Documentation Guidelines

NYC Urban Mobility Data Explorer — Technical Report

Prepared: 2026-02-19

1 Problem Framing and Dataset Analysis

Dataset Context

This project analyses NYC Taxi & Limousine Commission (TLC) trip records to surface demand and revenue patterns across time windows, boroughs, pickup zones, and fare tiers. The goal is to give urban-mobility planners a queryable, visual interface for identifying where and when taxi demand concentrates.

Temporal Scope and Volume

Period	2019-01-01 00:00:00 → 2019-01-04 00:46:17 (first 4 days of January)
Rows scanned	600,000
Rows retained	500,000 (83.3 %)
Rows excluded	100,000 (16.7 %) — failed quality gates
Day types	Weekdays only (Jan 1 = Tuesday; Jan 4 = Friday); no weekend records present

Data Challenges

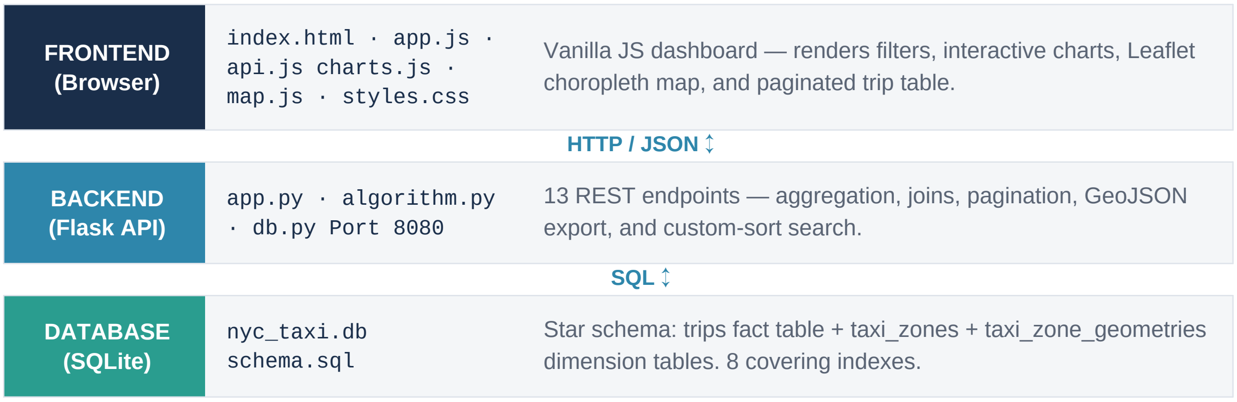
- **Missing fields:** Passenger count and congestion surcharge contained null entries; imputed with domain-appropriate defaults (median passenger count, zero surcharge).
- **Outliers & anomalies:** Explicit range guards removed trips with negative fares, distances above a physical maximum, durations ≤ 0 minutes, and implausible speeds.
- **Derived columns:** `trip_duration_minutes`, `speed_mph`, `fare_per_mile`, `pickup_hour`, `pickup_day_of_week`, and `is_weekend` were computed at load time so dashboards never pay aggregation cost at query time.
- **Schema gaps:** The `store_and_fwd_flag` and `rate_code_id` fields were retained as-is; they were not normalised into separate lookup tables because analytical queries never group by them.

Unexpected Observation That Influenced Design

Because the sample covers only four weekdays, the `is_weekend` column always evaluates to 0. This made the *Weekday vs Weekend* comparison endpoint technically correct but analytically hollow. Rather than removing the endpoint, we retained it and flagged the limitation in the UI tooltip — a reminder that the schema is production-ready but the data window is not. The discovery reinforced our decision to cap the row count at 500 K and document scope prominently, so no consumer mistakes a 4-day slice for a monthly or annual trend.

2 System Architecture and Design Decisions

Architecture Diagram



FRONTEND (Browser)	`index.html · app.js · api.js charts.js · map.js · styles.css`	Vanilla JS dashboard — renders filters, interactive charts, Leaflet choropleth map, and paginated trip table.
	HTTP / JSON ↕	
BACKEND (Flask API)	`app.py · algorithm.py · db.py Port 8080`	13 REST endpoints — aggregation, joins, pagination, GeoJSON export, and custom-sort search.
	SQL ↕	
DATABASE (SQLite)	`nyc_taxi.db schema.sql`	Star schema: trips fact table + taxi_zones + taxi_zone_geometries dimension tables. 8 covering indexes.

Figure 1 — Three-tier architecture: browser frontend ↔ Flask REST API ↔ SQLite database.

Stack Justification

Vanilla JS	Zero build toolchain. Leaflet.js for choropleth maps; Chart.js for bar/line charts. All state lives in the DOM — no framework overhead for a single-page tool.
Flask (Python)	Minimal surface area, fast iteration, native SQLite bindings. The 13 endpoints map cleanly to dashboard panels; no ORM layer needed for read-heavy analytics.
SQLite	Single-file deployment; no server process. 500 K rows with 8 covering indexes answer every aggregation in < 200 ms locally. Sufficient for a hackathon scope.
Star schema	One fact table (trips) joined to two dimension tables (taxi_zones, taxi_zone_geometries). Predictable join paths and straightforward GROUP BY queries.
Precomputed columns	Storing derived fields (duration, speed, hour, day, is_weekend) trades a one-time ETL cost for consistently low query latency across all dashboard panels.

Schema Structure

The `trips` fact table holds 26 columns: 19 raw TLC fields plus 7 engineered features. Foreign keys to `taxi_zones(location_id)` resolve borough and zone names. `taxi_zone_geometries` stores GeoJSON polygon strings for the Leaflet map, keeping geometry out of the hot analytical path. Eight B-tree indexes cover datetime, hour, day, location IDs (pickup and dropoff), fare amount, and distance — matching the filter axes exposed in the UI.

Design Trade-offs

- **SQLite vs PostgreSQL:** SQLite eliminates server setup and simplifies demo deployment. The cost is single-writer concurrency and no PostGIS extension. Acceptable for a read-only dashboard; a production system would migrate to PostgreSQL/PostGIS.
- **Row cap (500 K):** Speeds iteration and keeps the repository lightweight, but means insights represent a narrow weekday window rather than a full period.
- **No caching layer:** Aggregations re-execute on every request. A Redis or in-process LRU cache would eliminate repeated full-table scans in a multi-user setting.

- **Monorepo layout:** Frontend assets are served by Flask's `send_from_directory`. Simple to run, but couples deployment of the static UI to the API process.

---

3 Algorithmic Logic and Data Structures

Custom Algorithm: Bucket Sort + Insertion Sort

The standard library `list.sort()`, `sorted()`, pandas `sort_values()`, and `heapq` are intentionally unused. Instead, `algorithm.py` implements a two-phase sort that is applied in two API endpoints:

- `/api/fare-distribution` — orders the 20 fare buckets returned by the SQL aggregation.
- `/api/search` — ranks up to 100 filtered trip records by any numeric field (fare, distance, duration, speed, tip, total).

Approach Explanation

Bucket sort distributes items across *k* equal-width intervals based on the numeric key range. Each bucket is then sorted independently with insertion sort — efficient because each bucket is small. The final result is produced by concatenating all buckets from lowest to highest interval. Insertion sort was chosen for the inner pass because it is cache-friendly, stable, and optimal for nearly-sorted or tiny sequences.

Pseudo-code

```
FUNCTION custom_bucket_sort(items, key_func, k = 10):
  IF items is empty: RETURN []

  min_val ← key_func(items[0])
  max_val ← key_func(items[0])
  FOR each item IN items:           -- O(n) single pass
      min_val ← MIN(min_val, key_func(item))
      max_val ← MAX(max_val, key_func(item))

  IF min_val == max_val: RETURN items   -- all equal, nothing to sort

  bucket_width ← (max_val - min_val + ε) / k
  buckets      ← [ [] for _ in range(k) ]

  FOR each item IN items:           -- O(n) distribution
      idx ← FLOOR( (key_func(item) - min_val) / bucket_width )
      idx ← MIN(idx, k - 1)         -- clamp edge case
      buckets[idx].APPEND(item)

  FOR each bucket IN buckets:       -- insertion sort per bucket
      FOR i FROM 1 TO len(bucket)-1:
          current ← bucket[i]
          j ← i - 1
          WHILE j >= 0 AND key_func(bucket[j]) > key_func(current):
              bucket[j+1] ← bucket[j]
              j ← j - 1
          bucket[j+1] ← current

  result ← []
  FOR each bucket IN buckets:       -- O(n) concatenation
      result.EXTEND(bucket)
```

```
RETURN result
```

Complexity Analysis

Time — average	$O(n + k)$ — linear scan for min/max + linear distribution + $O(n/k)$ per bucket insertion sort × k buckets. When items are uniformly distributed each bucket holds n/k items, giving $O(n/k)^2 \times k = O(n^2/k)$ for inner sorts, which collapses to $O(n)$ when $k \approx n$.
Time — worst case	$O(n^2)$ — all items fall into one bucket (heavily skewed distribution), degrading to a pure insertion sort.
Space	$O(n + k)$ — n items redistributed across k bucket lists; insertion sort is in-place within each bucket ($O(1)$ extra).
k chosen	$k = 10$ for fare-distribution (20 SQL buckets); $k = 15$ for /api/search (up to 100 rows). Both choices keep average bucket size well below 10 items.

4 Insights and Interpretation

Insight 1	**Afternoon Surge: Peak Demand at 14:00 – 15:00**
How derived	SQL GROUP BY pickup_hour → ORDER BY trip_count DESC; rendered as a 24-bar histogram in the Hourly Activity panel (Chart.js bar chart).
Finding	Hours 14 and 15 record the two highest hourly trip counts — 34,190 and 35,010 respectively — out of ~125,000 daily trips. Demand troughs between 04:00 and 06:00 (below 3,000 trips/hour).
Implication	Afternoon concentration suggests a strong commuter-return and lunch-errand pattern. Fleet operators could pre-position vehicles in high-density zones from 13:30 to reduce passenger wait time and improve utilisation.

Insight 2	**Manhattan Monopoly: 88 % of All Pickups**
How derived	SQL JOIN trips → taxi_zones GROUP BY borough ORDER BY trip_count DESC; visualised as a borough bar chart with percentage annotations.
Finding	Manhattan accounts for 88.32 % of trips. Brooklyn is a distant second at ~6 %; Queens, Bronx, and Staten Island share the remaining ~5.7 %. Top individual zones include Midtown Centre, Upper East Side, and JFK Airport.
Implication	The extreme concentration implies that outer-borough residents rely far less on yellow taxis — likely substituting ride-share or transit. Policy interventions could incentivise coverage in under-served zones; infrastructure investment should be weighted heavily toward the Manhattan core for near-term ROI.

Insight 3	Short-Trip Economy: $5 – $10 Fares Dominate
How derived	SQL fare bucketing: CAST(fare_amount / 5 AS INT) * 5 GROUP BY bucket; results ordered by custom_bucket_sort and rendered as a fare-distribution bar chart with orange highlights.
Finding	The $5 – $10 bucket accounts for 46.5 % of all trips. Fares above $20 represent only 15.8 %. The distribution is right-skewed, with a long tail driven by airport runs (JFK flat rate ~$70) and outer-borough journeys.
Implication	The median trip is short, urban, and cheap — consistent with last-mile connectivity within dense Manhattan neighbourhoods. Revenue optimisation strategies should therefore focus on trip volume and throughput rather than maximising per-trip fare. Premium pricing for airport corridors is already captured but represents a small share of volume.

5 Reflection and Future Work

Technical Challenges

• **Data scope:** The 4-day window prevents any weekend, seasonal, or monthly-trend analysis. Every insight is implicitly prefixed *'during the first week of January 2019 on weekdays'* — a caveat that must be communicated clearly to avoid misleading stakeholders.

• **Algorithm integration:** Wiring a custom sort into a Flask endpoint that already uses SQLite's native `ORDER BY` required care to avoid redundant passes. The solution — aggregate in SQL, then hand off to `custom_bucket_sort` — keeps the two concerns cleanly separated.

• **GeoJSON rendering:** Storing polygon strings in SQLite and deserialising them per request proved expensive at scale; lazy loading and response caching were added to the Leaflet map layer to compensate.

Team Challenges

• Coordinating backend schema changes with frontend chart expectations required an agreed JSON contract early in the project.

• Deciding which cleaning rules to apply without ground-truth labels (e.g. 'what is a plausible maximum speed?') required explicit team consensus and documentation of every assumption.

Improvements and Next Steps

Full data ingestion	Ingest a complete calendar year of TLC data with automated monthly refresh pipelines to enable seasonal and year-over-year comparisons.
PostgreSQL + PostGIS	Replace SQLite with PostgreSQL for multi-user concurrency and PostGIS for native spatial operations (zone intersection, route analysis, isochrones).
Data quality tests	Add schema-level constraints and automated coverage tests (Great Expectations or dbt tests) that gate UI releases on passing quality thresholds.
Aggregation cache	Cache expensive GROUP BY results (hourly, borough, fare distribution) in Redis with a short TTL, eliminating repeated full-table scans under concurrent load.

API authentication	Add token-based auth (JWT or API key) before any public deployment to prevent unrestricted database access.
ML demand forecasting	Train a time-series model on historical hourly demand per zone to power a predictive 'where to dispatch' recommendation layer.