IEE 305 Final Project: NPS Visitation Data Analysis and Planning Tool

Esteban Dozal

Arizona State University

Information Systems Engineering IEE 305

Professor Gonzalez

December 9th, 2025

1. Introduction

Effective management of national parks requires accurate demand forecasting, capacity planning, and data-driven decision support. The National Park Service experiences extreme variability in visitation volumes across regions, park types, and seasons, which significantly impacts staffing, maintenance scheduling, congestion mitigation, environmental impact assessments, and long-term infrastructure planning. Industrial Engineers rely on past historical data and analytical tools to understand visitor demand, detect trends, and optimize resource allocation. However, the raw visitation data is complicated to analyze and manage manually due to its volume, irregularity, and geographic distribution.

This project develops a National Park Service Visitor Analytics and Planning Tool, a complete decision-support information system powered by a relational database, FastAPI backend, and a Streamlit-based frontend. The system integrates park information gathered from the NPS API and a monthly visitation records dataset into a normalized data model that supports complex SQL analysis and interactive visualization. The backend exposes analytical endpoints that compute year-over-year demand changes, peak season pressure, regional growth, park-level variability, and other key metrics. These insights enable planners to quickly identify operational bottlenecks, prioritize resource allocation, and evaluate changes in visitor behavior over time.

To achieve this, we implemented a complete client-server architecture. The backend (FastAPI + SQLModel) handles all data access, query execution, and domain logic. The frontend (Streamlit) provides a user-friendly interface that allows exploration through dropdowns, filters, charts, and maps. The backend is populated using a custom Python data integration pipeline that retrieves live NPS park metadata from the National Park Service API and merges it with structured visitation data from the NPS Statistics Office.

The analytical capabilities of the system include:

- Monthly and annual visitation summaries
- Demand threshold identification
- Multi-year averages
- Peak-season analysis
- Year-to-year growth measurement
- Region-wide ranking and comparisons
- Variability and volatility analysis
- Custom filtering by region, park, and thresholds
- Top-N park rankings
- Case-insensitive park and region lookup
- Rankings of other miscellaneous metrics

These analyses directly support Industrial Engineering tasks such as scheduling, predictive modeling, facility load balancing, and capacity planning.

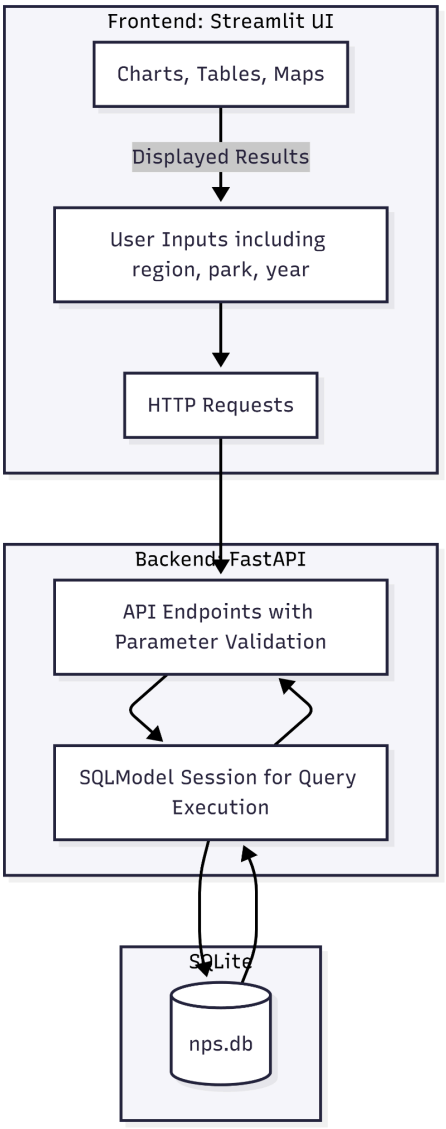
2. System Design

The system follows a clear client-server architecture featuring three significant components:

1. A SQLite relational database storing normalized NPS visitation and park metadata
2. A FastAPI backend providing secure analytical endpoints
3. A Streamlit frontend that enables user interaction, visualization, and filtering

The system design ensures scalability, modularity, and clean separation of concerns. The backend exposes only validated, parameterized interactions with the database, while the frontend performs no direct SQL access—ensuring security and maintainability.

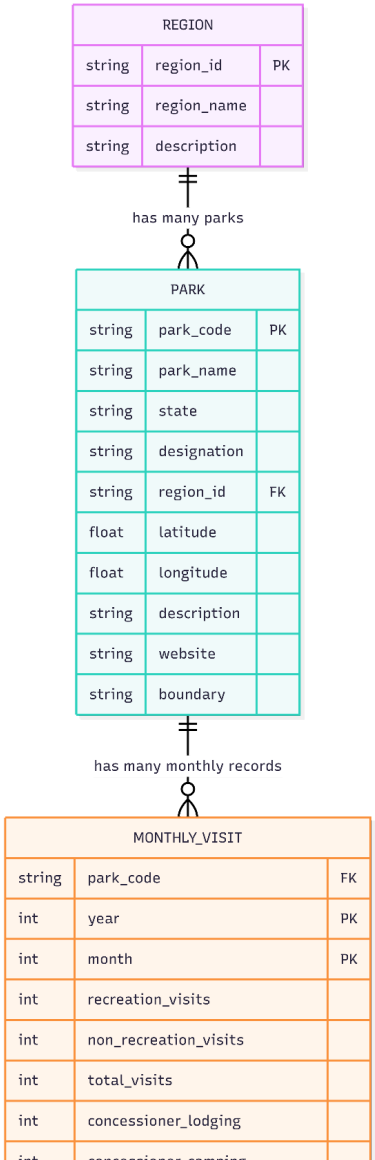
Client-Server Architecture Diagram:



Description:

- The frontend sends a structured request to FastAPI endpoints
- FastAPI validates all parameters using Pydantic and SQLModel
- Only parameterized SQL queries are executed, eliminating risks
- Results are returned as JSON and rendered into visualizations by Streamlit

Final ER Model:



Notes:

- Region → Park: one region has many parks.
- Park → MonthlyVisit: one park has up to 12 monthly records per year.
- Boundary stores the park boundary geometry as text (GeoJSON/WKT-style string) for potential mapping purposes.
- state, designation, website, and description enrich
- parks for real-world decision support.

API Endpoint Documentation:

#	HTTP	Path
-	GET	/metadata/years
-	GET	/regions/
-	GET	/parks/{park_code}/details
Q1	GET	/parks/{park_code}/monthly-visits
Q2	GET	/annual-visits/parks
Q3	GET	/visits/parks/average-monthly
Q4	GET	/visits/peak-season/above-threshold
Q5	GET	/visits/parks/above-system-average
Q6	GET	/annual-visits/top
Q11 (bonus)	GET	/annual-visits/parks/metrics
Q7	GET	/annual-visits/regions
Q8	GET	/parks/{park_code}/month-to-month-change
Q9	GET	/regions/{region_id}/growth
Q10	GET	/visits/parks/variability

Notes:

- The required SQL queries are implemented across Q1-Q10
- /annual-visits/parks/metrics (Q11) is an extra analytic endpoint that utilizes the extra miscellaneous data gathered from the CSV
- All endpoints are parameterized (no string concatenation SQL), and park/region codes are handled case-insensitively by uppercasing them in Python before querying

Technology choices:

Technology	Justification
SQLite	Lightweight, file-based DB, perfect for analytical workloads and small datasets; supports SQL + foreign keys.
SQLModel (SQLAlchemy + Pydantic)	Enables ORM, strict typing, and validation; integrates smoothly with FastAPI.
FastAPI	High performance, automatic documentation (/docs), async support, parameter validation, safe query execution.
Streamlit	An easy way to build an interactive analytics dashboard with minimal code. Provides charts, dropdowns, and maps. Familiar with it from previous projects.
NPS API	Reliable metadata source; enriches parks with real-world descriptions, URLs, boundaries, etc.
Python	Ideal for data manipulation, ETL, and full-stack integration in this project.

3. Database Implementation

Relational Schema - Exact schema used to create nps.db

```sql
DROP TABLE IF EXISTS monthly_visit;
DROP TABLE IF EXISTS park;
DROP TABLE IF EXISTS region;

CREATE TABLE region (
    region_id TEXT PRIMARY KEY,
    region_name TEXT NOT NULL,
    description TEXT
);

CREATE TABLE park (
    park_code TEXT PRIMARY KEY,
    park_name TEXT NOT NULL,
    state TEXT NOT NULL,
    designation TEXT NOT NULL,
    region_id TEXT,
    latitude REAL,
    longitude REAL,
    description TEXT,
    website TEXT,
    boundary TEXT,
    FOREIGN KEY (region_id) REFERENCES region(region_id)
);

CREATE TABLE monthly_visit (
    park_code TEXT NOT NULL,
    year INTEGER NOT NULL,
    month INTEGER NOT NULL,
    recreation_visits INTEGER,
    non_recreation_visits INTEGER,
    total_visits INTEGER,
    concessioner_lodging INTEGER,
    concessioner_camping INTEGER,
    tent_campers INTEGER,
    rv_campers INTEGER,
    backcountry INTEGER,
    nonrecreation_overnight_stays INTEGER,
    miscellaneous_overnight_stays INTEGER,
    PRIMARY KEY (park_code, year, month),
    FOREIGN KEY (park_code) REFERENCES park(park_code)
);
```

Normalization & Functional Dependencies

- region
    - PK: region_id, FD: region_id → {region_name, description}, 3NF
- park
    - PK: park_code, FD: park_code → {park_name, state, designation, region_id, latitude, longitude, description, website, boundary}, region_id is a foreign key but not a determinant for any other park attributes in this table → 3NF
- Monthly_visit
    - Composite PK: (park_code, year, month), FD: (park_code, year, month) → {all visitation metrics}, No attribute depends only on part of the key (e.g., year alone does not determine total_visits) and no transitive dependencies → 3NF

Data loading approach and record counts:

1. Schema Initialization
   The schema.sql file defines the three tables (region, park, and monthly_visit) and their foreign‑key relationships. I run a small helper script (create_db.py) that connects to SQLite and executes this schema script to create an empty nps.db database with the correct structure.

2. External API data for parks and boundaries
   Next, I use backend/fetch_data.py to pull park metadata directly from the National Park Service (NPS) public API. The script:
   Loads the NPS_API_KEY from a .env file.
   Calls https://developer.nps.gov/api/v1/parks with a limit of 474 to retrieve all parks in a single page.
   For each park, extract and clean the fields needed for the park table:
   park_code, park_name, state, designation, latitude, longitude, plus descriptive text and website URL.
   Clears any existing rows from the park table and inserts one row per park, leaving region_id temporarily NULL so the CSV loader can fill it in.

3. CSV visitor statistics into region and monthly_visit
   After the base park data is loaded, I run backend/load_csv.py, which reads the NPS monthly visitor statistics CSV file (2015–2024) from the data/ folder using pandas. The script:
   Normalizes and renames the CSV columns to match the monthly_visit schema (e.g., RecreationVisits, NonRecreationVisits, ConcessionerLodging, etc.).
   Derives total_visits as recreation_visits + non_recreation_visits to ensure a consistent metric across all parks.
   Builds the region table by inserting the seven official NPS regions (e.g., IMR, PWR, NER) with region_id and region_name.
   Uses a mapping from region names in the CSV to region_id codes and updates the park.region_id column for all parks that appear in the visitor data.
   Inserts one row per (park_code, year, month) into monthly_visit, populating all overnight‑stay metrics plus total_visits. Conflicts on the composite primary key are handled so the script can be rerun safely if needed.

In the final populated database, the region table contains seven records, the park table contains approximately 474 parks from the NPS API, and the monthly_visit table contains 46044 records covering the period from 2015 to 2024 for all parks with available data.

Indexing Strategy:

SQLite automatically creates indexes on all primary keys and foreign keys, which I rely on as the main indexing strategy:

- region(region_id) – primary key index supports fast lookups and joins from park.region_id
- park(park_code) – primary key index supports joins from monthly_visit.park_code and park‑level queries
- monthly_visit(park_code, year, month) – composite primary key index supports efficient filtering by park and year (and ordered by month), which is precisely how most analytical queries are written in the FastAPI backend

Given the relatively small size of the dataset (thousands, not millions, of rows), these automatic indexes are sufficient for interactive response times in the classroom and project setting.

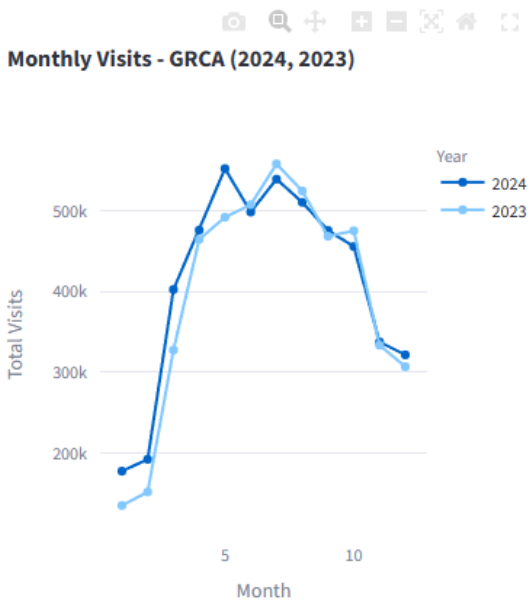
4. SQL Query Demonstrations (all queries are Parametized Queries)

Query 1 - Monthly visits by park and year
Endpoint: GET /parks/{park_code}/monthly-visits?year=YYYY&threshold=T
Files: main.py – function park_monthly_visits_with_threshold(...)
Business question:
For a given park and year, what are the monthly total visits, and how do they compare to other years? This helps operations planners identify peak months that may require increased staffing, shuttle frequency, or crowd management.
SQL statement

```
SELECT
    mv.month,
    mv.total_visits
FROM monthly_visit AS mv
WHERE
    mv.park_code = :park_code    -- e.g. 'GRCA'
    AND mv.year = :year          -- e.g. 2022
ORDER BY
    Mv.month;
```

Monthly Visits by Year

month	2023	2024
1	135139	177693
2	151927	192319
3	327573	402610
4	464731	476136
5	492023	552263
6	507642	498480
7	558123	539271
8	524542	510543
9	468536	475720
10	475260	455831



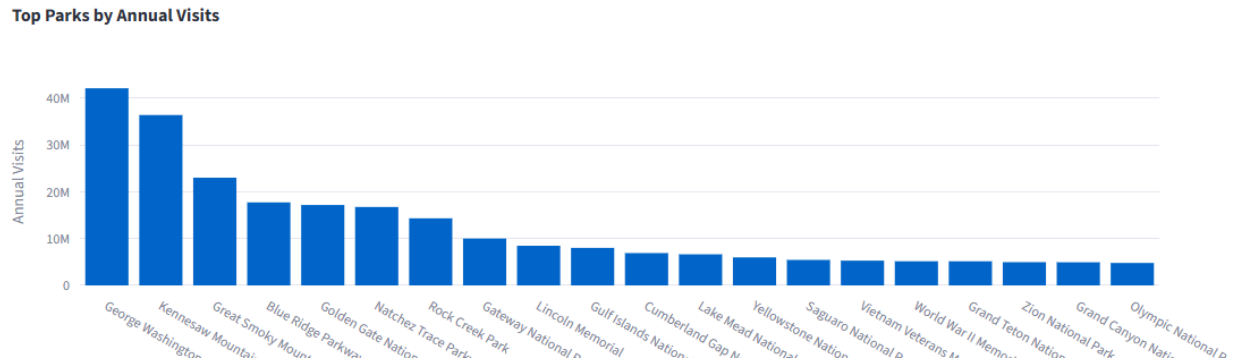
SQL concepts demonstrated

- Basic filtering with WHERE on park_code and year
- Ordering with ORDER BY month

Query 2 - Annual total visits per park (with flexible filters)
Endpoint: GET
/annual-visits/parks?year=YYYY&region_id=&park_code=&query=&min_total=&limit=
Files: main.py – function annual_visits_by_park(...)
Business question:
For a selected year, list all parks with their **region**, **location**, and **total annual visits**. Optionally filter by region, a specific park code, partial park name, and limit the number of returned parks. This supports ranking parks by demand and focusing on particular areas or high-volume units.
SQL statement

```
SELECT
    p.park_code,
    p.park_name,
    p.state,
    p.latitude,
    p.longitude,
    r.region_id,
    r.region_name,
    mv.year,
    SUM(mv.total_visits) AS annual_total
FROM park AS p
JOIN monthly_visit AS mv
    ON mv.park_code = p.park_code
LEFT JOIN region AS r
    ON r.region_id = p.region_id
WHERE
    mv.year = :year
    /* AND r.region_id = :region_id        (optional) */
    /* AND p.park_code = :park_code        (optional) */
    /* AND p.park_name LIKE '%' || :query || '%'  (optional name search) */
GROUP BY
    p.park_code,
    p.park_name,
    p.state,
    p.latitude,
    p.longitude,
    r.region_id,
    r.region_name,
    mv.year
HAVING
    (:min_total IS NULL OR SUM(mv.total_visits) >= :min_total)
ORDER BY
    annual_total DESC
LIMIT :limit;
```

	park_name	region_name	annual_total_visits
0	George Washington Memorial Parkway	National Capital	42140865
1	Kennesaw Mountain National Battlefield Park	Southeast	36421393
2	Great Smoky Mountains National Park	Southeast	23023834
3	Blue Ridge Parkway	Southeast	17743862
4	Golden Gate National Recreation Area	Pacific West	17187508
5	Natchez Trace Parkway	Southeast	16760441
6	Rock Creek Park	National Capital	14338439
7	Gateway National Recreation Area	Northeast	10013868
8	Lincoln Memorial	National Capital	8479349
9	Gulf Islands National Seashore	Southeast	8018582



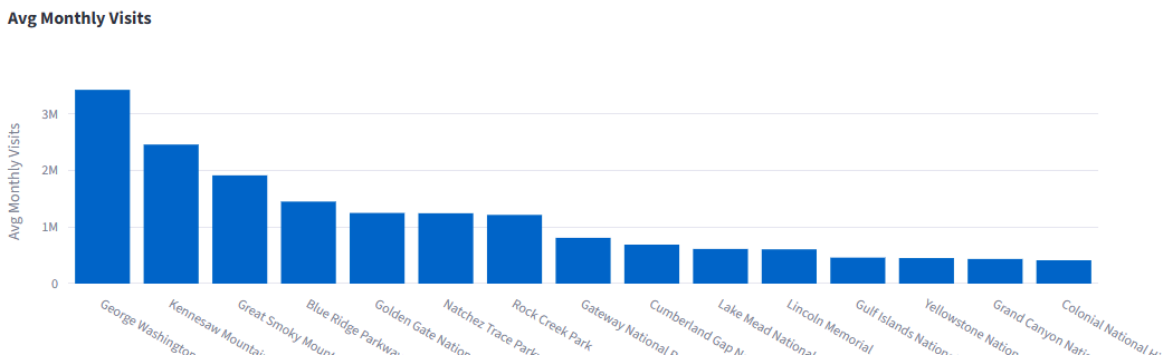
SQL concepts demonstrated

- JOIN operations: park ←→ monthly_visit and optional region
- Aggregation: SUM(mv.total_visits) to compute annual totals
- GROUP BY with aggregated values
- HAVING to filter on aggregated totals (min_total)
- ORDER BY with LIMIT for ranking and pagination

Query 3 - Average monthly visitation by park over a year range
Endpoint: GET
/visits/parks/average-monthly?start_year=&end_year=&region_id=&park_code=&query=&limit
=
Files: main.py – average_monthly_visits_by_park(...)
Business Question:
Over a selected multi-year window (e.g., 2022–2024), what is each park's average monthly total visitation? This helps planners understand typical demand levels, smoothing out short-term noise and unusual spikes.
SQL statement

```
SELECT
    p.park_code,
    p.park_name,
    r.region_id,
    r.region_name,
    AVG(mv.total_visits) AS avg_monthly
FROM park AS p
JOIN monthly_visit AS mv
    ON mv.park_code = p.park_code
LEFT JOIN region AS r
    ON r.region_id = p.region_id
WHERE
    mv.year BETWEEN :start_year AND :end_year
    /* AND r.region_id = :region_id                      (optional) */
    /* AND p.park_code = :park_code                      (optional) */
    /* AND p.park_name LIKE '%' || :query || '%'         (optional) */
GROUP BY
    p.park_code,
    p.park_name,
    r.region_id,
    r.region_name
ORDER BY
    avg_monthly DESC
LIMIT :limit;
```

	park_name	region_name	avg_monthly_visits
0	George Washington Memorial Parkway	National Capital	3418387
1	Kennesaw Mountain National Battlefield Park	Southeast	2453501
2	Great Smoky Mountains National Park	Southeast	1909456
3	Blue Ridge Parkway	Southeast	1447758
4	Golden Gate National Recreation Area	Pacific West	1246901
5	Natchez Trace Parkway	Southeast	1241210
6	Rock Creek Park	National Capital	1212621
7	Gateway National Recreation Area	Northeast	807600
8	Cumberland Gap National Historical Park	Southeast	688232
9	Lake Mead National Recreation Area	Pacific West	611782



SQL concepts demonstrated

- Basic filtering using BETWEEN start_year AND end_year
- Aggregation: AVG(mv.total_visits) across months and years
- JOINs across park, monthly_visit, and region
- GROUP BY to aggregate per park
- ORDER BY + LIMIT for ranking

Query 4 - Peak-season (June–August) average above threshold
Endpoint: GET /visits/peak-season/above-threshold?year=&threshold=&region_id=
Files: main.py – peak_season_above_threshold(...)
Business question:
In a given year, which parks have average peak-season (June–August) monthly visitation above a specified threshold? This helps identify parks that are at risk of peak-season congestion and may need operational interventions (reservations, shuttle systems, timed entry).

```
SQL Statement:
SELECT
    p.park_code,
    p.park_name,
    r.region_id,
    r.region_name,
    AVG(mv.total_visits) AS avg_monthly
FROM park AS p
JOIN monthly_visit AS mv
    ON mv.park_code = p.park_code
LEFT JOIN region AS r
    ON r.region_id = p.region_id
WHERE
    mv.year = :year
    AND mv.month IN (6, 7, 8)                -- peak season
    /* AND r.region_id = :region_id          (optional) */
GROUP BY
    p.park_code,
    p.park_name,
    r.region_id,
    r.region_name
HAVING
    AVG(mv.total_visits) >= :threshold
ORDER BY
    avg_monthly DESC;
```

Showing last Q4 results (Year 2024, Threshold 100000, Regions: All, Limit: 35)

	park_name	region_name	avg_monthly_visits
0	George Washington Memorial Parkway	National Capital	3745220
1	Kennesaw Mountain National Battlefield Park	Southeast	3065039
2	Great Smoky Mountains National Park	Southeast	2297761
3	Blue Ridge Parkway	Southeast	2208656
4	Golden Gate National Recreation Area	Pacific West	1789878
5	Natchez Trace Parkway	Southeast	1456661
6	Rock Creek Park	National Capital	1255935
7	Gateway National Recreation Area	Northeast	1127911
8	Yellowstone National Park	Intermountain	1106034
9	Grand Teton National Park	Intermountain	864579

SQL concepts demonstrated

- Filtering by year and a subset of months (IN (6,7,8))
- Aggregation: AVG of total_visits
- GROUP BY and HAVING on aggregated values
- JOIN between park, monthly_visit, and region

Query 5 - Parks above system-wide or region-wide average annual visits
Endpoint: GET /visits/parks/above-system-average?year=&region_id=&park_code=&query=
Files: main.py – parks_above_system_average(...)
Business question:
Which parks have total annual visits greater than the system-wide average (or the regional average if a region filter is provided) in the same year? This identifies parks that are demand outliers and may need extra resources, funding, and infrastructure.
SQL statement
Step 1 – compute annual totals per park and their average:

```
-- Annual totals per park (scope can be entire system or filtered to one region)
SELECT
    SUM(mv.total_visits) AS annual_total
FROM monthly_visit AS mv
JOIN park AS p
    ON p.park_code = mv.park_code
WHERE
    mv.year = :year
    /* AND p.region_id = :region_id        (optional) */
GROUP BY
    Mv.park_code;
```

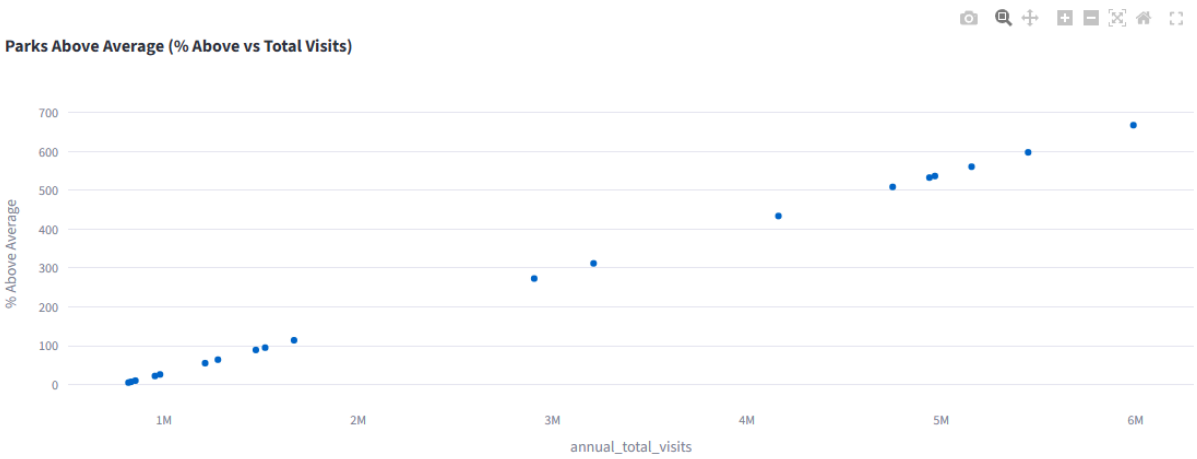
The API loads all annual_total values into Python and computes:

```
system_average = average of all annual_total values in this result set
```

Step 2 – retrieve parks whose total annual visits exceed that average:

```
SELECT
    p.park_code,
    p.park_name,
    r.region_id,
    r.region_name,
    mv.year,
    SUM(mv.total_visits) AS annual_total
FROM park AS p
JOIN monthly_visit AS mv
    ON mv.park_code = p.park_code
LEFT JOIN region AS r
    ON r.region_id = p.region_id
WHERE
    mv.year = :year
    /* AND r.region_id = :region_id        (optional) */
GROUP BY
    p.park_code,
    p.park_name,
    r.region_id,
    r.region_name,
    mv.year
HAVING
    SUM(mv.total_visits) > :system_average
ORDER BY
    annual_total DESC;
```

	park_name	region_name	annual_total_visits	percent_above_average
0	Yellowstone National Park	Intermountain	5989787	668
1	Saguaro National Park	Intermountain	5447967	598
2	Grand Teton National Park	Intermountain	5156987	561
3	Zion National Park	Intermountain	4968492	537
4	Grand Canyon National Park	Intermountain	4940173	533
5	Glen Canyon National Recreation Area	Intermountain	4750964	509
6	Rocky Mountain National Park	Intermountain	4163099	434
7	Glacier National Park	Intermountain	3211813	312
8	Bryce Canyon National Park	Intermountain	2906221	273
9	Chickasaw National Recreation Area	Intermountain	1670498	114



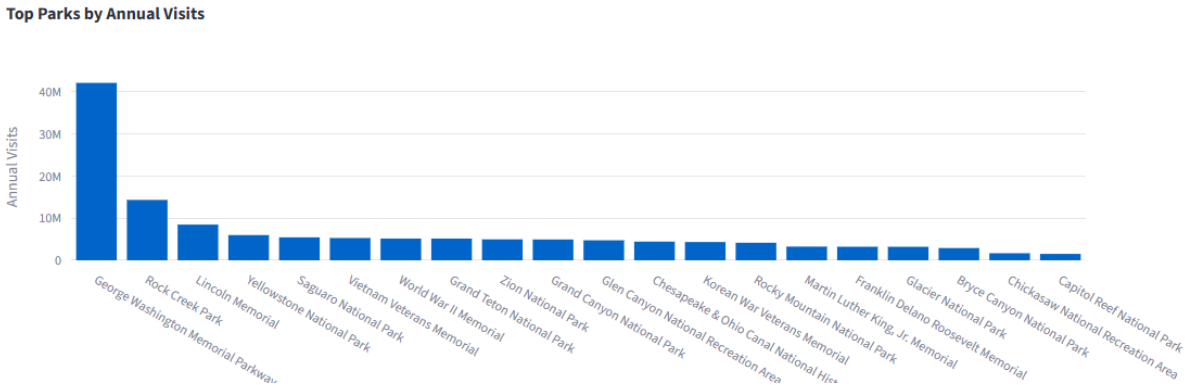
SQL concepts demonstrated

- Subquery-style aggregation: two-stage computation (annual totals, then comparison to average)
- Aggregation and GROUP BY on monthly_visit
- HAVING with parameterized average value
- JOINs with the park and region

Query 6 - Top N parks by annual visitation (with optional region filter)
Endpoint: GET /annual-visits/top?year=&limit=&region_id=&query=
Files: main.py – top_parks_by_year(...)
Business question:
What are the top N most visited parks in a given year (globally or within a specific region)? This supports ranking parks for resource allocation, marketing focus, and high-level strategic planning.
SQL statement:

```
SELECT
    p.park_code,
    p.park_name,
    mv.year,
    SUM(mv.total_visits) AS annual_total
FROM park AS p
JOIN monthly_visit AS mv
    ON mv.park_code = p.park_code
WHERE
    mv.year = :year
    /* AND p.region_id = :region_id        (optional) */
GROUP BY
    p.park_code,
    p.park_name,
    mv.year
ORDER BY
    annual_total DESC;
```

	park_name	region_name	annual_total_visits
82	George Washington Memorial Parkway	National Capital	42140865
83	Rock Creek Park	National Capital	14338439
84	Lincoln Memorial	National Capital	8479349
0	Yellowstone National Park	Intermountain	5989787
1	Saguaro National Park	Intermountain	5447967
85	Vietnam Veterans Memorial	National Capital	5295711
86	World War II Memorial	National Capital	5160769
2	Grand Teton National Park	Intermountain	5156987
3	Zion National Park	Intermountain	4968492
4	Grand Canyon National Park	Intermountain	4940173



SQL concepts demonstrated

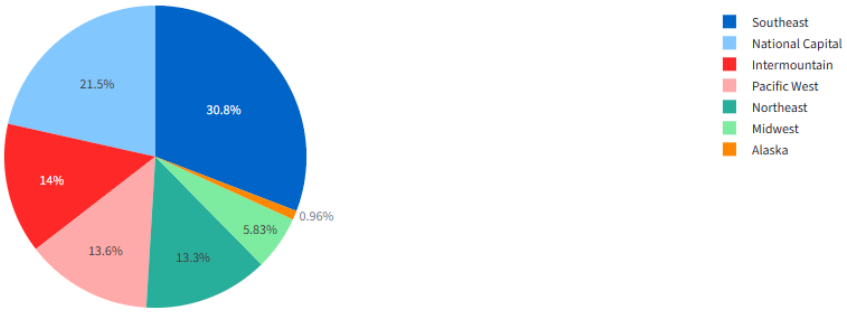
- Aggregation: SUM(total_visits) per park/year
- GROUP BY and ORDER BY to create a ranking
- ORDER BY with LIMIT pattern (limit is applied after in-memory filtering)

Query 7 - Total annual visits by region (ranked)
Endpoint: GET /annual-visits/regions?year=&region_id=
Files: main.py – annual_visits_by_region(...)
Business question:
For each NPS region, what is the total annual visitation in a given year, and how do regions rank from highest to lowest? This supports regional-level planning, budget allocation, and workload balancing.
SQL statement:

```
SELECT
    r.region_id,
    r.region_name,
    mv.year,
    SUM(mv.total_visits) AS annual_total
FROM region AS r
JOIN park AS p
    ON p.region_id = r.region_id
JOIN monthly_visit AS mv
    ON mv.park_code = p.park_code
WHERE
    mv.year = :year
    /* AND r.region_id = :region_id        (optional filter to one region) */
GROUP BY
    r.region_id,
    r.region_name,
    mv.year
ORDER BY
    annual_total DESC;
```

	region_id	region_name	year	annual_total_visits	rank
0	SER	Southeast	2024	141229421	1
1	NCR	National Capital	2024	98448183	2
2	IMR	Intermountain	2024	63963880	3
3	PWR	Pacific West	2024	62154003	4
4	NER	Northeast	2024	61014390	5
5	MWR	Midwest	2024	26693642	6
6	AKR	Alaska	2024	4396828	7

Regional Visit Distribution



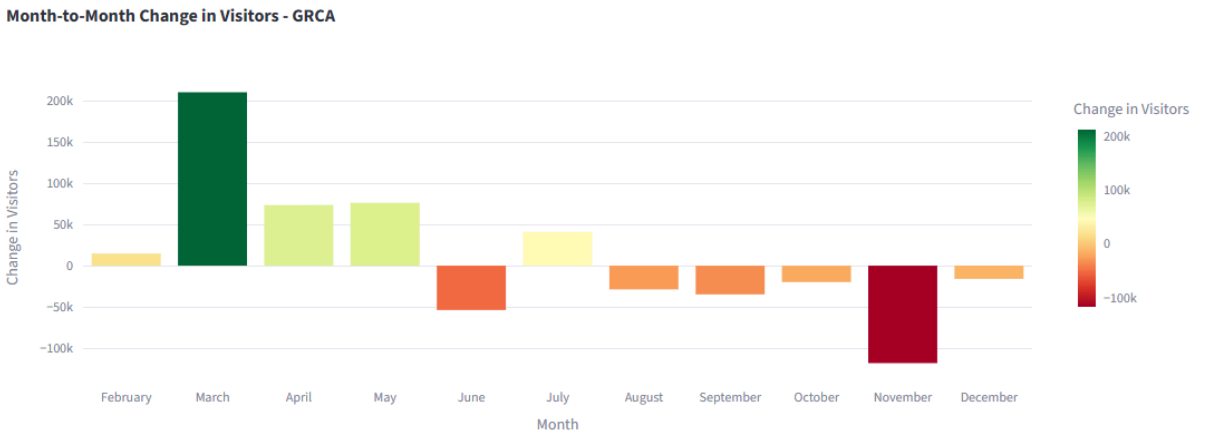
SQL concepts demonstrated

- 3-table JOIN (region–park–monthly_visit)
- Aggregation: total visits per region
- GROUP BY and ORDER BY to produce a regional ranking

Query 8 - Month-to-month change in visits for a park
Endpoint: GET /parks/{park_code}/month-to-month-change?year=
Files: main.py – month_to_month_change(...)
Business question:
For a selected park and year, what is the month-to-month change in total visits? This helps analysts identify sudden spikes or drops that may correspond to events, closures, or policy changes.
SQL statement

```
SELECT
    m1.month,
    m1.total_visits,
    (m1.total_visits - COALESCE(m2.total_visits, 0)) AS change_from_previous
FROM monthly_visit AS m1
LEFT JOIN monthly_visit AS m2
    ON m1.park_code = m2.park_code
    AND m1.year = m2.year
    AND m1.month = m2.month + 1
WHERE
    m1.park_code = :park_code
    AND m1.year = :year
ORDER BY
    m1.month;
```

	month_name	total_visits	change	change_percent
1	February	192319	14626	8.2311
2	March	402610	210291	109.3449
3	April	476136	73526	18.2623
4	May	552263	76127	15.9885
5	June	498480	-53783	-9.7387
6	July	539271	40791	8.1831
7	August	510543	-28728	-5.3272
8	September	475720	-34823	-6.8208
9	October	455831	-19889	-4.1808
10	November	337638	-118193	-25.9291



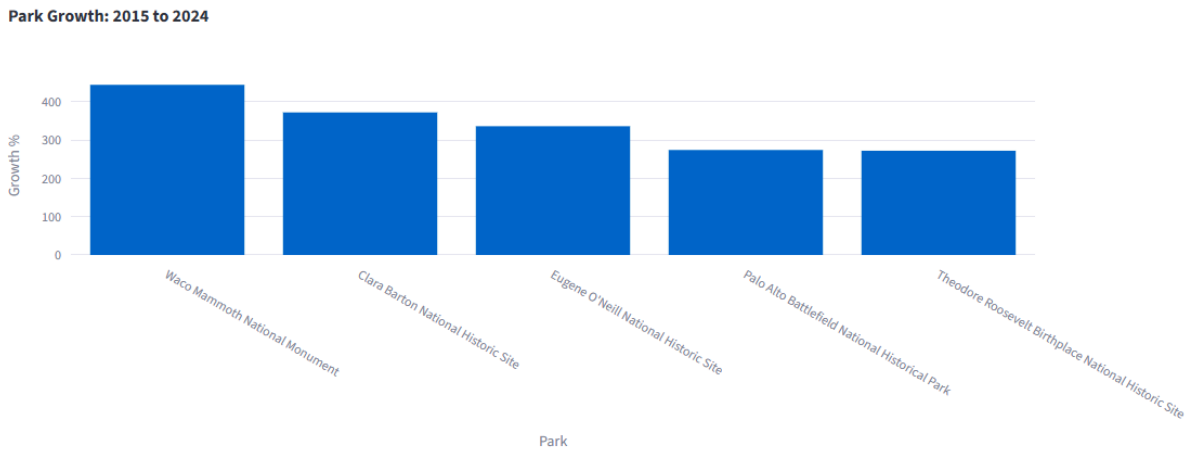
SQL concepts demonstrated

- Self-JOIN on the same table (monthly_visit)
- COALESCE to handle missing previous-month rows
- Filtering by park and year
- ORDER BY to get a chronological sequence

Query 9 - Growth in annual visits within a region over a time window
Endpoint: GET /regions/{region_id}/growth?start_year=&end_year=
Files: main.py – growth_by_region_over_time(...)
Business question:
For a given region and time window (e.g., 2022–2024), which parks show the highest percentage growth in total annual visitation? This identifies parks where demand is growing fastest, which is helpful for long-term capacity planning and investment decisions.
SQL statement

```
-- Annual totals by park and year (for start_year and end_year only)
WITH annual_totals AS (
    SELECT
        mv.park_code,
        mv.year,
        SUM(mv.total_visits) AS annual_total
    FROM monthly_visit AS mv
    JOIN park AS p
        ON p.park_code = mv.park_code
    WHERE
        p.region_id = :region_id
        AND mv.year IN (:start_year, :end_year)
    GROUP BY
        mv.park_code,
        mv.year
)
SELECT
    p.park_code,
    p.park_name,
    r.region_id,
    r.region_name,
    a_start.annual_total AS start_total,
    a_end.annual_total AS end_total
FROM park AS p
JOIN region AS r
    ON r.region_id = p.region_id
JOIN annual_totals AS a_start
    ON a_start.park_code = p.park_code
    AND a_start.year = :start_year
JOIN annual_totals AS a_end
    ON a_end.park_code = p.park_code
    AND a_end.year = :end_year
WHERE
    r.region_id = :region_id;
```

	park_code	park_name	region_id	region_name	start_year	end_year	start_total	end_total	growth_percent
15	WACO	Waco Mammoth National Monument	IMR	Intermountain	2015	2024	20597	112017	444
148	CLBA	Clara Barton National Historic Site	NCR	National Capital	2015	2024	2823	13334	372
249	EUON	Eugene O'Neill National Historic Site	PWR	Pacific West	2015	2024	4047	17653	336
16	PAAL	Palo Alto Battlefield National Historical Park	IMR	Intermountain	2015	2024	47866	178946	274
177	THRB	Theodore Roosevelt Birthplace National Historic Site	NER	Northeast	2015	2024	6776	25209	272



SQL concepts demonstrated

- Subquery / CTE-like pattern to compute annual totals by year
- JOINs between park, region, and the aggregated subquery
- Filtering by region and year window
- Aggregation and GROUP BY

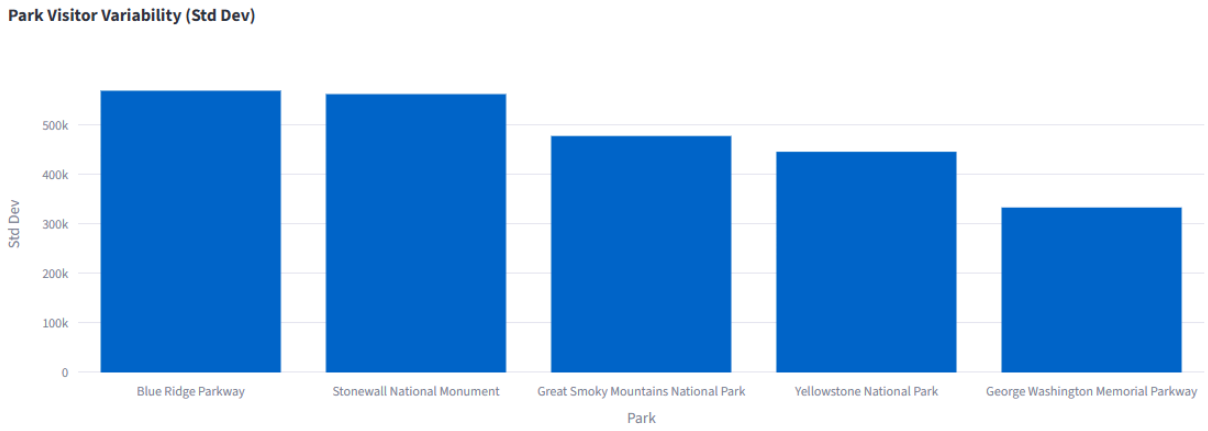
Query 10 - Variability (standard deviation) of monthly visits by park
Endpoint: GET /visits/parks/variability?year=&region_id=&park_code=&query=&limit=
Files: main.py – park_visit_variability(...)
Business question:
In a given year, which parks have the most variable (volatile) monthly visitation patterns? High variability indicates parks that are harder to staff and schedule (e.g., extreme peaks and troughs), whereas low-variability parks have more predictable demand.
SQL statement

```
SELECT
    p.park_code,
    p.park_name,
    r.region_id,
    r.region_name,
    mv.year,
    COUNT(mv.month) AS n_months,
    SUM(mv.total_visits) AS sum_v,
    SUM(mv.total_visits * mv.total_visits) AS sum_v2
FROM park AS p
JOIN monthly_visit AS mv
    ON mv.park_code = p.park_code
LEFT JOIN region AS r
    ON r.region_id = p.region_id
WHERE
    mv.year = :year
    /* AND r.region_id = :region_id              (optional) */
    /* AND p.park_code = :park_code              (optional) */
    /* AND p.park_name LIKE '%' || :query || '%' (optional) */
GROUP BY
    p.park_code,
    p.park_name,
    r.region_id,
    r.region_name,
    mv.year;
```

Then, in Python for each row:

```
n      = n_months
mean   = sum_v / n
var    = (sum_v2 / n) - (mean * mean)
stddev = sqrt(max(var, 0))
```

	park_name	region_name	std_dev_monthly_visits
0	Blue Ridge Parkway	Southeast	568951
1	Stonewall National Monument	Northeast	562093
2	Great Smoky Mountains National Park	Southeast	477548
3	Yellowstone National Park	Intermountain	445643
4	George Washington Memorial Parkway	National Capital	333092



SQL concepts demonstrated

- Aggregation: COUNT, SUM, and SUM(value²) in a single grouped query
- GROUP BY by park, region, and year
- Derived metrics (variance and standard deviation) computed in Python from aggregated SQL results

5. Backend API Implementation

The backend is implemented as a FastAPI application that exposes the NPS visitor analytics as a set of RESTful endpoints. The app is organized into three main modules:

- a database module that configures the SQLite engine and provides a get_session() dependency for opening and closing SQLModel sessions,
- a models module that defines the ORM entities (Region, Park, MonthlyVisit), and
- a main module that declares response schemas and implements all API routes.

All endpoints follow standard REST conventions: resources are grouped by path (/parks/..., /regions/..., /annual-visits/..., /visits/...) and accessed via HTTP GET. Path parameters are used for natural resources (e.g., /parks/{park_code}/monthly-visits, /regions/{region_id}/growth), while query parameters control filters and analytical options (e.g., year, start_year, end_year, region_id, threshold, limit, metric, search query). The API returns JSON responses suitable for programmatic consumption by the Streamlit frontend. CORS middleware is enabled, allowing the frontend to call the API from a different origin during local development.

Pydantic validation is handled through SQLModel response models, which are explicitly defined for each query (e.g., MonthlyThresholdOut, AnnualParkVisitsOut, ParkAboveAverageOut, RegionAnnualVisitsOut, GrowthOut, VariabilityOut, MetricParkOut, ParkDetailOut). These classes specify field types, optional vs. required attributes, and enforce consistent shapes for all responses. FastAPI automatically validates incoming parameters (e.g., int years, str region IDs) and serializes ORM results into these Pydantic models, ensuring that both backend and frontend work with well-typed data.

Error handling is implemented using FastAPI's HTTPException. Endpoints check for common failure cases and return informative HTTP status codes:

- 404 Not Found when a park, region, or year combination has no data (e.g., no rows in monthly_visit for the requested park/year)
- 400 Bad Request for invalid logical inputs (e.g., start_year > end_year in range-based queries)

This makes failure modes explicit and easier to handle on the frontend.

Database access is performed via SQLModel's ORM query builder (select, join, group_by, func.sum, func. avg, etc.) instead of string-concatenated SQL. This effectively gives parameterized queries "for free," because user inputs (such as park_code, region_id, year, thresholds, and metric names) are always bound as parameters or validated against a controlled mapping. For example, the variability and metrics endpoints map user-supplied metric names to explicit model attributes, rather than interpolating them into raw SQL, thereby preventing SQL injection. Each request receives its own short-lived session from get_session(), which is cleanly closed after use, ensuring correct ORM usage and avoiding connection leaks.

6. Frontend Implementation

The frontend is implemented as a lightweight Streamlit web app that acts as the "control panel" for the NPS Visitor Analytics API. Its primary goals are to (1) make the 10 analytical queries accessible to non-technical users and (2) present results in a way that directly supports planning and decision-making.

User interface design and features

The application is structured around a sidebar + main-view layout:

- Sidebar controls
    - Year
    - Optional region filter (drop-down populated from /regions/)
    - Park selector (populated from parks returned for the selected region)

- Main view
    - A short description card explaining what the current view shows ("What question is this answering?").
    - One or more tables of results (e.g., parks ranked by annual visits, monthly visits for a selected park).
    - Where appropriate, charts:
    - Line chart for monthly visits or month-to-month changes (Q1, Q8).
    - Bar chart for top parks or regions (Q6, Q7, Q11).
    - Optional growth/variability plots for Q9 and Q10.

This is intentionally kept to a single page with dynamic content rather than multiple separate pages, which aligns with the project requirement of "at least one frontend page" while still offering rich interaction.

How the frontend consumes the backend API

The Streamlit app communicates with the FastAPI backend using standard HTTP GET requests. For each interactive control, the app builds a URL to the appropriate endpoint, including query parameters derived from the user's selections. Examples include:

- /regions/ to populate the region selector.
- /parks/{park_code}/monthly-visits?year=2023&threshold=500000 for Q1.
- /annual-visits/parks?year=2023&region_id=IMR&min_total=0 for Q2/Q10.
- /annual-visits/top?year=2023&limit=10 for Q6.
- /regions/IMR/growth?start_year=2022&end_year=2024 for Q9.

Responses are returned as JSON, which Streamlit converts into Python dictionaries or Pandas DataFrames for easy display and plotting. All input is passed as parameters (never interpolated

into raw SQL), so the frontend interacts only with the FastAPI layer and never directly with the database.

Data visualization approach
The frontend uses a mix of:

- Tabular views for precise values and rankings:
    - Monthly visits per park (month, total, threshold flag).
-
    - Annual totals per park or region.
-
    - Growth and variability metrics.
-
- Charts to highlight trends and comparisons:
-
    - Line charts of monthly visits and month-to-month changes (Q1, Q8) to reveal seasonality, peaks, and drops.
    - Bar charts of:
    - Top N parks by annual visitation (Q6).
    - Annual visits by region (Q7).
    - Metric-specific sums (Q11).
- Geographic context (optional):

When park latitude/longitude (and boundary geometry) are available, the frontend can plot points or boundaries on a simple map to tie demand patterns back to location, which is helpful for planners coordinating multi-park or cross-region strategies.
The interactive map tab also provides the user with general information and a link to the official park website.

User interaction flow
A typical user flow looks like this:

1. Start with a high-level view.

    - Select a year and open the “Top Parks” or “Regions ranking” view.
    - Inspect which parks or regions dominate in total visits.

2. Drill down with filters.

    - Narrow to a specific region using the region selector.
    - The park selector updates to show only parks in that region (using /annual-visits/parks or /visits/parks/average-monthly results).

3. Zoom into a single park.

    - Choose a park from the list and switch to views such as:
        - “Monthly demand profile” (Q1).
        - “Month-to-month change” (Q8).
    - Adjust thresholds to see which months are “problem” periods for capacity.

4. Analyze patterns across years or metrics.

    - Use the multi-year average and growth views (Q3, Q9, Q10) to understand trends and volatility.
    - Use the metric-specific annual view (Q11) to focus on particular operational dimensions (e.g., concessioner lodging vs. tent campers).

This flow mirrors how an industrial engineer or park operations planner works: start with system-wide patterns, filter down by region, then focus on individual parks and specific time windows or metrics. The frontend’s job is to make all 10 analytical queries feel like natural, guided steps in that workflow rather than isolated SQL statements.

7. Setup and Usage Instructions

Because this project is implemented as a full client–server application, there are two main components to set up: the backend API (FastAPI + SQLite) and the frontend dashboard (Streamlit). Detailed, step-by-step installation commands are already documented in the project's README.md file, including instructions for creating a virtual environment, installing dependencies, configuring environment variables, and starting both the backend and frontend. In this report, I summarize the key steps and then focus on example usage scenarios from an end-user perspective.

Setup Summary (see README for full details)

At a high level, the setup process is:

1. Clone the repository and create a virtual environment
    - Clone the GitHub repo and create/activate a Python virtual environment.
    - Install all required dependencies with pip install -r backend/requirements.txt

2. Configure environment variables
    - Create a .env file in the project root and add the NPS API key (used only for initial data population).

3. Initialize and populate the database
    - Run database/create_db.py to create an empty nps.db with the schema.
    - Run backend/fetch_data.py to pull park metadata and boundaries from the NPS API into the park table.
    - Run backend/load_csv.py to load monthly visitor statistics into monthly_visit.

4. Start the backend API
    - From the backend/ folder, start FastAPI with Uvicorn (e.g., python -m uvicorn main:app --reload --host 127.0.0.1 --port 8000).
    - The interactive API docs are available at http://127.0.0.1:8000/docs.

5. Start the frontend dashboard.
    - From the project root, run python -m streamlit run frontend/app.py.
    - Streamlit opens the NPS Park Operations Dashboard at http://localhost:8501.

Example Usage Scenarios:

Scenario 1 – Analyze peak demand for a single park (operations manager):
A regional operations manager wants to understand summer demand at Grand Canyon National Park for staffing decisions.

1. Open the Streamlit dashboard and use the global park search to find “Grand Canyon” or GRCA.
2. Select analysis year (e.g., 2023) in the sidebar.
3. Choose Q1: Monthly Visits and run the query for GRCA.
4. The dashboard displays monthly total visits and a line chart, highlighting peak months (June–August).
5. The manager uses this to align seasonal hiring, shuttle operations, and maintenance shutdowns with observed demand peaks.

Scenario 2 – Compare parks within a region for resource allocation (regional planner):
A regional planner wants to see which parks in the Intermountain Region (IMR) have the highest annual visitation in 2024.

1. In the sidebar, select Region = IMR and Year = 2024.
2. Choose Q2: Annual by Park or Q6: Top Parks by Annual Visits and run the query.
3. The frontend calls GET /annual-visits/parks or GET /annual-visits/top with the selected filters and displays a sorted table, along with a bar chart.
4. The planner quickly identifies the busiest parks in the region and uses this ranking to prioritize staffing, funding, and capital projects.

8. Testing and Challenges

The system was tested iteratively throughout development, with a focus on correctness, stability, and real-world usability.

1. Unit-style testing of backend queries
Each FastAPI endpoint was tested independently using the automatic Swagger UI (/docs). This allowed quick validation of:

- SQL correctness
- Parameter handling and validation
- Case-insensitive park and region lookups
- Meaningful error messages (404 for missing parks/years, 400 for invalid ranges)

I would run every query with varying inputs to ensure robust behavior and ensure that each request ran successfully.
2. Frontend/Backend integration testing
When the Streamlit frontend was added, I tested:

- API connectivity (CORS working)
- Response parsing into tables and charts
- User interface behavior (filters, dropdowns, sliders)
- Handling of empty results or missing data
- Performance when retrieving large region-wide datasets

I would rigorously test the frontend to ensure that each part of it worked well without any visual bugs or errors.

One of the most challenging parts of the project was merging the two completely different data sources: the NPS API (which provides park metadata, coordinates, descriptions, and boundaries) and the CSV visitor dataset (which provides monthly visitation counts, overnight stays, and region names). The data sets did not align, but they both shared the same park_code, which I realized would be the key to loading them. But some problems included that many parks appear in one but not the other, fields use different naming conventions, and the API data arrives as nested JSON while the CSV is a flat table. Initially, I attempted to load everything into a single script, but this quickly became unmanageable and made debugging extremely difficult.
Solution:
I eventually realized the data loading process needed to be separated into two dedicated scripts, each responsible for only one dataset:

- fetch_data.py — loads park metadata from the NPS API
- load_csv.py — loads region and monthly visitation data from the CSV file

After they load independently, the scripts reconcile fields such as park_code and region_id within SQLite. This modular design made the process much easier to test, debug, and rerun. Separating the ETL flow also ensured that parks missing visitation data would not break the

database, while still allowing fully enriched metadata for all parks. The challenge ultimately improved the system's structure and reinforced the value of modular, maintainable data pipelines.

Lessons learned

I learned about the amount of work it takes to set up a personal information system. This project definitely took a lot of time, but it was also super fun and rewarding to eventually come up with a working solution. I learned that working through it from the back forward, meaning starting with the database and then the backend integration, followed by the frontend integration, was the best approach. From there, you are set up to add what you need, if anything, simply. This project demonstrated to me how these kinds of systems are built end-to-end, and it effectively highlighted the practical challenges of real data integration, SQL design, and interactive visualization. This was a much better experience than taking an exam, even if I spent many more hours working on this compared to what I would have spent on an exam. The lessons I learned from this I will definitely keep with me and implement in my future.

Generative AI Disclosure:

ChatGPT and GitHub Copilot were heavily utilized throughout the project's development to support productivity, accuracy, and organization. ChatGPT primarily served as a high-level planning and clarification tool, helping to break down project requirements, structure the workflow, identify dependencies, and explain unfamiliar concepts such as SQLModel patterns, API design, and database normalization. It also assisted in generating documentation drafts and refining the wording of analytical explanations. All final architectural decisions, code integration, and debugging steps were performed manually.

GitHub Copilot played a different but equally critical role by accelerating code writing within the editor. Copilot provided autocomplete suggestions for repetitive SQLModel patterns, endpoint scaffolding, and common FastAPI structures. It was beneficial during debugging, where it suggested corrections for function signatures, SQL joins, and model attributes. However, every Copilot suggestion was reviewed, tested, and often modified to match the specific schema, business logic, and analytical goals of the project.

Overall, AI tools enhanced efficiency and helped maintain consistency. Still, the core system design, data integration strategy, SQL query logic, testing, debugging, and frontend integration were completed through my own understanding and manual implementation.

Link to GitHub Repo

https://github.com/edozal3/ED305Project