a␣*single-day*␣GA␣(January␣weekday).
The␣GA␣optimizes:
␣␣␣␣(i)␣baseline␣distribution␣B␣(summing␣to␣500)
␣␣␣␣(ii)␣rebalancing␣rate␣r␣   ␣[0,␣1]␣applied␣at␣NIGHT␣(after␣drift)

Output␣per␣scenario:
␣␣␣␣-␣Optimized␣baseline␣distribution␣+␣r
␣␣␣␣-␣Start-of-day␣distribution␣=␣baseline
␣␣␣␣-␣End-of-day␣distribution␣after␣drift
␣␣␣␣-␣Night␣rebalanced␣distribution␣(what␣next␣day␣would␣start␣with)
␣␣␣␣-␣Trips,␣move␣cost,␣charging␣cost,␣total␣cost
"""

from __future__ import annotations

import argparse
import math
import random
from dataclasses import dataclass
from typing import Dict, List, Tuple, Optional
```

```
# -------------------------
# Problem constants
# -------------------------
ZONES = ["A", "B", "C", "D", "E"]

TOTAL_SCOOTERS = 500
MAX_TRIPS_PER_SCOOTER = 6
TOTAL_DAILY_DEMAND = TOTAL_SCOOTERS * MAX_TRIPS_PER_SCOOTER  # 3000
    trips/day capacity

DAILY_DRIFT_FRACTION = 0.35  # 35% migrate daily
MOVE_COST_PER_SCOOTER = 12.0  # TL per scooter moved

# Charging model
FULL_RANGE_KM = 25.0
FULL_CHARGE_COST_TL = 8.0

AVG_TRIP_MINUTES = 18.0 / 6.0  # 3.0 min per trip
AVG_SPEED_KMH = 15.4

# Propensity multipliers
STUDENT_PROPENSITY_MULTIPLIER = 1.30
TOURIST_PROPENSITY_MULTIPLIER = 0.65

# Student activity modifiers
WEEKEND_STUDENT_ACTIVITY_MULT = 0.90
STUDENT_SEASON_ACTIVITY = {
    "Winter": 1.00,
    "Spring": 1.00,
    "Summer": 0.55,
    "Autumn": 1.00,
}

# PT effect parameter
BETA_PT = 0.20

# Tourism drift bias
EPS_TOUR_DRIFT = 0.15


# -------------------------
# Qualitative     numeric mapping (percentile-anchor method)
# -------------------------
PASSENGER_LEVELS = {"low": 12.5, "mid": 37.5, "high": 62.5, "vhigh": 87.5}
```

```
# Public transport accessibility (15-min anchor percentiles)
PT_5MIN_LEVELS = {
    "vweak": 53.263690,
    "weak": 74.976128,
    "mid": 89.548728,
    "good": 94.718893,
    "vgood": 97.903829,
}

# Student share anchor values (already a population share)
STUDENT_SHARE_LEVELS = {
    "low":   0.055527,
    "mid":   0.075871,
    "high":  0.095667,
    "vhigh": 0.113784,
}

# Zone categories (example consistent mapping)
ZONE_CATS = {
    "A": {"pass": "vhigh", "stud": "high",  "tour": "mid",   "pt": "vgood"},
    "B": {"pass": "high",  "stud": "mid",   "tour": "low",   "pt": "good"},
    "C": {"pass": "mid",   "stud": "low",   "tour": "vhigh", "pt": "mid"},
    "D": {"pass": "high",  "stud": "mid",   "tour": "mid",   "pt": "weak"},
    "E": {"pass": "low",   "stud": "vhigh", "tour": "high",  "pt": "vweak"},
}

# Tourism medians (p50) (used only for seasonal intensity)
TOURISM_MONTHLY_P50 = {
    "M01": 127559.0, "M02": 139955.0, "M03": 177776.0, "M04": 211020.0,
    "M05": 298541.0, "M06": 380704.0, "M07": 549296.0, "M08": 562832.0,
    "M09": 345798.0, "M10": 235559.0, "M11": 154990.0, "M12": 145031.0,
}


def _season_tour_intensity() -> Dict[str, float]:
    winter = (TOURISM_MONTHLY_P50["M12"] + TOURISM_MONTHLY_P50["M01"] +
        TOURISM_MONTHLY_P50["M02"]) / 3.0
    spring = (TOURISM_MONTHLY_P50["M03"] + TOURISM_MONTHLY_P50["M04"] +
        TOURISM_MONTHLY_P50["M05"]) / 3.0
    summer = (TOURISM_MONTHLY_P50["M06"] + TOURISM_MONTHLY_P50["M07"] +
        TOURISM_MONTHLY_P50["M08"]) / 3.0
    autumn = (TOURISM_MONTHLY_P50["M09"] + TOURISM_MONTHLY_P50["M10"] +
        TOURISM_MONTHLY_P50["M11"]) / 3.0
```

```
    mean_all = (winter + spring + summer + autumn) / 4.0
    return {
        "Winter": winter / mean_all,
        "Spring": spring / mean_all,
        "Summer": summer / mean_all,
        "Autumn": autumn / mean_all,
    }


SEASON_TOUR_INTENSITY = _season_tour_intensity()


# -------------------------
# Helpers
# -------------------------
def normalize_weights(w: Dict[str, float]) -> Dict[str, float]:
    s = sum(max(0.0, v) for v in w.values())
    if s <= 0:
        return {k: 1.0 / len(w) for k in w}
    return {k: max(0.0, v) / s for k, v in w.items()}


def repair_distribution(vals: List[float], total: int = TOTAL_SCOOTERS) ->
    List[int]:
    vals = [max(0.0, float(v)) for v in vals]
    s = sum(vals)
    if s == 0:
        base = [total // len(vals)] * len(vals)
        for i in range(total - sum(base)):
            base[i] += 1
        return base

    scaled = [v * total / s for v in vals]
    ints = [int(round(v)) for v in scaled]

    diff = total - sum(ints)
    if diff != 0:
        order = sorted(range(len(ints)), key=lambda i: ints[i],
            reverse=True)
        step = 1 if diff > 0 else -1
        for k in range(abs(diff)):
            idx = order[k % len(order)]
            ints[idx] += step
            if ints[idx] < 0:
```

```
                ints[idx] = 0

        if sum(ints) != total:
            return repair_distribution([max(0.0, x) for x in ints],
                total=total)

    return ints


def l1_moved(a: List[int], b: List[int]) -> int:
    return int(round(0.5 * sum(abs(ai - bi) for ai, bi in zip(a, b))))


def charging_cost(total_trips: int) -> float:
    km_per_trip = AVG_SPEED_KMH * (AVG_TRIP_MINUTES / 60.0)
    cost_per_km = FULL_CHARGE_COST_TL / FULL_RANGE_KM
    return total_trips * km_per_trip * cost_per_km


# -------------------------
# Demand + drift model
# -------------------------
def tourism_fraction_by_zone(season: str, daytype: str) -> Dict[str, float]:
    base_level = {"low": 0.03, "mid": 0.07, "high": 0.12, "vhigh": 0.18}
    season_mult = SEASON_TOUR_INTENSITY[season]
    weekend_mult = 1.10 if daytype == "WE" else 1.00

    Tfrac = {}
    for z in ZONES:
        level = ZONE_CATS[z]["tour"]
        Tfrac[z] = min(0.45, base_level[level] * season_mult * weekend_mult)
    return Tfrac


def tourism_attractiveness_rank() -> Dict[str, float]:
    rank = {"low": 0.25, "mid": 0.50, "high": 0.75, "vhigh": 1.00}
    return {z: rank[ZONE_CATS[z]["tour"]] for z in ZONES}


def zone_feature_scores(season: str, daytype: str, scenario: Optional[Dict]
    = None
                        ) -> Tuple[Dict[str, float], Dict[str, float],
                            Dict[str, float]]:
    scenario = scenario or {}
```

```
passenger_multiplier: Dict[str, float] =
    scenario.get("passenger_multiplier", {})
demand_multiplier: Dict[str, float] = scenario.get("demand_multiplier",
    {})

# Passenger activity magnitude (NOT a probability yet)
rawP = {}
for z in ZONES:
    pctl = PASSENGER_LEVELS[ZONE_CATS[z]["pass"]]
    raw = pctl / 100.0
    raw *= float(passenger_multiplier.get(z, 1.0))   # Scenario 1 hits
        here
    rawP[z] = raw

# normalize passenger activity into weights (used as part of demand)
Pn = normalize_weights(rawP)

# PT score (0-1) and multiplier
PTscore = {z: PT_5MIN_LEVELS[ZONE_CATS[z]["pt"]] / 100.0 for z in ZONES}
fPT = {z: 1.0 + BETA_PT * PTscore[z] for z in ZONES}

# Student/tourist shares
Tfrac = tourism_fraction_by_zone(season, daytype)
Sshare = {z: STUDENT_SHARE_LEVELS[ZONE_CATS[z]["stud"]] for z in ZONES}

stu_activity = STUDENT_SEASON_ACTIVITY[season] *
    (WEEKEND_STUDENT_ACTIVITY_MULT if daytype == "WE" else 1.0)
S_eff = {z: min(0.35, Sshare[z] * stu_activity) for z in ZONES}

# Final effective demand weight (still relative)
PassEff = {}
for z in ZONES:
    local_frac = 1.0 - Tfrac[z]
    local_multiplier = (1.0 - S_eff[z]) * 1.0 + S_eff[z] *
        STUDENT_PROPENSITY_MULTIPLIER
    tourist_multiplier = TOURIST_PROPENSITY_MULTIPLIER

    PassEff[z] = Pn[z] * fPT[z] * (local_frac * local_multiplier +
        Tfrac[z] * tourist_multiplier)

    # Scenario 2 hits here (final demand boost)
    PassEff[z] *= float(demand_multiplier.get(z, 1.0))

TourAttr = tourism_attractiveness_rank()
```

```
    return PassEff, PTscore, TourAttr


def expected_daily_demand(prob: Dict[str, float], total: int =
    TOTAL_DAILY_DEMAND) -> Dict[str, int]:
    raw = {z: prob[z] * total for z in ZONES}
    base = {z: int(math.floor(raw[z])) for z in ZONES}
    remainder = total - sum(base.values())
    frac_order = sorted(ZONES, key=lambda z: raw[z] - base[z], reverse=True)
    for k in range(remainder):
        base[frac_order[k % len(ZONES)]] += 1
    return base


def drift_step(scooters_start: Dict[str, int], trips_realized: Dict[str,
    int], tour_attr: Dict[str, float]
              ) -> Dict[str, int]:
    # destination pull is trips + tiny tourism attractiveness
    raw = {z: float(trips_realized[z]) + EPS_TOUR_DRIFT *
        float(tour_attr[z]) for z in ZONES}
    pi = normalize_weights(raw)

    end_float = {}
    for z in ZONES:
        end_float[z] = (1.0 - DAILY_DRIFT_FRACTION) * scooters_start[z] +
            DAILY_DRIFT_FRACTION * TOTAL_SCOOTERS * pi[z]

    end_dist = repair_distribution([end_float[z] for z in ZONES],
        total=TOTAL_SCOOTERS)
    return {z: end_dist[i] for i, z in enumerate(ZONES)}


def night_rebalance(end_dist: Dict[str, int], baseline: Dict[str, int], r:
    float
                    ) -> Tuple[Dict[str, int], int, float]:
    """
␣␣␣␣Night␣repositioning␣AFTER␣the␣day:
␣␣␣␣␣␣␣␣next_start␣=␣(1-r)*end␣+␣r*baseline
␣␣␣␣Move␣cost␣is␣based␣on␣|next_start␣-␣end|.
␣␣␣␣"""
    r = min(1.0, max(0.0, float(r)))
    blended = {z: (1.0 - r) * end_dist[z] + r * baseline[z] for z in ZONES}

    next_list = repair_distribution([blended[z] for z in ZONES],
```

```
        total=TOTAL_SCOOTERS)
    next_start = {z: next_list[i] for i, z in enumerate(ZONES)}

    moved = l1_moved([end_dist[z] for z in ZONES], [next_start[z] for z in
        ZONES])
    move_cost = moved * MOVE_COST_PER_SCOOTER
    return next_start, moved, move_cost


# -------------------------
# One-day simulation (NORMAL)
# -------------------------
@dataclass
class OneDayResult:
    scenario_name: str
    baseline: Dict[str, int]
    r: float
    start_of_day: Dict[str, int]
    end_of_day: Dict[str, int]
    night_rebalanced: Dict[str, int]
    moved: int
    trips: int
    move_cost: float
    charge_cost: float
    total_cost: float


def simulate_one_day(baseline: Dict[str, int], r: float, scenario_name: str,
                     scenario: Optional[Dict] = None) -> OneDayResult:
    season = "Winter"
    daytype = "WD"

    #     NORMAL: day starts with baseline distribution exactly
    start = dict(baseline)

    PassEff, _, TourAttr = zone_feature_scores(season, daytype,
        scenario=scenario)
    prob = normalize_weights(PassEff)
    demand = expected_daily_demand(prob, total=TOTAL_DAILY_DEMAND)

    trips_realized = {}
    total_trips = 0
    for z in ZONES:
        cap = MAX_TRIPS_PER_SCOOTER * start[z]
```

```
        q = min(demand[z], cap)
        trips_realized[z] = q
        total_trips += q

    # drift     end-of-day distribution
    end = drift_step(start, trips_realized, TourAttr)

    # night rebalancing AFTER the day (optional)
    next_start, moved, move_cost = night_rebalance(end, baseline, r)

    charge = charging_cost(total_trips)
    total_cost = move_cost + charge

    return OneDayResult(
        scenario_name=scenario_name,
        baseline=baseline,
        r=r,
        start_of_day=start,
        end_of_day=end,
        night_rebalanced=next_start,
        moved=moved,
        trips=total_trips,
        move_cost=move_cost,
        charge_cost=charge,
        total_cost=total_cost,
    )


# -------------------------
# One-day GA
# -------------------------
@dataclass
class Individual:
    dist: List[int]
    r: float
    fitness: float = float("-inf")
    trips: int = 0
    cost: float = 0.0


def random_distribution(rng: random.Random) -> List[int]:
    cuts = sorted([0] + [rng.randint(0, TOTAL_SCOOTERS) for _ in
        range(len(ZONES) - 1)] + [TOTAL_SCOOTERS])
    return [cuts[i + 1] - cuts[i] for i in range(len(ZONES))]
```

```
def mutate_distribution(dist: List[int], rng: random.Random, max_delta: int
    = 30) -> List[int]:
    d = dist[:]
    i, j = rng.sample(range(len(ZONES)), 2)
    delta = rng.randint(1, max_delta)
    if d[i] >= delta:
        d[i] -= delta
        d[j] += delta
    return d


def crossover_distributions(a: List[int], b: List[int], rng: random.Random)
    -> List[int]:
    mix = [a[i] if rng.random() < 0.5 else b[i] for i in range(len(ZONES))]
    return repair_distribution(mix, total=TOTAL_SCOOTERS)


def evaluate_individual(ind: Individual, scenario_name: str, scenario:
    Optional[Dict], lambda_cost: float) -> Individual:
    baseline = {z: ind.dist[i] for i, z in enumerate(ZONES)}
    res = simulate_one_day(baseline, ind.r, scenario_name=scenario_name,
        scenario=scenario)

    # normalized objective to keep GA stable
    trips_norm = res.trips / float(TOTAL_DAILY_DEMAND)
    cost_norm = res.total_cost / 7000.0  # rough upper bound
    score = trips_norm - lambda_cost * cost_norm

    ind.fitness = score
    ind.trips = res.trips
    ind.cost = res.total_cost
    return ind


def run_ga_one_day(scenario_name: str, scenario: Optional[Dict], seed: int
    = 0,
                   pop_size: int = 120, generations: int = 200, elite_frac:
                       float = 0.08,
                   tournament_k: int = 3, mutation_rate: float = 0.35,
                       lambda_cost: float = 0.50
                  ) -> Tuple[Individual, OneDayResult]:
```

```
rng = random.Random(seed)
pop = [Individual(dist=random_distribution(rng), r=rng.random()) for _
    in range(pop_size)]
for ind in pop:
    evaluate_individual(ind, scenario_name, scenario, lambda_cost)

elite_n = max(1, int(round(elite_frac * pop_size)))

def tournament_select() -> Individual:
    cand = rng.sample(pop, tournament_k)
    cand.sort(key=lambda x: x.fitness, reverse=True)
    return cand[0]

best = max(pop, key=lambda x: x.fitness)

for _ in range(generations):
    pop.sort(key=lambda x: x.fitness, reverse=True)
    elites = pop[:elite_n]
    new_pop = elites[:]

    while len(new_pop) < pop_size:
        p1 = tournament_select()
        p2 = tournament_select()

        child_dist = crossover_distributions(p1.dist, p2.dist, rng)
        child_r = (p1.r + p2.r) / 2.0

        if rng.random() < mutation_rate:
            child_dist = mutate_distribution(child_dist, rng)
        if rng.random() < mutation_rate:
            child_r = min(1.0, max(0.0, child_r + rng.uniform(-0.10,
                0.10)))

        child = Individual(dist=child_dist, r=child_r)
        evaluate_individual(child, scenario_name, scenario, lambda_cost)
        new_pop.append(child)

    pop = new_pop
    gen_best = max(pop, key=lambda x: x.fitness)
    if gen_best.fitness > best.fitness:
        best = gen_best

best_baseline = {z: best.dist[i] for i, z in enumerate(ZONES)}
best_day = simulate_one_day(best_baseline, best.r,
```

```
        scenario_name=scenario_name, scenario=scenario)
    return best, best_day


# -------------------------
# Scenarios
# -------------------------
SCENARIOS = {
    "festival": {
        "name": "Scenario 1    Festival in C (+70% passenger density)",
        "mod": {"passenger_multiplier": {"C": 1.70}},
    },
    "strike": {
        "name": "Scenario 2    PT strike in B (2   scooter demand)",
        "mod": {"demand_multiplier": {"B": 2.00}},
    },
}


def format_dist(d: Dict[str, int]) -> str:
    return " ".join([f"{z}={d[z]}" for z in ZONES])


def print_result(day: OneDayResult) -> None:
    print("=" * 70)
    print(day.scenario_name)
    print("-" * 70)
    print(f"Optimized baseline (Day start): {format_dist(day.baseline)}")
    print(f"Optimized night rebalancing rate r: {day.r:.4f}\n")

    print(f"Start-of-day distribution : {format_dist(day.start_of_day)}")
    print(f"End-of-day distribution    : {format_dist(day.end_of_day)}")
    print(f"After-night distribution   :
        {format_dist(day.night_rebalanced)}\n")

    print(f"Moved scooters at night    : {day.moved}")
    print(f"Trips (realized)           : {day.trips}")
    print(f"Night move cost TL         : {day.move_cost:.2f}")
    print(f"Charging cost TL           : {day.charge_cost:.2f}")
    print(f"Total cost TL              : {day.total_cost:.2f}")
    print("=" * 70 + "\n")


def main():
```

```
    ap = argparse.ArgumentParser()
    ap.add_argument("--seed", type=int, default=0)
    ap.add_argument("--pop", type=int, default=120)
    ap.add_argument("--gens", type=int, default=200)
    ap.add_argument("--lambda_cost", type=float, default=0.50)
    args = ap.parse_args()

    for key in ["festival", "strike"]:
        info = SCENARIOS[key]
        _, day = run_ga_one_day(
            scenario_name=info["name"],
            scenario=info["mod"],
            seed=args.seed,
            pop_size=args.pop,
            generations=args.gens,
            lambda_cost=args.lambda_cost,
        )
        print_result(day)


if __name__ == "__main__":
    print("Hello␣world!␣yayayayayaayyayyayayay")
    main()
```