Dynamic Parking Pricing System
Developer Documentation

Ayush Singh (AyushLeft)

July 7, 2025

Contents

1 Introduction

This document provides a comprehensive, human-centric explanation of the **Dynamic Parking Pricing System** implemented in Google Colab using *Pathway* for real-time streaming, *Pandas* for data wrangling, and *Bokeh* for live visualizations. The goal is to simulate delayed sensor feeds, compute demand-aware prices for each parking space, and publish continuous pricing predictions that transparently respond to demand and competition.

2 High-Level Architecture

Figure 1 outlines the end-to-end pipeline: a data simulator replays historic CSV rows with artificial delays; Pathway ingests batches in timestamp order, enriches them with on-the-fly features, and passes each record through three pricing models. Results stream both to CSV sinks and to a Bokeh dashboard running inline in Colab.

architecture_diagram.png

Figure 1: End-to-end data-flow of the Dynamic Parking Pricing System.

2.1 Component Breakdown

1. **DataSimulator**: Replays historical dataset rows as if emitted by IoT sensors, injecting Gaussian noise into occupancy and queue length columns.
2. **Pathway UDFs**: Pure Python functions decorated with `@pw.udf` that convert raw fields to engineered features (e.g., vehicle and traffic weights) and compute prices.
3. **Pricing Models**: Three functions encapsulate pricing logic: Baseline Linear, Demand-Based, and Competitive.
4. **VisualizationEngine**: Streams live prices into Bokeh `ColumnDataSource` objects and renders interactive line charts.
5. **DynamicPricingEngine**: Orchestrates loading, preprocessing, pipeline construction, output sinks, and summary analytics.

3 Google Colab Setup

Open a new Colab notebook and run the following cell to install dependencies:

Listing 1: Dependency installation

```
!pip install pathway pandas numpy geopy bokeh seaborn matplotlib
```

Enable Bokeh output inside Colab:

```
from bokeh.io import output_notebook
output_notebook()
```

4 Dataset and Simulation

4.1 Input Schema

Each CSV row represents a snapshot of one parking space. Key columns:

- `SystemCodeNumber`: Identifier of the parking bay.
- `Capacity`: Maximum vehicle count.
- `Occupancy`: Current count of vehicles parked.
- `QueueLength`: Vehicles waiting for a spot.
- `IsSpecialDay`: Binary flag for holidays/events.
- `TrafficConditionNearby`: Categorical traffic indicator (*low*, *average*, *high*, *heavy*).
- Geolocation columns `Latitude` and `Longitude` support proximity calculations.

4.2 Simulator Logic

Listing 2 shows the core of `DataSimulator`. The class sorts records by timestamp, injects noise, and delivers mini-batches to emulate streaming ingestion.

Listing 2: Core streaming simulator

```
class DataSimulator:
    def __init__(self, df):
        self.df = df.sort_values('DateTime').copy()
        self.current_index = 0
        self._add_noise()

    def _add_noise(self):
        np.random.seed(42)
        self.df['Occupancy'] = np.clip(
            self.df['Occupancy'] + np.random.normal(0, 2, len(self.df))
                ,
            0, self.df['Capacity']
        ).astype(int)
        self.df['QueueLength'] = np.clip(
            self.df['QueueLength'] + np.random.normal(0, 1, len(self.df
                )),
            0, 50
        ).astype(int)

    def get_next_batch(self, batch_size: int = 100):
        end = min(self.current_index + batch_size, len(self.df))
        batch = self.df.iloc[self.current_index:end]
        self.current_index = end
        return batch
```

5 Feature Engineering

Pathway enriches each row with derived metrics: $\text{occupancy\_rate} = \text{Occupancy}_{\overline{Capacityvehicle\_weight=w_v(VehicleT}}$ where q is normalized queue length and S indicates a special day.

6 Pricing Models

6.1 Baseline Linear Model

$$P_{t+1} = P_t + \alpha \times occupancyRate \tag{1}$$

This conservative model keeps prices within [5, 20] and updates them incrementally.

6.2 Demand-Based Surge Model

$$P_t = P_0(1 + \lambda D) \tag{2}$$

Prices increase proportionally to the normalized demand score D.

6.3 Competitive Model

Blends the demand price with average prices from competitors in a $2\,km$ radius, with heuristics for high ($> 90\%$) and low ($< 30\%$) occupancy scenarios.

7 Visualization Layer

The `VisualizationEngine` registers a `ColumnDataSource` per parking space and streams three price lines (baseline, demand, competitive). Hover tool-tips reveal exact timestamps and dollar values. Figure 2 illustrates a sample output.

bokeh_screenshot.png

Figure 2: Real-time Bokeh dashboard with three concurrent pricing curves.

8 Analytics and Rerouting

After the Pathway run completes, a summary table compares average, min, max, and standard deviation of prices for each model. Lots with $> 90\%$ occupancy trigger a rerouting recommendation listing up to three nearby cheaper alternatives.

9 Configuration Reference

Key parameters centralised in `PricingConfig`:

- `BASE_PRICE` = \$10.00
- `PRICE_LOWER_BOUND` = \$5.00
- `PRICE_UPPER_BOUND` = \$20.00
- Demand coefficients $\alpha = 0.3$, $\beta = 0.15$, $\gamma = 0.2$, $\delta = 0.2$, $\varepsilon = 0.15$, $\lambda = 0.8$.
- Proximity radius: $2\,km$.

Modify these constants to tune sensitivity without editing model code.

10 Extensibility

Future enhancements include:

1. Replacing heuristics with gradient-boosted regressors or neural nets.
2. Publishing prices via REST or WebSocket for integration with signage.
3. Auto-detecting events (concerts, sports) to flip `IsSpecialDay`.

11 Running the Notebook End-to-End

1. Upload `dataset.csv` to Colab or mount Google Drive.
2. Execute cells top-to-bottom. The pipeline writes three CSVs under `/tmp/parking_pricing`. Download them for offline inspection.
3. Export the entire notebook as PDF via `File → Print` or use `jupyter nbconvert` if working locally.

12 Appendix: Key Classes

Listing 3 shows trimmed UDF implementations. Refer to the notebook for the complete source.

Listing 3: Selected pricing UDFs

```python
class PricingUDFs:
    @pw.udf
    def get_vehicle_weight(vehicle_type: str) -> float:
        return PricingConfig.VEHICLE_WEIGHTS.get(vehicle_type.lower(),
            1.0)

    @pw.udf
    def baseline_linear_pricing(system_code: str, occupancy: int,
        capacity: int) -> float:
        prev = pricing_memory.get_last_price(system_code)
        occ_rate = occupancy / max(capacity, 1)
        price = min(max(prev + PricingConfig.ALPHA * occ_rate,
                        PricingConfig.PRICE_LOWER_BOUND),
                    PricingConfig.PRICE_UPPER_BOUND)
        pricing_memory.update_price(system_code, price)
        return price
```