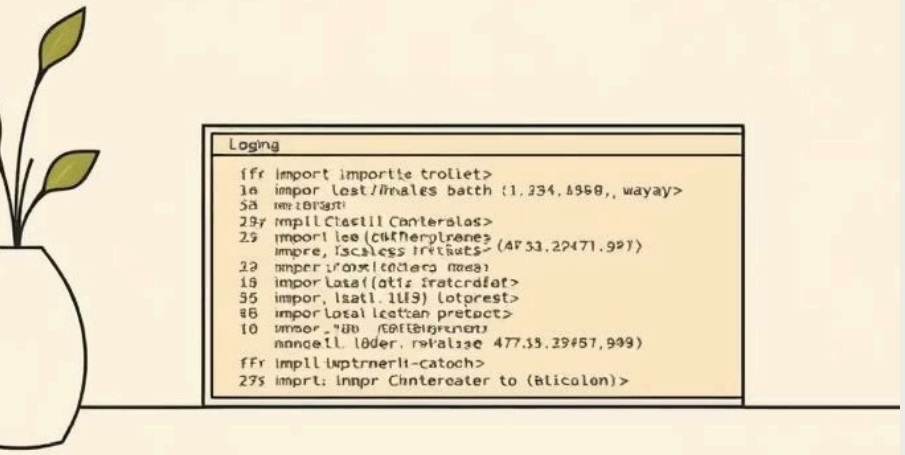
Milestone 1: Traffic Data Generation and MQTT Publishing

This presentation outlines the architecture and implementation of our real-time traffic data generator, focusing on its core components, data flows, and configuration. We'll explore the Python code, the configuration file, and an end-to-end system overview.



Imports and Logging Setup

- **import random**
  import time
  import json
  import yaml
  import logging
  import csv
  from datetime import datetime, timedelta
  import paho.mqtt.client as mqtt
  import os

These lines import necessary libraries. `paho.mqtt.client` is for MQTT communication, `time` for delays, `json` for data serialization, `random` for simulating variability, and `yaml` for configuration parsing. `logging` ensures proper message output, and `datetime`/`timedelta` handle time-based operations.

- **logging.basicConfig(level=logging.INFO,filename="traffic_mqtt.log", format='%(asctime)s - %(levelname)s - %(message)s')**

This sets up basic logging, providing timestamps and severity levels for each message, crucial for debugging and operational monitoring.

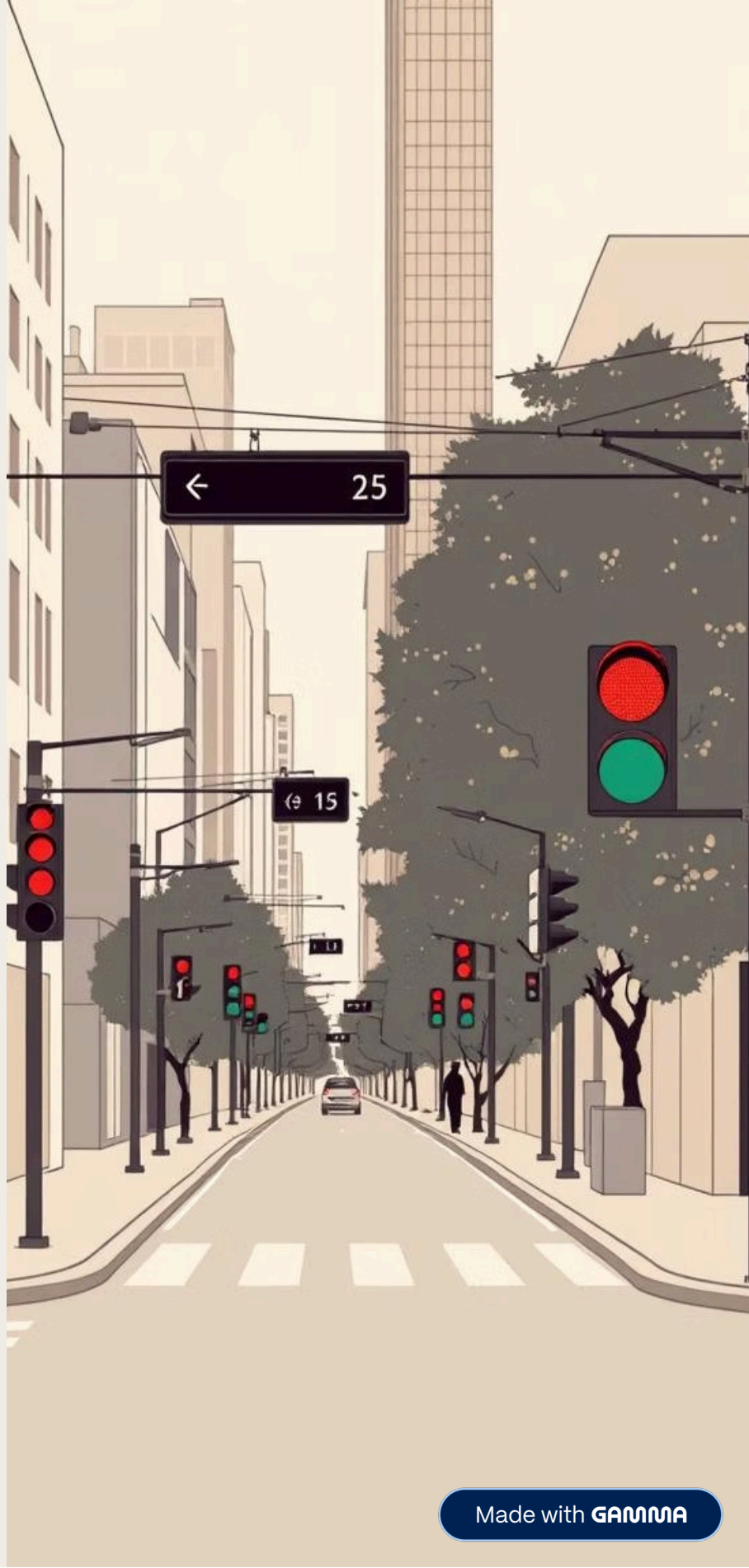
Class Definition: TrafficDataGeneratorMQTT

- class TrafficDataGeneratorMQTT:
  def __init__(self, config_path="config_m1.yaml"):
  # تحديد المسار الكامل للملف (نسبيًا للسكربت)
  base_dir = os.path.dirname(os.path.abspath(__file__))
  config_full_path = os.path.join(base_dir, config_path)

  # تحميل الكونفيج
  with open(config_full_path, "r") as f:
  self.config = yaml.safe_load(f)

  # قراءة إعدادات Milestone 1
  self.frequency = self.config["generator"]["frequency"]
  self.streets = self.config["generator"]["streets"]
  self.rush_hours = self.config["generator"]["rush_hours"]
  self.broker = self.config["mqtt"]["broker"]
  self.port = self.config["mqtt"]["port"]
  self.topic = self.config["mqtt"]["topic"]
  self.output_format = self.config["generator"]["output"]["format"]
  self.output_file = os.path.join(base_dir, self.config["generator"]
["output"]["file"])

The TrafficDataGeneratorMQTT class encapsulates the entire data generation and publishing logic. Its __init__ method loads the configuration, initializes the MQTT client, and sets up initial states for streets, weather, and traffic lights. This ensures the system starts in a well-defined state.



Helper Functions: Time and Environmental Factors

is_rush_hour(self):

Determines if the current time falls within defined rush hour periods. This is crucial for applying traffic multipliers, simulating realistic peak-hour congestion.

```
def is_rush_hour(self, now):
        current_time = now.strftime("%H:%M")
        for rh in self.rush_hours:
            if rh["start"] <= current_time <= rh["end"]:
                return rh["multiplier"]
        return 1
```

get_light_level(self):

Simulates light conditions (day/night) based on the current hour. This influences visibility, which can be a factor in accident risk or driver behavior in more advanced simulations.

```
def get_light_level(self, now):
        hour = now.hour
        if 6 <= hour < 17:
            return "day"
        elif 17 <= hour < 19:
            return "dusk"
        else:
            return "night"
```

Helper Functions: Weather and Traffic State

get_weather(self):

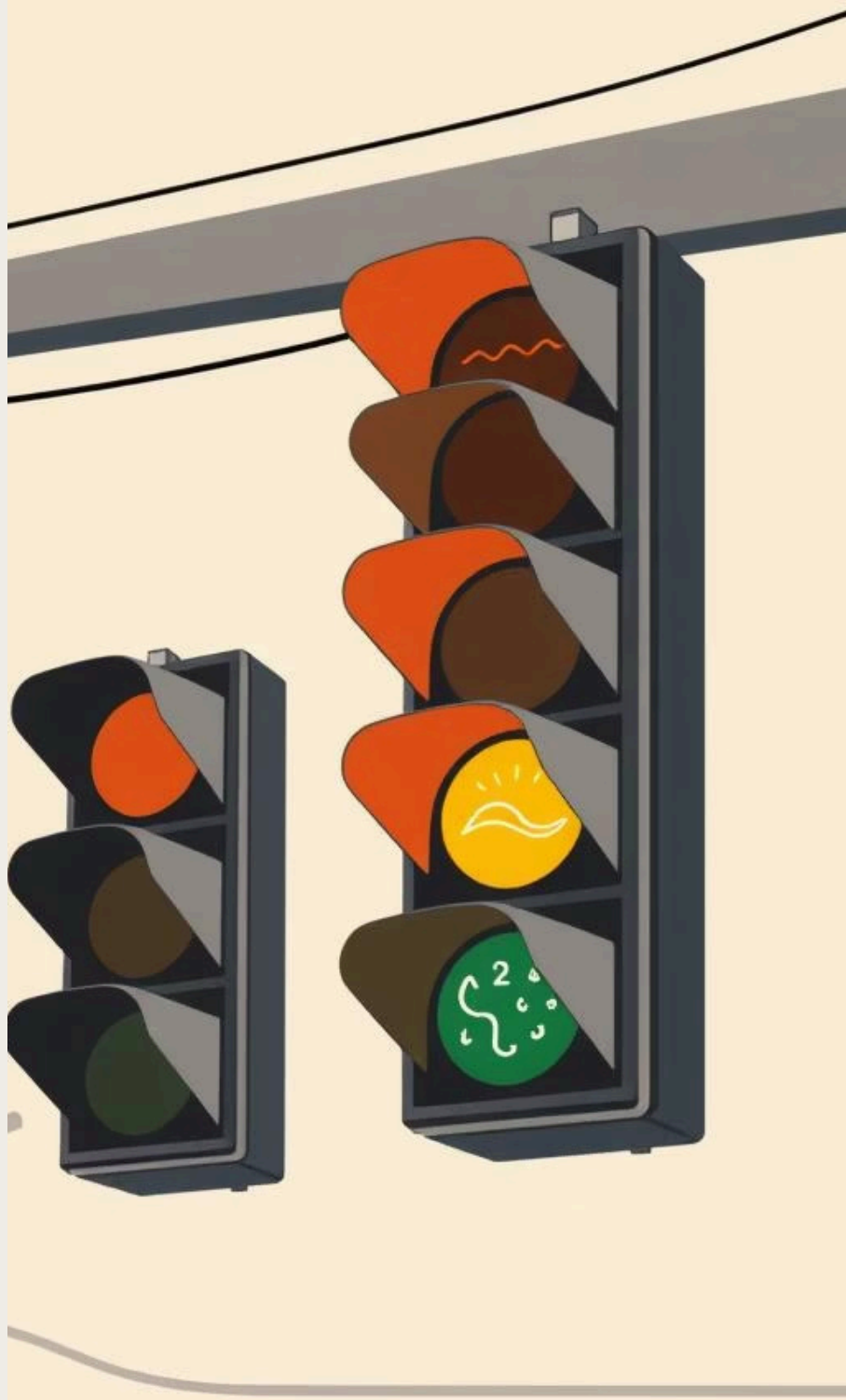
Randomly selects a weather condition (sunny, rainy, snowy, foggy). Weather significantly impacts traffic flow and safety, and this function periodically updates the environmental state.

```
def get_weather(self, now):
        update_interval = self.config["generator"]["weather"]
["update_interval"]
        if now - self.last_weather_update >
timedelta(seconds=update_interval):
            self.weather = random.choice(self.weather_states)
            self.last_weather_update = now
        return self.weather
```

get_traffic_light(self, street):

Randomly changes the state of a traffic light for a given street (green, yellow, red). This adds dynamic, localized variations to traffic flow, mimicking real-world traffic signal operations.

```
def get_traffic_light(self, now):
        if now - self.last_light_update > timedelta(seconds=30):
            self.current_light = random.choice(self.traffic_lights)
            self.last_light_update = now
        return self.current_light
```



Effect Application Functions

01

```
def apply_weather_effects(self, speed, count, weather):
        if weather == "rain":
            speed *= 0.8
            count *= 0.8
        elif weather == "fog":
            speed *= 0.7
            count *= 0.7
        elif weather == "storm":
            speed *= 0.5
            count *= 0.5
        return speed, count
```

Adjusts vehicle speed based on weather conditions. For example, rainy or snowy weather reduces speed and increases the chance of minor incidents, making the simulation more realistic.

02

```
def apply_traffic_light_effects(self, speed, count, light):
        if light == "red":
            speed = random.uniform(0, 10)
            count *= 1.5
        elif light == "yellow":
            speed *= 0.5
        return speed, count
```

Modifies speed based on traffic light status. A red light will bring speed to zero, while yellow might cause deceleration, simulating how drivers react to signals.

03

_load_config(self, config_path):

Loads configuration from a YAML file. This method makes the system highly configurable without requiring code changes, allowing easy adjustment of parameters like MQTT settings, streets, and intervals.

04

_setup_mqtt_client(self):

Initializes and connects the MQTT client to the broker specified in the configuration. It sets up callback functions for connection and message receipt, ensuring reliable communication.

Data Generation and Publishing

1

```
def generate_record(self):
        now = datetime.now()
        street = random.choice(self.streets)

        multiplier = self.is_rush_hour(now)
        vehicle_count = int(random.randint(5, 30) * multiplier)

        vehicle_speed = round(
            random.gauss(street["speed_mean"], street["speed_std"]),
2
        )
        vehicle_speed = max(5, min(vehicle_speed, 150))

        light_level = self.get_light_level(now)
        weather = self.get_weather(now)
        traffic_light = self.get_traffic_light(now)

        vehicle_speed, vehicle_count =
self.apply_weather_effects(vehicle_speed, vehicle_count, weather)
        vehicle_speed, vehicle_count =
self.apply_traffic_light_effects(vehicle_speed, vehicle_count,
traffic_light)

        vehicle_speed = round(max(0, min(vehicle_speed, 150)), 2)
        vehicle_count = max(0, int(vehicle_count))

        solar_energy_level = round(random.uniform(0.0, 1.0), 2)
        if light_level == "night" or weather in ["fog", "rain", "storm"]:
            lighting_demand = "high"
        elif light_level == "dusk":
            lighting_demand = "medium"
        else:
            lighting_demand = "low"

        record = {
            "timestamp": now.strftime("%Y-%m-%d %H:%M:%S"),
            "street_name": street["name"],
            "vehicle_count": vehicle_count,
            "vehicle_speed": vehicle_speed,
            "light_level": light_level,
            "weather": weather,
            "traffic_light": traffic_light,
            "solar_energy_level": solar_energy_level,
            "lighting_demand": lighting_demand
        }
        return record
```

This core function generates a single traffic data record. It pulls together all simulated factors: rush hour, light level, current weather, and traffic light status for a random street. It then calculates a realistic speed and generates a timestamped data payload.

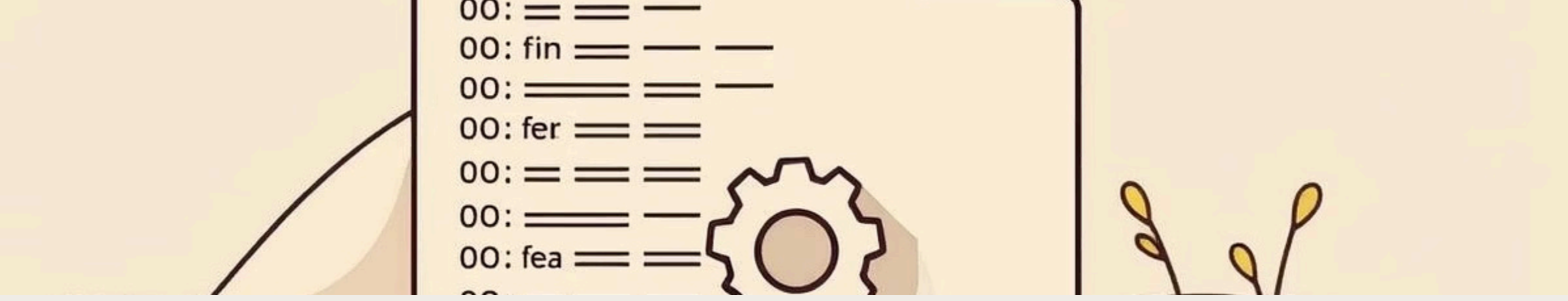
This function is the heart of the simulation, ensuring each data point reflects a complex interplay of environmental and operational factors.

2

```
def publish_record(self, record):
        payload = json.dumps(record)
        self.client.publish(self.topic, payload)
        logging.info(f"Published to MQTT: {payload}")
        print(f"Published to MQTT: {payload}")
```

Takes the generated record, serializes it to JSON, and publishes it to the configured MQTT topic. It also logs the publication, providing visibility into the system's operation.

This ensures that generated data is immediately available to subscribing systems, fulfilling the real-time data streaming requirement.



Configuration File: config_m1.yaml

```
# ============================================================
#   MILSTONE 1 CONFIG - TRAFFIC DATA GENERATOR
# ============================================================

# --------------------------
#   DATA GENERATOR SETTINGS
# --------------------------
generator:
  frequency: 5   # seconds between data points

  streets:
    - name: "Agricultural Rd - Menoufia NU (Cairo bound)"
      speed_mean: 55
      speed_std: 12
    - name: "Agricultural Rd - Menoufia NU (Alexandria bound)"
      speed_mean: 20
      speed_std: 5
    - name: "Overpass Ramp - Menoufia NU (bridge)"
      speed_mean: 30
      speed_std: 8

rush_hours:
  - start: "08:00"
    end: "09:30"
    multiplier: 4
  - start: "13:00"
    end: "14:00"
    multiplier: 4
  - start: "17:00"
    end: "18:00"
    multiplier: 4

weather:
  update_interval: 600   # seconds

solar_energy:
  min: 0.0
  max: 1.0

lighting_demand:
  night: high
  dusk: medium
  day: low
  weather_impact:
    fog: high
    rain: high
    storm: high

output:
  format: "csv"
    file: "./output/traffic_data.csv"

# --------------------------
#   MQTT SETTINGS (Used in M1)
# --------------------------
mqtt:
  broker: "broker.hivemq.com"
  port: 1883
  topic: "traffic/data/menofia_national_university"
```

The `config_m1.yaml` file centralizes all configurable parameters. `frequency_seconds` controls data generation rate. `streets` lists simulated locations. `rush_hours` defines peak traffic periods. `weather_update_interval_seconds` sets how often weather changes. `mqtt` specifies broker details and topic. `output_format` currently set to "json" for publishing.

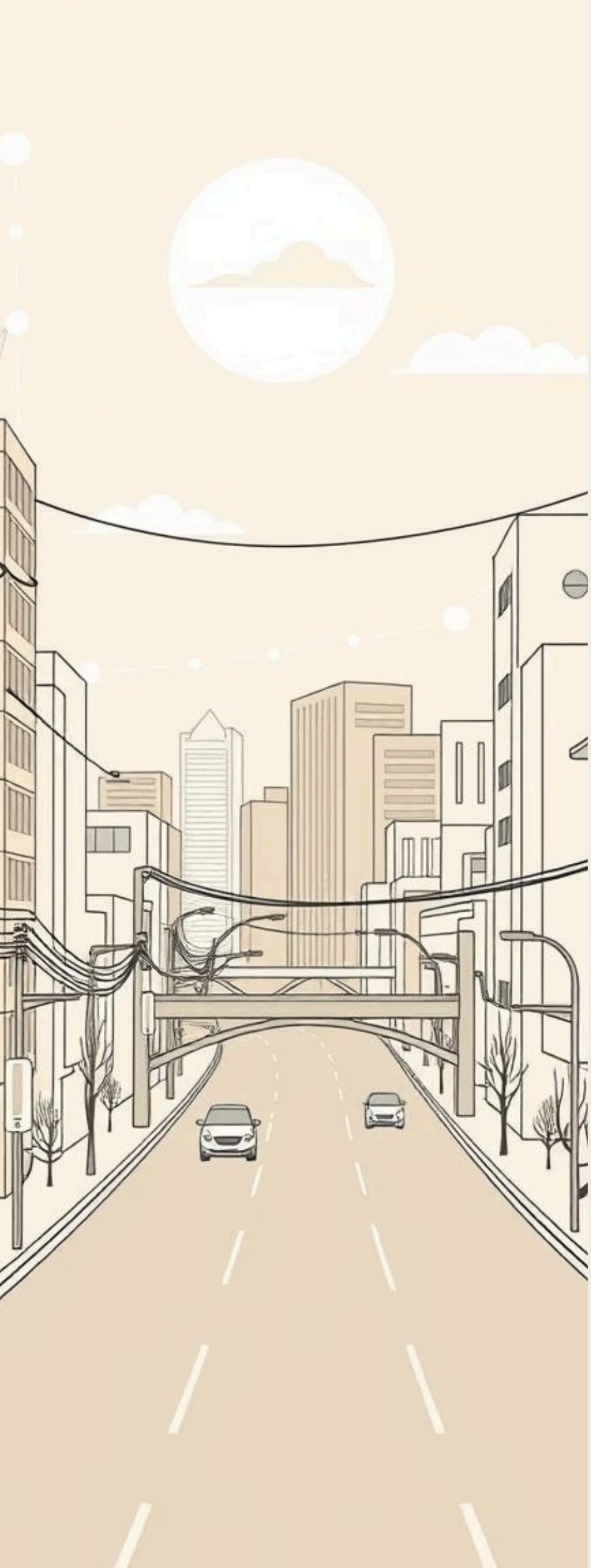
Main Loop and Data Persistence

```
def run(self):
        logging.info("Traffic MQTT + CSV data generation started.")
        while True:
            record = self.generate_record()
            self.publish_record(record)
            if self.output_format == "csv":
                self.save_to_csv(record)
            time.sleep(self.frequency)
```

The main execution loop. It continuously generates and publishes traffic records at the specified frequency. It also handles periodic weather updates, ensuring the simulation remains dynamic. This loop is the engine driving the continuous data stream.

```
def save_to_csv(self, record):
        with open(self.output_file, mode="a", newline="",
encoding="utf-8") as file:
            writer = csv.writer(file, delimiter=",")
            writer.writerow([
                record["timestamp"],
                record["street_name"],
                record["vehicle_count"],
                record["vehicle_speed"],
                record["light_level"],
                record["weather"],
                record["traffic_light"],
                record["solar_energy_level"],
                record["lighting_demand"]
            ])
        logging.info(f"Saved to CSV: {record}")
```

(Optional/Future Expansion) This method is designed to save generated traffic records to a CSV file. While not strictly used in the MQTT publishing flow, it provides a valuable capability for local data storage, debugging, or offline analysis.



End-to-End System Overview

Config File

Defines all simulation parameters and MQTT broker details.

Traffic Generator

Reads config, simulates traffic, weather, and light conditions to create records.

MQTT Broker

Receives and routes real-time traffic data messages.

Subscribing Systems

(Future) Consume data for analysis, visualization, or control actions.

Example Generated Traffic Record


```
{
  "timestamp": "2023-10-27 10:30:00",
  "street": "Main Street",
  "speed_mph": 35.5,
  "traffic_density": "moderate",
  "weather": "sunny",
  "light_level": "day",
  "traffic_light_status": "green",
  "incident": false
}
```


Milestone 1 successfully establishes a configurable, real-time traffic data generation and MQTT publishing system, laying the groundwork for complex IoT traffic management solutions.

Milestone 2 – Real-Time Traffic Analytics Pipeline

A real-time ingestion, transformation, alerting, and storage pipeline using MQTT, Kafka, Spark Streaming, SQL Server, and SSIS.

System Architecture

This architecture outlines the interconnected components that enable real-time processing and analysis of traffic data, from initial data capture to actionable insights and automated responses.

- **IoT MQTT Publisher:** Simulates traffic sensors sending data.
- **MQTT Broker:** Collects sensor data efficiently.
- **Kafka Bridge:** Seamlessly transfers MQTT data to Kafka.
- **Kafka (Docker):** Robust message queuing for data streams.
- **Spark Structured Streaming:** Real-time data processing and transformation.
- **Parquet Data Lake:** Optimized storage for historical analysis.
- **SQL Server:** Relational storage for structured, queried data.
- **Kafka Alerts Topic:** Publishes critical alerts for immediate action.
- **Lighting Actuator (MQTT):** Automated response to lighting alerts.
- **Telegram Notifier:** Delivers human-readable alerts.

MQTT → Kafka Bridge

The MQTT to Kafka Bridge is crucial for integrating data from IoT devices into our streaming platform. It ensures reliable data transfer and format consistency.

```
producer = KafkaProducer(
    bootstrap_servers='localhost:29092',
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

def on_message(client, userdata, msg):
    data = json.loads(msg.payload.decode())
    producer.send("traffic_topic", value=data)
    producer.flush()
```

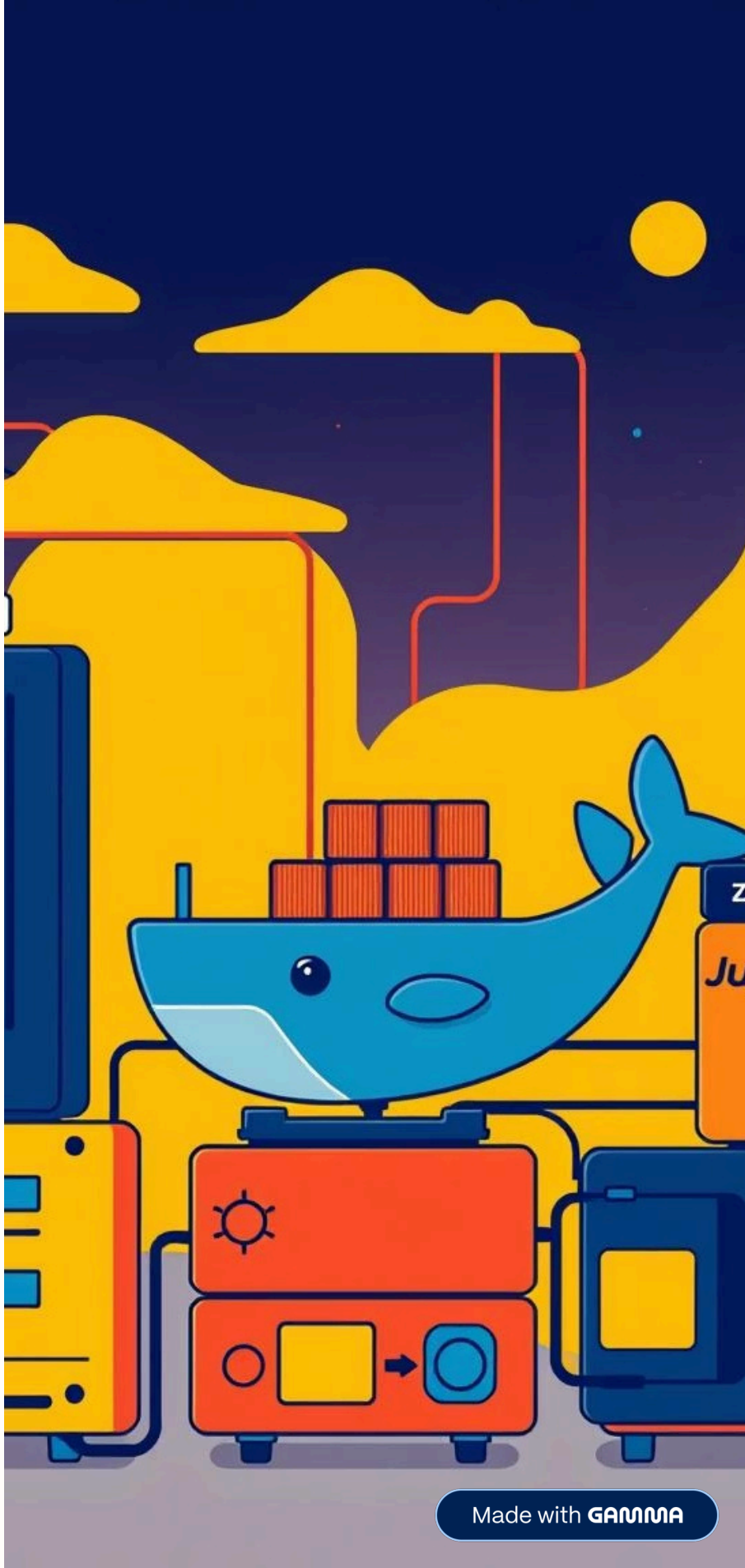
- Converts incoming MQTT messages into Kafka events, enabling seamless data flow.
- Maintains **Quality of Service (QoS) of 1** for message reliability, guaranteeing at least once delivery.
- Should incorporate robust error handling and retry mechanisms to prevent data loss during network interruptions or Kafka unavailability.

Docker Compose Infrastructure

Our entire streaming infrastructure is orchestrated using Docker Compose, ensuring a reproducible and scalable environment. This setup simplifies deployment and management of critical services.

```
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.5.0
  kafka:
    image: confluentinc/cp-kafka:7.5.0
    environment:
      KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://kafka:9092,PLAINTEXT_HOST://localhost:29092
  jupyter:
    image: jupyter/pyspark-notebook:spark-3.5.0
```

- Kafka is configured to be accessible both internally within the Docker network (port 9092) and externally for local development and testing (port 29092).
- Leverages health checks within the Docker Compose configuration to ensure that all services start up correctly and in the proper sequence, preventing issues caused by service dependencies.
- The Jupyter/PySpark notebook provides an interactive environment for developing and testing Spark Streaming applications.



Spark Streaming: Ingestion + Parsing

Spark Structured Streaming handles the continuous ingestion and parsing of raw traffic data from Kafka, preparing it for further processing and analysis.

```
df = spark.readStream.format("kafka") \
  .option("kafka.bootstrap.servers", "kafka:9092") \
  .option("subscribe", "traffic_topic") \
  .load()
json_df = df.selectExpr("CAST(value AS STRING)")
parsed_df = json_df.select(from_json(col("value"), schema).alias("data")).select("data.*")
```

The raw Kafka messages, which are JSON strings, are cast to STRING type and then parsed using a predefined schema:

- **Schema:** `timestamp` (timestamp), `street` (string), `vehicle_count` (int), `vehicle_speed` (float), `weather_condition` (string), `lighting_info` (string), `solar_energy_level` (float), `lighting_demand_kW` (float).
- **Windowing:** Data can be grouped and aggregated over time windows to analyze trends or detect events over specific periods.
- **Watermarking:** Handles late-arriving data by specifying a threshold for how long to wait before considering data complete for a given window.
- **JSON Parsing:** Extracts structured data from the raw JSON messages, making it queryable and usable for subsequent transformations.

Feature Engineering + Alert Rules

We enrich the raw data with engineered features and define alert rules to identify critical traffic conditions and energy anomalies in real-time.

```
df_transformed = parsed_df \
 .withColumn("hour_of_day", hour(to_timestamp(col("timestamp")))) \
 .withColumn("congestion_level",
  when((col("vehicle_count") >= 30) & (col("vehicle_speed") <= 20), "High")
  .when((col("vehicle_count") >= 15) & (col("vehicle_speed") <= 40), "Medium")
  .otherwise("Low")
 ) \
 .withColumn("lighting_consumption_kWh",
  when(col("lighting_demand_kW") - col("solar_energy_level") > 0, col("lighting_demand_kW") - col("solar_energy_level"))
  .otherwise(0.0)
 )
```

- **Hourly Features:** Extracting `hour_of_day` enables time-based analysis and pattern recognition.
- **Congestion Levels:** Defined based on `vehicle_count` and `vehicle_speed` thresholds:
  - **High Congestion:** Vehicle count **>= 30** AND Speed **<= 20** mph
  - **Medium Congestion:** Vehicle count **>= 15** AND Speed **<= 40** mph
  - **Low Congestion:** Otherwise
- **Lighting Consumption:** Calculates actual energy usage by subtracting `solar_energy_level` from `lighting_demand_kW`, ensuring positive consumption values.

Alerts Pipeline (Kafka Alerts Topic)

Critical alerts are published to a dedicated Kafka topic, enabling immediate notification and automated responses to significant events.

```
alerts_for_kafka = agg2.filter(col("is_alert") == True) \
  .select(to_json(struct(*agg2.columns)).alias("value"))
```

Any row in the transformed DataFrame marked with `is_alert = True` is converted to a JSON string and sent to the Kafka Alerts Topic. This ensures that only relevant alerts are propagated through the system.

- **Telegram Notifier:** A consumer listens to the alerts topic and sends instant notifications to a designated Telegram channel, keeping stakeholders informed in real-time.
- **Lighting Actuator:** Another consumer processes alerts related to lighting conditions or energy anomalies, triggering MQTT commands to adjust smart city lighting systems automatically.
- **Dashboard:** A real-time dashboard subscribes to the alerts topic, displaying active incidents and their details for operational monitoring.

Parquet Data Lake Output

Processed traffic data is continuously written to a Parquet data lake, providing an efficient and scalable solution for long-term storage and analytical queries.

```
df_transformed.writeStream \
  .format("parquet") \
  .option("path", "/data/processed") \
  .option("checkpointLocation", "/data/checkpoints") \
  .start()
```

- **Efficient Analytics:** Parquet is a columnar storage format, highly optimized for analytical queries, allowing faster data retrieval for specific columns.
- **Compression:** Parquet files offer excellent compression, significantly reducing storage costs and I/O operations.
- **Schema Evolution:** It supports schema evolution, allowing us to add or modify columns over time without breaking existing data.
- **Checkpointing:** The `checkpointLocation` ensures fault tolerance and exactly-once processing by recording the progress of the streaming query.

SQL Server Sink (3 Normalized Tables)

In parallel with the data lake, a subset of the processed data is stored in a normalized SQL Server database, designed for structured queries and reporting.

```
sensors_df.write.format("jdbc").options(
  url=JDBC_URL,
  user=DB_USER,
  password=DB_PASS,
  dbtable="traffic_sensors_data",
  driver="com.microsoft.sqlserver.jdbc.SQLServerDriver"
).mode("append").save()
```

The data is distributed across three normalized tables to minimize redundancy and improve data integrity:

- **1.** `traffic_sensors_data`**:** Stores core traffic measurements:
  - `timestamp` (DATETIME2)
  - `street_name` (NVARCHAR(200))
  - `vehicle_count` (INT)
  - `vehicle_speed` (FLOAT)
  - `hour_of_day` (INT)
  - `day_of_week` (NVARCHAR(20))
  - `is_peak_hour` (BIT)
  - `congestion_level` (NVARCHAR(20))
  - `is_congested` (BIT)
- **2.** `traffic_weather_conditions`**:** Stores weather-related information:
  - `timestamp` (DATETIME2)
  - `street_name` (NVARCHAR(200))
  - `weather_condition` (NVARCHAR(50))
  - `temperature_celsius` (FLOAT)
  - `precipitation_mm` (FLOAT)
- **3.** `traffic_energy_analysis`**:** Captures energy consumption data:
  - `timestamp` (DATETIME2)
  - `street_name` (NVARCHAR(200))
  - `lighting_demand_kW` (FLOAT)
  - `solar_energy_level` (FLOAT)
  - `lighting_consumption_kWh` (FLOAT)

SSIS Integration for Batch Processing

SQL Server Integration Services (SSIS) is used for robust batch operations, including database restoration, table creation, and ETL processes for historical data or specific reporting needs.

A) SQL Server Restore Script

```
-- Inspect file structure
RESTORE FILELISTONLY FROM DISK =
N'C:\\backups\\Final_Project_des.bak';

RESTORE DATABASE [FinalProjectDB]
FROM DISK = N'C:\\backups\\Final_Project_des.bak'
WITH MOVE 'FinalProjectDB_Data' TO
'C:\\SQLData\\FinalProjectDB.mdf',
MOVE 'FinalProjectDB_Log' TO
'C:\\SQLLog\\FinalProjectDB.ldf',
REPLACE;
```

- Update logical names based on `FILELISTONLY` output for correct file paths.
- Ensure SQL Server service account has read permissions to the `.bak` file.

B) SSIS Control Flow

- **Execute SQL Task:** Restores the database from a backup file.
- **Execute SQL Task:** Creates necessary tables if they do not exist.
- **Data Flow Task:** Handles ETL from flat files to SQL Server.
- **File System Task:** Archives processed input files.
- **Send Mail Task:** Notifies on package success or failure.

C) SSIS Create Table Scripts

```
IF NOT EXISTS (SELECT * FROM sys.objects WHERE
name = 'traffic_sensors_data')
BEGIN
   CREATE TABLE dbo.traffic_sensors_data(
      [timestamp] DATETIME2,
      [street_name] NVARCHAR(200),
      [vehicle_count] INT,
      [vehicle_speed] FLOAT,
      [hour_of_day] INT,
      [day_of_week] NVARCHAR(20),
      [is_peak_hour] BIT,
      [congestion_level] NVARCHAR(20),
      [is_congested] BIT
   );
END

-- Similar scripts for traffic_weather_conditions and
traffic_energy_analysis
```

These scripts ensure that the target tables exist before data insertion, preventing runtime errors.

D) SSIS Data Flow Mapping

Flat File Source columns → SQL columns:

- `timestamp` → `timestamp`
- `street_name` → `nvarchar`
- `vehicle_count` → `int`
- `vehicle_speed` → `float`

Data Conversion:

- `timestamp` → `DT_DBTIMESTAMP`
- `vehicle_speed` → `DT_R8`
- `bools` → `DT_BOOL`

Destination: OLE DB → FinalProjectDB

E) Script Task for Bulk Insert (C#)

```
public void Main() {
  string connStr =
Dts.Variables["User::DB_CONNECTION"].Value.ToStrin
g();
  string filePath =
Dts.Variables["User::CurrentFile"].Value.ToString();
  string sql = $@" BULK INSERT
dbo.traffic_sensors_data FROM '{filePath}' WITH
(FIRSTROW = 2, FIELDTERMINATOR = ',',
ROWTERMINATOR = '\n', TABLOCK); ";
  using (SqlConnection conn = new
SqlConnection(connStr)) {
    conn.Open();
    new SqlCommand(sql, conn).ExecuteNonQuery();
  }
  Dts.TaskResult = (int)ScriptResults.Success;
}
```

Automates high-performance bulk data loading from CSV files into SQL Server.

F) SSIS Configuration Variables

DB_CONNECTION	Data Source=SQL_HOST;Initial Catalog=FinalProjectDB;User ID=sa;Password=maya;
INPUT_FOLDER	C:\data\incoming
ARCHIVE_FOLDER	C:\data\archive
CurrentFile	File path during loop

Externalizes critical settings for flexible deployment and easy maintenance.

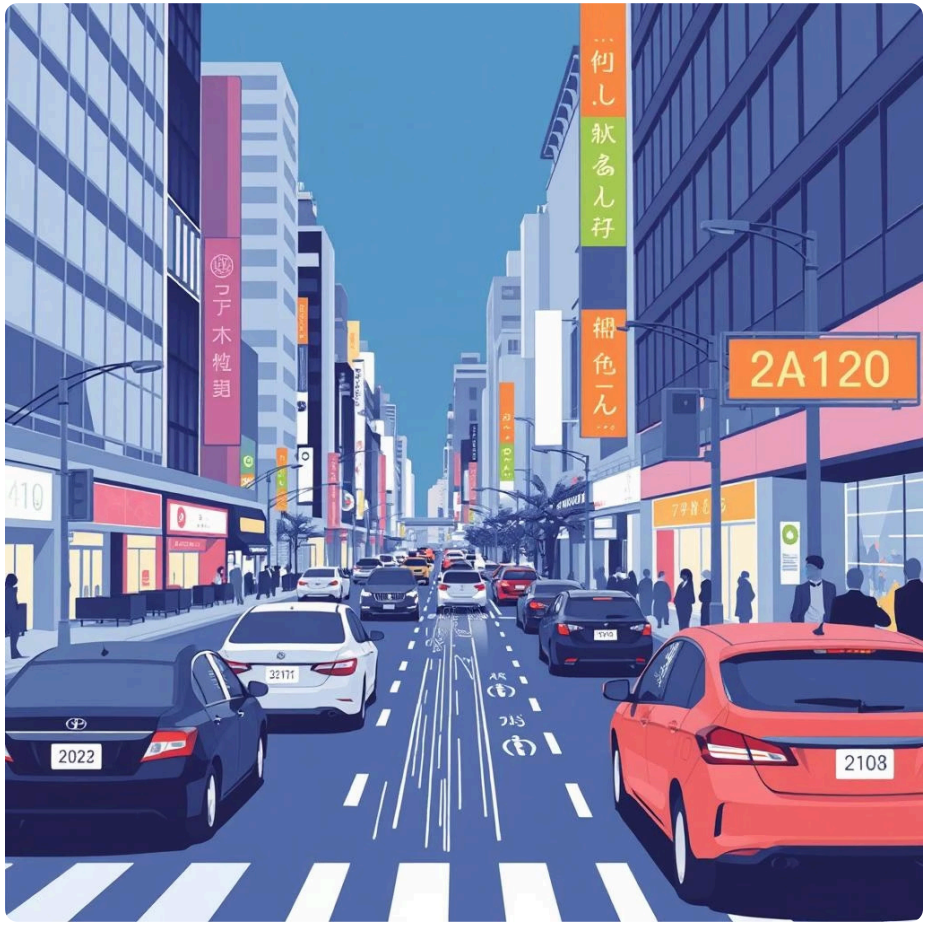
Milestone 3 – Real-Time Traffic Monitoring, Congestion Alerts & IoT Lighting Control

Introduction: Revolutionizing Traffic Management

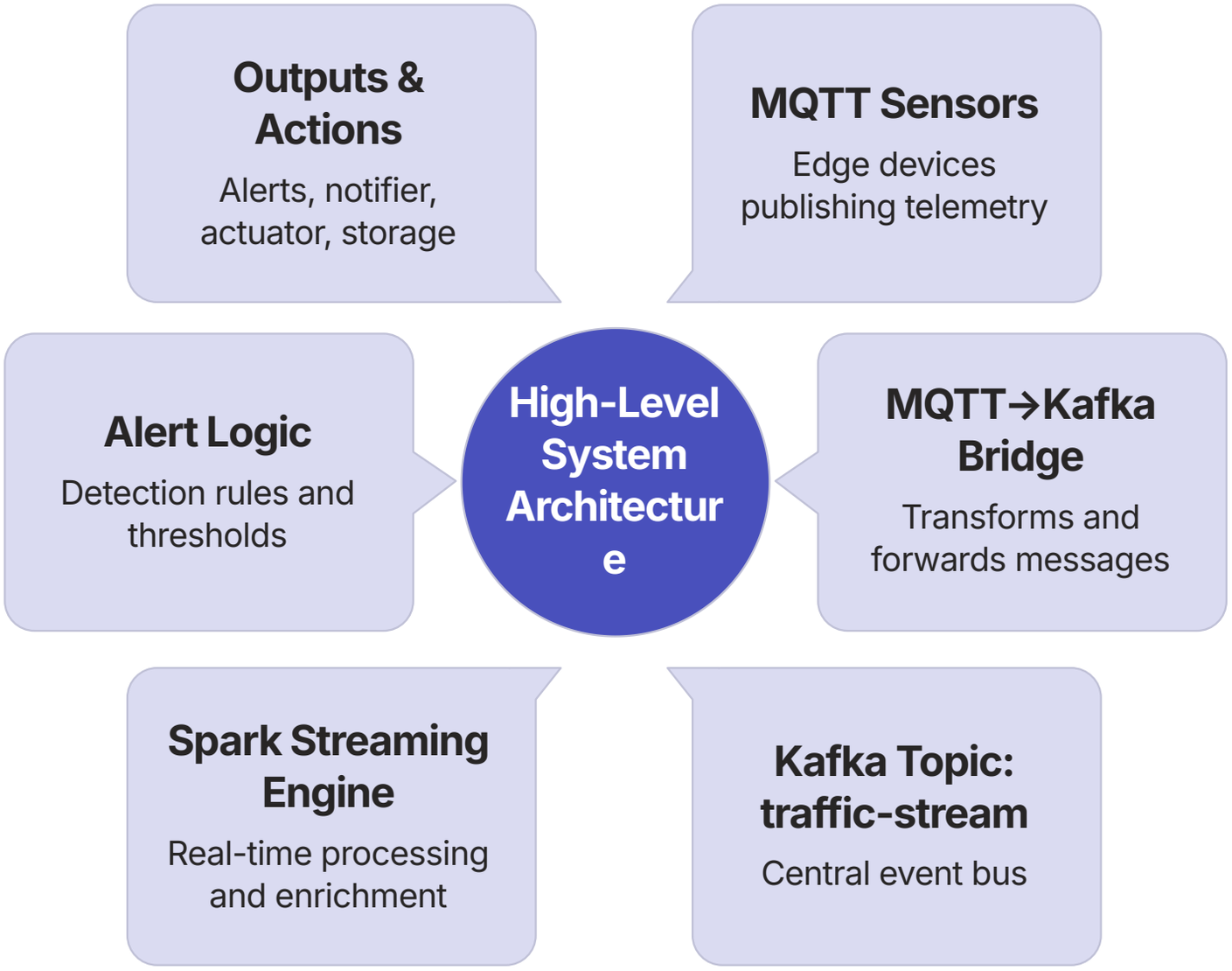
Our project focuses on transforming urban traffic management through real-time data and intelligent control. Milestone 3 significantly advances this vision by integrating sophisticated alert mechanisms and IoT automation.

Key Objectives:

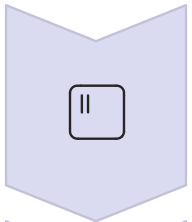
- Stream live traffic data from IoT sensors
- Detect traffic congestion with precision
- Generate instant, multi-factor alerts
- Integrate IoT lighting automation for dynamic street illumination
- Send critical notifications via Telegram



High-Level System Architecture

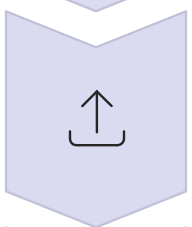


Data Flow: From Sensor to Action



IoT Sensors to MQTT

Traffic data (vehicle count, speed) is collected by IoT sensors and published to an MQTT broker.



MQTT Bridge to Kafka

Our custom bridge securely ingests MQTT messages and forwards them to a Kafka "traffic-stream" topic.

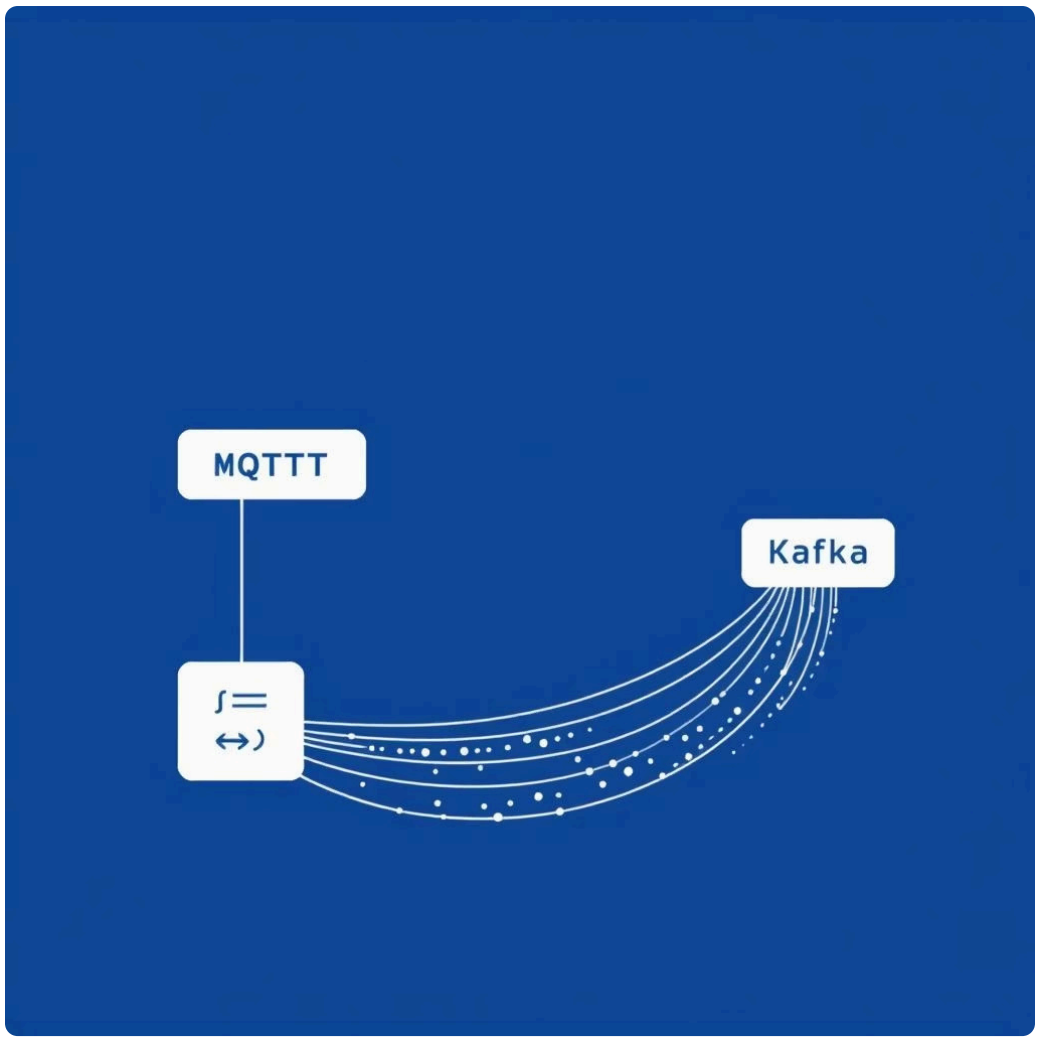
Spark Streaming Analytics

Apache Spark consumes the Kafka stream, performing real-time processing and congestion detection.

Alert Detection & Output

Congestion alerts are generated and dispatched to multiple destinations for immediate response and historical analysis.

Module 1: MQTT to Kafka Bridge



Purpose:

Seamlessly transfer real-time IoT sensor data from MQTT topics to Kafka's "traffic-stream" for scalable processing.

Key Functionality:

- Reads configuration dynamically from YAML files.
- Subscribes to designated MQTT sensor topics.
- Converts incoming JSON payloads into a structured format.
- Ensures reliable, real-time data ingestion into Kafka for downstream analytics.

Important Code Snippet:

```
def on_message(client, userdata, msg):
    try:
        payload = msg.payload.decode()
        # if payload is already JSON string
        data = json.loads(payload)
    except Exception:
        data = {"raw": payload}
    # publish to Kafka
    producer.send(KAFKA_TOPIC, value=data)
    producer.flush()
    print("Bridged to Kafka:", data)
```

Module 2: Spark Streaming Analytics

Purpose:

Perform real-time traffic congestion detection using sophisticated windowed aggregations on live data streams.

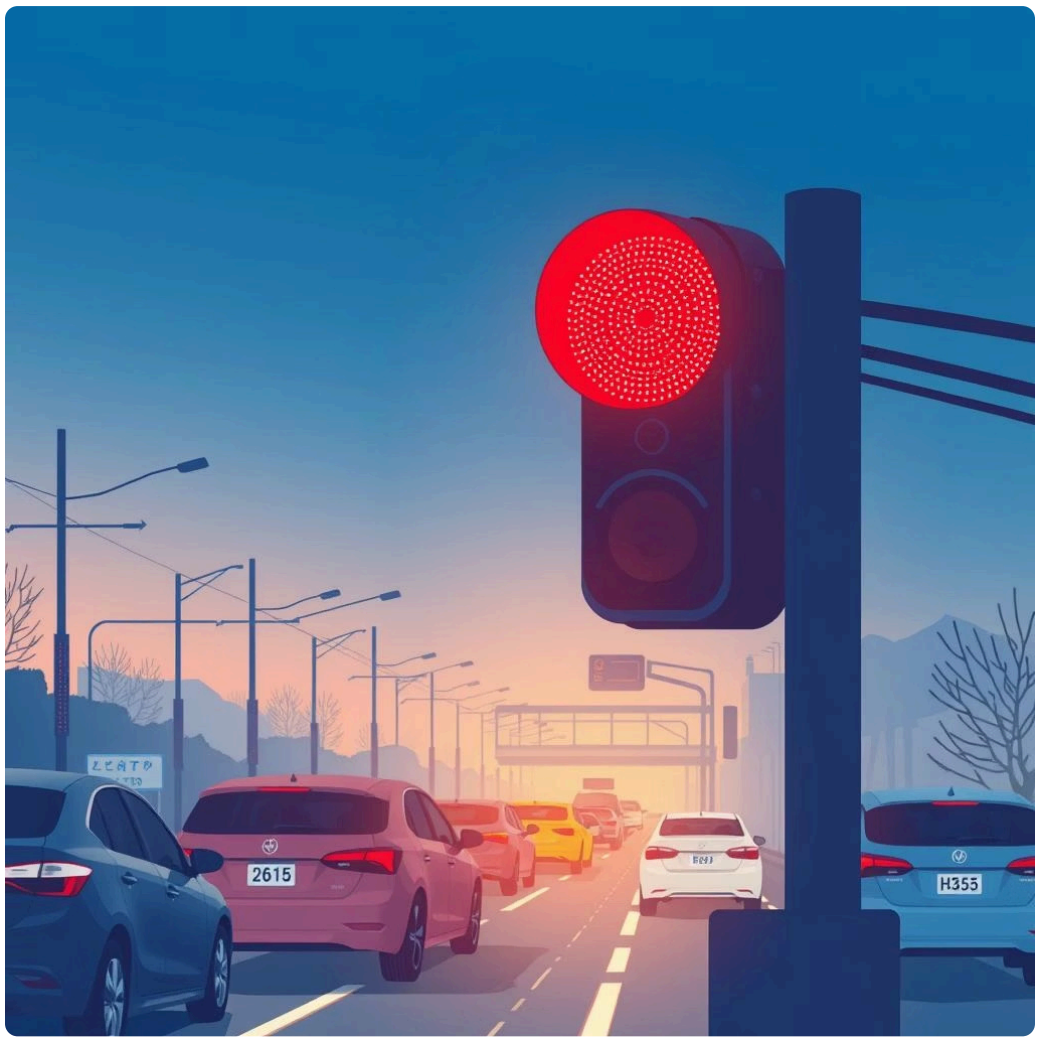
Key Processing Steps:

- Reads continuous data streams directly from Kafka.
- Parses incoming JSON messages into a structured format.
- Converts timestamps for accurate windowing operations.
- Applies sliding window aggregation (e.g., 1-minute window, 30-second slide) to calculate:
  - Sum of vehicle counts
  - Average speed within each window
- Builds robust alert logic based on aggregated metrics.

Important Code Snippet:

```
# windowed aggregations per street
agg = (parsed
      .groupBy(window(col("event_time"), w_dur, s_dur),
col("street_name"))
      .agg(_sum("vehicle_count").alias("vehicles_per_window"),
          _avg("avg_speed").alias("avg_speed"))
      .select(col("window").start.alias("window_start"),
            col("window").end.alias("window_end"),
            col("street_name"),
            col("vehicles_per_window"),
            col("avg_speed"))
)
```

Alert Logic: The Core of Congestion Detection



Multi-Factor Congestion Assessment:

Our alert system uses a combination of average speed and vehicle count within a given window to accurately determine congestion levels.

Alert Level Generation:

- Assigns a congestion level: HIGH, MEDIUM, or LOW.
- **HIGH:** Triggered by very low speeds or extremely high vehicle counts.
- **MEDIUM:** Triggered by moderately low speeds or high vehicle counts.
- **LOW:** Normal traffic conditions.
- **is_alert:** A boolean flag set to True when thresholds are exceeded.

Important Code Snippet:

```
# add alert level
agg2 = (agg
        .withColumn("alert_level",
                    when(col("vehicles_per_window") > thr_high,
"HIGH")
                    .when(col("vehicles_per_window") > thr_med,
"MEDIUM")
                    .otherwise("LOW"))
        .withColumn("is_alert", (col("vehicles_per_window") >
thr_med).cast("boolean"))
)
```

Real-Time Outputs & Automation

Kafka Output: Traffic Alerts Topic

Real-time JSON alert objects are published to a dedicated Kafka topic (e.g., `traffic-alerts`), including street name, alert level, and window details.

Example Alert JSON:

```
{
  "street_name": "El-Tahrir",
  "vehicles_per_window": 92,
  "avg_speed": 18.4,
  "alert_level": "HIGH",
  "is_alert": true
}
```

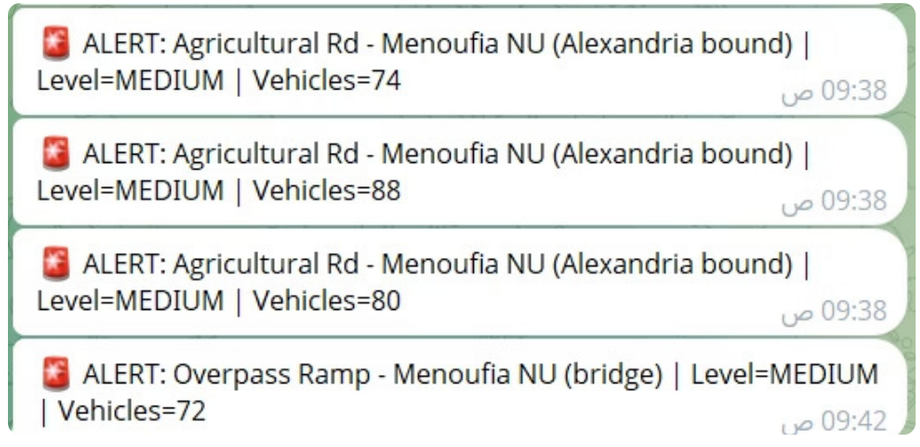
Module 3: Telegram Alert Notifier

Consumes alerts from Kafka and sends formatted messages to a Telegram channel for immediate stakeholder notification and emergency response.

Example Telegram Message:

🚨 ALERT: Overpass Ramp - Menoufia NU (bridge) | Level=MEDIUM | Vehicles=72

🚨 ALERT: Agricultural Rd - Menoufia NU (Alexandria bound) | Level=MEDIUM | Vehicles=80



System Configuration

The system uses a unified YAML configuration file (config.yaml) that centralizes all component settings. Here are the most critical configuration parameters explained:

Kafka Connection & Topics:

```
kafka:
  bootstrap_servers: "localhost:9092"
  input_topic: "traffic-stream"
  alert_topic: "traffic-alerts"
```

Explanation: Defines the Kafka broker address and two essential topics - `traffic-stream` receives raw sensor data, while `traffic-alerts` publishes processed congestion alerts.

Windowing & Aggregation:

```
streaming:
  window_duration: "1 minute"
  slide_duration: "30 seconds"
```

Explanation: Creates overlapping 1-minute windows that slide every 30 seconds, enabling continuous real-time analysis without gaps in traffic monitoring.

Congestion Thresholds:

```
streaming:
  vehicle_threshold_moderate: 60
  vehicle_threshold_high: 90
```

Explanation: Defines alert triggers - 60+ vehicles per window generates MEDIUM alerts, 90+ triggers HIGH alerts for critical congestion.

MQTT Broker Configuration:

```
mqtt:
  host: "broker.hivemq.com"
  port: 1883
  traffic_topic: "traffic/data/menofia_national_university"
```

Explanation: Connects to HiveMQ public broker for IoT communication. The traffic topic receives sensor data, while lighting commands use the prefix defined below.

Lighting Control Topics:

```
lighting_control:
  lighting_topic_prefix: "lighting"
```

Explanation: Sets the base topic for actuator commands. Full topics follow the pattern `lighting/<street_name>/set` for street-specific control.

Telegram Integration:

```
notifier:
  enable_telegram: true
  telegram_bot_token: "8357678603:AAG7Q3jLfiknkDJhNUOlxJ0j5z1FrPIk__g"
  telegram_chat_id: "8461510162"
```

Explanation: Enables instant notifications via Telegram bot. The bot token authenticates with Telegram API, and chat_id specifies the recipient channel or user.

Data Persistence:

```
paths:
  parquet_output: "./output/alerts_parquet"
  checkpoint_dir: "./checkpoint/spark"
```

Explanation: Stores processed alerts in Parquet format for historical analysis. Checkpoint directory enables Spark to recover from failures and maintain exactly-once processing semantics.

Module 4: IoT Lighting Control & actuator System :

System Overview:

The lighting control system consists of two components: a **Controller** that detects congestion and publishes alerts, and an **Actuator** that receives commands and physically adjusts street lights.

Controller Code (Alert Detection):

```
# Detect congestion and publish alerts
agg2 = (agg
    .withColumn("alert_level",
        when(col("avg_speed") < 15, "HIGH")
        .when(col("avg_speed") < 25, "MEDIUM")
        .when(col("vehicles_per_window") > 90, "HIGH")
        .when(col("vehicles_per_window") > 60, "MEDIUM")
        .otherwise("LOW"))
    .withColumn("is_alert",
        ((col("avg_speed") < 25) |
         (col("vehicles_per_window") > 60)).cast("boolean")))

# Publish to Kafka
alerts_query = (agg2
    .filter(col("is_alert") == True)
    .writeStream.format("kafka")
    .option("topic", "traffic-alerts")
    .start())
```

Function: Analyzes traffic data in real-time, detects congestion using speed and vehicle count thresholds, and publishes alerts to Kafka.

Actuator Code (Light Control):

```
def on_message(client, userdata, msg):
    payload = msg.payload.decode()
    try:
        data = json.loads(payload)
    except:
        data = {"raw": payload}
    print("Actuator received:", data)
    # update stored state
    current = {}
    if STATE_FILE.exists():
        current = json.loads(STATE_FILE.read_text())
    street = data.get("street","unknown")
    curr_level = data.get("level","LOW")
    current[street] = curr_level
    save_state(current)


client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect("localhost", 1883, 60)
client.loop_forever()
```

Function: Listens to MQTT lighting commands, maps alert levels to brightness (HIGH=100%, MEDIUM=75%, LOW=50%), and sends PWM signals to control physical street lights.

Complete Workflow:

Traffic Data → Controller (Spark) → Kafka Alerts → Actuator (MQTT) → Street Lights Adjust

Project Summary & Future Vision

Milestone 3 Accomplishments:

End-to-End Real-Time Pipeline

Successfully established a functional pipeline from IoT data ingestion to actionable alerts and automated responses.

Intelligent Congestion Detection

Implemented multi-factor alert logic for accurate identification of traffic bottlenecks.

Dynamic IoT Lighting Control

Demonstrated adaptive street lighting based on real-time traffic conditions.

Future Work:

Integrate Deep Learning

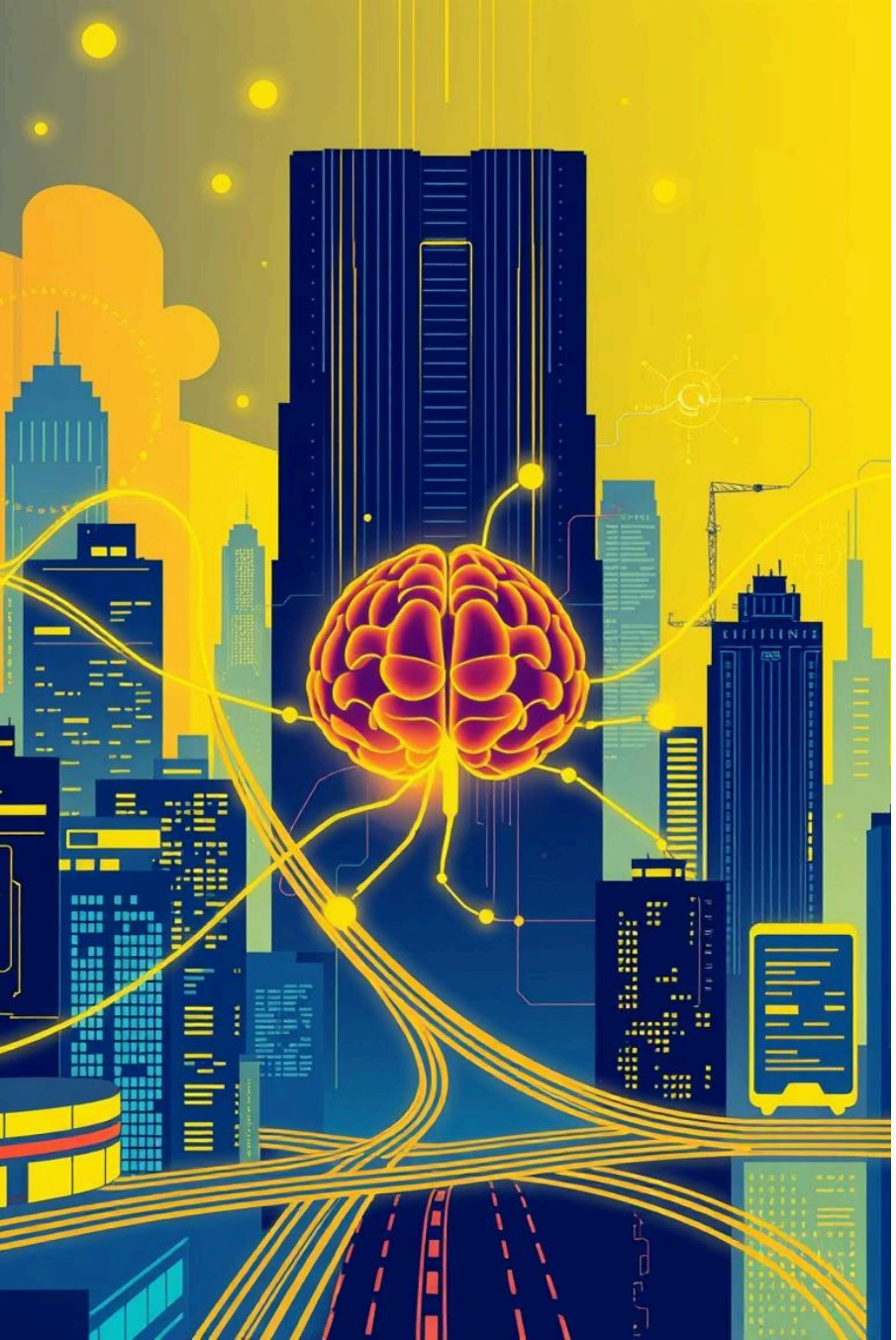
Develop predictive models for proactive congestion forecasting.

Build Comprehensive Dashboard

Create a user-friendly interface for monitoring and control.

ML-Based Congestion Modeling

Enhance predictive capabilities with advanced machine learning techniques.



Real-Time ML Predictions for Smart Traffic & Energy

Unveiling an advanced machine learning system for smart, automated predictions.

Key Code Snippets & Outputs from Advanced Model

Let's dive into the most critical parts of the training script and see what they produce.

Data Preprocessing & Feature Engineering

```
# Time-based features
df['hour'] = df['timestamp'].dt.hour
df['weekday'] = df['timestamp'].dt.weekday
df['is_weekend'] = df['weekday'].isin([5,6]).astype(int)

# Lag features for time series
LAGS = [1,2,3]
for lag in LAGS:
    df[f'veh_count_lag_{lag}'] = df.groupby('street_name')
['vehicle_count'].shift(lag)

# Rolling window statistics
df['veh_roll_3'] = df.groupby('street_name')
['vehicle_count'].rolling(window=3,
min_periods=1).mean().reset_index(level=0, drop=True)
```

✅ Loaded 8640 rows — columns: [timestamp, street_name, vehicle_count, vehicle_speed, solar_energy_level, lighting_demand, ...]

Model Training with RandomizedSearchCV

```
reg_search = RandomizedSearchCV(
    estimator=lgb_reg,
    param_distributions=reg_param_dist,
    n_iter=5,
    cv=tscv,  # TimeSeriesSplit
    scoring='neg_mean_absolute_error',
    random_state=RANDOM_STATE
)
reg_search.fit(X_train, y_train_reg)
best_reg = reg_search.best_estimator_
```

Best regressor params: {'subsample': 1.0, 'num_leaves': 63, 'n_estimators': 200, 'min_child_samples': 5, 'learning_rate': 0.1, 'colsample_bytree': 0.8}

Model Evaluation & Comparison

```
# Baseline persistence model
y_pred_baseline = X_test[f"veh_count_lag_{LAGS[0]}"]
baseline_mae = mean_absolute_error(y_test_reg, y_pred_baseline)

# Advanced model predictions
y_pred_lgb_reg = best_reg.predict(X_test)
lgb_reg_mae = mean_absolute_error(y_test_reg, y_pred_lgb_reg)
lgb_reg_rmse = rmse(y_test_reg, y_pred_lgb_reg)
```

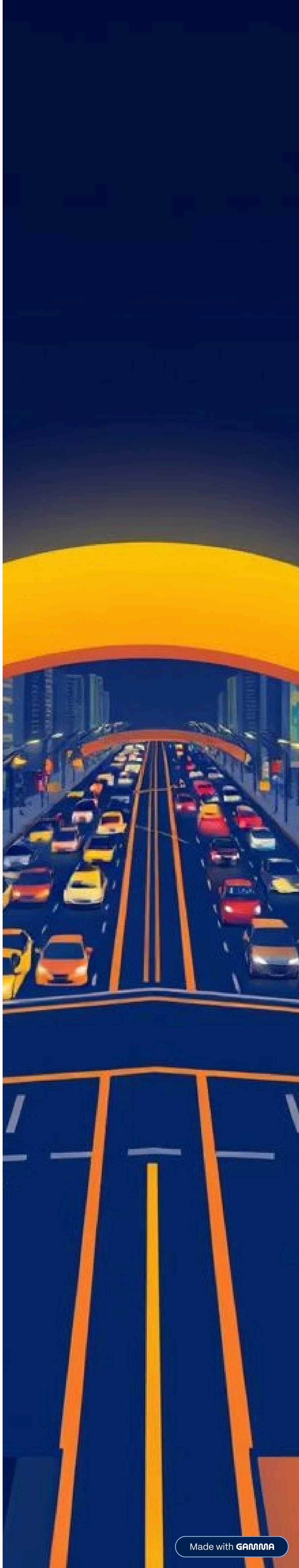
Baseline persistence MAE: 12.456
LGB Regressor — MAE: 3.821, RMSE: 5.234
🔔 Final chosen regressor: LightGBM Regressor

Feature Importance Analysis

```
perm_reg = permutation_importance(
    chosen_reg, X_test, y_test_reg,
    n_repeats=20, random_state=RANDOM_STATE
)
imp_reg = pd.Series(
    perm_reg.importances_mean,
    index=FEATURES
).sort_values(ascending=False)
```

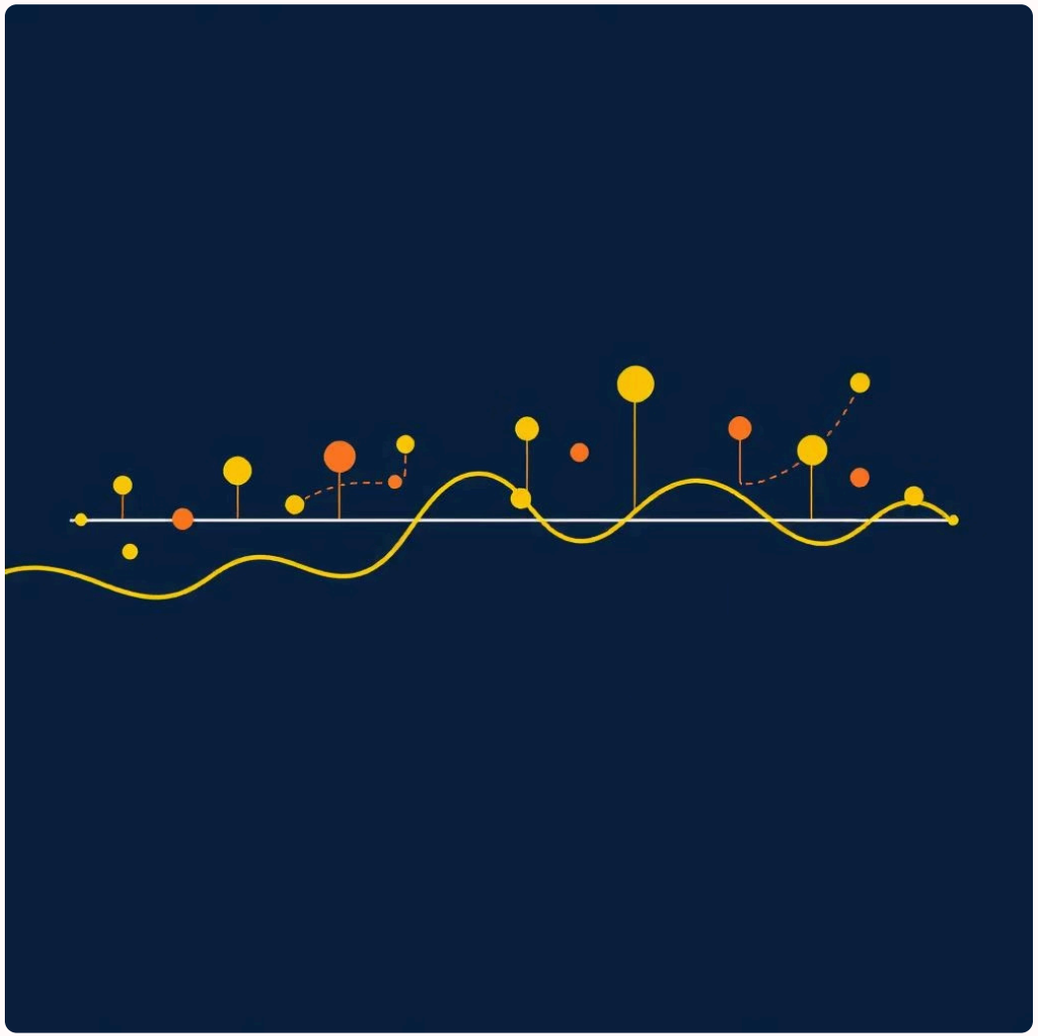
Top features: veh_count_lag_1 (0.342), vehicle_count (0.289), veh_roll_3 (0.156), hour (0.098), vehicle_speed (0.067)

All models, encoders, plots, and evaluation CSV saved to milestone4_outputs_advanced/ directory



Dataset & Preprocessing: Fueling Our Models

The model leverages `traffic_data.csv`, a time-series sensor dataset, with critical preprocessing steps.



- **Sort Data:** By `street_name + timestamp` for chronological integrity.
- **Time Features:** Extracting `hour`, `weekday`, and `is_weekend`.
- **Lag Features:** Creating `veh_count_lag_1`, `veh_count_lag_2`, `veh_count_lag_3` to capture past behavior.
- **Rolling Mean:** Incorporating `veh_roll_3` for trend analysis.
- **Label Encoding:** Converting categorical features like `street_name`, `light_level`, `weather`, and `traffic_light` into numerical representations.
- **Target Generation:** Defining `vehicle_count_next` for regression and encoding `lighting_demand` for classification.

Model Training Script: `milestone4_model_advanced.py`

This script orchestrates the entire model training process, ensuring robust and optimized predictive capabilities.

Data Loading & Feature Engineering

Ingests raw data and transforms it into a rich feature set suitable for time-series analysis.

Time-Based Data Split

Ensures no data leakage by splitting training/testing sets chronologically.

Regressor & Classifier Training

Trains RandomForest (and optionally LightGBM) for both regression and classification tasks.

Optional Hyperparameter Tuning

Utilizes `RandomizedSearchCV` for efficient and safe model optimization.

Model & Encoder Saving

Persists trained models (`.pkl`) and encoders for consistent deployment.

Integrated Evaluation & Inference

Generates evaluation CSVs, feature importance plots, and provides an inference helper.

Streaming Prediction Script:

milestone4_streaming_predictor.py

This script forms the backbone of our real-time prediction engine, connecting to Kafka for seamless data flow.

Kafka Topic Connections:

Purpose	Topic
Live incoming data	traffic-stream
Model predictions	traffic-predictions

ML Feature Set:

['vehicle_count', 'vehicle_speed',
'solar_energy_level', 'hour', 'weekday',
'is_weekend', 'veh_roll_3',
'veh_count_lag_1', 'veh_count_lag_2',
'veh_count_lag_3', 'street_name_enc',
'light_level_enc', 'weather_enc',
'traffic_light_enc']

Consistency is critical for accurate predictions.

Pipeline Flow:

Consumer Reads JSON

Ingests live sensor data from Kafka.

Feature Set Conversion

Transforms JSON into a DataFrame with required features.

Loads Pre-trained Models

Initializes regressor and classifier for inference.

Generates Predictions

Predicts next vehicle count and lighting demand.

Sends Output to Kafka

Publishes predictions as JSON to the output topic.

Prediction Outputs & Evaluation

Our system delivers precise numerical and categorical predictions, validated through comprehensive evaluation metrics.

Model Outputs:

Regression Output

`predicted_vehicle_count_next` : **float**

Classification Output

`predicted_lighting_demand` : **"Low" / "Medium" / "High"**

Example Combined Output:

```
{
  "timestamp": "2025-03-21T12:30:02",
  "street_name": "Street A",
  "predicted_vehicle_count_next": 32.75,
  "predicted_lighting_demand": "High"
}
```

Evaluation Metrics:

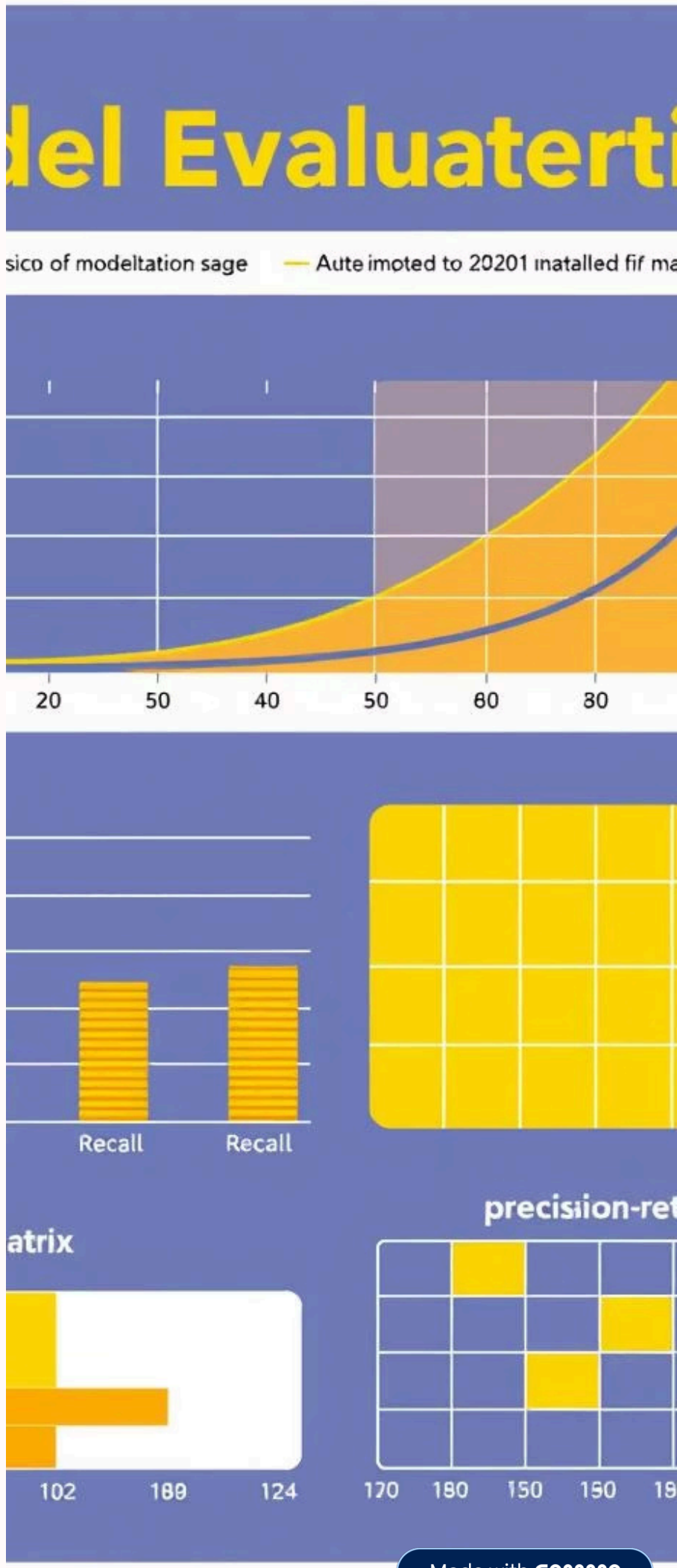
Regression

- Mean Absolute Error (MAE)
- Root Mean Squared Error (RMSE)
- Comparison with persistence baseline (lag_1)

Classification

- Accuracy
- Classification Report (precision, recall, f1-score)
- Confusion Matrix

LightGBM is chosen only if it significantly outperforms RandomForest; otherwise, RandomForest remains the selected model for deployment.

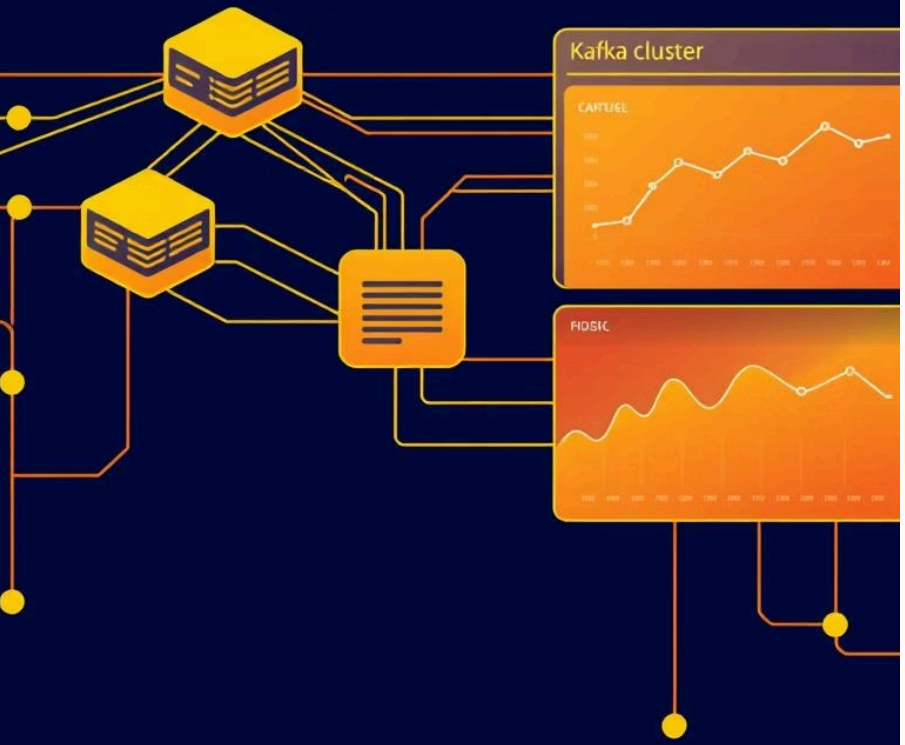


Saved Outputs Directory Structure

All generated artifacts from training and evaluation are meticulously organized within the `milestone4_outputs_advanced/` directory.

```
milestone4_outputs_advanced/
├── models/
│   ├── chosen_regressor.pkl
│   ├── chosen_classifier.pkl
│   ├── label_encoder_lighting.pkl
│   └── label_encoder_street_name.pkl (and others)
├── plots/
│   ├── feature_importance_reg.png
│   ├── feature_importance_clf.png
│   └── evaluation_predictions.csv
```

This structured output ensures easy access to models, encoders, and evaluation results for deployment and analysis.



Real-Time System Architecture: The Data Flow

A high-level overview of our real-time prediction system, from sensor data ingestion to dashboard integration.

Sensors

traffic-stream

Predictor

Predictions

This architecture provides a scalable and efficient pipeline for real-time smart traffic and energy management.

How to Run the System: Step-by-Step Guide

Follow these steps to activate the real-time prediction system and observe its capabilities.

Run Kafka

Initialize ZooKeeper and Kafka servers using their respective configuration files.

```
zookeeper-server-start.bat
config/zookeeper.properties
kafka-server-start.bat
config/server.properties
```

Start the Streaming Predictor

Execute the Python script for the real-time prediction service.

```
python
milestone4_streaming_predictor.py
```

Send Live Messages

Use a traffic generator or the Kafka console producer to send sample JSON data.

```
kafka-console-producer --topic traffic-stream --bootstrap-server localhost:9092
```


```
{"vehicle_count":28,"vehicle_speed":21.5,
"solar_energy_level":0.75,"street_name_enc":1,
"light_level_enc":0,"weather_enc":2,
"traffic_light_enc":1,"veh_roll_3":25,
"veh_count_lag_1":27,"veh_count_lag_2":29,
"veh_count_lag_3":30,"hour":14,"weekday":3,
"is_weekend":0}
```

Key Improvements & Final Deliverables

Milestone 4 significantly advances our system with critical enhancements and comprehensive outputs.

Key Improvements:

- Safe hyperparameter tuning (faster + Windows compatible)
- Time-based data splits for robust evaluation
- Real-time ready ML inference
- Advanced feature engineering for time-series data
- Fully modular + production-ready Kafka pipeline
- Complete outputs (models, plots, CSV, logs)

Final Deliverables:

Trained ML Models

Ready for deployment.

Streaming Predictor Code

Operational Kafka pipeline.

Feature Importance Reports

Insightful model analysis.

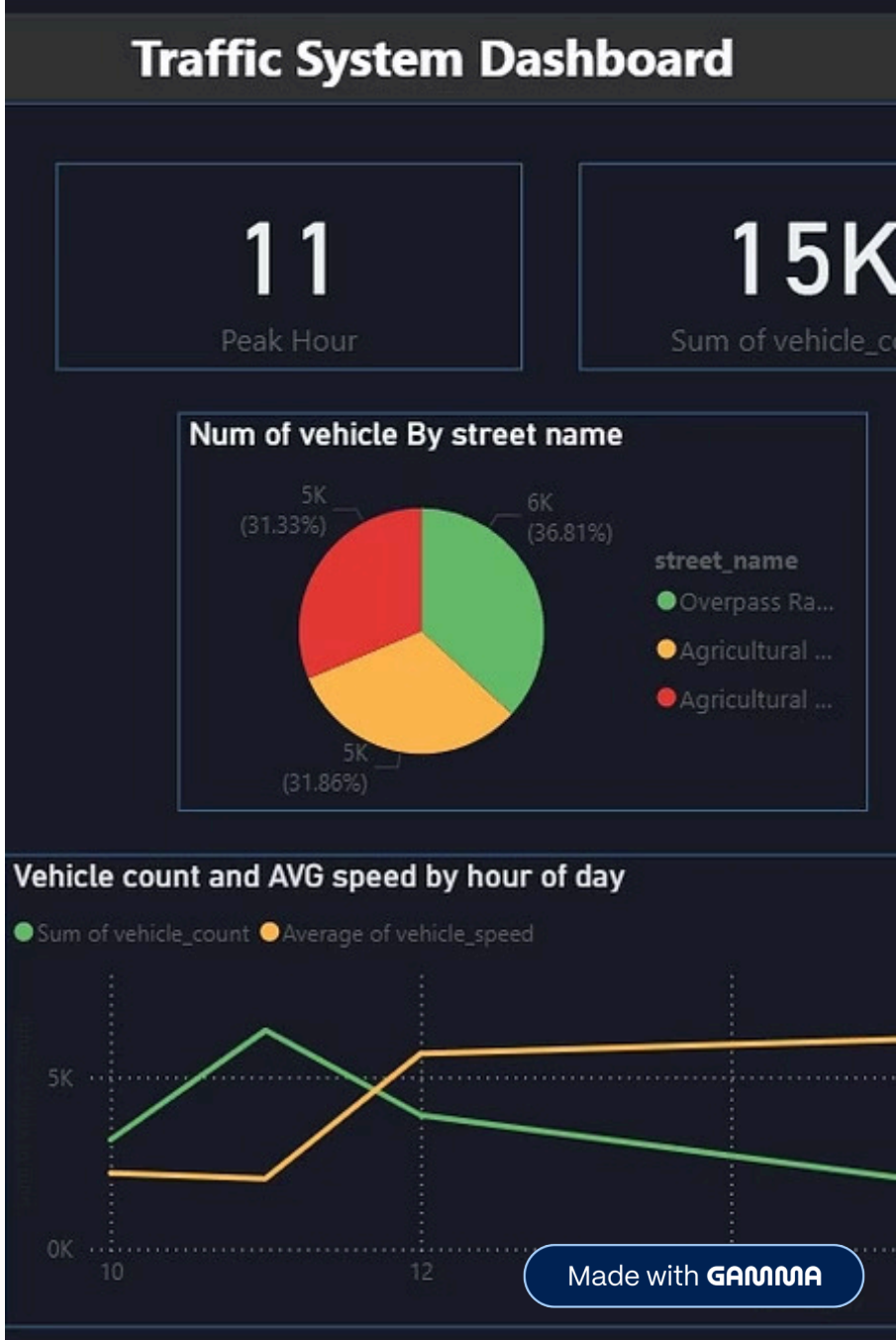
Evaluation CSV & JSON Output

Comprehensive performance data.

This milestone delivers a complete AI-powered real-time prediction pipeline, forming the "brain" of our Smart Traffic + Energy System, ready for integration with SSIS, Docker, and dashboards.

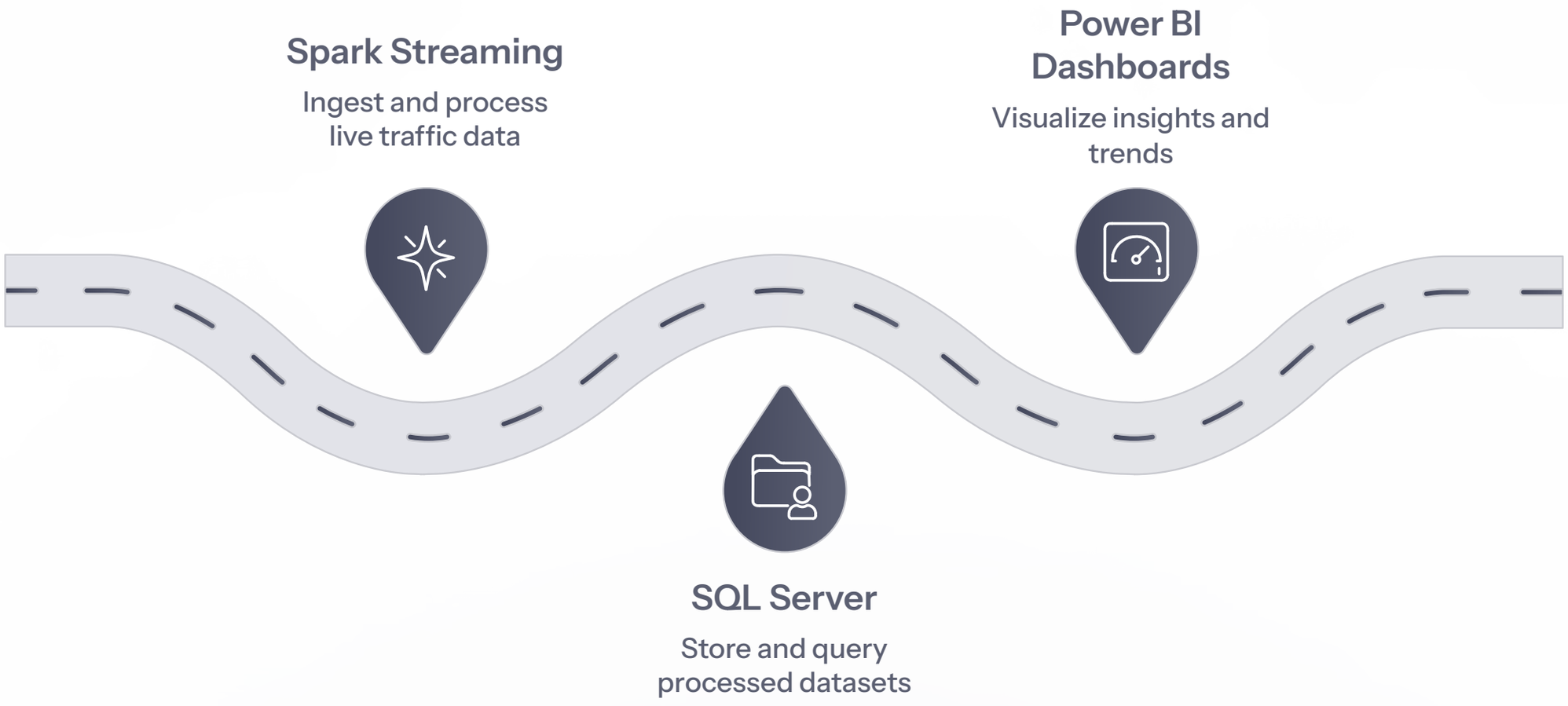
Smart Traffic Management System

Featuring three key dashboards: Traffic Overview, Congestion Analysis, and Energy & Lighting Management.



System Context: Data Flow Architecture

Our dashboards visualize processed traffic data originating from robust streaming pipelines, providing timely insights into complex traffic patterns.



This architecture ensures a seamless flow from raw data ingestion to interactive data visualization, supporting real-time decision-making.

Dashboard 1: Traffic Overview

This dashboard provides a holistic view of daily traffic behavior, highlighting real-time changes and key performance indicators.

Total Vehicles

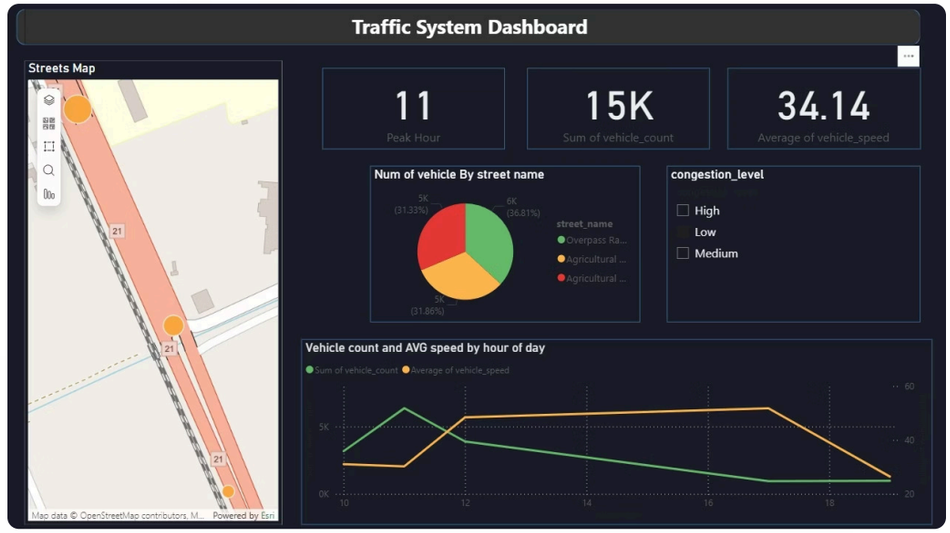
25,000+ per day

Average Speed

45 km/h (Real-time)

Peak Hours

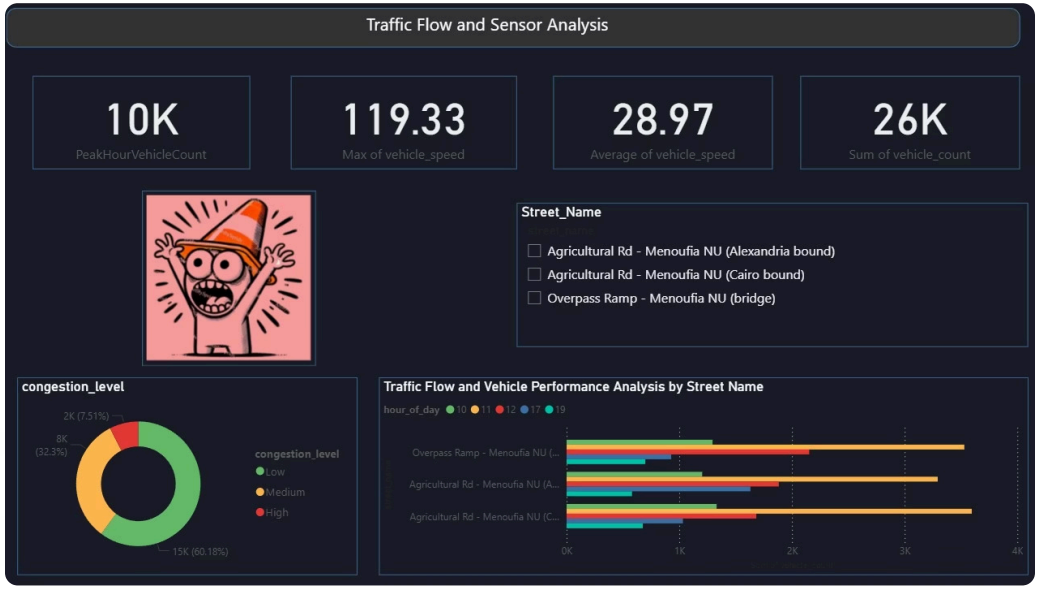
7-9 AM, 4-6 PM



All values update continuously as data flows through the Kafka, Spark, and SQL pipeline.

Dashboard 2: Congestion Analysis

Understand traffic bottlenecks and their severity with detailed heatmaps and comparative insights.



Heatmap by Street & Hour

Visualize congestion patterns across different routes and times.

Severity Indicators

Green: Low | Yellow: Medium | Red: High

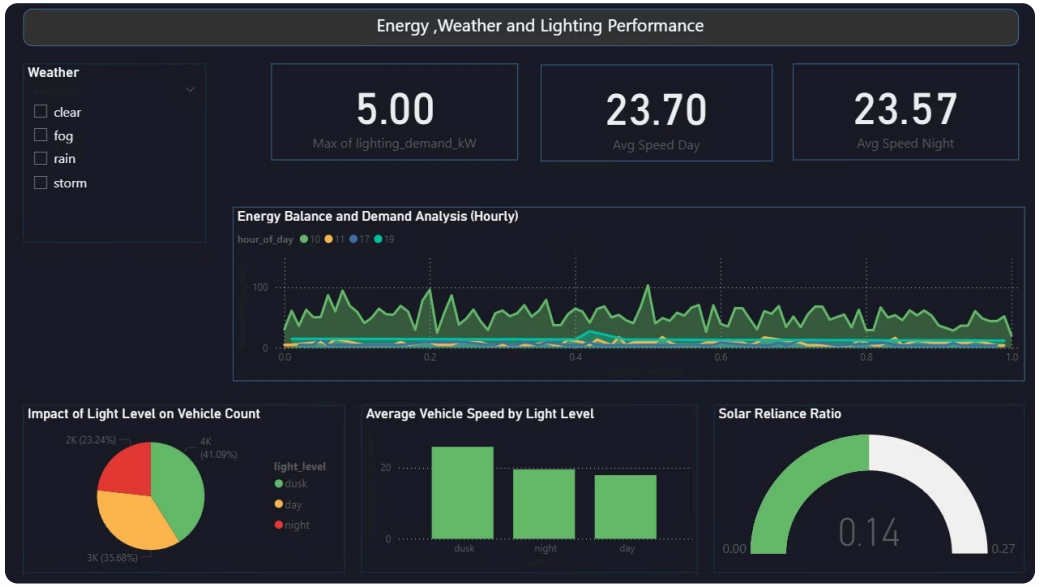
Peak vs. Off-Peak

Compare congestion levels during critical periods.

Congestion levels are computed dynamically based on vehicle count and average speed, offering granular insights.

Dashboard 3: Energy & Lighting Management

Monitor smart street lighting, tracking solar energy levels and demand to optimize energy consumption.



Solar Energy Trend

Track energy harvesting patterns.

Lighting Demand

Observe consumption relative to production.

Energy Alert Spikes

Identify unusual energy fluctuations.

This dashboard ensures street lighting adjusts efficiently, leveraging available solar energy and environmental conditions.

Stakeholder Value: Key Outcomes

Enhanced Traffic Insights

Gain comprehensive understanding of traffic patterns and congestion for optimized urban mobility.

Optimized Energy & Lighting

Achieve efficient street lighting management and energy usage through smart controls.

Proactive Anomaly Detection

Receive immediate alerts for critical events and unusual patterns across all urban systems.

These dashboards empower stakeholders with actionable insights for smarter urban mobility and resource management.

Future Enhancements

Our commitment to continuous improvement drives future developments aimed at increasing the system's predictive power and user engagement, specifically focusing on enhancing our three main dashboards.

Enhanced Traffic Prediction

Integrate advanced machine learning models to forecast traffic patterns and predict congestion, providing proactive insights for urban mobility dashboard.

Advanced Congestion Algorithms

Develop sophisticated algorithms to analyze and manage traffic flow, optimizing routes and reducing gridlock within the urban mobility dashboard.

Smart Energy Optimization

Implement intelligent controls and predictive analytics for energy consumption, enhancing the efficiency and sustainability features of the energy dashboard.

These improvements will further refine our analytics capabilities, providing even more comprehensive tools for urban planning and more actionable insights across all dashboards.

Thank You