Internal Data & Modeling Readiness Report (IR-05)

Date: 2025-10-25
Author: Xiel (THAT Le Quang)
Contact: fxlqthat@gmail.com / +84 33 863 6369
GitHub: thatlq1812
Scope: Complete data structure, collection pipeline, preprocessing, model inventory, and deployment guide for Academic v4.0 traffic forecasting system.

---

1. Data Structure & Schema

1.1 Database Schema (Postgres/Timescale)

```sql
-- Network topology (static metadata)
CREATE TABLE nodes (
    node_id TEXT PRIMARY KEY,            -- Format: 'node-{lat}-{lon}'
    lat DOUBLE PRECISION,                -- Latitude (-90 to 90)
    lon DOUBLE PRECISION,                -- Longitude (-180 to 180)
    road_name TEXT,                      -- Street name(s)
    road_type TEXT,                      -- motorway, trunk, primary, etc.
    attributes JSONB                     -- Additional OSM attributes
);

-- Model predictions history
CREATE TABLE forecasts (
    id SERIAL PRIMARY KEY,
    node_id TEXT REFERENCES nodes(node_id),
    ts_generated TIMESTAMPTZ DEFAULT NOW(),
    horizon_min INT,                     -- 5, 15, 30, or 60 minutes
    speed_kmh_pred FLOAT,                -- Predicted speed
    flow_pred INT,                       -- Predicted vehicle count
    congestion_pred INT,                 -- Congestion level 0-3
    model_name TEXT,                     -- Model identifier
    meta JSONB                           -- Confidence, features, etc.
);
CREATE INDEX idx_forecasts_node_ts ON forecasts(node_id, ts_generated);

-- External events (optional)
CREATE TABLE events (
    event_id TEXT PRIMARY KEY,
    title TEXT,
    category TEXT,                       -- concert, sports, construction, etc.
    start_time TIMESTAMPTZ,
    end_time TIMESTAMPTZ,
    venue_lat DOUBLE PRECISION,
    venue_lon DOUBLE PRECISION,
    source TEXT,
    metadata JSONB
);
CREATE INDEX idx_events_venue ON events USING GIST (point(venue_lon, venue_lat));

-- Ingestion log
CREATE TABLE ingest_log (
```

```sql
    id SERIAL PRIMARY KEY,
    source TEXT,                        -- overpass, google, open_meteo
    ts_ingested TIMESTAMPTZ DEFAULT NOW(),
    records_count INT,
    status TEXT                         -- success, partial, failed
);
```

1.2 Traffic History Cache (SQLite)

Database: `data/traffic_history.db`

```sql
CREATE TABLE traffic_snapshots (
    timestamp TEXT NOT NULL,            -- ISO 8601 timestamp
    node_id TEXT NOT NULL,              -- Node identifier
    avg_speed_kmh REAL,                 -- Average speed
    congestion_level INTEGER,           -- 0=free, 1=slow, 2=heavy, 3=jammed
    temperature_c REAL,                 -- Temperature in Celsius
    rain_mm REAL,                       -- Rainfall in mm
    wind_speed_kmh REAL,                -- Wind speed
    data_json TEXT NOT NULL,            -- Full JSON payload
    PRIMARY KEY (timestamp, node_id)
);
CREATE INDEX idx_timestamp ON traffic_snapshots(timestamp DESC);
CREATE INDEX idx_node_time ON traffic_snapshots(node_id, timestamp DESC);
```

Purpose: Enables lag feature computation (5/15/30/60 min lookbacks) without re-collecting data.
Retention: 7 days (configurable), cleaned via `TrafficHistoryStore.cleanup_old_data()`.

1.3 Validation Schemas (Pydantic)

TrafficNode — Intersection metadata:

```
{
    "node_id": "node-10.7688-106.7033",  # Unique ID
    "lat": 10.7688,                      # -90 to 90
    "lon": 106.7033,                     # -180 to 180
    "degree": 6,                         #   6 for major intersections
    "importance_score": 45.2,            #   40 for v4.0 config
    "road_type": "primary",              # motorway/trunk/primary
    "connected_road_types": ["primary", "trunk"],
    "street_names": ["Nguyen Hue", "Le Loi"],
    "way_ids": [123456, 789012]
}
```

TrafficSnapshot — Collected traffic data:

```
{
    "node_id": "node-10.7688-106.7033",
    "timestamp": "2025-10-25T10:30:00",
    "avg_speed_kmh": 32.5,               # 0-200 km/h
    "sample_count": 15,                  # Number of samples
    "congestion_level": 1,               # 0-3 scale
    "reliability": 0.95                  # 0-1 confidence
}
```

NodeFeatures — ML-ready feature vector (~60 columns):

```
{
    "node_id": "node-10.7688-106.7033",
    "timestamp": "2025-10-25T10:30:00",

    # Current state
    "avg_speed_kmh": 32.5,
    "congestion_level": 1,

    # Weather current
    "temperature_c": 28.5,
    "rain_mm": 0.0,
    "wind_speed_kmh": 12.3,

    # Weather forecasts (4 horizons)
    "forecast_temp_t5_c": 28.6, "forecast_temp_t15_c": 28.7, ...
    "forecast_rain_t5_mm": 0.0, "forecast_rain_t15_mm": 0.2, ...
    "forecast_wind_t5_kmh": 12.5, "forecast_wind_t15_kmh": 13.0, ...

    # Lag features (from traffic history DB)
    "speed_lag_5min": 30.2,
    "speed_lag_15min": 28.9,
    "speed_lag_30min": 27.5,
    "speed_lag_60min": 25.0,

    # Rolling statistics (15/30/60 min windows)
    "speed_rolling_mean_15min": 29.5,
    "speed_rolling_std_15min": 3.2,
    "speed_rolling_min_15min": 25.0,
    "speed_rolling_max_15min": 35.0,

    # Speed changes
    "speed_change_5min": 2.3,            # Absolute change
    "speed_pct_change_5min": 7.6,        # Percentage change

    # Temporal features (cyclical encoding)
    "hour_sin": 0.866, "hour_cos": 0.5,
    "day_of_week_sin": 0.0, "day_of_week_cos": 1.0,
    "is_rush_hour": true,
    "is_morning_rush": true,
    "is_evening_rush": false,
    "is_weekend": false,
    "is_holiday": false,

    # Spatial features (from neighbor nodes)
    "neighbor_avg_speed": 28.5,
    "neighbor_min_speed": 20.0,
    "neighbor_max_speed": 35.0,
    "neighbor_std_speed": 5.2,
    "neighbor_congested_count": 2,
    "neighbor_congested_fraction": 0.33
}
```

2. Data Collection Pipeline

2.1 Network Topology Acquisition (Overpass API)

Endpoint: `https://overpass-api.de/api/interpreter`
Query Strategy: Download road network for Ho Chi Minh City area using OpenStreetMap data.

Road Type Filters:

```
highway~"^(motorway|trunk|primary|secondary)$"
```

Retrieves only major roads suitable for traffic analysis.

Node Selection Criteria:

- **Degree Threshold:** 6 connected ways (major intersection)
- **Importance Score:** 40.0 (configurable in `configs/project_config.yaml`)
- **Importance Calculation:**

```
importance = degree * 5.0                          # Base connectivity weight
if 'motorway' in road_types: importance += 10.0   # Highway boost
if 'trunk' in road_types: importance += 8.0       # Arterial road boost
if 'primary' in road_types: importance += 6.0     # Primary road boost
```

Output: 64 high-priority nodes covering central HCMC (District 1, District 3, Binh Thanh, etc.)

2.2 Real-time Traffic Collection (Google Directions API)

Strategy: Mock mode enabled by default (`USE_GOOGLE=false` in environment). Production requires valid API key.

Sampling Protocol (per node):

1. Select k=3 nearest neighbor nodes within 1024m radius
2. For each neighbor pair (origin → destination):
   - Query `routes.distanceMatrix.v2` with `TRAFFIC_AWARE_OPTIMAL` routing
   - Extract `duration` (current travel time with traffic)
   - Extract `staticDuration` (free-flow baseline)
3. Compute congestion level:

```
congestion_ratio = duration / staticDuration
if ratio >= 2.0: congestion_level = 3 (jammed)
elif ratio >= 1.5: congestion_level = 2 (heavy)
elif ratio >= 1.2: congestion_level = 1 (slow)
else: congestion_level = 0 (free)
```

Rate Limits: 300 requests/minute enforced via token bucket. Delays between batches prevent quota exhaustion.

Cost Optimization (Academic v4.0):

- 25 collections/day (adaptive schedule)
- 64 nodes × 3 samples = 192 requests/collection
- **Daily:** 4,800 requests → **$24/day → $720/month**

2.3 Weather Data Collection (Open-Meteo)

Endpoint: `https://api.open-meteo.com/v1/forecast`
Parameters:

- `latitude`, `longitude` (per node)
- `current_weather=true` (instant temperature, wind, rain)
- `hourly=temperature_2m,precipitation,windspeed_10m` (forecast horizons)

Horizons: t+5min, t+15min, t+30min, t+60min (interpolated from hourly forecasts)

Features Extracted:

- `temperature_c` (current)
- `rain_mm` (current precipitation)
- `wind_speed_kmh` (current wind)
- `forecast_temp_t5_c`, `forecast_temp_t15_c`, `forecast_temp_t30_c`, `forecast_temp_t60_c`
- `forecast_rain_t5_mm`, `forecast_rain_t15_mm`, `forecast_rain_t30_mm`, `forecast_rain_t60_mm`
- `forecast_wind_t5_kmh`, `forecast_wind_t15_kmh`, `forecast_wind_t30_kmh`, `forecast_wind_t60_kmh`

Rate Limits: 10,000 requests/day (free tier). Academic v4.0 uses 64 nodes × 25 collections = 1,600 requests/day.

2.4 Adaptive Collection Schedule

Implementation: `traffic_forecast/scheduler/adaptive_scheduler.py`

Time Windows:

- **Peak Hours** (7:00-9:00, 17:00-19:00 weekdays): 30-minute intervals
- **Off-Peak** (9:00-17:00, 19:00-22:00 weekdays): 60-minute intervals
- **Night** (22:00-7:00): 90-minute intervals
- **Weekend**: 60-minute intervals

Daily Collection Count:

- Peak: 4 hours × 2 collections/hour = 8
- Off-Peak: 8 hours × 1 collection/hour = 8
- Night: 9 hours × 0.67 collections/hour 6
- Weekend: 17 hours × 1 collection/hour = 17
- **Weighted Average:** ~25 collections/day

Deployment: Systemd service `traffic-scheduler.service` triggers `collect_and_render.py --once` via cron-like loop.

---

3. Preprocessing & Feature Engineering

3.1 Lag Feature Computation

Purpose: Capture temporal autocorrelation (traffic 5 minutes ago predicts traffic now).

Implementation: `traffic_forecast/features/lag_features.py`

Features Generated:

```
# Direct lags (lookback to historical data)
speed_lag_5min   = snapshot[t - 5min]['avg_speed_kmh']
speed_lag_15min  = snapshot[t - 15min]['avg_speed_kmh']
speed_lag_30min  = snapshot[t - 30min]['avg_speed_kmh']
speed_lag_60min  = snapshot[t - 60min]['avg_speed_kmh']

# Rolling window statistics (15/30/60 min windows)
speed_rolling_mean_15min = mean(speeds[t-15min : t])
speed_rolling_std_15min  = std(speeds[t-15min : t])
```

```
speed_rolling_min_15min  = min(speeds[t-15min : t])
speed_rolling_max_15min  = max(speeds[t-15min : t])

speed_rolling_mean_30min = mean(speeds[t-30min : t])
speed_rolling_std_30min  = std(speeds[t-30min : t])
# ... (repeat for 60min window)

# Speed changes (rate of change)
speed_change_5min      = speed[t] - speed[t-5min]
speed_pct_change_5min  = (speed[t] - speed[t-5min]) / speed[t-5min] * 100
```

Data Source: SQLite `traffic_history.db` query via `TrafficHistoryStore.get_historical_snapshot()`.

3.2 Temporal Feature Encoding

Purpose: Encode cyclical time patterns (hour-of-day, day-of-week) as continuous features.

Implementation: `traffic_forecast/features/temporal_features.py`

Cyclical Encoding (sin/cos transformations):

```
hour = timestamp.hour                    # 0-23
hour_sin = sin(2 *   * hour / 24)
hour_cos = cos(2 *   * hour / 24)

day_of_week = timestamp.weekday()        # 0=Monday, 6=Sunday
day_of_week_sin = sin(2 *   * day_of_week / 7)
day_of_week_cos = cos(2 *   * day_of_week / 7)
```

Rush Hour Flags:

```
is_rush_hour        = (7   hour < 9) OR (17   hour < 19)
is_morning_rush     = (7   hour < 9)
is_evening_rush     = (17   hour < 19)
is_weekend          = day_of_week   5
is_holiday          = lookup(vietnam_holidays, date)  # Optional
```

3.3 Spatial Feature Aggregation

Purpose: Incorporate neighbor node states (upstream congestion propagates downstream).

Implementation: `traffic_forecast/features/spatial_features.py`

Neighbor Selection: Precomputed adjacency graph from Overpass OSM ways (nodes connected by shared road segments).

Aggregation Functions:

```
neighbors = get_connected_nodes(node_id)  # From OSM topology

neighbor_avg_speed        = mean([n.avg_speed_kmh for n in neighbors])
neighbor_min_speed        = min([n.avg_speed_kmh for n in neighbors])
neighbor_max_speed        = max([n.avg_speed_kmh for n in neighbors])
neighbor_std_speed        = std([n.avg_speed_kmh for n in neighbors])
neighbor_congested_count  = count([n for n in neighbors if n.congestion_level   2])
neighbor_congested_frac   = neighbor_congested_count / len(neighbors)
```

3.4 Normalization & Scaling

Method: StandardScaler (zero mean, unit variance)
Scope: All continuous numerical features (~60 columns)

Pipeline:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
joblib.dump(scaler, 'models/feature_scaler.pkl')
```

Saved Artifacts:

- `models/feature_scaler.pkl` (scaler object)
- `models/scaler.npy` (legacy NumPy format)

Usage in Production:

```
scaler = joblib.load('models/feature_scaler.pkl')
features_normalized = scaler.transform(features_raw)
```

---

4. Repository Data Layout & Configuration

4.1 Directory Structure

- `data/` — latest JSON exports, SQLite history, feature CSV outputs.
- `data/parquet/` — partitioned raw timeseries (node_id, ts, avg_speed_kmh, vehicle_count, raw_json).
- `data/processed/` — train/validation splits, scalers, and metadata (`metadata.json`).
- `models/` — serialized estimators (`*.pkl`, `*.keras`), feature scaler, research artifacts.
  - `models/research/` – experimental deep learning artifacts; now includes ASTGCN saves (`*_config.pkl`, `*_adjacency.npy`, `.keras`).
- `cache/` — API cache (Overpass/Open-Meteo/Google) governed by TTLs in `configs/project_config.yaml`.

4.2 Configuration Files

- `configs/project_config.yaml` — governs global area of interest, adaptive scheduler slots, collector options (Overpass, Google Directions, Open-Meteo), feature pipeline toggles, and default model registry entry.
- `.env` (from `.env.template`) — holds secrets for Google API keys, database URLs, and environment-specific overrides.
- `configs/nodes_schema_v2.json` — JSON schema used by validation layer to guarantee node payload shape before storage.

---

5. Collection Workflow & Deployment

5.1 Orchestration

Entry Point: `scripts/collect_and_render.py`

Workflow Steps:

1. **Scheduler** — Adaptive by default (`scheduler.mode=adaptive`), implemented in `traffic_forecast/scheduler/adapt`
   - Peak windows (06:30-07:30, 10:30-11:30, 13:00-13:30, 16:30-19:30) at 30-minute cadence.
   - Off-peak at 60 minutes; weekends optionally throttled to 90 minutes.

   - CLI: `python scripts/collect_and_render.py --adaptive` (or `--print-schedule` for cost preview).
2. **Collectors** — orchestrated via `scripts/collect_and_render.py` / `scripts/collect_with_history.py`:
   - **Overpass** (`traffic_forecast/collectors/overpass/collector.py`) retrieves OSM topology, filtered by `node_selection` config (min_degree 6, min_importance 40, motorway/trunk/primary only).
   - **Google Directions** (`traffic_forecast/collectors/google/collector.py`) samples k=3 nearest edges per node within 1.024 km radius. Development uses mock responses (`use_mock_api: true`); production toggles to real API with cost envelope ~$720/month for 64 nodes.
   - **Open-Meteo** (`traffic_forecast/collectors/open_meteo/collector.py`) enriches current and short-horizon weather features (t+5 to t+60 minutes).
3. **Post-processing** — `traffic_forecast/pipelines` modules construct temporal, lag, and spatial features according to the YAML pipeline definitions. Traffic history DB supports lag queries to avoid recomputation.
4. **Storage & Manifest** — each run writes a manifest under `data_runs/` (configurable via `globals.output_base`) and, when history capture is enabled, appends to `traffic_history.db` plus feature CSV outputs.

5.2 Deployment Platforms

Local / CI:

- Conda environment `dsp` (`environment.yml`)
- Runnable via helper scripts (`scripts/quick_start.sh`)
- Mock API keeps costs at zero

GCP / Cloud:

- **Production setup** uses provided shell scripts to configure a Google Cloud VM
- **Database:** Cloud SQL or TimescaleDB hosts relational layer
- **Storage:** Raw Parquet and model artifacts stored on GCS buckets
- **Scheduling:** Systemd timer, cron, or Cloud Scheduler invoking `collect_and_render.py`
- **Bootstrap:** `scripts/start_collection.sh` provides one-command production startup
- **Health Checks:** `scripts/health_check.sh` monitors system status

Maintenance:

- `scripts/cleanup_runs.py --days 14` prunes historical runs
- `TrafficHistoryStore.cleanup_old_data()` enforces rolling SQLite retention (7 days default)

---

6. Model Portfolio & Metrics

6.1 Feature Pipeline Summary

60+ engineered features organized by category:

Lag Features:

- Direct lags: 5, 15, 30, 60 minute shifts
- Rolling statistics: mean, std, min, max over 15/30/60 min windows
- Speed changes: absolute and percentage rate of change

Temporal Features:

- Cyclical encoding: hour_sin/cos, day_of_week_sin/cos
- Rush hour flags: is_rush_hour, is_morning_rush, is_evening_rush
- Calendar markers: is_weekend, is_holiday

Spatial Features:

- Neighbor aggregations: avg_speed, min_speed, max_speed, std_speed
- Congestion propagation: neighbor_congested_count, neighbor_congested_fraction

Weather Features:

- Current: temperature_c, rain_mm, wind_speed_kmh
- Forecasts: 4 horizons (t+5, t+15, t+30, t+60) for temp/rain/wind

Configuration: All feature groups can be enabled/disabled via `pipelines.preprocess` in `project_config.yaml`.

6.2 Production Models

Model Family	Purpose	Implementation	Latest Metrics
Linear Regression	Baseline	Fast inference; default `pipelines.model.type`	RMSE 8.2 km/h, MAE 6.1 km/h, R^2 0.89
Ridge / Lasso	Regularized baselines	Available via registry	Cross-val RMSE 8.5 ± 0.3
Random Forest / Gradient Boosting	Tree ensembles	Used in stacking ensemble	Contribute to final RMSE 8.2 km/h
XGBoost	High-bias/high-variance component	Weight 0.45 in ensemble	Improves non-linear capture
Stacking Ensemble	Production best	Combines XGBoost, RF, GB	RMSE 8.2 km/h, MAPE 12.5%
LSTM (attention)	Deep temporal model	`traffic_forecast/models/lstm_model.py`; 12-timestep window	RMSE ~8.2-8.5 km/h on validation

6.3 LSTM Architecture Details

Purpose: Capture long-term temporal dependencies in traffic patterns.

Implementation: `traffic_forecast/models/lstm_model.py`

Architecture:

```
sequence_length = 12  # 12 timesteps (60 minutes with 5-min sampling)

Input Layer: (batch_size, 12, num_features)
    ↓
LSTM Layer 1: 128 units, return_sequences=True
    ↓
Dropout: 0.2
    ↓
LSTM Layer 2: 64 units, return_sequences=False
    ↓
Dropout: 0.2
    ↓
Dense Layer: 32 units, ReLU activation
    ↓
Output Layer: 1 unit (speed prediction)
```

Training Configuration:

- Optimizer: Adam (lr=0.001)
- Loss: Mean Squared Error (MSE)
- Batch size: 32
- Epochs: 50 (early stopping on validation loss, patience=5)

- Validation split: 20%

Artifacts:

- `traffic_forecast/models/lstm_v2.keras` (full model)
- `traffic_forecast/models/lstm_v2.h5` (legacy format)
- `traffic_forecast/models/scaler.npy` (feature scaler for preprocessing)

6.4 ASTGCN (Research Model)

Full Name: Attention-based Spatial-Temporal Graph Convolutional Network

Purpose: Experimental model for capturing both spatial dependencies (via graph structure) and temporal patterns (via multi-component architecture).

Implementation: `traffic_forecast/models/research/astgcn.py`

Architecture Components:

1. Multi-Component Design:

```
# Recent Component (short-term patterns)
recent_window = 12       # Last 12 timesteps (1 hour with 5-min sampling)

# Daily Component (daily periodicity)
daily_window = 288       # 24 hours × 12 samples/hour = 288 timesteps

# Weekly Component (weekly patterns)
weekly_window = 2016     # 7 days × 288 samples/day = 2016 timesteps
```

2. Core Building Blocks:

Temporal Attention:

```
# Self-attention over time dimension
Q = W_Q @ X  # Query
K = W_K @ X  # Key
V = W_V @ X  # Value

attention_scores = softmax(Q @ K^T / sqrt(d_k))
output = attention_scores @ V
```

Spatial Attention:

```
# Graph-based attention using adjacency matrix
S = softmax(X @ X^T)          # Spatial similarity
S_attended = S   Adjacency    # Masked by road network topology
output = S_attended @ X
```

Chebyshev Graph Convolution:

```
# K-order Chebyshev polynomials on graph Laplacian
L_norm = 2 * L /  _max - I    # Normalized Laplacian
T_0 = I
T_1 = L_norm
T_k = 2 * L_norm @ T_{k-1} - T_{k-2}

output = Σ( _k * T_k @ X)     # Weighted sum over K orders
```

3. ASTGCN Block:

```
Input: (batch, timesteps, nodes, features)
    ↓
Temporal Attention → (batch, timesteps, nodes, features')
    ↓
Spatial Attention → (batch, timesteps, nodes, features')
    ↓
Chebyshev Graph Conv (order=3) → (batch, timesteps, nodes, features')
    ↓
Residual Connection + LayerNorm
    ↓
Output: (batch, timesteps, nodes, features')
```

4. Component Fusion:

```
# Each component (recent, daily, weekly) produces prediction
recent_pred = ASTGCN_Block(recent_input)
daily_pred = ASTGCN_Block(daily_input)
weekly_pred = ASTGCN_Block(weekly_input)

# Learnable fusion weights
final_pred = w_recent * recent_pred + w_daily * daily_pred + w_weekly * weekly_pred
# where w_recent + w_daily + w_weekly = 1 (softmax-normalized)
```

Configuration:

```
ASTGCNConfig(
    num_nodes=64,                       # Network size
    num_features=60,                    # Feature dimension
    num_blocks=2,                       # Stacked ASTGCN blocks
    chebyshev_k=3,                      # Chebyshev polynomial order
    hidden_dim=64,                      # Hidden layer size
    output_dim=1,                       # Single speed prediction
    recent_window=12,
    daily_window=288,
    weekly_window=2016,
    dropout=0.3
)
```

Training:

- Optimizer: Adam (lr=0.001)
- Loss: MSE + L2 regularization
- Batch size: 16 (memory-intensive due to graph ops)
- Epochs: 100 (early stopping patience=10)

Artifacts:

- `{model_name}_config.pkl` — serialized configuration
- `{model_name}_adjacency.npy` — precomputed adjacency matrix
- `{model_name}.keras` — full model weights

Status: Experimental. Not yet deployed to production. Requires adjacency matrix preprocessing and multi-component data preparation.

6.5 Model Registry & Deployment

Registry: `traffic_forecast/models/registry.py` exposes standardized builder interface:

```python
from traffic_forecast.models import get_model

model = get_model('linear')        # Linear regression baseline
model = get_model('xgboost')       # XGBoost regressor
model = get_model('stacking')      # Stacking ensemble
```

API Service: `traffic_forecast/api/main.py` loads trained models for real-time inference:

```
GET /predict?node_id=node-10.7688-106.7033&horizon=15
→ {"speed_kmh_pred": 32.5, "congestion_pred": 1, "model": "stacking"}
```

Retraining Schedule: Weekly cadence recommended. Model metadata recorded in `models/model_metadata.json` with scaler parity.

MLflow Integration: `traffic_forecast/models/advanced_training.py` supports experiment tracking for hyperparameter tuning and model versioning.

---

7. Pre-Cloud Deployment Checklist

7.1 Infrastructure Provisioning

GCP Resources:

- ☐ Provision Compute Engine VM (e2-standard-2 or n1-standard-2 minimum)
- ☐ Configure Cloud SQL PostgreSQL instance (db-f1-micro for testing, db-n1-standard-1 for production)
- ☐ Create GCS bucket for model artifacts and Parquet storage
- ☐ Set up VPC firewall rules (allow 8000 for API, SSH from trusted IPs only)
- ☐ Enable required APIs: Compute Engine, Cloud SQL, Cloud Storage, Cloud Scheduler

Credentials & Secrets:

- ☐ Generate Google Directions API key (set quota limits: 10,000 requests/day)
- ☐ Store API key in `.env` file (`GOOGLE_API_KEY=...`)
- ☐ Configure Cloud SQL connection string (`DATABASE_URL=postgresql://...`)
- ☐ Set up service account with Storage Admin + Cloud SQL Client roles
- ☐ Download service account JSON key, set `GOOGLE_APPLICATION_CREDENTIALS`

7.2 Database Initialization

Schema Setup:

```
# Connect to Cloud SQL
gcloud sql connect <instance-name> --user=postgres

# Run schema creation
\i infra/sql/schema.sql

# Verify tables
\dt
# Expected: nodes, forecasts, events, ingest_log

# Create indexes (if not in schema.sql)
CREATE INDEX idx_forecasts_node_ts ON forecasts(node_id, ts_generated);
CREATE INDEX idx_events_venue ON events USING GIST (point(venue_lon, venue_lat));
```

Initial Data Load:

```
# Load node topology (from local collection)
python scripts/export_nodes_info.py --output nodes_export.json
```

```
# Import to Cloud SQL
psql $DATABASE_URL -c "COPY nodes FROM 'nodes_export.json' CSV HEADER;"
```

7.3 Systemd Service Configuration

File: `/etc/systemd/system/traffic-scheduler.service`

```
[Unit]
Description=Traffic Data Collection Scheduler
After=network.target

[Service]
Type=simple
User=traffic
WorkingDirectory=/opt/traffic_forecast
ExecStart=/opt/miniconda3/envs/dsp/bin/python scripts/collect_and_render.py --adaptive --no-visualize
Restart=on-failure
RestartSec=30
Environment="PATH=/opt/miniconda3/envs/dsp/bin:/usr/local/bin:/usr/bin"
Environment="GOOGLE_API_KEY=<your-key>"
Environment="DATABASE_URL=postgresql://..."

[Install]
WantedBy=multi-user.target
```

Enable & Start:

```
sudo systemctl daemon-reload
sudo systemctl enable traffic-scheduler.service
sudo systemctl start traffic-scheduler.service
sudo systemctl status traffic-scheduler.service
```

7.4 API Service Deployment

File: `/etc/systemd/system/traffic-api.service`

```
[Unit]
Description=Traffic Forecast API Server
After=network.target

[Service]
Type=simple
User=traffic
WorkingDirectory=/opt/traffic_forecast
ExecStart=/opt/miniconda3/envs/dsp/bin/uvicorn traffic_forecast.api.main:app --host 0.0.0.0 --port 8000
Restart=always
Environment="PATH=/opt/miniconda3/envs/dsp/bin:/usr/local/bin:/usr/bin"
Environment="MODEL_PATH=/opt/traffic_forecast/models/stacking_ensemble.pkl"

[Install]
WantedBy=multi-user.target
```

Test API:

```
curl http://<vm-external-ip>:8000/health
# Expected: {"status": "healthy", "models_loaded": true}
```

```
curl "http://<vm-external-ip>:8000/predict?node_id=node-10.7688-106.7033&horizon=15"
# Expected: {"speed_kmh_pred": 32.5, "congestion_pred": 1, ...}
```

7.5 Monitoring & Logging

CloudWatch Equivalent (Stackdriver):

```
# Install logging agent
curl -sSO https://dl.google.com/cloudagents/add-logging-agent-repo.sh
sudo bash add-logging-agent-repo.sh
sudo apt-get update
sudo apt-get install google-fluentd
```

Log Configuration: `/etc/google-fluentd/config.d/traffic.conf`

```
<source>
  @type tail
  path /opt/traffic_forecast/logs/scheduler.log
  pos_file /var/lib/google-fluentd/pos/scheduler.pos
  tag traffic.scheduler
  format json
</source>
```

Alerts:

- ☐ Set up alert for API response time > 5s
- ☐ Alert on collection failures (3+ consecutive errors)
- ☐ Alert on disk usage > 80%
- ☐ Alert on database connection failures

7.6 Cost Validation

Expected Monthly Costs (Academic v4.0):

Resource	Configuration	Monthly Cost
Compute Engine VM	e2-standard-2	$48
Cloud SQL	db-f1-micro (dev)	$15
Cloud Storage	10 GB standard	$0.20
Google Directions API	4,800 req/day × 30	$720
Network Egress	~5 GB/month	$0.50
Total		**~$784/mo**

Cost Optimization:

- Use preemptible VM instances (saves ~70% on compute)
- Set Google API daily quota to 5,000 requests (budget cap)
- Enable Cloud SQL automatic backups only for production (skip dev)
- Use lifecycle policies on GCS to delete old Parquet files (>30 days)

7.7 Validation Tests

Pre-Launch Checklist:

- ☐ Run `scripts/health_check.sh` — all services green
- ☐ Verify database connectivity: `psql $DATABASE_URL -c "SELECT COUNT(*) FROM nodes;"`
- ☐ Test single collection: `python scripts/collect_and_render.py --once --no-visualize`
- ☐ Validate features: Check `data/processed/features.csv` has ~60 columns

- ☐ Test API prediction: `curl http://localhost:8000/predict?node_id=node-10.7688-106.7033&horizon=15`
- ☐ Check logs: `journalctl -u traffic-scheduler -n 50`
- ☐ Monitor resource usage: `htop` (CPU < 50%, RAM < 4GB)
- ☐ Verify adaptive schedule: `python scripts/collect_and_render.py --print-schedule`

Smoke Test Script:

```bash
#!/bin/bash
# File: scripts/smoke_test.sh

set -e

echo "1. Testing database connection..."
psql $DATABASE_URL -c "SELECT 1;" > /dev/null

echo "2. Running single collection..."
python scripts/collect_and_render.py --once --no-visualize

echo "3. Checking feature output..."
test -f data/processed/features.csv || exit 1

echo "4. Testing API endpoint..."
curl -f http://localhost:8000/health || exit 1

echo "All smoke tests passed!"
```

End of Report
Last Updated: 2025-10-25
Version: IR-05 (Standalone Edition)