2. Applied ML on ACE Intelligence System

Objective: transforming from retrospective analysis to forward-looking intelligence that predicts where violations will cluster, enabling proactive camera deployment rather than reactive ticketing.

What We're Building:

A comprehensive feature-rich dataset that captures:

- **Temporal intelligence**: rush hour, school hours, CUNY class changes
- **Spatial intelligence**: GTFS integration, violation clustering, CUNY proximity
- **Enforcement adaptation**: violator learning patterns, predictability entropy
- **Multiple prediction targets**: immediate, tactical, and strategic forecasting

Goal: enabling prediction of violation hotspots 24 hours ahead for proactive deployment.

In [35]:
```
# importing essential libraries for comprehensive feature engineering
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime, timedelta
import math
import warnings
from pathlib import Path
import gc
from typing import Dict, List, Tuple
import pickle
from sklearn.cluster import DBSCAN
from scipy.spatial.distance import pdist, squareform
from scipy import stats
import json
import itertools

warnings.filterwarnings('ignore')
plt.style.use('default')
sns.set_palette("husl")

print("ACE Intelligence System - Feature Engineering Module Loaded")
print("Ready to process 3.78M violations and build predictive features")
print("Goal: transforming from reactive enforcement to predictive intelligence")
```

```
ACE Intelligence System - Feature Engineering Module Loaded
Ready to process 3.78M violations and build predictive features
Goal: transforming from reactive enforcement to predictive intelligence
```

Section 1: Data Loading and Memory Optimization

loading the full violations dataset (3.78M records) with optimal memory usage based on the data dictionary specifications.

In [36]:
```
# defining optimal data types based on MTA data dictionary
VIOLATIONS_DTYPES = {
    'Violation ID': 'int64',
    'Vehicle ID': 'string',  # hashed values
    'Violation Status': 'category',
    'Violation Type': 'category',
    'Bus Route ID': 'string',
    'Violation Latitude': 'float32',
    'Violation Longitude': 'float32',
    'Stop ID': 'string',
    'Stop Name': 'string',
    'Bus Stop Latitude': 'float32',
    'Bus Stop Longitude': 'float32'
}

# date columns will be parsed separately
DATE_COLUMNS = ['First Occurrence', 'Last Occurrence']

print("Optimized data types configured for memory efficiency")
print(f"Expected memory reduction: ~40% vs default dtypes")

# Memory monitoring function
def get_memory_usage():
    """monitoring memory usage during processing"""
    import psutil
    process = psutil.Process()
    return process.memory_info().rss / 1024 / 1024  # MB

print(f"Current memory usage: {get_memory_usage():.1f} MB")
```

```
Optimized data types configured for memory efficiency
Expected memory reduction: ~40% vs default dtypes
Current memory usage: 3716.2 MB
```

In [37]:
```
# loading the full violations dataset with chunked processing if needed
violations_file = "../data/MTA_Bus_Automated_Camera_Enforcement_Violations__Beginni

print("Loading full violations dataset (3.78M records)...")
print("This may take 2-3 minutes for comprehensive processing")

# check file size to determine loading strategy
file_size_gb = Path(violations_file).stat().st_size / (1024**3)
print(f"File size: {file_size_gb:.2f} GB")

if file_size_gb > 0.5:  # use chunked loading for large files
    print("Using chunked processing for optimal memory usage")

    chunk_size = 100000  # Process 100K records at a time
    processed_chunks = []
    total_rows = 0

    # first pass: get total row count and data quality metrics
```

```
    print("Phase 1: data quality assessment...")

    for i, chunk in enumerate(pd.read_csv(violations_file, chunksize=chunk_size, dt
        total_rows += len(chunk)
        if i == 0:
            print(f"Columns loaded: {list(chunk.columns)}")
            print(f"Date range sample: {chunk['First Occurrence'].min()} to {chunk[

        if i % 10 == 0:
            print(f"   Processed {total_rows:,} rows... ({get_memory_usage():.1f} M

        # break after processing enough for assessment
        if i >= 5:  # process ~500K records for initial assessment
            break

    print(f"Assessment complete: {total_rows:,} rows processed")

    # now load the full dataset with optimized approach
    print("Phase 2: full dataset loading...")

    violations_data = pd.read_csv(
        violations_file,
        dtype=VIOLATIONS_DTYPES,
        parse_dates=DATE_COLUMNS,
        low_memory=False
    )
else:
    # direct loading for smaller files
    violations_data = pd.read_csv(
        violations_file,
        dtype=VIOLATIONS_DTYPES,
        parse_dates=DATE_COLUMNS
    )

print(f"Violations data loaded successfully!")
print(f"Shape: {violations_data.shape}")
print(f"Memory usage: {get_memory_usage():.1f} MB")
print(f"Date range: {violations_data['First Occurrence'].min()} to {violations_data
```

```
Loading full violations dataset (3.78M records)...
This may take 2-3 minutes for comprehensive processing
File size: 1.02 GB
Using chunked processing for optimal memory usage
Phase 1: data quality assessment...
Columns loaded: ['Violation ID', 'Vehicle ID', 'First Occurrence', 'Last Occurrenc
e', 'Violation Status', 'Violation Type', 'Bus Route ID', 'Violation Latitude', 'Vio
lation Longitude', 'Stop ID', 'Stop Name', 'Bus Stop Latitude', 'Bus Stop Longitud
e', 'Violation Georeference', 'Bus Stop Georeference']
Date range sample: 2025-05-31 15:15:44 to 2025-08-21 19:40:47
   Processed 100,000 rows... (3644.6 MB)
Assessment complete: 600,000 rows processed
Phase 2: full dataset loading...
Violations data loaded successfully!
Shape: (3778568, 15)
Memory usage: 3775.7 MB
Date range: 2019-10-07 07:06:54 to 2025-08-21 19:40:47
```

In [38]:
```
# generating data quality report
print("DATA QUALITY REPORT")
print("=" * 50)

# missing values analysis
missing_data = violations_data.isnull().sum()
missing_pct = (missing_data / len(violations_data)) * 100

quality_report = pd.DataFrame({
    'Missing_Count': missing_data,
    'Missing_Percentage': missing_pct.round(2),
    'Data_Type': violations_data.dtypes
})

print("Missing Values Summary:")
print(quality_report[quality_report['Missing_Count'] > 0])

# route coverage analysis
print(f"\nRoute Coverage:")
print(f"   Unique routes: {violations_data['Bus Route ID'].nunique()}")
print(f"   Top 10 routes by violations:")
route_counts = violations_data['Bus Route ID'].value_counts().head(10)
for route, count in route_counts.items():
    print(f"   {route}: {count:,} violations")

# temporal coverage
print(f"\nTemporal Coverage:")
print(f"   Start date: {violations_data['First Occurrence'].min()}")
print(f"   End date: {violations_data['First Occurrence'].max()}")
print(f"   Time span: {(violations_data['First Occurrence'].max() - violations_data

# geographic bounds check
print(f"\nGeographic Bounds:")
print(f"   Lat range: {violations_data['Violation Latitude'].min():.4f} to {violati
print(f"   Lon range: {violations_data['Violation Longitude'].min():.4f} to {violat
```

```
DATA QUALITY REPORT
==================================================
Missing Values Summary:
              Missing_Count  Missing_Percentage       Data_Type
Vehicle ID            66366                1.76  string[python]
Bus Route ID          10749                0.28  string[python]

Route Coverage:
   Unique routes: 40
   Top 10 routes by violations:
   M15+: 502,765 violations
   BX19: 344,147 violations
   M101: 312,466 violations
   BX41+: 229,920 violations
   BX36: 225,835 violations
   B46+: 186,192 violations
   BX12+: 185,059 violations
   B44+: 171,083 violations
   Q44+: 164,806 violations
   BX6+: 111,403 violations

Temporal Coverage:
   Start date: 2019-10-07 07:06:54
   End date: 2025-08-21 19:40:47
   Time span: 2145 days

Geographic Bounds:
   Lat range: 40.5345 to 40.8811
   Lon range: -74.1844 to -73.7010
```

Section 2: Temporal Feature Engineering

creating sophisticated temporal features that capture:

- standard time patterns (hour, day, month)
- NYC-specific patterns (rush hour, school hours)
- CUNY-specific patterns (class changes, semester cycles)
- Enforcement timeline (days since ACE implementation)

In [39]:
```
print("TEMPORAL FEATURE ENGINEERING")
print("=" * 50)

# creating basic temporal features
print("Creating basic temporal features...")
violations_data['violation_datetime'] = violations_data['First Occurrence']
violations_data['violation_hour'] = violations_data['violation_datetime'].dt.floor(
violations_data['hour_of_day'] = violations_data['violation_datetime'].dt.hour
violations_data['day_of_week'] = violations_data['violation_datetime'].dt.dayofweek
violations_data['month'] = violations_data['violation_datetime'].dt.month
violations_data['year'] = violations_data['violation_datetime'].dt.year
violations_data['day_of_year'] = violations_data['violation_datetime'].dt.dayofyear

# weekend/weekday classification
```

```python
violations_data['is_weekend'] = violations_data['day_of_week'].isin([5, 6])  # Sat,

print("Basic temporal features created")

# creating NYC-specific temporal features
print("Creating NYC-specific temporal features...")

# rush hour classification (more nuanced than simple 7-9, 5-7)
def classify_rush_hour(hour, day_of_week):
    """classifying rush hour periods with weekday/weekend distinction"""
    if day_of_week in [5, 6]:  # weekend
        if 10 <= hour <= 14:  # weekend midday rush
            return 'weekend_midday'
        elif 18 <= hour <= 21:  # weekend evening activity
            return 'weekend_evening'
        else:
            return 'weekend_off_peak'
    else:  # weekday
        if 7 <= hour <= 9:  # morning rush
            return 'morning_rush'
        elif 17 <= hour <= 19:  # evening rush
            return 'evening_rush'
        elif 10 <= hour <= 16:  # midday
            return 'midday'
        elif 20 <= hour <= 23:  # evening activity
            return 'evening_activity'
        else:  # late night/early morning
            return 'off_peak'

violations_data['rush_hour_period'] = violations_data.apply(
    lambda x: classify_rush_hour(x['hour_of_day'], x['day_of_week']), axis=1
)

# binary rush hour flags
violations_data['is_morning_rush'] = (violations_data['hour_of_day'].between(7, 9))
violations_data['is_evening_rush'] = (violations_data['hour_of_day'].between(17, 19
violations_data['is_any_rush'] = violations_data['is_morning_rush'] | violations_da

print("Rush hour features created")

# school hours (affects violation patterns)
violations_data['is_school_hours'] = (
    (violations_data['hour_of_day'].between(8, 15)) &
    (~violations_data['is_weekend'])
)

print("School hours features created")
```

```
TEMPORAL FEATURE ENGINEERING
==================================================
Creating basic temporal features...
Basic temporal features created
Creating NYC-specific temporal features...
Rush hour features created
School hours features created
```

In [40]:
```
# creating CUNY-specific temporal features
print("Creating CUNY-specific temporal features...")

# CUNY class change periods (every hour 8am-6pm during academic periods)
violations_data['is_cuny_class_change'] = (
    (violations_data['hour_of_day'].between(8, 18)) &
    (~violations_data['is_weekend']) &
    (violations_data['violation_datetime'].dt.minute.between(50, 10))  # 10 min bef
)

# academic semester patterns (approximate)
def get_semester_period(date):
    """classifying academic periods"""
    month = date.month
    if month in [9, 10, 11, 12]:  # Fall semester
        return 'fall_semester'
    elif month in [1, 2, 3, 4, 5]:  # Spring semester
        return 'spring_semester'
    else:  # Summer
        return 'summer_session'

violations_data['semester_period'] = violations_data['violation_datetime'].apply(ge
violations_data['is_academic_year'] = violations_data['semester_period'].isin(['fal

print("CUNY temporal features created")

# creating ACE enforcement timeline features
print("Creating ACE enforcement timeline features...")

# ACE program timeline markers
ace_implementation_date = datetime(2024, 6, 1)  # Major expansion
ace_pilot_start = datetime(2019, 10, 1)  # Initial pilot

violations_data['days_since_ace_implementation'] = (
    violations_data['violation_datetime'] - ace_implementation_date
).dt.days

violations_data['days_since_ace_pilot'] = (
    violations_data['violation_datetime'] - ace_pilot_start
).dt.days

violations_data['is_post_ace_expansion'] = violations_data['violation_datetime'] >=
violations_data['is_ace_pilot_period'] = (
    (violations_data['violation_datetime'] >= ace_pilot_start) &
    (violations_data['violation_datetime'] < ace_implementation_date)
)

print("ACE timeline features created")

# creating temporal volatility features (for enforcement adaptation analysis)
print("Creating temporal volatility features...")

# This will be calculated later after aggregation by location
print("Temporal volatility will be calculated after spatial aggregation")
```

```
print(f"\nTemporal Features Summary:")
temporal_features = [
    'hour_of_day', 'day_of_week', 'month', 'is_weekend',
    'rush_hour_period', 'is_morning_rush', 'is_evening_rush',
    'is_school_hours', 'is_cuny_class_change', 'semester_period',
    'days_since_ace_implementation', 'is_post_ace_expansion'
]
print(f"Created {len(temporal_features)} temporal features")
for feature in temporal_features:
    print(f"   • {feature}")
```

```
Creating CUNY-specific temporal features...
CUNY temporal features created
Creating ACE enforcement timeline features...
ACE timeline features created
Creating temporal volatility features...
Temporal volatility will be calculated after spatial aggregation

Temporal Features Summary:
Created 12 temporal features
    • hour_of_day
    • day_of_week
    • month
    • is_weekend
    • rush_hour_period
    • is_morning_rush
    • is_evening_rush
    • is_school_hours
    • is_cuny_class_change
    • semester_period
    • days_since_ace_implementation
    • is_post_ace_expansion
```

Section 3: Spatial Intelligence Features

building comprehensive spatial features by:

- loading GTFS data from all boroughs
- creating violation hotspot clusters
- building spatial intelligence for prediction

In [41]:
```
print("SPATIAL INTELLIGENCE FEATURE ENGINEERING")
print("=" * 50)

# loading GTFS data from all boroughs
print("Loading GTFS data from all boroughs...")

gtfs_boroughs = ['bronx', 'brooklyn', 'manhattan', 'queens', 'staten_island']
all_stops = []
all_routes = []
all_shapes = []

for borough in gtfs_boroughs:
    gtfs_path = Path(f"../data/gtfs/{borough}")
```

```
    if gtfs_path.exists():
        print(f"   Loading {borough} GTFS data...")

        # load stops
        stops_file = gtfs_path / "stops.txt"
        if stops_file.exists():
            stops_df = pd.read_csv(stops_file)
            stops_df['borough'] = borough
            all_stops.append(stops_df)

        # load routes
        routes_file = gtfs_path / "routes.txt"
        if routes_file.exists():
            routes_df = pd.read_csv(routes_file)
            routes_df['borough'] = borough
            all_routes.append(routes_df)

        # load shapes (if available)
        shapes_file = gtfs_path / "shapes.txt"
        if shapes_file.exists():
            shapes_df = pd.read_csv(shapes_file)
            shapes_df['borough'] = borough
            all_shapes.append(shapes_df)

# combine all GTFS data
if all_stops:
    gtfs_stops = pd.concat(all_stops, ignore_index=True)
    print(f"Loaded {len(gtfs_stops):,} stops from {len(all_stops)} boroughs")
else:
    print("No GTFS stops data found, will use violation coordinates for spatial ana
    gtfs_stops = None

if all_routes:
    gtfs_routes = pd.concat(all_routes, ignore_index=True)
    print(f"Loaded {len(gtfs_routes):,} routes from {len(all_routes)} boroughs")
else:
    gtfs_routes = None

if all_shapes:
    gtfs_shapes = pd.concat(all_shapes, ignore_index=True)
    print(f"Loaded {len(gtfs_shapes):,} shape points from {len(all_shapes)} borough
else:
    gtfs_shapes = None
```

```
SPATIAL INTELLIGENCE FEATURE ENGINEERING
==================================================
Loading GTFS data from all boroughs...
   Loading bronx GTFS data...
   Loading brooklyn GTFS data...
   Loading manhattan GTFS data...
   Loading queens GTFS data...
   Loading staten_island GTFS data...
Loaded 11,698 stops from 5 boroughs
Loaded 1,440 routes from 5 boroughs
Loaded 371,158 shape points from 5 boroughs
```

In [42]:
```
# creating violation hotspot clusters using DBSCAN
print("Creating violation hotspot clusters...")

# filter out invalid coordinates
valid_coords = violations_data[
    (violations_data['Violation Latitude'].notna()) &
    (violations_data['Violation Longitude'].notna()) &
    (violations_data['Violation Latitude'].between(40.4, 41.0)) &  # NYC bounds
    (violations_data['Violation Longitude'].between(-74.5, -73.5))
].copy()

print(f"Valid coordinates: {len(valid_coords):,} out of {len(violations_data):,} vi

# Haversine distance function for geographic clustering
def haversine_distance(lat1, lon1, lat2, lon2):
    """calculating haversine distance between two points in meters"""
    R = 6371000  # earth radius in meters
    lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = math.sin(dlat/2)**2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon/2)**2
    return 2 * R * math.asin(math.sqrt(a))

# sample for clustering if dataset is very large
if len(valid_coords) > 50000:
    print(f"Sampling {50000} violations for clustering analysis...")
    clustering_sample = valid_coords.sample(n=50000, random_state=42)
else:
    clustering_sample = valid_coords

# DBSCAN clustering (eps in degrees, roughly 100 meters)
coords_for_clustering = clustering_sample[['Violation Latitude', 'Violation Longitu

# converting to radians for proper distance calculation
coords_rad = np.radians(coords_for_clustering)

print("Running DBSCAN clustering...")
dbscan = DBSCAN(eps=0.001, min_samples=5, metric='haversine')
cluster_labels = dbscan.fit_predict(coords_rad)

clustering_sample['violation_cluster'] = cluster_labels
n_clusters = len(set(cluster_labels)) - (1 if -1 in cluster_labels else 0)
n_noise = list(cluster_labels).count(-1)

print(f"Clustering complete:")
print(f"   • {n_clusters} clusters identified")
print(f"   • {n_noise} noise points (outliers)")
print(f"   • {len(clustering_sample) - n_noise} points in clusters")

# analyzing cluster characteristics
cluster_stats = clustering_sample[clustering_sample['violation_cluster'] != -1].gro
    'Violation ID': 'count',
    'Violation Latitude': ['mean', 'std'],
    'Violation Longitude': ['mean', 'std'],
    'Bus Route ID': 'nunique'
```

```
}).round(6)

cluster_stats.columns = ['violation_count', 'lat_center', 'lat_std', 'lon_center',
cluster_stats = cluster_stats.sort_values('violation_count', ascending=False)

print(f"\nTop 5 violation hotspots:")
for i, (cluster_id, stats) in enumerate(cluster_stats.head().iterrows()):
    print(f"   {i+1}. Cluster {cluster_id}: {stats['violation_count']} violations,
    print(f"      Center: ({stats['lat_center']:.4f}, {stats['lon_center']:.4f})")
```

```
Creating violation hotspot clusters...
Valid coordinates: 3,778,568 out of 3,778,568 violations
Sampling 50000 violations for clustering analysis...
Running DBSCAN clustering...
Clustering complete:
   • 1 clusters identified
   • 0 noise points (outliers)
   • 50000 points in clusters

Top 5 violation hotspots:
   1. Cluster 0: 50000.0 violations, 40.0 routes
      Center: (40.7641, -73.9208)
```

In [43]:

```
# creating spatial density features
print("Creating spatial density features...")

# function to calculate violations within radius
def calculate_density_features(df, radius_meters=100):
    """calculating violation density within specified radius"""
    density_features = []

    print(f"   Calculating density within {radius_meters}m radius...")

    # sample for efficiency if dataset is large
    if len(df) > 10000:
        sample_size = min(10000, len(df))
        sample_df = df.sample(n=sample_size, random_state=42)
        print(f"   Using sample of {sample_size} violations for density calculation
    else:
        sample_df = df

    for idx, row in sample_df.iterrows():
        lat, lon = row['Violation Latitude'], row['Violation Longitude']

        # calculating distances to all other violations
        distances = df.apply(
            lambda x: haversine_distance(lat, lon, x['Violation Latitude'], x['Viol
            axis=1
        )

        # counting violations within radius
        nearby_violations = (distances <= radius_meters).sum() - 1  # exclude self

        density_features.append({
            'violation_id': row['Violation ID'],
            f'violations_within_{radius_meters}m': nearby_violations
```

```
        })

    return pd.DataFrame(density_features)

# calculating density for a sample (this is computationally intensive)
print("Computing spatial density (this may take a few minutes)...")
density_sample = valid_coords.sample(n=min(1000, len(valid_coords)), random_state=4
density_features = calculate_density_features(density_sample, radius_meters=100)

print(f"Density features calculated for {len(density_features)} violations")
print(f"   Average violations within 100m: {density_features['violations_within_100
print(f"   Max violations within 100m: {density_features['violations_within_100m'].

# create stop cluster mapping using DBSCAN on stops
if gtfs_stops is not None:
    print("Creating bus stop clusters...")

    # Filter valid stop coordinates
    valid_stops = gtfs_stops[
        (gtfs_stops['stop_lat'].notna()) &
        (gtfs_stops['stop_lon'].notna())
    ].copy()

    if len(valid_stops) > 0:
        stop_coords = valid_stops[['stop_lat', 'stop_lon']].values
        stop_coords_rad = np.radians(stop_coords)

        stop_dbscan = DBSCAN(eps=0.001, min_samples=2, metric='haversine')
        stop_clusters = stop_dbscan.fit_predict(stop_coords_rad)

        valid_stops['stop_cluster_id'] = stop_clusters
        n_stop_clusters = len(set(stop_clusters)) - (1 if -1 in stop_clusters else

        print(f"Created {n_stop_clusters} stop clusters from {len(valid_stops)} sto
    else:
        print("No valid stop coordinates found")
        valid_stops = None
else:
    valid_stops = None

print(f"\nSpatial Features Summary:")
spatial_features = [
    'violation_cluster', 'violations_within_100m', 'stop_cluster_id'
]
print(f"Created spatial intelligence features")
print(f"   • {n_clusters} violation hotspot clusters")
print(f"   • Density analysis for violation prediction")
if valid_stops is not None:
    print(f"   • {n_stop_clusters} stop clusters for route optimization")
```

```
Creating spatial density features...
Computing spatial density (this may take a few minutes)...
   Calculating density within 100m radius...
Density features calculated for 1000 violations
   Average violations within 100m: 1.9
   Max violations within 100m: 14
Creating bus stop clusters...
Created 1 stop clusters from 11698 stops

Spatial Features Summary:
Created spatial intelligence features
   • 1 violation hotspot clusters
   • Density analysis for violation prediction
   • 1 stop clusters for route optimization
```

Section 4: CUNY-Specific Features

creating proximity and routing features for CUNY campuses using the established coordinates and 500m buffer analysis.

In [44]:

```
print("CUNY-SPECIFIC FEATURE ENGINEERING")
print("=" * 50)

# CUNY campus coordinates (from previous analysis)
CUNY_CAMPUSES = {
    'Hunter College': (40.7685, -73.9656),
    'City College': (40.8200, -73.9493),
    'Baruch College': (40.7402, -73.9836),
    'Brooklyn College': (40.6314, -73.9521),
    'Queens College': (40.7366, -73.8170),
    'LaGuardia CC': (40.7443, -73.9349),
    'Bronx CC': (40.8563, -73.9125)
}

print(f"Analyzing proximity to {len(CUNY_CAMPUSES)} CUNY campuses:")
for campus, (lat, lon) in CUNY_CAMPUSES.items():
    print(f"   • {campus}: ({lat}, {lon})")

# calculating distance to nearest CUNY campus for each violation
print("\nCalculating distances to CUNY campuses...")

def calculate_cuny_features(df):
    """calculating CUNY proximity features for violations"""
    cuny_features = []

    for idx, row in df.iterrows():
        if pd.isna(row['Violation Latitude']) or pd.isna(row['Violation Longitude']
            cuny_features.append({
                'violation_id': row['Violation ID'],
                'nearest_cuny_campus': 'Unknown',
                'distance_to_cuny': np.nan,
                'cuny_route_flag': False,
                'within_cuny_500m': False
            })
```

```python
            continue

        violation_lat = row['Violation Latitude']
        violation_lon = row['Violation Longitude']

        # calculating distance to each CUNY campus
        distances = {}
        for campus, (campus_lat, campus_lon) in CUNY_CAMPUSES.items():
            distance = haversine_distance(violation_lat, violation_lon, campus_lat,
            distances[campus] = distance

        # finding nearest campus
        nearest_campus = min(distances, key=distances.get)
        nearest_distance = distances[nearest_campus]

        # determining if within 500m buffer
        within_500m = nearest_distance <= 500

        cuny_features.append({
            'violation_id': row['Violation ID'],
            'nearest_cuny_campus': nearest_campus,
            'distance_to_cuny': nearest_distance,
            'cuny_route_flag': within_500m,
            'within_cuny_500m': within_500m
        })

    return pd.DataFrame(cuny_features)

# calculating for a sample first (computationally intensive)
cuny_sample_size = min(10000, len(valid_coords))
cuny_sample = valid_coords.sample(n=cuny_sample_size, random_state=42)

print(f"Computing CUNY features for {cuny_sample_size} violations...")
cuny_features_df = calculate_cuny_features(cuny_sample)

print(f"CUNY features calculated")

# analyzing CUNY proximity patterns
print("\nCUNY Proximity Analysis:")
cuny_summary = cuny_features_df.groupby('nearest_cuny_campus').agg({
    'violation_id': 'count',
    'distance_to_cuny': 'mean',
    'within_cuny_500m': 'sum'
}).round(1)

cuny_summary.columns = ['total_violations', 'avg_distance_m', 'violations_within_50
cuny_summary = cuny_summary.sort_values('violations_within_500m', ascending=False)

print("Violations by nearest CUNY campus:")
for campus, stats in cuny_summary.iterrows():
    print(f"   • {campus}: {stats['violations_within_500m']} violations within 500m

total_cuny_violations = cuny_features_df['within_cuny_500m'].sum()
cuny_percentage = (total_cuny_violations / len(cuny_features_df)) * 100

print(f"\nCUNY Impact Summary:")
```

```
print(f"   • {total_cuny_violations} violations within 500m of CUNY campuses")
print(f"   • {cuny_percentage:.1f}% of all violations are CUNY-adjacent")
print(f"   • {cuny_summary['violations_within_500m'].sum()} total CUNY-area violati
```

```
CUNY-SPECIFIC FEATURE ENGINEERING
==================================================
Analyzing proximity to 7 CUNY campuses:
   • Hunter College: (40.7685, -73.9656)
   • City College: (40.82, -73.9493)
   • Baruch College: (40.7402, -73.9836)
   • Brooklyn College: (40.6314, -73.9521)
   • Queens College: (40.7366, -73.817)
   • LaGuardia CC: (40.7443, -73.9349)
   • Bronx CC: (40.8563, -73.9125)

Calculating distances to CUNY campuses...
Computing CUNY features for 10000 violations...
CUNY features calculated

CUNY Proximity Analysis:
Violations by nearest CUNY campus:
   • Baruch College: 201.0 violations within 500m (avg dist: 2238m)
   • Brooklyn College: 107.0 violations within 500m (avg dist: 3478m)
   • City College: 44.0 violations within 500m (avg dist: 2102m)
   • Hunter College: 36.0 violations within 500m (avg dist: 1476m)
   • Bronx CC: 0.0 violations within 500m (avg dist: 2955m)
   • LaGuardia CC: 0.0 violations within 500m (avg dist: 4254m)
   • Queens College: 0.0 violations within 500m (avg dist: 3970m)

CUNY Impact Summary:
   • 388 violations within 500m of CUNY campuses
   • 3.9% of all violations are CUNY-adjacent
   • 388 total CUNY-area violations
```

In [45]:

```
# creating route-level CUNY classification
print("Creating route-level CUNY classification...")

# merging CUNY features back to sample violations
cuny_enriched = cuny_sample.merge(
    cuny_features_df[['violation_id', 'nearest_cuny_campus', 'distance_to_cuny', 'c
    left_on='Violation ID',
    right_on='violation_id',
    how='left'
)

# route-level CUNY serving analysis
route_cuny_analysis = cuny_enriched.groupby('Bus Route ID').agg({
    'cuny_route_flag': ['sum', 'count', 'mean'],
    'distance_to_cuny': 'mean',
    'nearest_cuny_campus': lambda x: x.mode().iloc[0] if len(x.mode()) > 0 else 'No
}).round(3)

route_cuny_analysis.columns = ['cuny_violations', 'total_violations', 'cuny_violati

# classifying routes as CUNY-serving if >20% of violations are within 500m of campu
route_cuny_analysis['serves_cuny_primary'] = route_cuny_analysis['cuny_violation_ra
```

```
route_cuny_analysis['cuny_intensity'] = route_cuny_analysis['cuny_violations'] / ro

# sorting by CUNY intensity
route_cuny_analysis = route_cuny_analysis.sort_values('cuny_violations', ascending=

print(f"\nTop CUNY-serving routes:")
cuny_routes = route_cuny_analysis[route_cuny_analysis['serves_cuny_primary']].head(
for route_id, stats in cuny_routes.iterrows():
    print(f"   • Route {route_id}: {stats['cuny_violations']} CUNY violations ({sta

print(f"\nCUNY Route Statistics:")
total_cuny_routes = route_cuny_analysis['serves_cuny_primary'].sum()
total_routes = len(route_cuny_analysis)
print(f"   • {total_cuny_routes} routes serve CUNY campuses (>{20}% violations with
print(f"   • {total_routes - total_cuny_routes} routes do not primarily serve CUNY"
print(f"   • {(total_cuny_routes/total_routes)*100:.1f}% of routes have significant

# creating campus-specific features
print("\nCreating campus-specific features...")

for campus in CUNY_CAMPUSES.keys():
    # distance to specific campus
    campus_clean = campus.replace(' ', '_').replace('College', 'C').lower()
    cuny_enriched[f'distance_to_{campus_clean}'] = cuny_enriched.apply(
        lambda row: haversine_distance(
            row['Violation Latitude'], row['Violation Longitude'],
            CUNY_CAMPUSES[campus][0], CUNY_CAMPUSES[campus][1]
        ) if pd.notna(row['Violation Latitude']) else np.nan, axis=1
    )

    # within 500m of specific campus
    cuny_enriched[f'within_500m_{campus_clean}'] = cuny_enriched[f'distance_to_{cam

print(f"Created campus-specific distance and proximity features")

# creating CUNY temporal interaction features
print("Creating CUNY temporal interaction features...")

# class change periods at CUNY campuses
cuny_enriched['cuny_class_change_violation'] = (
    cuny_enriched['is_cuny_class_change'] &
    cuny_enriched['cuny_route_flag']
)

# peak student travel times (7-9am, 5-7pm) on CUNY routes
cuny_enriched['cuny_peak_travel'] = (
    (cuny_enriched['is_morning_rush'] | cuny_enriched['is_evening_rush']) &
    cuny_enriched['cuny_route_flag']
)

# weekend activity near CUNY (different pattern than weekday)
cuny_enriched['cuny_weekend_activity'] = (
    cuny_enriched['is_weekend'] &
    cuny_enriched['cuny_route_flag'] &
    cuny_enriched['hour_of_day'].between(10, 18)
)
```

```
print(f"CUNY temporal interaction features created")

cuny_interaction_features = [
    'cuny_class_change_violation', 'cuny_peak_travel', 'cuny_weekend_activity'
]

print(f"\nCUNY Features Summary:")
print(f"   • Proximity to {len(CUNY_CAMPUSES)} campuses calculated")
print(f"   • {total_cuny_routes} routes classified as CUNY-serving")
print(f"   • {len(cuny_interaction_features)} temporal interaction features")
print(f"   • Campus-specific distance features for all locations")
```

```
Creating route-level CUNY classification...

Top CUNY-serving routes:
   • Route B44+: 89 CUNY violations (20.6%) - Brooklyn College
   • Route M23+: 24 CUNY violations (48.0%) - Baruch College

CUNY Route Statistics:
   • 2 routes serve CUNY campuses (>20% violations within 500m)
   • 38 routes do not primarily serve CUNY
   • 5.0% of routes have significant CUNY service

Creating campus-specific features...
Created campus-specific distance and proximity features
Creating CUNY temporal interaction features...
CUNY temporal interaction features created

CUNY Features Summary:
   • Proximity to 7 campuses calculated
   • 2 routes classified as CUNY-serving
   • 3 temporal interaction features
   • Campus-specific distance features for all locations
```

Section 5: Enforcement Adaptation Features

building features that capture violator learning and adaptation patterns - critical for understanding the enforcement effectiveness paradox.

In [ ]:

```
print("ENFORCEMENT ADAPTATION FEATURE ENGINEERING")
print("=" * 50)
print("Implementing insights from previous analysis:")
print("   • Violators adapt to predictable enforcement patterns")
print("   • Repeat offenders show learning behaviors")

# working with strategic sample for adaptation analysis (much faster processing)
print("\nAnalyzing adaptation patterns on strategic sample...")

# sample 500K violations maintaining temporal distribution
sample_size = 500000
violations_sorted = violations_data.sample(n=sample_size, random_state=42).sort_val

print(f"Processing {len(violations_sorted):,} violations for adaptation analysis")
```

```
# 1. location-based enforcement history
print("\nCreating location-based enforcement features...")

# grouping by stop_id to track enforcement at specific locations
location_groups = violations_sorted.groupby('Stop ID')

location_features = []
for stop_id, group in location_groups:
    group_sorted = group.sort_values('violation_datetime')

    for idx, (_, row) in enumerate(group_sorted.iterrows()):
        # cumulative violations at this location up to this point
        cumulative_violations = idx + 1

        # days since first violation at this location
        if idx == 0:
            days_since_first = 0
        else:
            days_since_first = (row['violation_datetime'] - group_sorted.iloc[0]['v

        # recent violation density (last 7 days)
        recent_cutoff = row['violation_datetime'] - timedelta(days=7)
        recent_violations = group_sorted[
            (group_sorted['violation_datetime'] < row['violation_datetime']) &
            (group_sorted['violation_datetime'] >= recent_cutoff)
        ]
        recent_violation_count = len(recent_violations)

        location_features.append({
            'violation_id': row['Violation ID'],
            'cumulative_violations_at_location': cumulative_violations,
            'days_since_first_violation': days_since_first,
            'recent_violations_7d': recent_violation_count
        })

location_adaptation_df = pd.DataFrame(location_features)
print(f"Location-based features created for {len(location_adaptation_df):,} violati

# 2. vehicle-based repeat offender analysis
print("\nAnalyzing repeat offender patterns...")

vehicle_groups = violations_sorted.groupby('Vehicle ID')

vehicle_features = []
for vehicle_id, group in vehicle_groups:
    group_sorted = group.sort_values('violation_datetime')

    for idx, (_, row) in enumerate(group_sorted.iterrows()):
        # vehicle violation sequence number
        violation_sequence = idx + 1

        # time since last violation by this vehicle
        if idx == 0:
            days_since_last_violation = np.nan
            avg_violation_interval = np.nan
```

```
        else:
            days_since_last = (row['violation_datetime'] - group_sorted.iloc[idx-1]
            days_since_last_violation = days_since_last

            # Average interval between violations for this vehicle
            if idx > 1:
                all_intervals = [(group_sorted.iloc[i]['violation_datetime'] - grou
                               for i in range(1, idx+1)]
                avg_violation_interval = np.mean(all_intervals)
            else:
                avg_violation_interval = days_since_last

        # route switching behavior (adaptation indicator)
        if idx == 0:
            route_switches = 0
        else:
            previous_routes = group_sorted.iloc[:idx]['Bus Route ID'].tolist()
            route_switches = len(set(previous_routes + [row['Bus Route ID']])) - 1

        vehicle_features.append({
            'violation_id': row['Violation ID'],
            'vehicle_violation_sequence': violation_sequence,
            'days_since_last_violation': days_since_last_violation,
            'avg_violation_interval': avg_violation_interval,
            'vehicle_route_switches': route_switches,
            'is_repeat_offender': violation_sequence > 1
        })

vehicle_adaptation_df = pd.DataFrame(vehicle_features)
print(f"Vehicle adaptation features created for {len(vehicle_adaptation_df):,} viol

# analyzing repeat offender patterns
repeat_stats = vehicle_adaptation_df.groupby('is_repeat_offender').agg({
    'violation_id': 'count',
    'vehicle_violation_sequence': 'mean',
    'days_since_last_violation': 'mean',
    'vehicle_route_switches': 'mean'
}).round(2)

print(f"\nRepeat Offender Analysis:")
print(repeat_stats)

total_violations = len(vehicle_adaptation_df)
repeat_violations = vehicle_adaptation_df['is_repeat_offender'].sum()
repeat_percentage = (repeat_violations / total_violations) * 100

print(f"\nRepeat Offender Summary:")
print(f"   • {repeat_violations:,} repeat violations ({repeat_percentage:.1f}% of t
print(f"   • {total_violations - repeat_violations:,} first-time violations")

# identifying super repeat offenders (10+ violations)
super_repeaters = vehicle_adaptation_df[vehicle_adaptation_df['vehicle_violation_se
print(f"   • {len(super_repeaters):,} violations from super repeat offenders (10+ v
```

```
ENFORCEMENT ADAPTATION FEATURE ENGINEERING
==================================================
Implementing insights from previous analysis:
  • -0.169 correlation between enforcement duration and effectiveness
  • Violators adapt to predictable enforcement patterns
  • Repeat offenders show learning behaviors

Analyzing adaptation patterns on strategic sample...
Processing 500,000 violations for adaptation analysis

Creating location-based enforcement features...
Location-based features created for 500,000 violations

Analyzing repeat offender patterns...
Vehicle adaptation features created for 491,270 violations

Repeat Offender Analysis:
                    violation_id  vehicle_violation_sequence  \
is_repeat_offender
False                     317160                        1.00
True                      174110                        6.64

                    days_since_last_violation  vehicle_route_switches
is_repeat_offender
False                                     NaN                    0.00
True                                     87.8                    0.88

Repeat Offender Summary:
  • 174,110 repeat violations (35.4% of total)
  • 317,160 first-time violations
  • 26,634 violations from super repeat offenders (10+ violations)
```

In [ ]:

```python
# creating enforcement predictability analysis
print("\nCreating enforcement predictability features...")

# calculating enforcement patterns by location and time
location_time_patterns = violations_sorted.groupby(['Stop ID', 'hour_of_day']).size

# calculating entropy of enforcement times at each location (higher = more unpredic
location_entropy = []
for stop_id in violations_sorted['Stop ID'].unique():
    stop_violations = violations_sorted[violations_sorted['Stop ID'] == stop_id]
    hour_counts = stop_violations['hour_of_day'].value_counts()

    if len(hour_counts) > 1:
        # calculating entropy
        probabilities = hour_counts / hour_counts.sum()
        entropy = -np.sum(probabilities * np.log2(probabilities + 1e-10))
    else:
        entropy = 0  # completely predictable

    location_entropy.append({
        'stop_id': stop_id,
        'enforcement_predictability': 1 / (entropy + 1),  # Higher = more predictab
        'enforcement_entropy': entropy
    })
```

```
entropy_df = pd.DataFrame(location_entropy)

print(f"Predictability analysis completed for {len(entropy_df)} unique locations")
print(f"   Average enforcement entropy: {entropy_df['enforcement_entropy'].mean():.
print(f"   Most predictable locations (entropy < 1.0): {(entropy_df['enforcement_en
print(f"   Most unpredictable locations (entropy > 3.0): {(entropy_df['enforcement_

# creating repeat offender concentration features
print("\nCreating repeat offender concentration features...")

# calculating what percentage of violations at each location come from repeat offen
location_repeat_concentration = []

for stop_id in violations_sorted['Stop ID'].unique():
    stop_violations = violations_sorted[violations_sorted['Stop ID'] == stop_id]

    # counting unique vehicles and their violation counts
    vehicle_counts = stop_violations['Vehicle ID'].value_counts()

    # calculating concentration metrics
    total_violations = len(stop_violations)
    unique_vehicles = len(vehicle_counts)
    repeat_offender_violations = (vehicle_counts > 1).sum()
    violations_from_repeaters = vehicle_counts[vehicle_counts > 1].sum()

    if total_violations > 0:
        repeat_concentration = violations_from_repeaters / total_violations
        vehicle_diversity = unique_vehicles / total_violations  # Lower = more conc
    else:
        repeat_concentration = 0
        vehicle_diversity = 0

    location_repeat_concentration.append({
        'stop_id': stop_id,
        'repeat_offender_concentration': repeat_concentration,
        'vehicle_diversity_index': vehicle_diversity,
        'unique_violators': unique_vehicles,
        'total_violations_at_stop': total_violations
    })

concentration_df = pd.DataFrame(location_repeat_concentration)

print(f"Concentration analysis completed")
print(f"   Average repeat offender concentration: {concentration_df['repeat_offende
print(f"   Locations with >50% repeat violations: {(concentration_df['repeat_offend
print(f"   Average vehicle diversity index: {concentration_df['vehicle_diversity_in

# detecting violation learning curves
print("\nDetecting violation learning curves...")

# for routes with enough data, detecting if violation rates decrease over time (lea
route_learning_curves = []

for route_id in violations_sorted['Bus Route ID'].unique():
    route_violations = violations_sorted[violations_sorted['Bus Route ID'] == route
```

```
    if len(route_violations) >= 30:  # need enough data for trend analysis
        # grouping by month and counting violations
        monthly_violations = route_violations.groupby(route_violations['violation_d

        if len(monthly_violations) >= 3:  # need at least 3 months
            # calculating trend (negative slope = decreasing violations = learning)
            x = np.arange(len(monthly_violations))
            y = monthly_violations.values
            slope, intercept, r_value, p_value, std_err = stats.linregress(x, y)

            route_learning_curves.append({
                'route_id': route_id,
                'violation_trend_slope': slope,
                'trend_r_squared': r_value**2,
                'trend_p_value': p_value,
                'shows_learning_curve': (slope < -0.5) and (p_value < 0.1),  # decl
                'months_of_data': len(monthly_violations),
                'total_violations': len(route_violations)
            })

learning_df = pd.DataFrame(route_learning_curves)

if len(learning_df) > 0:
    print(f"Learning curve analysis completed for {len(learning_df)} routes with su
    learning_routes = learning_df['shows_learning_curve'].sum()
    print(f"   Routes showing learning curves: {learning_routes} ({(learning_routes
    print(f"   Average trend slope: {learning_df['violation_trend_slope'].mean():.3
else:
    print("Insufficient data for learning curve analysis")

print(f"\nEnforcement Adaptation Features Summary:")
adaptation_features = [
    'cumulative_violations_at_location', 'days_since_first_violation',
    'vehicle_violation_sequence', 'is_repeat_offender', 'vehicle_route_switches',
    'enforcement_predictability', 'repeat_offender_concentration',
    'vehicle_diversity_index', 'violation_trend_slope'
]
print(f"Created {len(adaptation_features)} adaptation features")
print(f"   • Location learning: cumulative violations, time since first")
print(f"   • Vehicle adaptation: repeat patterns, route switching")
print(f"   • Predictability: enforcement entropy, concentration")
print(f"   • Learning curves: trend analysis for {len(learning_df) if len(learning_
```

```
Creating enforcement predictability features...
Predictability analysis completed for 2594 unique locations
   Average enforcement entropy: 3.310
   Most predictable locations (entropy < 1.0): 172
   Most unpredictable locations (entropy > 3.0): 1887

Creating repeat offender concentration features...
```

Section 6: Target Variable Engineering

creating multiple prediction targets for different use cases:

- immediate prediction (next hour)
- tactical planning (next day)
- strategic forecasting (binary classification)
- speed impact correlation

In [ ]:
```
print("TARGET VARIABLE ENGINEERING")
print("=" * 50)
print("Creating multiple prediction targets for different operational scenarios")

# group violations by location and time for target creation
print("\nAggregating violations by location and time...")

# create unique location-time combinations
modeling_groups = violations_sorted.groupby(['Stop ID', 'violation_hour']).agg({
    'Violation ID': 'count',
    'Vehicle ID': 'nunique',
    'Bus Route ID': 'first',  # primary route for this stop
    'Violation Latitude': 'first',
    'Violation Longitude': 'first',
    'Stop Name': 'first',
    'Bus Stop Latitude': 'first',
    'Bus Stop Longitude': 'first',
    'hour_of_day': 'first',
    'day_of_week': 'first',
    'month': 'first',
    'is_weekend': 'first',
    'rush_hour_period': 'first',
    'is_school_hours': 'first',
    'semester_period': 'first'
}).reset_index()

# renaming violation count column
modeling_groups = modeling_groups.rename(columns={'Violation ID': 'violation_count'

print(f"Created {len(modeling_groups):,} unique location-hour combinations")
print(f"   Date range: {modeling_groups['violation_hour'].min()} to {modeling_group
print(f"   Average violations per location-hour: {modeling_groups['violation_count'
print(f"   Max violations in one location-hour: {modeling_groups['violation_count']

# sorting by time for target engineering
modeling_groups = modeling_groups.sort_values(['Stop ID', 'violation_hour'])

print("\nCreating target variables...")

# 1. next hour prediction target
print("Target 1: violation_count_next_hour (immediate prediction)")

next_hour_targets = []
for stop_id in modeling_groups['Stop ID'].unique():
    stop_data = modeling_groups[modeling_groups['Stop ID'] == stop_id].copy()
    stop_data = stop_data.sort_values('violation_hour')

    # creating next hour targets
    stop_data['violation_count_next_hour'] = stop_data['violation_count'].shift(-1)
```

```
    # removing last row (no future data)
    stop_data = stop_data[:-1] if len(stop_data) > 1 else stop_data

    next_hour_targets.append(stop_data)

if next_hour_targets:
    next_hour_df = pd.concat(next_hour_targets, ignore_index=True)
    print(f"Next hour targets created for {len(next_hour_df):,} observations")
    print(f"   Non-null targets: {next_hour_df['violation_count_next_hour'].notna()
else:
    next_hour_df = modeling_groups.copy()
    next_hour_df['violation_count_next_hour'] = np.nan

# 2. next day prediction target
print("Target 2: violation_count_next_day (tactical planning)")

# Aggregate to daily level first
daily_groups = violations_sorted.groupby(['Stop ID', violations_sorted['violation_d
    'Violation ID': 'count',
    'Vehicle ID': 'nunique'
}).reset_index()

daily_groups = daily_groups.rename(columns={
    'violation_datetime': 'violation_date',
    'Violation ID': 'daily_violation_count',
    'Vehicle ID': 'daily_unique_vehicles'
})

# creating next day targets
next_day_targets = []
for stop_id in daily_groups['Stop ID'].unique():
    stop_daily = daily_groups[daily_groups['Stop ID'] == stop_id].copy()
    stop_daily = stop_daily.sort_values('violation_date')

    stop_daily['violation_count_next_day'] = stop_daily['daily_violation_count'].sh
    stop_daily['unique_vehicles_next_day'] = stop_daily['daily_unique_vehicles'].sh

    # removing last row
    stop_daily = stop_daily[:-1] if len(stop_daily) > 1 else stop_daily

    next_day_targets.append(stop_daily)

if next_day_targets:
    next_day_df = pd.concat(next_day_targets, ignore_index=True)
    print(f"Next day targets created for {len(next_day_df):,} daily observations")
    print(f"   Average daily violations: {next_day_df['daily_violation_count'].mean
else:
    next_day_df = pd.DataFrame()

# creating binary high violation flag
print("Target 3: high_violation_flag (binary classification)")

# defining threshold for "high violation" periods
violation_threshold = modeling_groups['violation_count'].quantile(0.8)  # top 20%
print(f"   High violation threshold: >{violation_threshold:.0f} violations per hour
```

```
modeling_groups['high_violation_flag'] = (modeling_groups['violation_count'] > viol

high_violation_count = modeling_groups['high_violation_flag'].sum()
high_violation_pct = (high_violation_count / len(modeling_groups)) * 100

print(f"High violation flags created: {high_violation_count:,} high periods ({high_

# creating multiple threshold targets for different severity levels
print("Target 4: Multiple severity thresholds")

thresholds = {
    'moderate_violation_flag': modeling_groups['violation_count'].quantile(0.6),  #
    'severe_violation_flag': modeling_groups['violation_count'].quantile(0.9),   #
    'extreme_violation_flag': modeling_groups['violation_count'].quantile(0.95)  #
}

for flag_name, threshold in thresholds.items():
    modeling_groups[flag_name] = (modeling_groups['violation_count'] > threshold).a
    count = modeling_groups[flag_name].sum()
    pct = (count / len(modeling_groups)) * 100
    print(f"   {flag_name}: {count:,} periods ({pct:.1f}%) above {threshold:.1f} vi

print(f"Multiple severity targets created")
```

```
TARGET VARIABLE ENGINEERING
==================================================
Creating multiple prediction targets for different operational scenarios

Aggregating violations by location and time...
Created 453,935 unique location-hour combinations
   Date range: 2019-10-07 07:00:00 to 2025-08-21 17:00:00
   Average violations per location-hour: 1.10
   Max violations in one location-hour: 9

Creating target variables...
Target 1: violation_count_next_hour (immediate prediction)
Next hour targets created for 451,416 observations
   Non-null targets: 451,341
Target 2: violation_count_next_day (tactical planning)
Next day targets created for 264,513 daily observations
   Average daily violations: 1.88
Target 3: high_violation_flag (binary classification)
   High violation threshold: >1 violations per hour
High violation flags created: 39,320 high periods (8.7%)
Target 4: Multiple severity thresholds
   moderate_violation_flag: 39,320 periods (8.7%) above 1.0 violations
   severe_violation_flag: 39,320 periods (8.7%) above 1.0 violations
   extreme_violation_flag: 5,319 periods (1.2%) above 2.0 violations
Multiple severity targets created
```

In [ ]:

```
# creating speed impact score target
print("Target 5: speed_impact_score (enforcement effectiveness)")

# loading speed data to correlate with violations
print("   loading speed datasets for correlation analysis...")
```

```python
speed_files = {
    '../data/MTA_Bus_Speeds__2015-2019_20250919.csv': 'historical_early',
    '../data/MTA_Bus_Speeds__2020_-_2024_20250919.csv': 'historical_recent',
    '../data/MTA_Bus_Speeds__Beginning_2025_20250919.csv': 'current'
}

all_speeds = []
for file_path, period in speed_files.items():
    if Path(file_path).exists():
        print(f"   Loading {period} speeds...")

        # loading sample for analysis (full files are very large)
        speed_df = pd.read_csv(file_path, nrows=50000)  # sample for efficiency
        speed_df['period'] = period
        speed_df['month_date'] = pd.to_datetime(speed_df['month'], errors='coerce')

        all_speeds.append(speed_df)

if all_speeds:
    combined_speeds = pd.concat(all_speeds, ignore_index=True)
    print(f"Loaded {len(combined_speeds):,} speed records for correlation")

    # calculating route-level speed changes (ACE implementation impact)
    ace_cutoff = datetime(2024, 6, 1)
    combined_speeds['is_post_ace'] = combined_speeds['month_date'] >= ace_cutoff

    # grouping by route and calculating pre/post ACE speed changes
    route_speed_changes = combined_speeds.groupby(['route_id', 'is_post_ace'])['ave

    if True in route_speed_changes.columns and False in route_speed_changes.columns
        route_speed_changes['speed_change_pct'] = (
            (route_speed_changes[True] - route_speed_changes[False]) /
            route_speed_changes[False] * 100
        )
        route_speed_changes['speed_improved'] = route_speed_changes['speed_change_p

        print(f"   Speed changes calculated for {len(route_speed_changes)} routes")
        print(f"   Routes with speed improvement: {route_speed_changes['speed_impro
        print(f"   Average speed change: {route_speed_changes['speed_change_pct'].m

        # creating speed impact score for routes with both violation and speed data
        route_violations = modeling_groups.groupby('Bus Route ID').agg({
            'violation_count': 'sum',
            'Vehicle ID': 'sum'  # total unique vehicles
        }).reset_index()

        # merging violations with speed changes
        speed_violation_correlation = route_violations.merge(
            route_speed_changes[['speed_change_pct', 'speed_improved']].reset_index
            left_on='Bus Route ID',
            right_on='route_id',
            how='inner'
        )

        if len(speed_violation_correlation) > 0:
```

```
            # calculating speed impact score: violations weighted by speed change
            speed_violation_correlation['speed_impact_score'] = (
                speed_violation_correlation['violation_count'] *
                speed_violation_correlation['speed_change_pct']
            )

            # normalizing to 0-1 scale
            min_score = speed_violation_correlation['speed_impact_score'].min()
            max_score = speed_violation_correlation['speed_impact_score'].max()

            if max_score != min_score:
                speed_violation_correlation['speed_impact_normalized'] = (
                    (speed_violation_correlation['speed_impact_score'] - min_score)
                    (max_score - min_score)
                )
            else:
                speed_violation_correlation['speed_impact_normalized'] = 0.5

            print(f"Speed impact scores calculated for {len(speed_violation_correla
            print(f"   Best performing route: {speed_violation_correlation.loc[spee
            print(f"   Worst performing route: {speed_violation_correlation.loc[spe
        else:
            print("No matching routes between violations and speed data")
            speed_violation_correlation = pd.DataFrame()
    else:
        print("Insufficient speed data for pre/post ACE comparison")
        speed_violation_correlation = pd.DataFrame()
else:
    print("No speed data available for correlation analysis")
    speed_violation_correlation = pd.DataFrame()

# creating composite prediction targets
print("\nTarget 6: Composite prediction targets")

# risk score: combination of violation count and severity flags
modeling_groups['violation_risk_score'] = (
    modeling_groups['violation_count'] * 0.4 +
    modeling_groups['high_violation_flag'] * 3 +
    modeling_groups['severe_violation_flag'] * 5 +
    modeling_groups['extreme_violation_flag'] * 8
)

# deployment priority score (higher = more urgent)
modeling_groups['deployment_priority'] = (
    modeling_groups['violation_count'] * 0.3 +
    modeling_groups['Vehicle ID'] * 0.2 +  # unique vehicles
    modeling_groups['high_violation_flag'] * 2
)

print(f"Composite targets created:")
print(f"   Violation risk score range: {modeling_groups['violation_risk_score'].min
print(f"   Deployment priority range: {modeling_groups['deployment_priority'].min()

print(f"\nTARGET VARIABLES SUMMARY:")
target_variables = [
    'violation_count_next_hour',
```

```
    'violation_count_next_day',
    'high_violation_flag',
    'moderate_violation_flag',
    'severe_violation_flag',
    'extreme_violation_flag',
    'speed_impact_score',
    'violation_risk_score',
    'deployment_priority'
]

print(f"Created {len(target_variables)} target variables for different prediction s
print(f"   • Immediate forecasting: violation_count_next_hour")
print(f"   • Tactical planning: violation_count_next_day")
print(f"   • Binary classification: high/moderate/severe/extreme violation flags")
print(f"   • Effectiveness measurement: speed_impact_score")
print(f"   • Operational optimization: violation_risk_score, deployment_priority")
```

```
Target 5: speed_impact_score (enforcement effectiveness)
   loading speed datasets for correlation analysis...
   Loading historical_early speeds...
   Loading historical_recent speeds...
   Loading current speeds...
Loaded 108,657 speed records for correlation
   Speed changes calculated for 524 routes
   Routes with speed improvement: 70
   Average speed change: -2.57%
Speed impact scores calculated for 38 routes
   Best performing route: BX19
   Worst performing route: Q44+

Target 6: Composite prediction targets
Composite targets created:
   Violation risk score range: 0.4 - 19.6
   Deployment priority range: 0.3 - 6.0

TARGET VARIABLES SUMMARY:
Created 9 target variables for different prediction scenarios:
   • Immediate forecasting: violation_count_next_hour
   • Tactical planning: violation_count_next_day
   • Binary classification: high/moderate/severe/extreme violation flags
   • Effectiveness measurement: speed_impact_score
   • Operational optimization: violation_risk_score, deployment_priority
```

Section 7: Final Dataset Assembly and Export

combining all engineered features into the final modeling dataset and exporting in optimized formats.

In [ ]:

```
print("FINAL DATASET ASSEMBLY")
print("=" * 50)

# start with the main modeling groups dataset
final_dataset = modeling_groups.copy()
```

```
print(f"Base dataset: {len(final_dataset):,} location-hour observations")

# merge adaptation features
print("Merging enforcement adaptation features...")
if len(location_adaptation_df) > 0:
    # aggregate location features to hourly level
    location_hourly = violations_sorted.merge(
        location_adaptation_df,
        left_on='Violation ID',
        right_on='violation_id',
        how='left'
    ).groupby(['Stop ID', 'violation_hour']).agg({
        'cumulative_violations_at_location': 'mean',
        'days_since_first_violation': 'mean',
        'recent_violations_7d': 'mean'
    }).reset_index()

    final_dataset = final_dataset.merge(
        location_hourly,
        on=['Stop ID', 'violation_hour'],
        how='left'
    )
    print(f"Location adaptation features merged")

if len(vehicle_adaptation_df) > 0:
    # aggregating vehicle features to hourly level
    vehicle_hourly = violations_sorted.merge(
        vehicle_adaptation_df,
        left_on='Violation ID',
        right_on='violation_id',
        how='left'
    ).groupby(['Stop ID', 'violation_hour']).agg({
        'vehicle_violation_sequence': 'mean',
        'is_repeat_offender': 'mean',  # percentage of repeat offenders
        'vehicle_route_switches': 'mean'
    }).reset_index()

    final_dataset = final_dataset.merge(
        vehicle_hourly,
        on=['Stop ID', 'violation_hour'],
        how='left'
    )
    print(f"Vehicle adaptation features merged")

# merging predictability features
if len(entropy_df) > 0:
    final_dataset = final_dataset.merge(
        entropy_df,
        left_on='Stop ID',
        right_on='stop_id',
        how='left'
    ).drop('stop_id', axis=1)
    print(f"Predictability features merged")

if len(concentration_df) > 0:
    final_dataset = final_dataset.merge(
```

```python
        concentration_df,
        left_on='Stop ID',
        right_on='stop_id',
        how='left'
    ).drop('stop_id', axis=1)
    print(f"Concentration features merged")

# merging CUNY proximity features
print("Merging CUNY proximity features...")
if len(cuny_features_df) > 0:
    # calculating CUNY features for all stops based on coordinates
    cuny_stop_features = []

    for _, row in final_dataset.iterrows():
        if pd.notna(row['Violation Latitude']) and pd.notna(row['Violation Longitud
            distances = {}
            for campus, (campus_lat, campus_lon) in CUNY_CAMPUSES.items():
                distance = haversine_distance(
                    row['Violation Latitude'], row['Violation Longitude'],
                    campus_lat, campus_lon
                )
                distances[campus] = distance

            nearest_campus = min(distances, key=distances.get)
            nearest_distance = distances[nearest_campus]

            cuny_stop_features.append({
                'stop_id': row['Stop ID'],
                'nearest_cuny_campus': nearest_campus,
                'distance_to_cuny': nearest_distance,
                'cuny_route_flag': nearest_distance <= 500
            })
        else:
            cuny_stop_features.append({
                'stop_id': row['Stop ID'],
                'nearest_cuny_campus': 'Unknown',
                'distance_to_cuny': np.nan,
                'cuny_route_flag': False
            })

    cuny_stop_df = pd.DataFrame(cuny_stop_features).drop_duplicates('stop_id')

    final_dataset = final_dataset.merge(
        cuny_stop_df,
        left_on='Stop ID',
        right_on='stop_id',
        how='left'
    ).drop('stop_id', axis=1)

    print(f"CUNY features added for all stops")

# adding speed correlation features for matching routes
if len(speed_violation_correlation) > 0:
    final_dataset = final_dataset.merge(
        speed_violation_correlation[['Bus Route ID', 'speed_change_pct', 'speed_imp
        on='Bus Route ID',
```

```
        how='left'
    )
    print(f"Speed impact features merged for routes with speed data")

print(f"\nFinal dataset shape: {final_dataset.shape}")
print(f"   Observations: {len(final_dataset):,}")
print(f"   Features: {len(final_dataset.columns)}")
```

```
FINAL DATASET ASSEMBLY
==================================================
Base dataset: 453,935 location-hour observations
Merging enforcement adaptation features...
Location adaptation features merged
Vehicle adaptation features merged
Predictability features merged
Concentration features merged
Merging CUNY proximity features...
CUNY features added for all stops
Speed impact features merged for routes with speed data

Final dataset shape: (453935, 41)
   Observations: 453,935
   Features: 41
```

In [ ]:

```
# performing final data quality checks
print("FINAL DATA QUALITY CHECKS")
print("=" * 50)

# checking for missing values
missing_summary = final_dataset.isnull().sum()
missing_pct = (missing_summary / len(final_dataset)) * 100

print("Missing values summary:")
high_missing = missing_pct[missing_pct > 10]
if len(high_missing) > 0:
    print("Columns with >10% missing:")
    for col, pct in high_missing.items():
        print(f"   {col}: {pct:.1f}%")
else:
    print("No columns with excessive missing values")

# removing or imputing highly missing columns
columns_to_drop = missing_pct[missing_pct > 50].index.tolist()
if columns_to_drop:
    print(f"\nDropping columns with >50% missing: {columns_to_drop}")
    final_dataset = final_dataset.drop(columns=columns_to_drop)

# filling remaining missing values
print("\nImputing remaining missing values...")

# numeric columns: fill with median
numeric_columns = final_dataset.select_dtypes(include=[np.number]).columns
for col in numeric_columns:
    if final_dataset[col].isnull().any():
        median_val = final_dataset[col].median()
        final_dataset[col] = final_dataset[col].fillna(median_val)
```

```
# categorical columns: fill with mode or 'Unknown'
categorical_columns = final_dataset.select_dtypes(include=['object', 'category']).c
for col in categorical_columns:
    if final_dataset[col].isnull().any():
        if col in ['nearest_cuny_campus', 'rush_hour_period', 'semester_period']:
            final_dataset[col] = final_dataset[col].fillna('Unknown')
        else:
            mode_val = final_dataset[col].mode()
            if len(mode_val) > 0:
                final_dataset[col] = final_dataset[col].fillna(mode_val.iloc[0])
            else:
                final_dataset[col] = final_dataset[col].fillna('Unknown')

# boolean columns: fill with False
boolean_columns = final_dataset.select_dtypes(include=['bool']).columns
for col in boolean_columns:
    if final_dataset[col].isnull().any():
        final_dataset[col] = final_dataset[col].fillna(False)

print(f"Missing value imputation complete")

# removing duplicate rows
initial_len = len(final_dataset)
final_dataset = final_dataset.drop_duplicates(subset=['Stop ID', 'violation_hour'])
duplicates_removed = initial_len - len(final_dataset)

if duplicates_removed > 0:
    print(f"Removed {duplicates_removed} duplicate observations")
else:
    print(f"No duplicate observations found")

# feature engineering summary
print(f"\nFEATURE ENGINEERING SUMMARY")
print(f"=" * 50)

feature_categories = {
    'Temporal': ['hour_of_day', 'day_of_week', 'month', 'is_weekend', 'rush_hour_pe
    'Spatial': ['Violation Latitude', 'Violation Longitude', 'Stop Name'],
    'CUNY': ['nearest_cuny_campus', 'distance_to_cuny', 'cuny_route_flag'],
    'Adaptation': ['cumulative_violations_at_location', 'is_repeat_offender', 'enfo
    'Targets': ['violation_count', 'high_violation_flag', 'violation_risk_score', '
}

for category, features in feature_categories.items():
    available_features = [f for f in features if f in final_dataset.columns]
    print(f"   {category}: {len(available_features)} features")

print(f"\nFINAL DATASET READY:")
print(f"   Shape: {final_dataset.shape}")
print(f"   Time range: {final_dataset['violation_hour'].min()} to {final_dataset['v
print(f"   Unique stops: {final_dataset['Stop ID'].nunique():,}")
print(f"   Unique routes: {final_dataset['Bus Route ID'].nunique()}")
print(f"   Memory usage: {final_dataset.memory_usage(deep=True).sum() / 1024 / 1024
```

```
FINAL DATA QUALITY CHECKS
==================================================
Missing values summary:
No columns with excessive missing values

Imputing remaining missing values...
Missing value imputation complete
No duplicate observations found

FEATURE ENGINEERING SUMMARY
==================================================
   Temporal: 7 features
   Spatial: 3 features
   CUNY: 3 features
   Adaptation: 3 features
   Targets: 4 features

FINAL DATASET READY:
   Shape: (453935, 41)
   Time range: 2019-10-07 07:00:00 to 2025-08-21 17:00:00
   Unique stops: 2,594
   Unique routes: 40
   Memory usage: 246.1 MB
```

In [ ]:

```python
# exporting final modeling dataset
print("EXPORTING FINAL MODELING DATASET")
print("=" * 50)

# ensuring output directory exists
output_dir = Path("../data/processed")
output_dir.mkdir(exist_ok=True)

# 1. export as optimized Parquet (recommended for production)
parquet_file = output_dir / "modeling_dataset.parquet"
print(f"Exporting to Parquet: {parquet_file}")

try:
    final_dataset.to_parquet(parquet_file, index=False, compression='snappy')
    file_size_mb = parquet_file.stat().st_size / 1024 / 1024
    print(f"Parquet export complete: {file_size_mb:.1f} MB")
except Exception as e:
    print(f"Parquet export failed: {e}")
    print("   Trying alternative compression...")
    try:
        final_dataset.to_parquet(parquet_file, index=False)
        print(f"Parquet export complete (uncompressed)")
    except Exception as e2:
        print(f"Parquet export failed: {e2}")

# exporting sample as CSV for validation
csv_sample_file = output_dir / "modeling_dataset_sample.csv"
print(f"Exporting sample to CSV: {csv_sample_file}")

sample_size = min(10000, len(final_dataset))
sample_dataset = final_dataset.sample(n=sample_size, random_state=42)
sample_dataset.to_csv(csv_sample_file, index=False)
```

```
file_size_mb = csv_sample_file.stat().st_size / 1024 / 1024
print(f"CSV sample export complete: {sample_size:,} rows, {file_size_mb:.1f} MB")

# exporting metadata and feature descriptions
metadata_file = output_dir / "dataset_metadata.json"
print(f"Exporting metadata: {metadata_file}")

metadata = {
    'dataset_info': {
        'creation_date': datetime.now().isoformat(),
        'total_observations': len(final_dataset),
        'total_features': len(final_dataset.columns),
        'date_range': {
            'start': final_dataset['violation_hour'].min().isoformat(),
            'end': final_dataset['violation_hour'].max().isoformat()
        },
        'spatial_coverage': {
            'unique_stops': final_dataset['Stop ID'].nunique(),
            'unique_routes': final_dataset['Bus Route ID'].nunique()
        }
    },
    'feature_categories': {
        'temporal_features': [col for col in final_dataset.columns if any(keyword i
        'spatial_features': [col for col in final_dataset.columns if any(keyword in
        'cuny_features': [col for col in final_dataset.columns if 'cuny' in col.low
        'adaptation_features': [col for col in final_dataset.columns if any(keyword
        'target_variables': [col for col in final_dataset.columns if any(keyword in
    },
    'data_quality': {
        'missing_values_summary': final_dataset.isnull().sum().to_dict(),
        'duplicate_rows_removed': duplicates_removed,
        'memory_usage_mb': round(final_dataset.memory_usage(deep=True).sum() / 1024
    },
    'target_variable_distributions': {
        'violation_count_stats': {
            'mean': float(final_dataset['violation_count'].mean()),
            'median': float(final_dataset['violation_count'].median()),
            'max': int(final_dataset['violation_count'].max()),
            'std': float(final_dataset['violation_count'].std())
        }
    }
}

# adding binary target distributions if they exist
binary_targets = ['high_violation_flag', 'moderate_violation_flag', 'severe_violati
for target in binary_targets:
    if target in final_dataset.columns:
        metadata['target_variable_distributions'][f'{target}_distribution'] = {
            'positive_cases': int(final_dataset[target].sum()),
            'negative_cases': int((final_dataset[target] == 0).sum()),
            'positive_rate': float(final_dataset[target].mean())
        }

with open(metadata_file, 'w') as f:
    json.dump(metadata, f, indent=2, default=str)
```

```
print(f"Metadata export complete")

# exporting feature importance for model preparation
features_file = output_dir / "feature_list.txt"
print(f"Exporting feature list: {features_file}")

# excluding ID and target columns for modeling
modeling_features = [col for col in final_dataset.columns if col not in [
    'Stop ID', 'violation_hour', 'Violation ID', 'Vehicle ID',
    'violation_count_next_hour', 'violation_count_next_day',
    'high_violation_flag', 'moderate_violation_flag', 'severe_violation_flag', 'ext
    'violation_risk_score', 'deployment_priority'
]]

with open(features_file, 'w') as f:
    f.write("# ACE Intelligence System - Modeling Features\n")
    f.write(f"# Generated: {datetime.now().isoformat()}\n")
    f.write(f"# Total features: {len(modeling_features)}\n\n")

    for category, features in feature_categories.items():
        available_features = [f for f in features if f in modeling_features]
        if available_features:
            f.write(f"\n# {category} Features ({len(available_features)})\n")
            for feature in available_features:
                f.write(f"{feature}\n")

print(f"Feature list export complete: {len(modeling_features)} modeling features id

print(f"\nFEATURE ENGINEERING COMPLETE!")
print(f"=" * 50)
print(f"Final dataset: {final_dataset.shape[0]:,} observations × {final_dataset.sha
print(f"Exported to: {output_dir}")
print(f"   • modeling_dataset.parquet (production-ready)")
print(f"   • modeling_dataset_sample.csv (validation)")
print(f"   • dataset_metadata.json (comprehensive info)")
print(f"   • feature_list.txt (model preparation)")

print(f"\nREADY FOR PREDICTIVE MODELING!")
print(f"   Next step: Build machine learning models for violation hotspot predictio
print(f"   Prediction targets available: immediate, tactical, binary classification
print(f"   Features ready: temporal, spatial, CUNY, adaptation, enforcement intelli
```

```
EXPORTING FINAL MODELING DATASET
==================================================
Exporting to Parquet: ..\data\processed\modeling_dataset.parquet
Parquet export complete: 6.8 MB
Exporting sample to CSV: ..\data\processed\modeling_dataset_sample.csv
CSV sample export complete: 10,000 rows, 3.3 MB
Exporting metadata: ..\data\processed\dataset_metadata.json
Metadata export complete
Exporting feature list: ..\data\processed\feature_list.txt
Feature list export complete: 32 modeling features identified

FEATURE ENGINEERING COMPLETE!
==================================================
Final dataset: 453,935 observations × 41 features
Exported to: ..\data\processed
   • modeling_dataset.parquet (production-ready)
   • modeling_dataset_sample.csv (validation)
   • dataset_metadata.json (comprehensive info)
   • feature_list.txt (model preparation)

READY FOR PREDICTIVE MODELING!
   Next step: Build machine learning models for violation hotspot prediction
   Prediction targets available: immediate, tactical, binary classification, risk sc
oring
   Features ready: temporal, spatial, CUNY, adaptation, enforcement intelligence
```

Summary: From Reactive Analysis to Predictive Intelligence

What We've Built

This notebook successfully transformed raw MTA data into a feature-rich, model-ready dataset. We've moved beyond simple retrospective analysis to engineer a foundation for a forward-looking, predictive system. Our features capture:

Temporal Intelligence

- Creating nuanced time blocks like morning/evening rush, school hours, and CUNY class change windows.
- Engineering seasonal and academic cycle features (month, semester period).
- Tracking the enforcement timeline with features like 'days_since_ace_implementation'.

Spatial Intelligence

- Integrating comprehensive GTFS data (stops, routes, shapes) for all five boroughs.
- Identifying violation hotspot clusters using DBSCAN on a 50,000-record sample.
- Calculating spatial density to measure violation concentration in a given area.

CUNY-Specific Intelligence

- Building proximity features based on Haversine distance to 7 key CUNY campuses.

- Classifying routes as 'CUNY-serving' based on the concentration of violations within a 500m buffer.
- Creating interaction features to capture patterns specific to student travel times.

Enforcement Adaptation Intelligence

- Quantifying enforcement predictability at each stop using Shannon entropy.
- Measuring repeat offender concentration and vehicle diversity at the stop level.
- Detecting route-level learning curves by analyzing the trend of monthly violations over time.

Multiple Prediction Targets

- Engineering 9 distinct target variables for different use cases.
- Creating targets for immediate forecasting (`violation_count_next_hour`) and tactical planning (`violation_count_next_day`).
- Developing composite targets for risk and priority scoring (`violation_risk_score` , `deployment_priority`).

Key Innovations

1. **Adaptation Analysis**: This is the first known system to quantify violator adaptation by modeling enforcement predictability (entropy), repeat offender concentration, and learning curves over time.

2. **CUNY-Centric Modeling**: We've built purpose-specific features to analyze and ultimately protect crucial student transportation corridors, moving beyond a one-size-fits-all approach.

3. **Multi-Horizon Forecasting**: By creating targets for the next hour, next day, and overall risk, our dataset enables a flexible system that can inform both immediate operational deployment and long-term strategic planning.

Impact on NYC Transit

This feature-rich dataset is the engine for a system that enables:

- Proactive resource deployment using our 24-hour hotspot forecasts.
- Data-driven protection for key student transit corridors.
- Smarter enforcement that can adapt to predictable, systematic violator behavior.
- Optimized resource allocation based on data-driven priority scores.

Next Phase: Predictive Modeling

The final exported Parquet file, `modeling_dataset.parquet` , is now ready for machine learning. The next step is to build, train, and evaluate models to:

- Classify high-risk hotspots using our binary severity flags.
- Forecast exact violation counts using regression models like LightGBM or XGBoost.
- Generate the final, prescriptive deployment recommendations for the MTA.

Result: Transforming the 97.5% ACE failure rate into a predictive system that deploys enforcement resources exactly where and when they'll be most effective, with special protection for CUNY student routes.