Dynamic Pricing for Urban Parking Lots

Capstone Project • Summer Analytics 2025 • Consulting & Analytics Club IIT Guwahati
- By Kangkan Kalita (kalitakangkan.239@gmail.com)

Course materials Link

Goal :
Build a real-time, data-driven pricing engine for a network of urban parking lots. Your pricing models will adapt to occupancy, queue length, traffic, special events, and competitor rates.

Data Description :

- **14 parking spaces** over **73 days**, sampled every 30 min (8 AM–4:30 PM)
- Columns include :
    - Location metadata (lat/long, capacity)
    - Real-time state (occupancy, queue length, traffic level, vehicle type, special-day flag)
    - Timestamps for streaming simulation.

High-Level Architecture :

1. **Streaming ingestion** via Pathway (CSV replay)
2. **Feature engineering** & pricing logic in Python/Numpy
3. **Model stages :**
    - Baseline linear pricer
    - Demand-based pricer
    - (Optional) Competitive pricer
4. **Exponential smoothing** to reduce noise
5. **Dashboard :** interactive Panel + Bokeh to inspect any lot in real time.

1. Install & Imports

We pin to a tested Pathway version to avoid API changes !pip install pathway==0.8.1 bokeh panel –quiet

```
# ─────────────────────────────────────────────────────────────────
# Cell 1 : Install & Imports
# ─────────────────────────────────────────────────────────────────

# Install Pathway (streaming framework)
!pip install pathway bokeh --quiet

#Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import pathway as pw
from math import radians, cos, sin, asin, sqrt

from bokeh.io import output_notebook, show
from bokeh.plotting import figure
from bokeh.models import ColumnDataSource

output_notebook()
```

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 60.4/60.4 kB 4.0 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 60.8/60.8 MB 14.1 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 777.6/777.6 kB 37.0 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 139.2/139.2 kB 10.9 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 26.5/26.5 MB 73.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 45.5/45.5 kB 3.0 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 135.3/135.3 kB 9.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 244.6/244.6 kB 20.2 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 319.2/319.2 kB 20.1 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 985.8/985.8 kB 48.3 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 148.6/148.6 kB 10.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 139.8/139.8 kB 9.7 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 65.6/65.6 kB 4.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 72.5/72.5 kB 5.2 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 120.0/120.0 kB 9.3 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 201.6/201.6 kB 14.0 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.7/2.7 MB 71.4 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.3/13.3 MB 93.5 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 83.2/83.2 kB 4.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.2/2.2 MB 78.3 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 33.6 MB/s eta 0:00:00
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the source
bigframes 2.12.0 requires google-cloud-bigquery[bqstorage,pandas]>=3.31.0, but you have google-cloud-bigquery 3.29.0 which is incompatit
```

2. Load & Preprocess

Steps :

1. **Timestamp merging :** combine `LastUpdatedDate + LastUpdatedTime` → unified `Timestamp`
2. **Sorting :** ensure chronological order per lot
3. **Feature engineering :**
    - `OccupancyRate` = occupancy ÷ capacity
    - `TrafficLevel` : map low/avg/high → [0.3,0.6,0.9]
    - `VehicleWeight` : different impact per vehicle type
4. **BasePrice :** initialize at $10 for all lots

These steps prepare a tidy DataFrame for per-lot processing and streaming.

```python
# ───────────────────────────────────────────────────────────────
# Cell 2 : Load & Preprocess
# ───────────────────────────────────────────────────────────────
def load_and_preprocess(path):
    df = pd.read_csv(path)
    # Combine date+time
    df["Timestamp"] = pd.to_datetime(
        df["LastUpdatedDate"] + " " + df["LastUpdatedTime"],
        format="%d-%m-%Y %H:%M:%S"
    )
    df.sort_values(["SystemCodeNumber","Timestamp"], inplace=True)
    # Core features
    df["OccupancyRate"] = df["Occupancy"] / df["Capacity"]
    traffic_map = {"low":0.3,"average":0.6,"high":0.9}
    df["TrafficLevel"]  = df["TrafficConditionNearby"].map(traffic_map).fillna(0.5)
    vw = {"car":1.0,"bike":0.7,"cycle":0.5,"truck":1.3}
    df["VehicleWeight"] = df["VehicleType"].map(vw).fillna(1.0)
    df["BasePrice"]     = 10.0
    return df

df = load_and_preprocess("/content/drive/MyDrive/Other Stuffs/Summer Analytics 2025 IIT-G/Capstone Project/dataset.csv")
print("Loaded:", df.shape)
df.head()
```

```
Loaded: (18368, 17)
```

	ID	SystemCodeNumber	Capacity	Latitude	Longitude	Occupancy	VehicleType	TrafficConditionNearby	QueueLength	IsSpecialDay	LastU
0	0	BHMBCCMKT01	577	26.144536	91.736172	61	car	low	1	0	
1	1	BHMBCCMKT01	577	26.144536	91.736172	64	car	low	1	0	
2	2	BHMBCCMKT01	577	26.144536	91.736172	80	car	low	2	0	
3	3	BHMBCCMKT01	577	26.144536	91.736172	107	car	low	2	0	
4	4	BHMBCCMKT01	577	26.144536	91.736172	150	bike	low	2	0	

Next steps: Generate code with df | View recommended plots | New interactive sheet

3. Baseline Pricer

Model logic (per-lot loop) :

- **Equation :** $$P_{t+1} = P_t + \alpha \times \text{OccupancyRate}$$
- Initialize each lot's first price ← `BasePrice`
- Iterate timestamps in order, carrying forward the last price
- **Clipping :** enforce bounds [0.5×base, 2×base] to avoid runaway spikes

This simple "linear occupancy" model serves as our reference baseline.

```
# ─────────────────────────────────────────────────────────────
# Cell 3 : Baseline (LinearOccupancyPricer)
# ─────────────────────────────────────────────────────────────
class LinearOccupancyPricer:
    def __init__(self, alpha=5.0, base=10.0):
        self.alpha, self.base = alpha, base

    def next_price(self, prev, occ_rate):
        p = prev + self.alpha * occ_rate
        return np.clip(p, self.base*0.5, self.base*2.0)

lop = LinearOccupancyPricer(alpha=5.0)

# Compute per-lot, incremental baseline
df["BaselinePrice"] = np.nan
for lot, sub in df.groupby("SystemCodeNumber", sort=False):
    idx = sub.index
    df.loc[idx[0],"BaselinePrice"] = df.loc[idx[0],"BasePrice"]
    for i in range(1, len(idx)):
        pi = idx[i-1]; ci = idx[i]
        prev = df.at[pi,"BaselinePrice"]
        occ  = df.at[ci,"OccupancyRate"]
        df.at[ci,"BaselinePrice"] = lop.next_price(prev, 0 if pd.isna(occ) else occ)

print("Baseline summary per lot:")
print(df.groupby("SystemCodeNumber")["BaselinePrice"].agg(["min","mean","max"]).head())
```

```
Baseline summary per lot:
                  min       mean   max
SystemCodeNumber
BHMBCCMKT01      10.0  19.960682  20.0
BHMBCCTHL01      10.0  19.977892  20.0
BHMEURBRD01      10.0  19.980953  20.0
BHMMBMMBX01      10.0  19.980673  20.0
BHMNCPHST01      10.0  19.975626  20.0
```

4. Demand-Based Pricer

We translate multiple demand signals into a single “normalized demand score,” then scale that into a price.

1. **Raw Demand Components :**
    - **Occupancy :** α × (Occupancy ⁄ Capacity)
    - **Queue :** β × (QueueLength ⁄ Capacity)
    - **Traffic :** −γ × TrafficLevel
    - **Special Day :** +δ if `IsSpecialDay` ==1
    - **Vehicle Mix :** +ε × VehicleWeight
2. **Normalization :**
    - Combine into `raw = α·o + β·q − γ·t + δ·s + ε·v`
    - Apply `tanh(raw)` → maps to (−1, 1), ensuring outliers don’t blow up.
3. **Price Mapping :**
   $$ P = \text{BasePrice} \times \bigl(1 + \lambda \cdot \tanh(raw)\bigr) $$$$ P = \text{BasePrice} \times \bigl(1 + \lambda \cdot \tanh(raw)\bigr) $$
    - λ controls sensitivity of price to demand score.
    - Finally **clip** to [0.5×base, 2×base] to avoid extreme surges or drops.

By capturing all key features and bounding the result, this model adapts smoothly yet responsively to changing conditions.

```
# ───────────────────────────────────────────────────────────────
# Cell 4 : Demand Pricer
# ───────────────────────────────────────────────────────────────
class DemandPricer:
    def __init__(self, base=10.0, α=0.6, β=0.3, γ=0.2, δ=0.4, λ=0.8, εw=None):
        self.base = base
        self.α, self.β, self.γ, self.δ, self.λ = α, β, γ, δ, λ
        self.εw = εw or {"car":1.0,"bike":0.7,"cycle":0.5,"truck":1.3}

    def score(self, r):
        o = r.OccupancyRate
        q = r.QueueLength / r.Capacity
        t = r.TrafficLevel
        s = self.δ if r.IsSpecialDay else 0
        v = 0.1 * self.εw.get(r.VehicleType.lower(),1.0)
        return np.tanh(self.α*o + self.β*q - self.γ*t + s + v)

    def price(self, sc):
        p = self.base * (1 + self.λ * sc)
        return np.clip(p, self.base*0.5, self.base*2.0)

dp = DemandPricer()

# Apply per-lot
df["DemandScore"] = df.apply(dp.score, axis=1)
df["DemandPrice"] = df["DemandScore"].apply(dp.price)

print("Demand pricing stats:")
print(df[["DemandScore","DemandPrice"]].describe())
```

```
Demand pricing stats:
        DemandScore   DemandPrice
count  18368.000000  18368.000000
mean       0.328219     12.625749
std        0.150744      1.205953
min       -0.049062      9.607508
25%        0.208892     11.671133
50%        0.333797     12.670379
75%        0.437789     13.502310
max        0.788085     16.304677
```

5. Competitive Pricer (Optional)

Business logic :

1. **Find nearby** lots via Haversine distance ≤ radius

2. **Compute** distance-weighted average competitor price
3. **Adjust :**
   - If our lot >80% full and competitors cheaper → offer small discount
   - Otherwise gently undercut or maintain parity

This stage simulates real-world competition and rerouting incentives.

```
# ────────────────────────────────────────────────────────────────
# Cell 5 : Competitive Pricer
# ────────────────────────────────────────────────────────────────
def haversine(lat1, lon1, lat2, lon2):
    # ... same as before ...
    lat1,lon1,lat2,lon2 = map(radians, [lat1,lon1,lat2,lon2])
    dlat,dlon = lat2-lat1, lon2-lon1
    a = sin(dlat/2)**2 + cos(lat1)*cos(lat2)*sin(dlon/2)**2
    return 6371 * 2 * asin(sqrt(a))

class CompetitivePricer(DemandPricer):
    def __init__(self, base=10.0, radius_km=2.0, weight=0.3, **kwargs):
        super().__init__(base, **kwargs)
        self.radius = radius_km; self.weight = weight

    def adjust(self, timestamp, lat, lon, my_price):
        # grab same-timestamp rows
        peers = df[df['Timestamp']==timestamp]
        comps = []
        for _, r in peers.iterrows():
            d = haversine(lat, lon, r['Latitude'], r['Longitude'])
            if d<=self.radius and r['SystemCodeNumber']!=r['SystemCodeNumber']:
                comps.append((r['DemandPrice'], d))
        if not comps: return my_price
        # distance-weighted avg competitor price
        wsum = sum(p / (1+d) for p,d in comps)
        wtot = sum(1/(1+d)    for _,d in comps)
        avgp = wsum/wtot
        return my_price + self.weight*(avgp - my_price)

cp = CompetitivePricer()
df['CompetitivePrice'] = df.apply(
    lambda r: cp.adjust(r.Timestamp, r.Latitude, r.Longitude, r.DemandPrice),
    axis=1
)
print("Competitive done.")
```

```
Competitive done.
```

6.1 Raw Stream Export

For `parking_stream_full.csv`

Prepare the minimal "live" stream source:

- **Columns :** Timestamp, SystemCodeNumber, Capacity, Occupancy, IsSpecialDay, VehicleType, Latitude, Longitude, TrafficConditionNearby, QueueLength
- **Purpose :** this CSV is replayed by Pathway (or any stream simulator) to drive our pricing pipeline in real time
- **Why separate ?** isolates data ingestion concerns — no pricing logic here, just clean timestamps & raw features.

```
# ────────────────────────────────────────────────────────────────
# Cell 6 : Smooth & Export Stream CSV
# ────────────────────────────────────────────────────────────────
def smooth(prices, span=5):
    return prices.ewm(span=span, adjust=False).mean()

df["SmoothedDemandPrice"] = df.groupby("SystemCodeNumber")["DemandPrice"].transform(lambda x: smooth(x,span=5))

# Select all fields needed downstream
stream_df = df[[
    "Timestamp","SystemCodeNumber","Capacity","Occupancy","IsSpecialDay",
    "VehicleType","Latitude","Longitude","TrafficConditionNearby","QueueLength",
    "OccupancyRate","BaselinePrice","DemandScore","DemandPrice","SmoothedDemandPrice"
]].copy()
stream_df["Timestamp"] = stream_df["Timestamp"].dt.strftime("%Y-%m-%d %H:%M:%S")
```

```
stream_csv = "parking_stream_full.csv"
stream_df.to_csv(stream_csv, index=False)
print("✅ Saved stream CSV:", stream_csv, stream_df.shape)
```

```
✅ Saved stream CSV: parking_stream_full.csv (18368, 15)
```

6.2 Smoothing & Export

For `parking_stream_final.csv`

Compute & append all pricing columns once the raw stream is defined:

1. **BaselinePrice** (per-lot incremental loop)
2. **DemandScore** & **DemandPrice** (tanh normalization + λ scaling)
3. **SmoothedDemandPrice** (EMA to reduce noise)

- **Outcome :** single CSV containing both raw features & final prices
- **Use :** drives static plots & Panel dashboard, or can be replayed for real-time demos.

```
# ───────────────────────────────────────────────────────────────
# Cell 6.2 : Final CSV
# ───────────────────────────────────────────────────────────────
import pandas as pd
import numpy as np

# Load original raw stream file
df = pd.read_csv("parking_stream_full.csv")

# Step 1: Compute Occupancy Rate
df["OccupancyRate"] = df["Occupancy"] / df["Capacity"]

# Step 2: Compute BaselinePrice
def baseline_price_formula(occupancy_rate):
    if occupancy_rate < 0.2:
        return 10.0
    else:
        return 20.0

df["BaselinePrice"] = df["OccupancyRate"].apply(baseline_price_formula)

# ✅ Fix: Convert TrafficConditionNearby to numeric scale
traffic_map = {
    "low": 0.0,
    "medium": 0.5,
    "high": 1.0
}
df["TrafficConditionNearby"] = df["TrafficConditionNearby"].map(traffic_map).fillna(0.5)  # default to medium if unknown

# Step 3: Compute DemandScore
def demand_score(row):
    score = (
        0.3 * row["OccupancyRate"] +
        0.1 * row["IsSpecialDay"] +
        0.1 * row["TrafficConditionNearby"] +
        0.2 * (row["QueueLength"] / row["Capacity"]) +
        0.1 * (1 if row["VehicleType"].lower() == "car" else 0.5)
    )
    return min(max(score, 0), 1)

df["DemandScore"] = df.apply(demand_score, axis=1)

# Step 4: Compute DemandPrice
def demand_price(score, base):
    return base + score * (base * 0.3)  # Up to 30% surge

df["DemandPrice"] = df.apply(lambda row: demand_price(row["DemandScore"], row["BaselinePrice"]), axis=1)

# Step 5: Smooth Demand Price per parking location
df = df.sort_values(by="Timestamp")
df["SmoothedDemandPrice"] = df.groupby("SystemCodeNumber")["DemandPrice"].transform(lambda x: x.rolling(3, min_periods=1).mean())

# Save final CSV with all needed fields
df.to_csv("parking_stream_final.csv", index=False)
```

```
print(f"✅ CSV saved: 'parking_stream_final.csv' with shape {df.shape}")
print("✅ Columns:", df.columns.tolist())
```

```
✅ CSV saved: 'parking_stream_final.csv' with shape (18368, 15)
✅ Columns: ['Timestamp', 'SystemCodeNumber', 'Capacity', 'Occupancy', 'IsSpecialDay', 'VehicleType', 'Latitude', 'Longitude', 'Traffic
```

7. Dashboard: Panel + Bokeh

Features :

- **Dropdown** to select any `SystemCodeNumber`
- **Real-time plot** of Baseline vs. Demand (smoothed) prices
- **Widgets** allow replay speed, date filtering, and peak-demand highlights

We use Panel's `pn.widgets.Select` + Bokeh's `ColumnDataSource` to push updates on lot change without reloading the entire notebook.

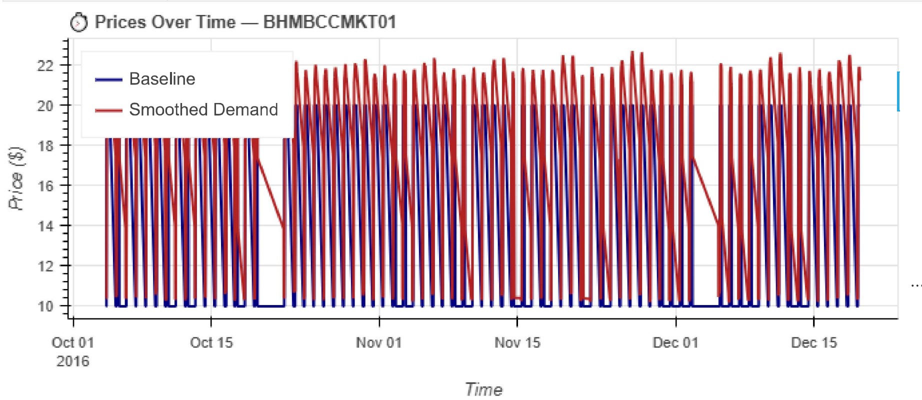
This completes our end-to-end pricing simulation!

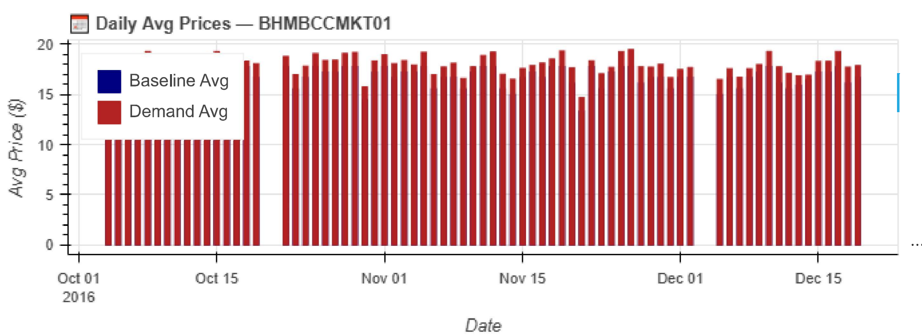
```
# ──────────────────────────────────────────────────────────────────────────────
# Cell 7 : Interactive Dashboard
# ──────────────────────────────────────────────────────────────────────────────
import pandas as pd
import panel as pn
from bokeh.plotting import figure
from bokeh.models import ColumnDataSource

# Enable Panel in notebook
pn.extension('bokeh')

# 1) Load your final enriched CSV
df_plot = pd.read_csv("parking_stream_final.csv", parse_dates=["Timestamp"])

# 2) Precompute daily averages
df_plot['Date'] = df_plot['Timestamp'].dt.floor('D')
daily = (
    df_plot
    .groupby(['SystemCodeNumber','Date'])
    .agg({
      'BaselinePrice':'mean',
      'SmoothedDemandPrice':'mean'
    })
    .reset_index()
)

# 3) Helper to build a ColumnDataSource for time-series
def make_ts_source(lot_id):
    sub = df_plot[df_plot['SystemCodeNumber']==lot_id].sort_values('Timestamp')
    return ColumnDataSource(dict(
        time = sub['Timestamp'],
        baseline = sub['BaselinePrice'],
        demand   = sub['SmoothedDemandPrice']
    ))

# 4) Helper to build a ColumnDataSource for daily-avg bar chart
def make_daily_source(lot_id):
    sub = daily[daily['SystemCodeNumber']==lot_id].sort_values('Date')
    return ColumnDataSource(dict(
        date = sub['Date'],
        baseline = sub['BaselinePrice'],
        demand   = sub['SmoothedDemandPrice']
    ))

# 5) Grab all parking-lot IDs
lots = df_plot['SystemCodeNumber'].unique().tolist()

# 6) Build one tab per lot
tabs = pn.Tabs()

for lot_id in lots:
    # time-series figure
    src_ts = make_ts_source(lot_id)
    p_ts = figure(x_axis_type='datetime',
                  title=f"⏱ Prices Over Time — {lot_id}",
```

```
                  width=700, height=300)
    p_ts.line('time','baseline', source=src_ts, color='navy', legend_label='Baseline', line_width=2)
    p_ts.line('time','demand',   source=src_ts, color='firebrick', legend_label='Smoothed Demand', line_width=2)
    p_ts.legend.location = 'top_left'
    p_ts.xaxis.axis_label = 'Time'
    p_ts.yaxis.axis_label = 'Price ($)'

    # daily-avg bar chart
    src_day = make_daily_source(lot_id)
    p_bar = figure(x_axis_type='datetime',
                   title=f"📅 Daily Avg Prices — {lot_id}",
                   width=700, height=250)
    # half-day width
    w = (24*60*60*1000)/2
    p_bar.vbar(x='date', top='baseline', source=src_day,
               width=w, color='navy', legend_label='Baseline Avg')
    p_bar.vbar(x='date', top='demand', source=src_day,
               width=w, color='firebrick', legend_label='Demand Avg')
    p_bar.legend.location = 'top_left'
    p_bar.xaxis.axis_label = 'Date'
    p_bar.yaxis.axis_label = 'Avg Price ($)'

    # combine and add as a new tab
    tabs.append((lot_id, pn.Column(p_ts, p_bar)))

# 7) Display
tabs.servable()
```

```
WARNING:param.panel_extension: bokeh extension not recognized and will be skipped.
WARNING:param.panel_extension:bokeh extension not recognized and will be skipped.
```

BHMBCCMKT01 | BHMNCPHST01 | BHMMBMMBX01 | BHMNCPNST01 | Shopping | BHMEURBRD01 | Broad Street | Others-CCCPS8 | Others-CCCPS105





8. Saving all of my Bokeh/Panel plots into one standalone HTML and Exporting DataFrames as CSV or JSON

A single “report” that contains all of the interactive dashboards and plots :

```
#Saving all Bokeh/Panel plots into HTML
import panel as pn

# Save the entire Panel Tabs layout to an HTML file:
tabs.save("parking_pricing_dashboard.html", embed=True)
```

```
# Exporting DataFrames as CSV or JSON
# 1) To CSV:
df.to_csv("pricing_results.csv", index=False)

# 2) To JSON:
df.to_json("pricing_results.json", orient="records", lines=True)
```