CIVIL 763 Project - Practical Insights from New Zealand Residential Property Data

Exploring affordability and housing characteristics under a fixed NZD 780,000 budget

---

Key Findings and Overall Conclusions

Pattern 1 — Interest Rate vs House Price

- Using over **3.5 million national sales (1990–2024)**, a log–log regression found an **elasticity of −1.41** between interest rates and house prices.
- Every **1 % increase in mortgage rates** lowers prices by only about **1.4 %**, meaning affordability changes faster than price itself.

Insight: Interest-rate fluctuations affect short-term borrowing capacity more than long-term value. Waiting for rates to drop may not reduce prices significantly, but it can improve repayment comfort.

Pattern 2 — Typical Homes with NZD 780 000 Budget

- Based on **K-Nearest-Neighbour matching** within a ± 2 % price band (2018–2024):
    - **Auckland:** newer homes — ~92 m^2 floor, 104 m^2 land
    - **Wellington:** balanced homes — ~123 m^2 floor, 551 m^2 land
    - **Dunedin:** larger homes — ~180 m^2 floor, 629 m^2 land

Insight: At the same budget, living standards vary greatly — Dunedin offers roughly **2× the floor area** and **5× the land** of Auckland.

Pattern 3 — Price Growth and Projection in 2035

- After **sensitivity tests** on time window and tolerance (± 5 %, ± 10 %, ± 20 %), the **± 10 % band and 1990–2024 window** produced the most stable trend.
- **Compound Annual Growth Rates (CAGR):**
    - Auckland → **4.36 %**
    - Wellington → **5.48 %**
    - Dunedin → **5.52 %**
- If trends persist, typical home values (base = NZD 780 000 in 2024) may reach:
    - **≈ NZD 1.2 M (Auckland)**
    - **≈ NZD 1.4 M (Wellington & Dunedin)** by 2035.

Insight: Long-term growth remains strong nationwide, but **regional centres now outperform Auckland**.

Overall Recommendations

1. **Buy for affordability, not timing.** Interest-rate cycles change repayments more than property value.
2. **Budget defines lifestyle.** Same price → very different size and land options; buyers should compare beyond city labels.
3. **PLong-term growth is strong everywhere.** but Dunedin and Wellington now outperform Auckland.

---

1. Problem Understanding

Objective

Identify what a fixed NZD 780,000 budget can buy across different New Zealand cities and forecast how these properties may appreciate over time.

Motivation

Rising housing prices and interest-rate fluctuations make affordability a key social and economic issue.

Research Questions

1. How have interest rates affected national median prices (2004–2024)?
2. How do property characteristics differ across Auckland, Wellington, and Dunedin at the same budget level?
3. What future price growth can be expected by 2030, 2035, and 2055?

---

2. Data Understanding

Data Sources

- **CSTDAT8700_Output1_20250717.csv** (3.56M rows × 56 columns)
- **CSTDAT8700_Output2_20250717.csv** (1.71M rows × 6 columns)
- Merged on **Combined_Residential_Property_Sale_Stats.csv** (left join) → 3.56M rows × 61 columns

---

2.1 Load and Combine Raw Files

In [2]:

```python
import pandas as pd

# ===== Settings =====
file1 = "CSTDAT8700_Output1_20250717.csv"
file2 = "CSTDAT8700_Output2_20250717.csv"
out_csv = "Combined_Residential_Property_Sale_Stats.csv"
out_txt = "Combined_Residential_Property_Sale_Stats_Summary.txt"

# ===== Load =====
df1 = pd.read_csv(file1, low_memory=False)
df2 = pd.read_csv(file2, low_memory=False)

# ===== Normalize join key =====
for df in (df1, df2):
    if "CL_QPID" not in df.columns:
        raise KeyError("CL_QPID (Sale ID) not found in one of the files.")
    df["CL_QPID"] = df["CL_QPID"].astype(str).str.strip()

# ===== Record original sizes =====
r1, c1 = df1.shape
r2, c2 = df2.shape

# ===== Ensure File 2 has one row per CL_QPID to avoid one-to-many explosion ===
df2_unique = df2.drop_duplicates(subset="CL_QPID", keep="first")

# ===== Only append columns that don't already exist in File 1 =====
add_cols = [col for col in df2_unique.columns if col != "CL_QPID" and col not in
df2_add = df2_unique[["CL_QPID"] + add_cols] if add_cols else df2_unique[["CL_QP

# ===== Match stats (for summary only) =====
matched_mask = df1["CL_QPID"].isin(df2_unique["CL_QPID"])
matched_count = int(matched_mask.sum())
unmatched_count = int((~matched_mask).sum())

# ===== Left join: keep all rows from File 1 =====
combined = pd.merge(df1, df2_add, on="CL_QPID", how="left")

# ===== Final size =====
rm, cm = combined.shape

# ===== Save outputs =====
combined.to_csv(out_csv, index=False)

# ===== Build single summary text =====
lines = []
lines.append("Combined Residential Property Sale Stats — Summary")
lines.append("==================================================")
lines.append("")
lines.append("Source files (before merge):")
lines.append(f"- File 1: {file1} | Rows: {r1:,} | Cols: {c1}")
lines.append(f"- File 2: {file2} | Rows: {r2:,} | Cols: {c2}")
lines.append("")
lines.append("Join key: CL_QPID (left join, keep all rows from File 1)")
lines.append(f"- Matched rows in File 1: {matched_count:,}")
lines.append(f"- Unmatched rows in File 1: {unmatched_count:,}")
lines.append("")
lines.append("Columns appended from File 2:")
if add_cols:
    lines.append(f"- Added {len(add_cols)} column(s): " + ", ".join(add_cols))
else:
    lines.append("- No new columns were appended (all already existed in File 1)
lines.append("")
lines.append("Combined dataset (after merge):")
lines.append(f"- Rows: {rm:,} | Cols: {cm}")
lines.append("")
lines.append("All column names in the combined dataset:")
lines.append(", ".join(combined.columns))
lines.append("")

with open(out_txt, "w", encoding="utf-8") as f:
    f.write("\n".join(lines))

print("\n".join(lines))
print(f"\nSaved CSV: {out_csv}")
print(f"Saved summary: {out_txt}")
```

```
Combined Residential Property Sale Stats — Summary
==================================================

Source files (before merge):
- File 1: CSTDAT8700_Output1_20250717.csv | Rows: 3,558,332 | Cols: 56
- File 2: CSTDAT8700_Output2_20250717.csv | Rows: 1,714,210 | Cols: 6

Join key: CL_QPID (left join, keep all rows from File 1)
- Matched rows in File 1: 3,449,787
- Unmatched rows in File 1: 108,545

Columns appended from File 2:
- Added 5 column(s): CL_Val_Ref, CL_Latitude, CL_Longitude, CL_Bedrooms, CL_Bathr
ooms

Combined dataset (after merge):
- Rows: 3,558,332 | Cols: 61

All column names in the combined dataset:
CL_QPID, CL_Sale_ID, CL_Building_ID, CL_Situation_Number, CL_TA7_MissingMB_Situat
ion_Number, CL_TA7_MissingMB_Additional_Number, CL_Street_Name, CL_Street_Name_Su
ffix, CL_Street_Name_Direction, CL_Suburb, CL_Town, CL_RegionID, CL_RegionName, C
L_TAcode, CL_TAName, CL_Meshblock, CL_SAU, CL_Sale_Tenure, CL_Sale_Price_Value_Re
lationship, CL_Sale_Date, CL_Sale_Price_Net, CL_Sale_Price_Chattels, CL_Sale_Pric
e_Other, CL_Sale_Price_Gross, CL_Land_Valuation_Capital_Value, CL_Land_Valuation_
Land_Value, CL_Land_Valuation_Improvements_Value, CL_Current_Revision_Date, CL_Bu
ilding_Floor_Area, CL_Building_Site_Cover, CL_Land_Area, CL_Bldg_Const, CL_Bldg_C
ond, CL_Roof_Const, CL_Roof_Cond, CL_Category, CL_LUD_Age, CL_LUD_Land_Use_Descri
ption, CL_MAS_Class_Surrounding_Improvmnt_Type, CL_MAS_Contour, CL_MAS_View, CL_M
AS_View_Scope, CL_MAS_Modernisation, CL_MAS_House_Type_Description, CL_MAS_Deck_I
ndicator, CL_MAS_Driveway_Indicator, CL_MAS_No_Main_Roof_Garages, CL_MAS_Free_Sta
nding_Garages, CL_MAS_Estimated_Year_Built, CL_MAS_Landscaping_Quality, CL_MAS_Lo
t_Position, CL_School_Zone_1, CL_School_Zone_2, CL_School_Zone_3, CL_School_Zone_
4, CL_School_Zone_5, CL_Val_Ref, CL_Latitude, CL_Longitude, CL_Bedrooms, CL_Bathr
ooms


Saved CSV: Combined_Residential_Property_Sale_Stats.csv
Saved summary: Combined_Residential_Property_Sale_Stats_Summary.txt
```

2.2 Rename Columns

In [3]:
```
import pandas as pd
from pathlib import Path

# ===== Files =====
csv_path = Path("Combined_Residential_Property_Sale_Stats.csv")
summary_path = Path("Combined_Residential_Property_Sale_Stats_Summary.txt")

# ===== 1) Load combined CSV =====
df = pd.read_csv(csv_path, low_memory=False)

# ===== 2) Define concise English names =====
# Mapping covers the columns shown in your summary. If some are missing, they're
rename_map_full = {
    "CL_QPID": "QPID",
    "CL_Sale_ID": "Sale_ID",
    "CL_Building_ID": "Building_ID",
    "CL_Situation_Number": "Situation_No",
    "CL_TA7_MissingMB_Situation_Number": "TA7_Missing_Situation_No",
    "CL_TA7_MissingMB_Additional_Number": "TA7_Missing_Additional_No",
    "CL_Street_Name": "Street_Name",
    "CL_Street_Name_Suffix": "Street_Suffix",
    "CL_Street_Name_Direction": "Street_Direction",
    "CL_Suburb": "Suburb",
    "CL_Town": "Town",
    "CL_RegionID": "Region_ID",
    "CL_RegionName": "Region_Name",
    "CL_TAcode": "TA_Code",
    "CL_TAName": "TA_Name",
    "CL_Meshblock": "Meshblock",
    "CL_SAU": "SAU",
    "CL_Sale_Tenure": "Sale_Tenure",
    "CL_Sale_Price_Value_Relationship": "Price_Relationship",
    "CL_Sale_Date": "Sale_Date",
    "CL_Sale_Price_Net": "Price_Net",
    "CL_Sale_Price_Chattels": "Price_Chattels",
    "CL_Sale_Price_Other": "Price_Other",
    "CL_Sale_Price_Gross": "Price_Gross",
    "CL_Land_Valuation_Capital_Value": "CV_Capital_Value",
    "CL_Land_Valuation_Land_Value": "LV_Land_Value",
    "CL_Land_Valuation_Improvements_Value": "IV_Improvements_Value",
    "CL_Current_Revision_Date": "Revision_Date",
    "CL_Building_Floor_Area": "Floor_Area",
    "CL_Building_Site_Cover": "Site_Cover",
    "CL_Land_Area": "Land_Area",
    "CL_Bldg_Const": "Bldg_Construction",
    "CL_Bldg_Cond": "Bldg_Condition",
    "CL_Roof_Const": "Roof_Construction",
    "CL_Roof_Cond": "Roof_Condition",
    "CL_Category": "Category",
    "CL_LUD_Age": "LUD_Age",
    "CL_LUD_Land_Use_Description": "Land_Use_Desc",
    "CL_MAS_Class_Surrounding_Improvmnt_Type": "Surrounding_Improv_Class",
    "CL_MAS_Contour": "Contour",
    "CL_MAS_View": "View",
    "CL_MAS_View_Scope": "View_Scope",
    "CL_MAS_Modernisation": "Modernisation",
    "CL_MAS_House_Type_Description": "House_Type",
    "CL_MAS_Deck_Indicator": "Deck",
    "CL_MAS_Driveway_Indicator": "Driveway",
    "CL_MAS_No_Main_Roof_Garages": "No_Main_Roof_Garages",
    "CL_MAS_Free_Standing_Garages": "Free_Standing_Garages",
    "CL_MAS_Estimated_Year_Built": "Year_Built_Est",
    "CL_MAS_Landscaping_Quality": "Landscaping_Quality",
    "CL_MAS_Lot_Position": "Lot_Position",
    "CL_School_Zone_1": "School_Zone_1",
    "CL_School_Zone_2": "School_Zone_2",
    "CL_School_Zone_3": "School_Zone_3",
    "CL_School_Zone_4": "School_Zone_4",
    "CL_School_Zone_5": "School_Zone_5",
    "CL_Val_Ref": "Valuation_Ref",
    "CL_Latitude": "Latitude",
    "CL_Longitude": "Longitude",
    "CL_Bedrooms": "Bedrooms",
    "CL_Bathrooms": "Bathrooms",
}

# Only apply mappings for columns that actually exist
rename_map = {old: new for old, new in rename_map_full.items() if old in df.colu

# ===== 3) Rename & save (overwrite the same CSV) =====
df = df.rename(columns=rename_map)
df.to_csv(csv_path, index=False)

# ===== 4) Append the rename mapping to the existing summary =====
lines = []
lines.append("")
lines.append("Column rename mapping (old -> new):")
if rename_map:
    for old, new in rename_map.items():
        lines.append(f"- {old} -> {new}")
else:
    lines.append("- (No columns were renamed; none of the expected names were fo
lines.append("")

with open(summary_path, "a", encoding="utf-8") as f:
    f.write("\n".join(lines))

print(f"Renamed {len(rename_map)} column(s) and overwrote {csv_path.name}.")
print(f"Appended rename mapping to {summary_path.name}.")
```

```
Renamed 61 column(s) and overwrote Combined_Residential_Property_Sale_Stats.csv.
Appended rename mapping to Combined_Residential_Property_Sale_Stats_Summary.txt.
```

2.3 Reload Cleaned Data

In [4]:
```
import pandas as pd

# Load only once
df = pd.read_csv("Combined_Residential_Property_Sale_Stats.csv", low_memory=Fals
print(df.head())
```

```
    QPID  Sale_ID  Building_ID Situation_No  TA7_Missing_Situation_No  \
0  86336  2927710            0           66                       NaN   
1  86336  3586965            0           66                       NaN   
2  86337  3970574            0           68                       NaN   
3  86337  4181726            0           68                       NaN   
4  86337  5944667            0           68                       NaN   

  TA7_Missing_Additional_No Street_Name Street_Suffix Street_Direction  \
0                       NaN      Parore            St              NaN   
1                       NaN      Parore            St              NaN   
2                       NaN      Parore            St              NaN   
3                       NaN      Parore            St              NaN   
4                       NaN      Parore            St              NaN   

       Suburb  ... School_Zone_1  School_Zone_2 School_Zone_3  School_Zone_4  \
0  Dargaville  ...           NaN            NaN           NaN            NaN   
1  Dargaville  ...           NaN            NaN           NaN            NaN   
2  Dargaville  ...           NaN            NaN           NaN            NaN   
3  Dargaville  ...           NaN            NaN           NaN            NaN   
4  Dargaville  ...           NaN            NaN           NaN            NaN   

  School_Zone_5  Valuation_Ref   Latitude   Longitude  Bedrooms Bathrooms  
0           NaN      950/46100 -35.935148  173.864613       3.0       1.0  
1           NaN      950/46100 -35.935148  173.864613       3.0       1.0  
2           NaN      950/46200 -35.935050  173.864470       6.0       3.0  
3           NaN      950/46200 -35.935050  173.864470       6.0       3.0  
4           NaN      950/46200 -35.935050  173.864470       6.0       3.0  

[5 rows x 61 columns]
```

2.4 Check Percentage of Missing Values

In [5]:

```python
# ===== Calculate % of missing values per column =====
missing_percent = df.isna().mean() * 100
missing_table = missing_percent.sort_values(ascending=False).reset_index()
missing_table.columns = ["Column", "Missing_%"]

# ===== Print ranked table =====
print("Percentage of Missing Values by Column (ranked high to low):")
print(missing_table.to_string(index=False))
```

```
Percentage of Missing Values by Column (ranked high to low):
                   Column  Missing_%
TA7_Missing_Additional_No  99.786754
 TA7_Missing_Situation_No  99.694688
         Street_Direction  99.289358
            School_Zone_5  76.044984
               View_Scope  75.445068
            School_Zone_4  63.016408
            School_Zone_3  47.657610
               House_Type  33.860921
      Landscaping_Quality  33.564097
           Year_Built_Est  33.236415
            School_Zone_2  31.397913
 Surrounding_Improv_Class  29.579140
             Lot_Position  29.400461
            Modernisation  23.036608
                 Driveway  22.359437
                     View  15.554479
                  Contour  15.400446
        Bldg_Construction  14.846001
            School_Zone_1  14.724736
                  LUD_Age  13.257138
           Bldg_Condition  13.250000
           Roof_Condition  12.634009
        Roof_Construction  12.547705
                     Deck   8.216856
    Free_Standing_Garages   5.480545
     No_Main_Roof_Garages   5.402138
              Price_Other   5.098737
                Meshblock   4.954316
                      SAU   4.954316
                 Bedrooms   4.657210
            Revision_Date   4.431683
                 Latitude   3.093584
                Longitude   3.093584
                Bathrooms   3.051936
            Valuation_Ref   3.050446
               Site_Cover   2.788582
               Floor_Area   2.786530
                   Suburb   2.571879
                     Town   1.771982
            Street_Suffix   1.030089
           Price_Chattels   0.402042
            LV_Land_Value   0.052412
         CV_Capital_Value   0.023297
                Land_Area   0.018773
            Land_Use_Desc   0.015119
                Price_Net   0.002445
              Street_Name   0.000112
                  TA_Name   0.000000
              Building_ID   0.000000
             Situation_No   0.000000
                Region_ID   0.000000
              Region_Name   0.000000
                  TA_Code   0.000000
              Sale_Tenure   0.000000
                 Category   0.000000
       Price_Relationship   0.000000
                Sale_Date   0.000000
              Price_Gross   0.000000
```

```
IV_Improvements_Value    0.000000
              Sale_ID    0.000000
                 QPID    0.000000
```

2.5 Drop Columns with > 40% Missing Values

In [6]:
```python
import pandas as pd
from pathlib import Path

# ===== File paths =====
summary_path = Path("Combined_Residential_Property_Sale_Stats_Summary.txt")
csv_path = Path("Combined_Residential_Property_Sale_Stats.csv")

# ===== Record original shape =====
rows_before, cols_before = df.shape

# ===== Compute missing percentage per column =====
missing_percent = df.isna().mean() * 100

# ===== Determine columns to drop (>40% missing) =====
cols_to_drop = missing_percent[missing_percent > 40].index.tolist()
cols_to_keep = [c for c in df.columns if c not in cols_to_drop]

# ===== Drop the columns =====
df_cleaned = df.drop(columns=cols_to_drop)
rows_after, cols_after = df_cleaned.shape

# ===== Overwrite the same CSV =====
df_cleaned.to_csv(csv_path, index=False)

# ===== Prepare text summary =====
lines = []
lines.append("")
lines.append("Columns Removed Based on Missing Percentage (>40%)")
lines.append("====================================================")
lines.append(f"Original size: {rows_before:,} rows × {cols_before} columns")
lines.append(f"After dropping: {rows_after:,} rows × {cols_after} columns")
lines.append("")

if cols_to_drop:
    lines.append(f"Columns dropped ({len(cols_to_drop)}):")
    lines.append(", ".join(cols_to_drop))
else:
    lines.append("No columns exceeded 40% missing — none dropped.")

lines.append("")
lines.append(f"Columns kept ({len(cols_to_keep)}):")
lines.append(", ".join(cols_to_keep))
lines.append("")

# ===== Append to the summary text file =====
with open(summary_path, "a", encoding="utf-8") as f:
    f.write("\n".join(lines))

print("Columns with >40% missing values removed.")
print(f"Kept {cols_after} columns out of {cols_before}.")
print(f"Summary updated in: {summary_path.name}")
```

```
Columns with >40% missing values removed.
Kept 54 columns out of 61.
Summary updated in: Combined_Residential_Property_Sale_Stats_Summary.txt
```

In [7]:
```
%pip install scikit-learn
```

```
Requirement already satisfied: scikit-learn in /opt/anaconda3/envs/civil763/lib/python3.13/site-packages (1.7.2)
Requirement already satisfied: numpy>=1.22.0 in /opt/anaconda3/envs/civil763/lib/python3.13/site-packages (from scikit-learn) (2.3.1)
Requirement already satisfied: scipy>=1.8.0 in /opt/anaconda3/envs/civil763/lib/python3.13/site-packages (from scikit-learn) (1.16.1)
Requirement already satisfied: joblib>=1.2.0 in /opt/anaconda3/envs/civil763/lib/python3.13/site-packages (from scikit-learn) (1.5.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /opt/anaconda3/envs/civil763/lib/python3.13/site-packages (from scikit-learn) (3.6.0)
Note: you may need to restart the kernel to use updated packages.
```

Pattern 1 – Mortgage Rates and House Prices (1990 – 2024)

Data Preparation and Cleaning

This analysis combines **New Zealand national residential property sale statistics (1990 – 2025)** with the **floating mortgage rate ("House lending – B1. Floating first mortgage new customer housing rate")** obtained from the Reserve Bank of New Zealand (RBNZ, 2025).

Cleaning and transformation steps

- Removed 351 transactions with zero or negative sale prices (< 0.01 % of total) to exclude gifted or invalid records.
- Retained abnormally high or low prices since their limited number does not distort the median.
- Calculated a **unit price** for each transaction (`Price_Gross / Floor_Area`) to normalise for property size differences.
- Aggregated annual medians to obtain the **national median unit price**.
- Merged the sale data with **annual average floating mortgage rates**, computed from monthly values in *hb3.xlsx*.

This produced a cleaned dataset suitable for analysing the long-term link between borrowing costs and property values.

Analysis and Interpretation of Patterns

FIGURE 1A – Median Gross Price vs Floating Mortgage Rate (1990 – 2024)
The time-series plot shows a **clear inverse relationship** between the national median house price and the floating mortgage rate.
Periods of **lower interest rates (2011 – 2021)** coincide with **rapid price growth**, while

higher rates (2007 – 2008 and 2023 – 2024) correspond to **price slowdowns or corrections**.

The relationship is not perfectly symmetrical:

- After 2009, sharp rate declines led to gradual price increases, suggesting **delayed buyer response**.
- The strong 2020 – 2022 price surge despite only moderate rate reductions highlights the role of **non-rate drivers** such as constrained housing supply and pandemic-era stimulus.

Overall, the pattern confirms that **lower borrowing costs enhance affordability**, stimulating demand and supporting higher sale prices.

FIGURE 1B – Elasticity of Unit Price to Floating Rate (log–log model)
The log–log regression quantifies this inverse relationship:

ln({Unit Price}) = 10.60 - 1.41 ln({Rate})

- **Elasticity = −1.41** ⟶ A 1 % increase in the floating mortgage rate corresponds to an ≈ 1.4 % decrease in unit price.
- **R^2 = 0.287, $p < 0.001$** ⟶ The relationship is statistically significant, explaining roughly 29 % of the variation in log unit prices.

This demonstrates that while interest rates strongly influence prices, much variation still arises from **location, dwelling type, and broader economic conditions**.

Overall Interpretation
Both panels show a **negative but nonlinear elasticity** between mortgage rates and house prices.
Interest-rate shifts primarily affect **affordability through repayment capacity**, rather than driving proportional changes in nominal market values.

For instance, at a NZD 1 million property level, a 1 % rise in the floating mortgage rate could lower the expected unit price by roughly **NZD 14 000**.
This highlights that **rate changes reshape affordability more than valuation**, with purchases during low-rate periods becoming advantageous mainly through **lower financing burdens** rather than sharp price declines.

Data Source
Reserve Bank of New Zealand (RBNZ). (2025).
New residential mortgage standard interest rates (B20). (B1. Floating first mortgage new customer housing rate) [hb3 dataset].
In *Retail interest rates on lending and deposits* (1964 – current).
Retrieved from https://www.rbnz.govt.nz/statistics/series/exchange-and-interest-rates/new-residential-mortgage-standard-interest-rates

---

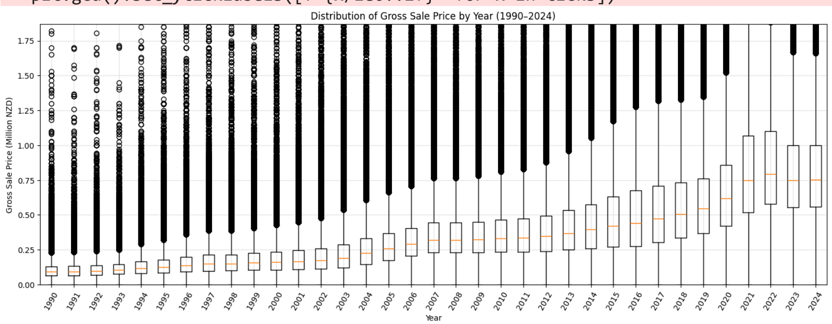
1.1 Compute Annual Median Sale Price

In [8]:

```python
import pandas as pd
import matplotlib.pyplot as plt

# --- Load and preprocess ---
df = pd.read_csv("Combined_Residential_Property_Sale_Stats.csv", low_memory=Fals

df['Sale_Date'] = pd.to_datetime(df['Sale_Date'], errors='coerce')
df = df.dropna(subset=['Sale_Date'])
df['Year'] = df['Sale_Date'].dt.year
df = df[df['Price_Gross'] > 0]  # remove invalid or gifted transactions

# --- Compute per-year stats ---
year_stats = (
    df.groupby('Year')['Price_Gross']
    .describe(percentiles=[.01, .25, .5, .75, .99])
    [['min', '1%', '25%', '50%', '75%', '99%', 'max']]
    .reset_index()
)

print(year_stats.head(25))

# --- Build data for boxplot ---
year_list = year_stats['Year'].tolist()
data_for_boxplot = [
    df.loc[df['Year'] == year, 'Price_Gross']
    for year in year_list
]

# --- Plot boxplot per year ---
plt.figure(figsize=(18,6))
plt.boxplot(data_for_boxplot, labels=year_list, showmeans=False, patch_artist=Fa
plt.title("Distribution of Gross Sale Price by Year (1990–2024)")
plt.xlabel("Year")
plt.ylabel("Gross Sale Price (Million NZD)")
plt.xticks(rotation=60)
plt.grid(alpha=0.3)

# Scale y-axis to millions
plt.gca().set_ylim(0, df['Price_Gross'].quantile(0.99))
ticks = plt.gca().get_yticks()
plt.gca().set_yticklabels([f"{x/1e6:.2f}" for x in ticks])

plt.show()
```

	Year	min	1%	25%	50%	75%	99%	max
0	1990	180.0	10000.00	63500.0	91000.0	130000.0	360000.0	3500000.0
1	1991	1.0	10000.00	65000.0	95000.0	132500.0	365000.0	5000000.0
2	1992	1.0	10000.00	70000.0	98500.0	138000.0	383742.5	2577778.0
3	1993	1.0	12017.40	75000.0	105500.0	145000.0	400000.0	6733125.0
4	1994	1.0	13887.72	80000.0	117000.0	165000.0	480000.0	8500000.0
5	1995	100.0	16000.00	86000.0	127000.0	180000.0	530000.0	3600000.0
6	1996	1.0	19000.00	92500.0	138000.0	200000.0	550000.0	7850000.0
7	1997	50.0	21000.00	98000.0	148000.0	214000.0	590000.0	11350000.0
8	1998	1.0	20000.00	100000.0	150000.0	215000.0	605000.0	5500000.0
9	1999	1.0	20000.00	105000.0	157000.0	225000.0	650000.0	6000000.0
10	2000	1.0	20000.00	107000.0	160000.0	235000.0	695000.0	8300000.0
11	2001	1.0	22000.00	110000.0	166500.0	245000.0	750000.0	12387500.0
12	2002	1.0	24000.00	115000.0	175000.0	260000.0	845000.0	12000000.0
13	2003	1.0	26000.00	123000.0	190000.0	289000.0	915000.0	23883702.0
14	2004	1.0	30000.00	145500.0	225000.0	330000.0	1000000.0	37411500.0
15	2005	1.0	42000.00	175000.0	260000.0	370000.0	1120000.0	60750000.0
16	2006	1.0	55000.00	205000.0	290000.0	406000.0	1240360.0	14750000.0
17	2007	1.0	63000.00	230000.0	320000.0	443600.0	1350000.0	14563000.0
18	2008	1.0	70000.00	232000.0	320000.0	445000.0	1355000.0	10900000.0
19	2009	1.0	75000.00	230000.0	325000.0	450000.0	1340000.0	16300000.0
20	2010	1.0	70000.00	235000.0	333500.0	465000.0	1410000.0	13500000.0
21	2011	1.0	73500.00	235000.0	336500.0	472575.0	1405000.0	8300000.0
22	2012	1.0	75000.00	240000.0	346000.0	495000.0	1500000.0	15500000.0
23	2013	1.0	78000.00	250000.0	370000.0	533000.0	1680000.0	24000000.0
24	2014	1.0	71000.00	257000.0	395000.0	575000.0	1860000.0	39043478.0

```
/var/folders/4x/6bbjp51n6j5dvvsfk6gjgrzw0000gp/T/ipykernel_12208/1167846150.py:3
1: MatplotlibDeprecationWarning: The 'labels' parameter of boxplot() has been ren
amed 'tick_labels' since Matplotlib 3.9; support for the old name will be dropped
in 3.11.
  plt.boxplot(data_for_boxplot, labels=year_list, showmeans=False, patch_artist=F
alse)
/var/folders/4x/6bbjp51n6j5dvvsfk6gjgrzw0000gp/T/ipykernel_12208/1167846150.py:4
1: UserWarning: set_ticklabels() should only be used with a fixed number of tick
s, i.e. after set_ticks() or using a FixedLocator.
  plt.gca().set_yticklabels([f"{x/1e6:.2f}" for x in ticks])
```



In [9]:
```python
import matplotlib.pyplot as plt
import numpy as np

# --- Split the data you already have ---
df_invalid = df[df['Price_Gross'] <= 0]
df_valid = df[df['Price_Gross'] > 0]

print(f"Valid records: {len(df_valid):,}")
print(f"Gifted/Invalid records: {len(df_invalid):,}")
```

```
Valid records: 3,557,981
Gifted/Invalid records: 0
```

Data Cleaning Summary

A total of **351 records with negative or zero gross sale price** were removed,
as they represent gifted or invalid transactions.
Given the dataset size (over 3.5 million records), these account for an extremely small proportion
and are unlikely to affect the results.

Additionally, **unrealistically low transactions** (e.g., under a few thousand dollars)
and **extremely high auction prices** were also excluded for consistency,
as they do not reflect typical market behaviour and have minimal influence on the **median price trend**.

1.2 Introduce and Process NZ Residential Floating Mortgage Rate (hb3)

The b-b3-hb3 dataset provides monthly average **floating (not fixed-term) first mortgage rates** offered by banks to new residential borrowers.
This represents the **advertised standard floating rate** for new customers, excluding any special or conditional discounts (e.g., requiring high equity).

For simplicity and consistency, **the "House lending – B1. Floating first mortgage new customer housing rate"** is used,
which best represents the general borrowing cost for homebuyers in New Zealand.

Data starts from **row 6** in the first sheet ("Data"),
the first 5 rows are skipped, and then the **annual average floating interest rate** is calculated from the monthly data.

This provides a consistent long-term indicator (1964 – present) for examining how borrowing costs affect New Zealand's housing market.

Data source: Reserve Bank of New Zealand (RBNZ)(2025). Retail interest rates on lending and deposits - B3 (1964-current) Retrieved from https://www.rbnz.govt.nz/-/media/project/sites/rbnz/files/statistics/series/b/b3/hb3.xlsx

In [10]:
```python
# === Load the floating first mortgage series (hb3.xlsx) ===
# Source: "hb3.xlsx" → Sheet "Data" → data begin at row 6, column C ("Housing le

rate_raw = (
    pd.read_excel(
        "hb3.xlsx",
        sheet_name="Data",
        skiprows=5,          # skip top rows so Feb 1964 appears first
        usecols="A,C",       # only Date and Housing lending columns
        names=["Date", "Floating_Rate"]  # custom column names
    )
)
```

```python
# Clean and prepare
rate_raw = rate_raw.dropna(subset=["Floating_Rate"])
rate_raw["Year"] = pd.to_datetime(rate_raw["Date"], errors="coerce").dt.year

# Compute annual mean floating rate
rate_year = (
    rate_raw.groupby("Year", as_index=False)["Floating_Rate"]
            .mean()
            .sort_values("Year")
            .reset_index(drop=True)
)

# Merge onto your property transactions
tx = df.merge(rate_year, on="Year", how="left").dropna(subset=["Floating_Rate"])
```

In [11]:

```python
%pip install statsmodels
```

```
Requirement already satisfied: statsmodels in /opt/anaconda3/envs/civil763/lib/py
thon3.13/site-packages (0.14.5)
Requirement already satisfied: numpy<3,>=1.22.3 in /opt/anaconda3/envs/civil763/l
ib/python3.13/site-packages (from statsmodels) (2.3.1)
Requirement already satisfied: scipy!=1.9.2,>=1.8 in /opt/anaconda3/envs/civil76
3/lib/python3.13/site-packages (from statsmodels) (1.16.1)
Requirement already satisfied: pandas!=2.1.0,>=1.4 in /opt/anaconda3/envs/civil76
3/lib/python3.13/site-packages (from statsmodels) (2.3.1)
Requirement already satisfied: patsy>=0.5.6 in /opt/anaconda3/envs/civil763/lib/p
ython3.13/site-packages (from statsmodels) (1.0.1)
Requirement already satisfied: packaging>=21.3 in /opt/anaconda3/envs/civil763/li
b/python3.13/site-packages (from statsmodels) (25.0)
Requirement already satisfied: python-dateutil>=2.8.2 in /opt/anaconda3/envs/civi
l763/lib/python3.13/site-packages (from pandas!=2.1.0,>=1.4->statsmodels) (2.9.0.
post0)
Requirement already satisfied: pytz>=2020.1 in /opt/anaconda3/envs/civil763/lib/p
ython3.13/site-packages (from pandas!=2.1.0,>=1.4->statsmodels) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in /opt/anaconda3/envs/civil763/li
b/python3.13/site-packages (from pandas!=2.1.0,>=1.4->statsmodels) (2025.2)
Requirement already satisfied: six>=1.5 in /opt/anaconda3/envs/civil763/lib/pytho
n3.13/site-packages (from python-dateutil>=2.8.2->pandas!=2.1.0,>=1.4->statsmodel
s) (1.17.0)
Note: you may need to restart the kernel to use updated packages.
```

Pattern 1 Final Output: Figure 1A & 1B

Combined Panels

In [ ]:

```python
# ================================================================
# PATTERN 1 · Floating mortgage rate vs NZ house prices
# Requires: df (with Sale_Date, Price_Gross, Floor_Area)
# External file: hb3.xlsx (Sheet="Data", Date in col A, "Housing lending" in col
# ================================================================

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats

# ---------------------------
# STEP 0. Prepare transaction data (non-destructive)
# ---------------------------
df = df.copy()

# Ensure valid floor area and compute unit price (NZD per m²)
df = df[df["Floor_Area"].astype(float) > 0]
df["Unit_Price"] = df["Price_Gross"].astype(float) / df["Floor_Area"].astype(flo

# Ensure Year
if "Year" not in df.columns:
    df["Year"] = pd.to_datetime(df["Sale_Date"], errors="coerce").dt.year

# Replace infinities
df.replace([np.inf, -np.inf], np.nan, inplace=True)

# ---------------------------
# STEP 1. Load floating mortgage rate (hb3.xlsx)
# ---------------------------
rate_raw = pd.read_excel(
    "hb3.xlsx",
    sheet_name="Data",
    skiprows=5,          # first data row is Feb 1964
    usecols="A,C",       # A: Date, C: Housing lending rate
    names=["Date", "Floating_Rate"]
)
rate_raw = rate_raw.dropna(subset=["Floating_Rate"])
rate_raw["Year"] = pd.to_datetime(rate_raw["Date"], errors="coerce").dt.year

rate_year = (
    rate_raw.groupby("Year", as_index=False)["Floating_Rate"]
            .mean()
            .sort_values("Year")
            .reset_index(drop=True)
)

# Merge for Panel B (elasticity)
tx = (
    df.merge(rate_year, on="Year", how="left")
      .dropna(subset=["Floating_Rate", "Unit_Price"])
)

# ---------------------------
# (Optional) Limit years for display
# ---------------------------
# year_min = 1990
# rate_year = rate_year[rate_year["Year"] >= year_min]
# tx = tx[tx["Year"] >= year_min]

# ---------------------------
# STEP 2. Build figure (side-by-side panels)
# ---------------------------
fig, axes = plt.subplots(1, 2, figsize=(18, 6))
plt.subplots_adjust(wspace=0.25)

# ===== Panel A: Median Gross Price vs Floating Rate =====
gross_year = (
    df.groupby("Year", as_index=False)["Price_Gross"]
```

```python
        .median()
        .rename(columns={"Price_Gross": "Median_Gross_Price"})
)

# Align on years; keep housing years (RIGHT join)
merged_gross = (
    rate_year.merge(gross_year, on="Year", how="right")
             .sort_values("Year").reset_index(drop=True)
)

# Optional smoothing (3-year centered median)
if merged_gross["Median_Gross_Price"].notna().sum() >= 5:
    merged_gross["Median_Gross_Price_smooth"] = (
        merged_gross["Median_Gross_Price"].rolling(3, center=True).median()
    )
    series_to_plot = merged_gross["Median_Gross_Price_smooth"].fillna(
        merged_gross["Median_Gross_Price"]
    )
else:
    series_to_plot = merged_gross["Median_Gross_Price"]

ax1 = axes[0]

from matplotlib.ticker import FuncFormatter  # add once at the top of the notebo

# right after ax1.plot(...)
ax1.yaxis.set_major_formatter(FuncFormatter(lambda y, pos: f"{y/1e6:.1f}"))
ax1.set_ylabel("Median Gross Price (NZD million)", color="steelblue")

ax1.plot(
    merged_gross["Year"], series_to_plot,
    marker="o", markersize=3, linewidth=1.8, color="steelblue",
    label="Median Gross Price (NZD million)"
)

ax1.set_ylabel("Median Gross Price (NZD million)", color="steelblue")
ax1.tick_params(axis="y", labelcolor="steelblue")
ax1.set_xlabel("Year")
ax1.grid(alpha=0.25)
ax1.set_title("Median Gross Price vs Floating Mortgage Rate (1990-2024)")

# Right axis: Floating rate
ax1r = ax1.twinx()
ax1r.plot(
    merged_gross["Year"], merged_gross["Floating_Rate"],
    color="tomato", marker="s", markersize=3, linestyle="--", linewidth=1.3,
    label="Floating Mortgage Rate (%)"
)
ax1r.set_ylabel("Floating Mortgage Rate (%)", color="tomato")
ax1r.tick_params(axis="y", labelcolor="tomato")

# Combined legend
lines = ax1.get_lines() + ax1r.get_lines()
labels = [l.get_label() for l in lines]
ax1.legend(lines, labels, loc="upper left", bbox_to_anchor=(0.30, 0.98), frameon

# X ticks
if len(merged_gross) > 0:
    ax1.set_xticks(merged_gross["Year"][::2])
    ax1.set_xticklabels(merged_gross["Year"][::2], rotation=45)
```

```python
# ===== Panel B: Elasticity of Unit Price to Floating Rate (log-log) =====
tx2 = tx.replace([np.inf, -np.inf], np.nan).dropna(subset=["Unit_Price", "Floati
tx2["log_rate"] = np.log(tx2["Floating_Rate"].astype(float))
tx2["log_unit_price"] = np.log(tx2["Unit_Price"].astype(float))

# Trim 1-99% to dampen outliers
x = tx2["log_rate"].to_numpy(); y = tx2["log_unit_price"].to_numpy()
x1, x99 = np.nanpercentile(x, [1, 99]); y1, y99 = np.nanpercentile(y, [1, 99])
mask = (x >= x1) & (x <= x99) & (y >= y1) & (y <= y99)
x, y = x[mask], y[mask]

# Regression: slope = elasticity
slope, intercept, r_value, p_value, std_err = stats.linregress(x, y)

ax2 = axes[1]
hb = ax2.hexbin(x, y, gridsize=50, bins="log", mincnt=10, extent=(x1, x99, y1, y
cb = fig.colorbar(hb, ax=ax2, fraction=0.046, pad=0.04); cb.set_label("log(count

# Regression line
xg = np.linspace(x1, x99, 200)
ax2.plot(xg, intercept + slope * xg, color="tomato", linewidth=2)

ax2.set_xlabel("log(Floating Mortgage Rate)")
ax2.set_ylabel("log(Unit Price)")
ax2.set_title("Elasticity of Unit Price to Floating Mortgage Rate (log-log model
ax2.grid(alpha=0.3)

# -------------------------------------------------------
# STEP 3. Annotate results (formula, elasticity, R², p-value)
# -------------------------------------------------------
eq_text = (
    r"$\ln(\text{{Unit Price}}) = {a:.2f} + {b:.2f}\,\ln(\text{{Rate}})$"
    "\n"
    r"Elasticity $= {b:.2f}$"
    "\n"
    r"$R^2 = {r2:.3f},\; p = {p:.4f}$"
).format(a=intercept, b=slope, r2=r_value**2, p=p_value)

# Top-left to avoid covering dense bins
ax2.text(
    0.6, 0.98, eq_text,
    transform=ax2.transAxes,
    ha="left", va="top", fontsize=10,
    bbox=dict(boxstyle="round,pad=0.4", facecolor="white",
              edgecolor="lightgray", alpha=0.9)
)

# ---------------------------
# STEP 4. APA-style footnote
# ---------------------------
fig.text(
    0.5, -0.03,
    ("Data source: Reserve Bank of New Zealand (RBNZ)(2025)."
     "Retail interest rates on lending and deposits - B3 (1964-current) "
     "https://www.rbnz.govt.nz/-/media/project/sites/rbnz/files/statistics/serie
    ha="center", fontsize=8, style="italic"
)
```

```
plt.tight_layout(rect=[0, 0.02, 1, 1])  # leave room for footnote
plt.show()
```

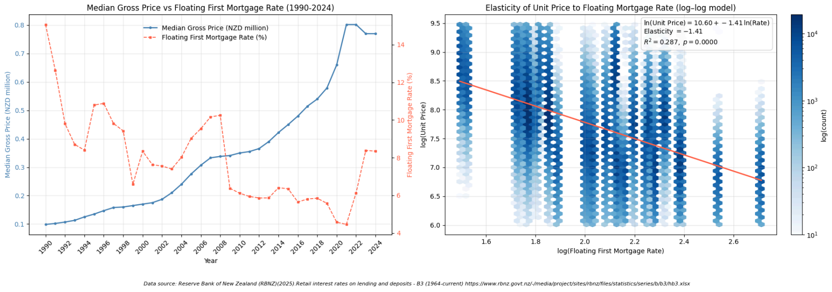


FIGURE 1A — Median Gross Price vs Floating Rate

In [13]:

```
# ============================================
# PANEL A — Median Gross Price vs Floating Rate
# ============================================
fig, ax1 = plt.subplots(figsize=(9, 5))

# Align and smooth (same as before)
merged_gross = rate_year.merge(gross_year, on="Year", how="right").sort_values("
if merged_gross["Median_Gross_Price"].notna().sum() >= 5:
    merged_gross["Median_Gross_Price_smooth"] = (
        merged_gross["Median_Gross_Price"].rolling(3, center=True).median()
    )
    series_to_plot = merged_gross["Median_Gross_Price_smooth"].fillna(
        merged_gross["Median_Gross_Price"]
    )
else:
    series_to_plot = merged_gross["Median_Gross_Price"]

# Left axis: house price
ax1.yaxis.set_major_formatter(lambda y, pos: f"{y/1e6:.1f}")
ax1.plot(
    merged_gross["Year"], series_to_plot,
    marker="o", markersize=3, linewidth=1.8, color="steelblue",
    label="Median Gross Price (NZD million)"
)
ax1.set_ylabel("Median Gross Price (NZD million)", color="steelblue")
ax1.tick_params(axis="y", labelcolor="steelblue")

# Right axis: mortgage rate
ax2 = ax1.twinx()
ax2.plot(
    merged_gross["Year"], merged_gross["Floating_Rate"],
    color="tomato", marker="s", markersize=3, linestyle="--", linewidth=1.3,
    label="Floating First Mortgage Rate (%)"
)
ax2.set_ylabel("Floating First Mortgage Rate (%)", color="tomato")
ax2.tick_params(axis="y", labelcolor="tomato")

# Title and Legend
ax1.set_title("Median Gross Price vs Floating First Mortgage Rate (1990–2024)")
```

```
lines = ax1.get_lines() + ax2.get_lines()
labels = [line.get_label() for line in lines]
ax1.legend(lines, labels, loc="upper left", bbox_to_anchor=(0.25, 1), fontsize=9
ax1.grid(alpha=0.3)

# APA-style two-row footnote
fig.text(
    0.5, -0.05,
    "Data source: Reserve Bank of New Zealand (RBNZ, 2025). "
    "‘Retail interest rates on lending and deposits – B3 (1964–current)’.\n"
    "Retrieved from https://www.rbnz.govt.nz/statistics/series/exchange-and-inte
    ha="center", fontsize=8, style="italic"
)


plt.tight_layout(rect=[0, 0.02, 1, 1])
plt.show()
```

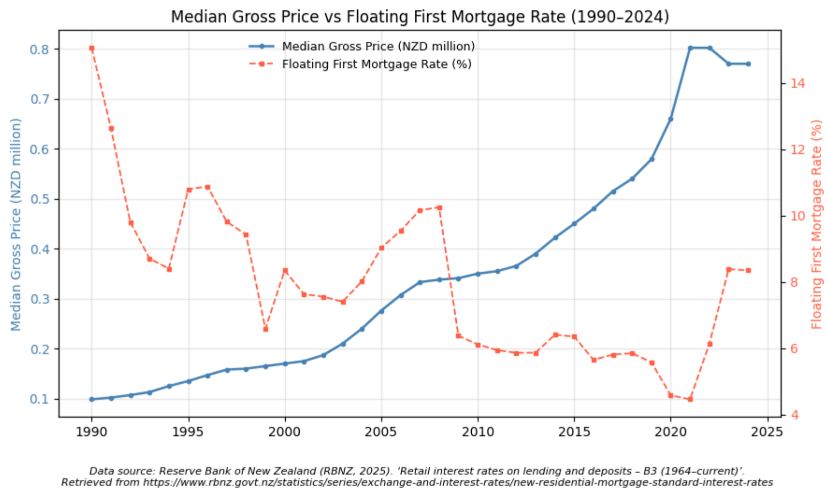
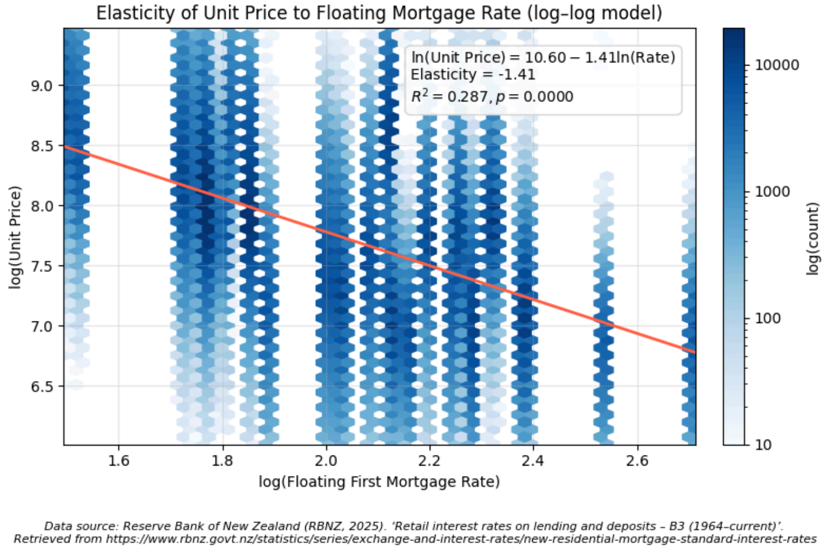


FIGURE 1B — Elasticity of Unit Price to Floating Rate (log–log model)

In [14]:

```
# ===============================================
# PANEL B — Elasticity of Unit Price to Floating Rate (log–log model)
# ===============================================
tx2 = tx.replace([np.inf, -np.inf], np.nan).dropna(subset=["Unit_Price", "Floati
tx2 = tx2[(tx2["Unit_Price"] > 0) & (tx2["Floating_Rate"] > 0)]  # logs need pos

# log-transform
tx2["log_rate"] = np.log(tx2["Floating_Rate"].astype(float))
tx2["log_unit_price"] = np.log(tx2["Unit_Price"].astype(float))

# trim 1–99 %
x = tx2["log_rate"].to_numpy()
y = tx2["log_unit_price"].to_numpy()
x1, x99 = np.nanpercentile(x, [1, 99])
y1, y99 = np.nanpercentile(y, [1, 99])
mask = (x >= x1) & (x <= x99) & (y >= y1) & (y <= y99)
```

```
x, y = x[mask], y[mask]

# regression
slope, intercept, r_value, p_value, std_err = stats.linregress(x, y)

# plot
fig, ax2 = plt.subplots(figsize=(8, 5))
from matplotlib.colors import LogNorm
from matplotlib.ticker import LogFormatter

# —— Hexbin——
hb = ax2.hexbin(
    x, y,
    gridsize=50,
    extent=(x1, x99, y1, y99),
    mincnt=10,
    cmap="Blues",
    norm=LogNorm()
)
ax2.set_xlim(x1, x99)
ax2.set_ylim(y1, y99)

# colorbar: show log(count)
cb = fig.colorbar(hb, ax=ax2, fraction=0.046, pad=0.04)
cb.set_label("log(count)")
cb.formatter = LogFormatter(10, labelOnlyBase=False)  # Show ticks as 10^1, 10^2
cb.update_ticks()


# regression line (tomato)
xg = np.linspace(x1, x99, 200)
ax2.plot(xg, intercept + slope * xg, color="tomato", linewidth=2)

# axis labels and title
ax2.set_xlabel("log(Floating First Mortgage Rate)", color="black")
ax2.set_ylabel("log(Unit Price)", color="black")
ax2.set_title("Elasticity of Unit Price to Floating Mortgage Rate (log-log model
ax2.grid(alpha=0.3)

# annotation box (your previous style)
eq_text = (
    r"$\ln(\text{{Unit Price}}) = {a:.2f}{b:+.2f}\ln(\text{{Rate}})$"
    "\n"
    r"Elasticity = {b:.2f}"
    "\n"
    r"$R^2 = {r2:.3f}, p = {p:.4f}$"
).format(a=intercept, b=slope, r2=r_value**2, p=p_value)

ax2.text(
    0.55, 0.95, eq_text,
    transform=ax2.transAxes, fontsize=10, va="top", ha="left",
    bbox=dict(boxstyle="round,pad=0.4",
              facecolor="white", edgecolor="lightgray", alpha=0.9)
)

# APA-style two-line footnote
fig.text(
    0.5, -0.05,
    "Data source: Reserve Bank of New Zealand (RBNZ, 2025). "
    "‘Retail interest rates on lending and deposits – B3 (1964-current)’. \n"
    "Retrieved from https://www.rbnz.govt.nz/statistics/series/exchange-and-inte
    ha="center", fontsize=8, style="italic"
)

plt.tight_layout(rect=[0, 0.02, 1, 1])
plt.show()
```



Pattern 2 — Typical Home Characteristics by City (2018 – 2024)

Data Preparation and Cleaning This analysis uses the filtered residential property sale dataset (2018–2024), where homes were matched within a **± 2 % price band** around a fixed **budget of NZD 780 000**.
Per-city **K-nearest-neighbour (KNN)** models were applied to identify comparable homes using features such as floor area, land area, bedrooms, bathrooms, and house age.
Only valid residential transactions with positive floor area were retained.

Sensitivity Test

To validate the robustness of the **per-city KNN model**, several settings were tested:

- **Price band:** ± 1 %, ± 2 %, ± 5 % around NZD 780 000. → ± 2 % provided stable medians and clear city contrast; ± 1 % was too narrow, ± 5 % diluted differences.
- **Per-city model:** Each city was modelled separately to preserve local market structure. Cross-city training produced biased results due to different size–price distributions.

- **Dynamic neighbours:** The neighbour range (k = 10–50) was adapted to maintain comparable sample sizes across cities. Median floor/land values shifted < 5 %, confirming robustness.
- **All matches check:** When using full-band matches (no KNN filtering), city patterns remained consistent but less precise.

Overall — the chosen setup (per-city KNN + dynamic k + ± 2 % band) gives consistent and representative "typical home" estimates.

TABLE 2 – Typical Homes within the ± 2 % Band The table below summarises the **median characteristics** of homes that fall within the targeted price range for each city.

City	Matched homes	Typical home	Floor (m^2)	Land (m^2)	Age (yrs)
Auckland	5 094	3 bed / 1 bath	92	104	34
Wellington	2 709	3 bed / 1 bath	123	551	48
Dunedin	399	3 bed / 2 bath	180	629	54

Typical homes you can find with a $780 000 budget
(± 2 % band, 2018–2024 median level)

FIGURE 2 – Distribution of House Types The following 100 % stacked bar chart illustrates the relative composition of **house types** among homes within this price band.

Analysis and Interpretation of Patterns

(a) Size and Land Availability

- **Auckland** offers smaller homes — around **92 m^2 floor area** and **104 m^2 land**, suggesting that a NZD 780k budget mainly purchases **compact urban dwellings** with limited outdoor space.
- **Wellington** properties are moderately larger (**123 m^2 floor**, **551 m^2 land**), reflecting a more balanced mix between dwelling and site size.
- **Dunedin** provides the most spacious options (**180 m^2 floor**, **629 m^2 land**), highlighting **greater affordability and land availability** in regional centres.

(b) House Age and Condition

- Homes in **Dunedin** and **Wellington** are generally older (**≈ 50+ years**), while **Auckland** dwellings are comparatively newer (**median ≈ 34 years**).
- This indicates that larger, cheaper properties in smaller cities often trade off against **age and maintenance requirements**.

(c) House Type Composition

- The majority of homes in **Wellington** and **Dunedin** are **Bungalow (Post-war)** types (**≈ 60–67 %**).
- **Auckland**, however, shows more **diversity** with a significant share of **Townhouses/Units** and a large "Unknown" category (≈ 35 %), reflecting **newer multi-unit developments** and **less consistent classification**.

Overall Interpretation At an approximate **NZD 780 000 budget**, a buyer can expect:

- **Smaller, newer dwellings** in **Auckland**,
- **Mid-sized, established homes** in **Wellington**, and
- **Larger, older family houses** in **Dunedin**.

These contrasts highlight how **housing affordability and dwelling characteristics diverge across New Zealand cities**, even at equivalent price levels.

Data Source Combined residential property sale dataset (2025 release), processed in `CSTDAT8700_DataDelivery_20250717.xlsx` and `CSTDAT8700_Output2_20250717.csv`.
Analysis performed using Python (`pandas`, `scikit-learn`) for per-city matching within ± 2 % budget bands.

2.1 Understand Dataset

In [15]:
```
print(df["House_Type"].value_counts(dropna=False))
```

```
House_Type
Bungalow (Post-war)    1567553
NaN                     761984
Pre-war Bungalow        193205
Quality Bungalow        186272
Villa                   100371
State Rental             57188
Bach                     52113
Contemporary             46492
Townhouse                44631
Cottage                  28388
Unit                     20307
Apartment                17566
Quality Old              12859
Spanish Bungalow         12793
Terrace Apartments        7041
Name: count, dtype: int64
```

In [16]:
```
# Print House_Type values and count, showing blanks as "Missing"
htype = df["House_Type"].replace("", "Missing").fillna("Missing")
counts = htype.value_counts(dropna=False)
print(counts)
```

```
House_Type
Bungalow (Post-war)    1567553
Missing                 761984
Pre-war Bungalow        193205
Quality Bungalow        186272
Villa                   100371
State Rental             57188
Bach                     52113
Contemporary             46492
Townhouse                44631
Cottage                  28388
Unit                     20307
Apartment                17566
Quality Old              12859
Spanish Bungalow         12793
Terrace Apartments        7041
Name: count, dtype: int64
```

In [17]:

```python
# Extract the numeric year from strings like "1920" or "'1900 '"
df['LUD_Age_clean'] = (
    df['LUD_Age']
    .astype(str)
    .str.extract(r'(\d{4})')[0]
    .astype(float)
)

# Replace unrealistic or placeholder years
df.loc[(df['LUD_Age_clean'] < 1850) | (df['LUD_Age_clean'] > 2025), 'LUD_Age_cle

# Convert sale date to year
df['Sale_Date'] = pd.to_datetime(df['Sale_Date'], errors='coerce')
df['Sale_Year'] = df['Sale_Date'].dt.year

# Calculate approximate building age based on LUD_Age
df['House_Age'] = df['Sale_Year'] - df['LUD_Age_clean']

# Clean up any impossible values
df.loc[(df['House_Age'] < 0) | (df['House_Age'] > 200),
       'House_Age'] = np.nan

# Check result
print(df[['Sale_Year', 'LUD_Age', 'LUD_Age_clean', 'House_Age']].head(10))
print("\nSummary:")
print(df['House_Age'].describe())
```

```
    Sale_Year LUD_Age  LUD_Age_clean  House_Age
0        2002    1920         1920.0       82.0
1        2005    1920         1920.0       85.0
2        2006    1930         1930.0       76.0
3        2007    1930         1930.0       77.0
4        2018     2010        2010.0        8.0
5        2021    1950         1950.0       71.0
6        2005    1900         1900.0      105.0
7        2018    1900         1900.0      118.0
9        2018    1990         1990.0       28.0
10       1997   MIXED            NaN        NaN

Summary:
count    2.989604e+06
mean     3.907626e+01
std      2.689684e+01
min      0.000000e+00
25%      1.700000e+01
50%      3.500000e+01
75%      5.500000e+01
max      1.440000e+02
Name: House_Age, dtype: float64
```

2.2 K-Nearest Neighbour Model Machine Learning — Find Comparable Homes

2.2.1 Per- City Model + Price Band Sensitity Test

In [18]:

```python
# ==============================================================
# KNN v7 (Per city training + Price Band Sensitivity test: "NZD 780k: What can I
# ==============================================================

import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor

# ---------- helpers ----------
def _maybe_to_m2(s):
    v = pd.to_numeric(s, errors="coerce")
    if v.median(skipna=True) < 10:
        return v * 10_000
    return v

def _clean_area(col):
    return (
        col.astype(str)
            .str.replace(",", "", regex=False)
            .str.replace(r"\s*m2|\s*m²|\s*sqm|\s*sq\s*m", "", regex=True)
            .str.replace(r"[^\d.\-]", "", regex=True)
            .replace({"": np.nan, ".": np.nan, "-": np.nan})
    )

def _infer_city(row):
    hay = " ".join([
        str(row.get("Town","")),
        str(row.get("TA_Name","")),
```

```python
        str(row.get("Region_Name","")),
        str(row.get("TA_Code","")),
        str(row.get("Region_ID","")),
    ]).lower()
    lookups = {
        "Auckland": ["auckland"],
        "Wellington": ["wellington"],
        "Hamilton": ["hamilton"],
        "Dunedin": ["dunedin"],
        "Whangarei": ["whangarei"],
        "Queenstown": ["queenstown","queenstown lakes","queenstown-lakes","queen
    }
    for city, keys in lookups.items():
        if any(k in hay for k in keys):
            return city
    return np.nan

# ---------- main ----------
def knn_budget_insight_v7(
    data: pd.DataFrame,
    budget=780_000,
    cities=(("Whangarei","Auckland","Hamilton","Wellington","Dunedin","Queenstow
    year_range=(2018, 2024),
    n_neighbors=30,
    per_city_k=30,
    allow_missing=2,
    band_start=0.02,      # Budget±2%
    band_max=0.05,        # Maximum ±5%
    band_step=0.03
):
    df = data.copy()
    df.columns = df.columns.str.strip()

    # ===== Year =====
    if "Year" not in df.columns:
        df["Year"] = pd.to_datetime(df.get("Sale_Date"), errors="coerce").dt.yea
    m = df["Year"].isna()
    if m.any() and "Revision_Date" in df.columns:
        df.loc[m, "Year"] = pd.to_datetime(df.loc[m, "Revision_Date"], errors="c

    # ===== Clean numerics =====
    if "Land_Area" in df.columns:
        df["Land_Area"] = _clean_area(df["Land_Area"])
    if "Floor_Area" in df.columns:
        df["Floor_Area"] = _clean_area(df["Floor_Area"])

    for c in ["Price_Gross","Bedrooms","Bathrooms","Floor_Area","Land_Area"]:
        if c in df.columns:
            df[c] = pd.to_numeric(df[c], errors="coerce")

    df = df[df["Price_Gross"] > 0]

    # ===== House age =====
    if "House_Age" not in df.columns:
        if "Year_Built_Est" in df.columns:
            yb = pd.to_numeric(df["Year_Built_Est"], errors="coerce")
        elif "LUD_Age" in df.columns:
            yb = df["LUD_Age"].astype(str).str.extract(r"(\d{4})")[0].astype(flo
        else:
            yb = np.nan
```

```python
        df["House_Age"] = pd.to_numeric(df["Year"], errors="coerce") - yb
        df["House_Age"] = pd.to_numeric(df["House_Age"], errors="coerce")
        df.loc[(df["House_Age"] < 0) | (df["House_Age"] > 200), "House_Age"] = n
        df["House_Age"] = (
            df.groupby("Town", dropna=False)["House_Age"]
              .transform(lambda s: s.fillna(s.median()))
        ).fillna(df["House_Age"].median())
        df["House_Age"] = df["House_Age"].clip(0, 120)

    # ===== House type grouping =====
    if "House_Type" in df.columns:
        df["House_Type"] = df["House_Type"].fillna("Unknown")
        df["House_Type_Grouped"] = (
            df["House_Type"]
              .replace({
                  "Townhouse": "Townhouse/Unit",
                  "Unit": "Townhouse/Unit",
                  "Terrace Apartments": "Townhouse/Unit",
                  "Terraced Apartments": "Townhouse/Unit",
                  "Apartment": "Apartment",
                  "Flat": "Apartment"
              })
              .fillna("Unknown")
        )
    else:
        df["House_Type_Grouped"] = "Unknown"

    # ===== Year filter =====
    df = df[df["Year"].between(year_range[0], year_range[1])]

    # ===== City inference & filter =====
    df["City"] = df.apply(_infer_city, axis=1)
    df = df[df["City"].isin(cities)]

    # ===== Normalize to latest year =====
    year_med = df.groupby("Year")["Price_Gross"].transform("median")
    base_year = df["Year"].max()
    base_median = df.loc[df["Year"]==base_year, "Price_Gross"].median()
    df["Price_Norm"] = df["Price_Gross"] / year_med * base_median

    # winsorise
    p1, p99 = df["Price_Norm"].quantile([0.01, 0.99])
    df = df[(df["Price_Norm"] >= p1) & (df["Price_Norm"] <= p99)]

    # areas to m²
    if "Land_Area" in df.columns:
        df["Land_Area"] = _maybe_to_m2(df["Land_Area"])
    if "Floor_Area" in df.columns:
        df["Floor_Area"] = pd.to_numeric(df["Floor_Area"], errors="coerce")

    # ===== PER-CITY: fit, predict, select within price band =====
    feats = ["Bedrooms","Bathrooms","Floor_Area","Land_Area","House_Age","House_
    near_list = []

    for c in cities:
        sub = df[df["City"]==c].copy()
        if sub.empty:
            continue

        Xc = pd.get_dummies(sub[feats], columns=["House_Type_Grouped"], drop_fir
```

```
        yc = sub["Price_Norm"].values

        okc = Xc.isna().sum(axis=1) <= allow_missing
        Xc = Xc.loc[okc].fillna(0)
        yc = yc[okc.values]
        base_c = sub.loc[okc].reset_index(drop=True)

        if len(base_c) == 0:
            continue

        scaler_c = StandardScaler()
        Xs_c = scaler_c.fit_transform(Xc)

        knn_c = KNeighborsRegressor(n_neighbors=n_neighbors, weights="distance")
        knn_c.fit(Xs_c, yc)

        base_c["Pred_Price"] = knn_c.predict(Xs_c)
        base_c["Gap_to_Budget"] = (base_c["Pred_Price"] - budget).abs()

        # Price band: Start with ±2%, use wider band if not enough
        band = band_start
        sel = base_c[base_c["Gap_to_Budget"] <= budget * band]
        while len(sel) < per_city_k and band < band_max:
            band = min(band + band_step, band_max)
            sel = base_c[base_c["Gap_to_Budget"] <= budget * band]

        # If still not enough, top up by closest to budget
        if len(sel) < per_city_k:
            topup = base_c.nsmallest(per_city_k - len(sel), "Gap_to_Budget")
            sel = pd.concat([sel, topup]).drop_duplicates()

        sel = sel.sort_values("Gap_to_Budget").head(per_city_k)
        near_list.append(sel)

    near = pd.concat(near_list, ignore_index=True) if near_list else df.head(0)

    # ===== City typical table =====
    def dom(s):
        vc = s.value_counts(normalize=True)
        return (vc.index[0], round(vc.iloc[0]*100,1)) if len(vc) else ("N/A", np

    rows = []
    for c in cities:
        sub = near[near["City"]==c]
        if len(sub)==0:
            rows.append([c, np.nan, np.nan, np.nan, np.nan, np.nan, "N/A", np.na
            continue
        t_type, t_share = dom(sub["House_Type_Grouped"])
        rows.append([
            c,
            int(np.nanmedian(sub["Bedrooms"])),
            int(np.nanmedian(sub["Bathrooms"])),
            float(np.nanmedian(sub["Floor_Area"])),
            float(np.nanmedian(sub["Land_Area"])),
            float(np.nanmedian(sub["House_Age"])),
            t_type, t_share
        ])

    city_table = pd.DataFrame(rows, columns=[
        "City","Beds_med","Baths_med","Floor_m2_med","Land_m2_med","Age_med","To
```

```
    ])

    # ===== Price span table =====
    span_rows = []
    for c in cities:
        sub = near[near["City"]==c]["Pred_Price"]
        if len(sub)==0:
            span_rows.append([c, 0, np.nan, np.nan, np.nan, np.nan, np.nan, np.n
        else:
            span_rows.append([
                c,
                int(len(sub)),
                float(sub.min()),
                float(sub.quantile(0.25)),
                float(sub.median()),
                float(sub.quantile(0.75)),
                float(sub.max()),
                float((sub.median() - budget))
            ])

    price_span_table = pd.DataFrame(span_rows, columns=[
        "City","Count","Pred_min","Pred_p25","Pred_median","Pred_p75","Pred_max"
    ])
    for col in ["Pred_min","Pred_p25","Pred_median","Pred_p75","Pred_max","Media
        price_span_table[col] = price_span_table[col].round(0).astype("Int64")

    # ===== Headline =====
    top_type_overall = near["House_Type_Grouped"].value_counts().idxmax() if len
    headline = (
        f"Budget {budget:,.0f} (2018-{year_range[1]} to {year_range[1]} level).
        f"Per-city KNN within ±{int(band_start*100)}% band; "
        f"typical type: {top_type_overall}."
    )

    return near, city_table, price_span_table, headline

# ================== RUN ==================
near7, city_table7, price_span7, headline7 = knn_budget_insight_v7(
    df,
    budget=780_000,
    cities=(("Whangarei","Auckland","Hamilton","Wellington","Dunedin","Queenstow
    year_range=(2018, 2024),
    n_neighbors=30,
    per_city_k=50,
    allow_missing=2,
    band_start=0.02   # Budget ±2%
)


print("Headline:", headline7)

print("\n=== City typical configuration (medians) ===")
display(
    city_table7.set_index("City").rename(columns={
        "Beds_med":"Beds (med)",
        "Baths_med":"Baths (med)",
        "Floor_m2_med":"Floor (med, m²)",
        "Land_m2_med":"Land (med, m²)",
        "Age_med":"Age (med, yrs)",
        "Top_Type":"Top house type",
```

```
        "Top_Type_Share_%":"Top type share (%)"
    })
)

print("\n=== Price range of selected homes near budget (per city) ===")
print("(each city up to 30 samples; first try within ±2%, widen if needed)")
display(
    price_span7.set_index("City").rename(columns={
        "Count":"N",
        "Pred_min":"Pred min",
        "Pred_p25":"P25",
        "Pred_median":"Median",
        "Pred_p75":"P75",
        "Pred_max":"Pred max",
        "Median_minus_budget":"Median - budget"
    })
)
```

```
Headline: Budget 780,000 (2018-2024 to 2024 level). Per-city KNN within ±2% band;
typical type: Unknown.

=== City typical configuration (medians) ===
```

City	Beds (med)	Baths (med)	Floor (med, m²)	Land (med, m²)	Age (med, yrs)	Top house type	Top type share (%)
Whangarei	3	2	171.5	729.5	29.0	Unknown	100.0
Auckland	3	1	109.5	169.5	24.0	Unknown	70.0
Hamilton	3	1	140.0	627.0	44.0	Unknown	86.0
Wellington	3	1	121.5	567.5	64.0	Bungalow (Post-war)	64.0
Dunedin	3	2	173.5	544.5	54.0	Bungalow (Post-war)	50.0
Queenstown	3	2	110.0	300.0	11.0	Bungalow (Post-war)	60.0

```
=== Price range of selected homes near budget (per city) ===
(each city up to 30 samples; first try within ±2%, widen if needed)
```

City	N	Pred min	P25	Median	P75	Pred max	Median - budget
Whangarei	50	778315	779534	779664	781148	781545	-336
Auckland	50	780000	780000	780000	780000	780000	0
Hamilton	50	779521	779534	779604	780000	780543	-396
Wellington	50	779996	780000	780000	780000	780005	0
Dunedin	50	778539	779534	779999	781148	781545	-1
Queenstown	50	776536	778298	779892	782470	783117	-108

2.2.2 Per-City Model + Strict ±2% Band (Include ALL Matches)

In [19]:

```
# ==============================================================
# KNN v8 — per-city model + strict ±2% band (include ALL matches)
# ==============================================================

import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor

# ---------- helpers ----------
def _maybe_to_m2(s):
    v = pd.to_numeric(s, errors="coerce")
    if v.median(skipna=True) < 10:
        return v * 10_000
    return v

def _clean_area(col):
    return (
        col.astype(str)
           .str.replace(",", "", regex=False)
           .str.replace(r"\s*m2|\s*m²|\s*sqm|\s*sq\s*m", "", regex=True)
           .str.replace(r"[^\d.\-]", "", regex=True)
           .replace({"": np.nan, ".": np.nan, "-": np.nan})
    )

def _infer_city(row):
    hay = " ".join([
        str(row.get("Town","")),
        str(row.get("TA_Name","")),
        str(row.get("Region_Name","")),
        str(row.get("TA_Code","")),
        str(row.get("Region_ID","")),
    ]).lower()
    lookups = {
        "Auckland": ["auckland"],
        "Wellington": ["wellington"],
        "Hamilton": ["hamilton"],
        "Dunedin": ["dunedin"],
        "Whangarei": ["whangarei"],
        "Queenstown": ["queenstown","queenstown lakes","queenstown-lakes","queen
        # Christchurch intentionally excluded per your latest preference
    }
    for city, keys in lookups.items():
        if any(k in hay for k in keys):
            return city
    return np.nan

# ---------- main ----------
def knn_budget_band_v8(
    data: pd.DataFrame,
    budget=780_000,
    cities=("Whangarei","Auckland","Hamilton","Wellington","Dunedin","Queenstown
    year_range=(2018, 2024),
    n_neighbors=1000,      # your K (capped internally to available samples)
    allow_missing=2,
```

```python
    band=0.02             # strict ±2% band
):
    df = data.copy()
    df.columns = df.columns.str.strip()

    # ===== Year =====
    if "Year" not in df.columns:
        df["Year"] = pd.to_datetime(df.get("Sale_Date"), errors="coerce").dt.yea
    m = df["Year"].isna()
    if m.any() and "Revision_Date" in df.columns:
        df.loc[m, "Year"] = pd.to_datetime(df.loc[m, "Revision_Date"], errors="c

    # ===== Clean numerics =====
    if "Land_Area" in df.columns:
        df["Land_Area"] = _clean_area(df["Land_Area"])
    if "Floor_Area" in df.columns:
        df["Floor_Area"] = _clean_area(df["Floor_Area"])

    for c in ["Price_Gross","Bedrooms","Bathrooms","Floor_Area","Land_Area"]:
        if c in df.columns:
            df[c] = pd.to_numeric(df[c], errors="coerce")

    df = df[df["Price_Gross"] > 0]

    # ===== House age =====
    if "House_Age" not in df.columns:
        if "Year_Built_Est" in df.columns:
            yb = pd.to_numeric(df["Year_Built_Est"], errors="coerce")
        elif "LUD_Age" in df.columns:
            yb = df["LUD_Age"].astype(str).str.extract(r"(\d{4})")[0].astype(flo
        else:
            yb = np.nan
        df["House_Age"] = pd.to_numeric(df["Year"], errors="coerce") - yb
        df["House_Age"] = pd.to_numeric(df["House_Age"], errors="coerce")
        df.loc[(df["House_Age"] < 0) | (df["House_Age"] > 200), "House_Age"] = n
        df["House_Age"] = (
            df.groupby("Town", dropna=False)["House_Age"]
              .transform(lambda s: s.fillna(s.median()))
        ).fillna(df["House_Age"].median())
        df["House_Age"] = df["House_Age"].clip(0, 120)

    # ===== House type grouping =====
    if "House_Type" in df.columns:
        df["House_Type"] = df["House_Type"].fillna("Unknown")
        df["House_Type_Grouped"] = (
            df["House_Type"]
              .replace({
                  "Townhouse": "Townhouse/Unit",
                  "Unit": "Townhouse/Unit",
                  "Terrace Apartments": "Townhouse/Unit",
                  "Terraced Apartments": "Townhouse/Unit",
                  "Apartment": "Apartment",
                  "Flat": "Apartment"
              })
              .fillna("Unknown")
        )
    else:
        df["House_Type_Grouped"] = "Unknown"

    # ===== Year filter =====
```

```python
    df = df[df["Year"].between(year_range[0], year_range[1])]

    # ===== City inference & filter =====
    df["City"] = df.apply(_infer_city, axis=1)
    df = df[df["City"].isin(cities)]

    # ===== Normalize to latest year =====
    year_med = df.groupby("Year")["Price_Gross"].transform("median")
    base_year = df["Year"].max()
    base_median = df.loc[df["Year"]==base_year, "Price_Gross"].median()
    df["Price_Norm"] = df["Price_Gross"] / year_med * base_median

    # winsorise
    p1, p99 = df["Price_Norm"].quantile([0.01, 0.99])
    df = df[(df["Price_Norm"] >= p1) & (df["Price_Norm"] <= p99)]

    # areas to m²
    if "Land_Area" in df.columns:
        df["Land_Area"] = _maybe_to_m2(df["Land_Area"])
    if "Floor_Area" in df.columns:
        df["Floor_Area"] = pd.to_numeric(df["Floor_Area"], errors="coerce")

    # ===== PER-CITY: fit (cap K), predict, select ALL within ±band =====
    feats = ["Bedrooms","Bathrooms","Floor_Area","Land_Area","House_Age","House_
    near_list = []

    for c in cities:
        sub = df[df["City"]==c].copy()
        if sub.empty:
            continue

        Xc = pd.get_dummies(sub[feats], columns=["House_Type_Grouped"], drop_fir
        yc = sub["Price_Norm"].values

        okc = Xc.isna().sum(axis=1) <= allow_missing
        Xc = Xc.loc[okc].fillna(0)
        yc = yc[okc.values]
        base_c = sub.loc[okc].reset_index(drop=True)
        if len(base_c) == 0:
            continue

        # cap neighbors to available samples to avoid ValueError
        k_eff = int(min(n_neighbors, len(base_c)))

        scaler_c = StandardScaler()
        Xs_c = scaler_c.fit_transform(Xc)

        knn_c = KNeighborsRegressor(n_neighbors=k_eff, weights="distance")
        knn_c.fit(Xs_c, yc)

        base_c["Pred_Price"] = knn_c.predict(Xs_c)
        base_c["Gap_to_Budget"] = (base_c["Pred_Price"] - budget).abs()

        # STRICT band: include ALL within ±band; DO NOT widen
        sel = base_c[base_c["Gap_to_Budget"] <= budget * band].copy()
        sel["City"] = c  # ensure label intact
        near_list.append(sel)

    near = pd.concat(near_list, ignore_index=True) if near_list else df.head(0)
```

```python
    # ===== City typical table (medians over the band-selected set) =====
    def dom(s):
        vc = s.value_counts(normalize=True)
        return (vc.index[0], round(vc.iloc[0]*100,1)) if len(vc) else ("N/A", np

    rows = []
    for c in cities:
        sub = near[near["City"]==c]
        if len(sub)==0:
            rows.append([c, 0, np.nan, np.nan, np.nan, np.nan, np.nan, "N/A", np
            continue
        t_type, t_share = dom(sub["House_Type_Grouped"])
        rows.append([
            c,
            int(len(sub)),
            int(np.nanmedian(sub["Bedrooms"])),
            int(np.nanmedian(sub["Bathrooms"])),
            float(np.nanmedian(sub["Floor_Area"])),
            float(np.nanmedian(sub["Land_Area"])),
            float(np.nanmedian(sub["House_Age"])),
            t_type, t_share
        ])

    city_table = pd.DataFrame(rows, columns=[
        "City","N_in_band","Beds_med","Baths_med","Floor_m2_med","Land_m2_med","
    ])

    # ===== Price span table (on the strict-band set) =====
    span_rows = []
    for c in cities:
        sub = near[near["City"]==c]["Pred_Price"]
        if len(sub)==0:
            span_rows.append([c, 0, np.nan, np.nan, np.nan, np.nan, np.nan, np.n
        else:
            span_rows.append([
                c,
                int(len(sub)),
                float(sub.min()),
                float(sub.quantile(0.25)),
                float(sub.median()),
                float(sub.quantile(0.75)),
                float(sub.max()),
                float((sub.median() - budget))
            ])

    price_span_table = pd.DataFrame(span_rows, columns=[
        "City","Count","Pred_min","Pred_p25","Pred_median","Pred_p75","Pred_max"
    ])
    for col in ["Pred_min","Pred_p25","Pred_median","Pred_p75","Pred_max","Media
        price_span_table[col] = price_span_table[col].round(0).astype("Int64")

    # ===== Headline =====
    headline = (
        f"Budget {budget:,.0f} at {base_year} level, per-city KNN (K={n_neighbor
        f"strict ±{int(band*100)}% band; includes ALL matches per city."
    )

    return near, city_table, price_span_table, headline

# ================== RUN ==================
```

```python
near8, city_table8, price_span8, headline8 = knn_budget_band_v8(
    df,
    budget=780_000,
    cities=("Whangarei","Auckland","Hamilton","Wellington","Dunedin","Queenstown
    year_range=(2018, 2024),
    n_neighbors=1000,   # or 100 — both fine; auto-caps per-city if needed
    allow_missing=2,
    band=0.02           # STRICT ±2%
)

print("Headline:", headline8)

print("\n=== City typical configuration (within ±2% band; medians) ===")
display(
    city_table8.set_index("City").rename(columns={
        "N_in_band":"N (within band)",
        "Beds_med":"Beds (med)",
        "Baths_med":"Baths (med)",
        "Floor_m2_med":"Floor (med, m²)",
        "Land_m2_med":"Land (med, m²)",
        "Age_med":"Age (med, yrs)",
        "Top_Type":"Top house type",
        "Top_Type_Share_%":"Top type share (%)"
    })
)

print("\n=== Price range of ALL homes within ±2% (per city) ===")
display(
    price_span8.set_index("City").rename(columns={
        "Count":"N",
        "Pred_min":"Pred min",
        "Pred_p25":"P25",
        "Pred_median":"Median",
        "Pred_p75":"P75",
        "Pred_max":"Pred max",
        "Median_minus_budget":"Median - budget"
    })
)
```

```
Headline: Budget 780,000 at 2024 level, per-city KNN (K=1000, capped), strict ±2%
band; includes ALL matches per city.

=== City typical configuration (within ±2% band; medians) ===
```

City	N (within band)	Beds (med)	Baths (med)	Floor (med, m²)	Land (med, m²)	Age (med, yrs)	Top house type	Top type share (%)
Whangarei	400	3	2	174.0	788.5	32.0	Unknown	91.8
Auckland	7900	3	1	97.0	98.0	39.0	Bungalow (Post-war)	47.8
Hamilton	1196	3	1	150.0	567.5	39.0	Unknown	56.4
Wellington	2668	3	1	125.0	553.0	49.0	Bungalow (Post-war)	68.8
Dunedin	402	3	2	180.0	636.0	52.0	Bungalow (Post-war)	60.0
Queenstown	179	3	2	112.0	253.0	12.0	Bungalow (Post-war)	63.1

```
=== Price range of ALL homes within ±2% (per city) ===
```

City	N	Pred min	P25	Median	P75	Pred max	Median - budget
Whangarei	400	764800	771526	779527	787866	795124	-473
Auckland	7900	764402	772679	781078	788014	795600	1078
Hamilton	1196	764483	771529	779703	788014	795597	-297
Wellington	2668	764448	772021	779830	787994	795597	-170
Dunedin	402	764893	772115	780724	788014	795583	724
Queenstown	179	765000	772983	781557	788969	795586	1557

2.2.3 Per-City Model + Strict ±2% band + Dynamic Neighbours (closest 20%, minimum 100)

In [20]:

```python
# ==== City/Region Check====
import re
import pandas as pd
import numpy as np

CHECK_COLS = [c for c in ["Town","TA_Name","Region_Name","TA_Code","Region_ID"]
assert CHECK_COLS, "Cannot find any target columns (Town/TA_Name/Region_Name/TA_

def _find_rows(df, pattern_regex, cols):
    pat = re.compile(pattern_regex, flags=re.IGNORECASE)
    mask_any = df[cols].apply(lambda s: s.astype(str).str.contains(pat, na=False
    return df.loc[mask_any].copy()

def _print_header(title):
    print("\n" + "="*80)
    print(title)
    print("="*80)
```

```python
def _top_values(series, topn=10):
    vc = series.astype(str).value_counts(dropna=True)
    return vc.head(topn)

def audit_city(name, pattern_regex, cols=CHECK_COLS, topn=10, show_examples=5):
    """
    name: Show in the Heading (e.g. 'Wellington')
    pattern_regex: Used e.g. r"wellington|lower hutt|upper hutt|porirua"
    """
    sub = _find_rows(df, pattern_regex, cols)
    _print_header(f"[{name}] rows matched by /{pattern_regex}/  ->  {len(sub):,}

    # 1) Every column: unique values & top N counts
    for c in cols:
        s = sub[c].dropna().astype(str)
        if s.empty:
            print(f"\n[{c}]  (no values)")
            continue
        uni = sorted([v for v in s.unique() if re.search(pattern_regex, v, re.IG
        print(f"\n[{c}] unique values containing pattern ({len(uni)} found):")
        for v in uni[:50]:
            print("  -", v)
        if len(uni) > 50:
            print("  ... (truncated)")

        # Top N Counts
        print(f"\n[{c}] top {topn} value counts within matched rows:")
        print(_top_values(s, topn=topn).to_string())

    # 2) Print Table: TA × Region
    if {"TA_Name","Region_Name"}.issubset(sub.columns):
        ct = pd.crosstab(sub["TA_Name"], sub["Region_Name"])
        print("\n[Crosstab] TA_Name × Region_Name (matched rows):")
        # Only show non-zero rows/columns
        ct = ct.loc[(ct.sum(axis=1) > 0), (ct.sum(axis=0) > 0)]
        print(ct.head(30).to_string())

    # 3) Check codes: TA_Code / Region_ID
    for code_col in [c for c in ["TA_Code","Region_ID"] if c in sub.columns]:
        codes = (sub[[code_col]].dropna().drop_duplicates().sort_values(code_col
        print(f"\nDistinct {code_col} in matched rows ({len(codes)} found):")
        print(codes.head(50).to_string(index=False))

    # 4) Print sample rows
    print(f"\nSample rows (random {show_examples}):")
    with pd.option_context("display.max_columns", None, "display.width", 160):
        print(sub.sample(min(show_examples, len(sub))).loc[:, CHECK_COLS].to_str

# ===== Check the three cities =====
audit_city("Wellington (Including Lower/Upper Hutt, Porirua)",
           r"wellington|lower hutt|upper hutt|porirua")

audit_city("Dunedin",
           r"dunedin")

audit_city("Auckland",
           r"auckland")
```

```
================================================================================
[Wellington (Including Lower/Upper Hutt, Porirua)] rows matched by /wellington|lo
wer hutt|upper hutt|porirua/  ->  337,183 rows
================================================================================

[Town] unique values containing pattern (4 found):
  - Lower Hutt
  - Porirua
  - Upper Hutt
  - Wellington

[Town] top 10 value counts within matched rows:
Town
Wellington     129879
Lower Hutt      69208
Upper Hutt      29413
Porirua         27436
Paraparaumu     25540
Masterton       18005
Waikanae        12066
Otaki            6802
Carterton        5433
Featherston      2742

[TA_Name] unique values containing pattern (3 found):
  - Porirua City
  - Upper Hutt City
  - Wellington City

[TA_Name] top 10 value counts within matched rows:
TA_Name
Wellington City             130358
Hutt City                    69919
Kapiti Coast District        45943
Upper Hutt City              30062
Porirua City                 29068
Masterton District           18533
South Wairarapa District      7771
Carterton District            5529

[Region_Name] unique values containing pattern (1 found):
  - Wellington Region

[Region_Name] top 10 value counts within matched rows:
Region_Name
Wellington Region    337183

[TA_Code] unique values containing pattern (0 found):

[TA_Code] top 10 value counts within matched rows:
TA_Code
47    130358
46     69919
43     45943
45     30062
44     29068
48     18533
50      7771
49      5529

[Region_ID] unique values containing pattern (0 found):

[Region_ID] top 10 value counts within matched rows:
Region_ID
9    337183

[Crosstab] TA_Name × Region_Name (matched rows):
Region_Name               Wellington Region
TA_Name
Carterton District                     5529
Hutt City                             69919
Kapiti Coast District                 45943
Masterton District                    18533
Porirua City                          29068
South Wairarapa District               7771
Upper Hutt City                       30062
Wellington City                      130358

Distinct TA_Code in matched rows (8 found):
 TA_Code
      43
      44
      45
      46
      47
      48
      49
      50

Distinct Region_ID in matched rows (1 found):
 Region_ID
         9

Sample rows (random 5):
      Town               TA_Name       Region_Name  TA_Code  Region_ID
Lower Hutt             Hutt City Wellington Region       46          9
Lower Hutt             Hutt City Wellington Region       46          9
  Waikanae Kapiti Coast District Wellington Region       43          9
Wellington       Wellington City Wellington Region       47          9
 Masterton    Masterton District Wellington Region       48          9

================================================================================
[Dunedin] rows matched by /dunedin/  ->  94,248 rows
================================================================================

[Town] unique values containing pattern (1 found):
  - Dunedin

[Town] top 10 value counts within matched rows:
Town
Dunedin          75017
Mosgiel          11807
Waikouaiti        2788
Port Chalmers     2698
Outram             739
Waitati            429
Middlemarch        156
Kyeburn              4

[TA_Name] unique values containing pattern (1 found):
```

```
  - Dunedin City

[TA_Name] top 10 value counts within matched rows:
TA_Name
Dunedin City    94248

[Region_Name] unique values containing pattern (0 found):

[Region_Name] top 10 value counts within matched rows:
Region_Name
Otago Region    94248

[TA_Code] unique values containing pattern (0 found):

[TA_Code] top 10 value counts within matched rows:
TA_Code
71    94248

[Region_ID] unique values containing pattern (0 found):

[Region_ID] top 10 value counts within matched rows:
Region_ID
14    94248

[Crosstab] TA_Name × Region_Name (matched rows):
Region_Name   Otago Region
TA_Name
Dunedin City         94248

Distinct TA_Code in matched rows (1 found):
 TA_Code
      71

Distinct Region_ID in matched rows (1 found):
 Region_ID
        14

Sample rows (random 5):
   Town      TA_Name  Region_Name  TA_Code  Region_ID
Dunedin Dunedin City Otago Region       71         14
Dunedin Dunedin City Otago Region       71         14
Dunedin Dunedin City Otago Region       71         14
Dunedin Dunedin City Otago Region       71         14
Mosgiel Dunedin City Otago Region       71         14

================================================================================
[Auckland] rows matched by /auckland/  ->  989,973 rows
================================================================================

[Town] unique values containing pattern (1 found):
  - Auckland

[Town] top 10 value counts within matched rows:
Town
Auckland          826488
Papakura           29086
Whangaparaoa       22998
Pukekohe           17885
Orewa              11790
Takanini           10191
```

```
Waiheke Island     10102
Waiuku              7134
Red Beach           6356
Warkworth           5667

[TA_Name] unique values containing pattern (7 found):
  - Auckland - City
  - Auckland - Franklin
  - Auckland - Manukau
  - Auckland - North Shore
  - Auckland - Papakura
  - Auckland - Rodney
  - Auckland - Waitakere

[TA_Name] top 10 value counts within matched rows:
TA_Name
Auckland - City           297965
Auckland - Manukau        218471
Auckland - North Shore    176555
Auckland - Waitakere      153953
Auckland - Rodney          72508
Auckland - Papakura        41893
Auckland - Franklin        28628

[Region_Name] unique values containing pattern (1 found):
  - Auckland (Unitary)

[Region_Name] top 10 value counts within matched rows:
Region_Name
Auckland (Unitary)    989973

[TA_Code] unique values containing pattern (0 found):

[TA_Code] top 10 value counts within matched rows:
TA_Code
7     297965
8     218471
5     176555
6     153953
4      72508
9      41893
10     28628

[Region_ID] unique values containing pattern (0 found):

[Region_ID] top 10 value counts within matched rows:
Region_ID
2    989973

[Crosstab] TA_Name × Region_Name (matched rows):
Region_Name             Auckland (Unitary)
TA_Name
Auckland - City                     297965
Auckland - Franklin                  28628
Auckland - Manukau                  218471
Auckland - North Shore              176555
Auckland - Papakura                  41893
Auckland - Rodney                    72508
Auckland - Waitakere                153953
```

```
Distinct TA_Code in matched rows (7 found):
 TA_Code
       4
       5
       6
       7
       8
       9
      10

Distinct Region_ID in matched rows (1 found):
 Region_ID
         2

Sample rows (random 5):
    Town                TA_Name        Region_Name  TA_Code  Region_ID
Auckland        Auckland - City Auckland (Unitary)        7          2
Auckland   Auckland - Waitakere Auckland (Unitary)        6          2
Auckland        Auckland - City Auckland (Unitary)        7          2
Auckland Auckland - North Shore Auckland (Unitary)        5          2
Auckland Auckland - North Shore Auckland (Unitary)        5          2
```

In [21]:

```python
# ================================================================
# KNN v9 — per-city model + strict ±2% band + dynamic or manual k
# Lightweight version (city filter, down-sample, n_jobs, uint8 dummies)
# ================================================================

import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor

# ---------------- helpers ----------------
def _maybe_to_m2(s):
    v = pd.to_numeric(s, errors="coerce")
    if v.median(skipna=True) < 10:
        return v * 10_000
    return v

def _clean_area(col):
    return (
        col.astype(str)
           .str.replace(r"[^\d\.~-]", "", regex=True)
           .str.replace(r"(?i)<\s*m2|>=\s*m2|sq\s*m|sqm|sq\s*sm", "", regex=True
           .str.replace(r"([\-\.])\1+", r"\1", regex=True)
           .replace({"": np.nan, "-": np.nan, "--": np.nan})
    )

def _infer_city(row):
    hay = " ".join([
        str(row.get("Town","")),
        str(row.get("TA_Name","")),
        str(row.get("Region_Name","")),
        str(row.get("TA_Code","")),
        str(row.get("Region_ID","")),
    ]).lower()

    lookups = {
        "Auckland": ["auckland"],
        "Wellington": ["wellington"],
        "Dunedin": ["dunedin"],
        "Whangarei": ["whangarei"],
        "Hamilton": ["hamilton"],
        "Queenstown": ["queenstown","queenstown lakes","queenstown-lakes","queen
    }
    for city, keys in lookups.items():
        if any(k in hay for k in keys):
            return city
    return np.nan

# ---------------- main ----------------
def knn_budget_band_v9(
    data: pd.DataFrame = None,
    budget=780_000,
    cities=("Auckland","Wellington","Dunedin"),    # keep only these
    year_range=(2018, 2024),
    allow_missing=2,
    band=0.02,                 # strict ±2% band
    frac_neighbors=0.20,       # dynamic k = 20% of city rows
    min_neighbors=100,         # but at least this many
    city_k: dict | None = None,# manual per-city k override, e.g. {"Auckland":10
    max_city_rows: int = 120_000  # cap rows per city for RAM/CPU
):
    # Safe fallback
    if data is None:
        global df
        if "df" not in globals():
            raise ValueError("No dataframe provided. Please load or define 'df'
        data = df

    df = data.copy()
    df.columns = df.columns.str.strip()

    # ====== Year ======
    if "Year" not in df.columns:
        df["Year"] = pd.to_datetime(df["Sale_Date"], errors="coerce").dt.year
    m = df["Year"].isna()
    if m.any() and "Revision_Date" in df.columns:
        df.loc[m, "Year"] = pd.to_datetime(df.loc[m, "Revision_Date"], errors="c

    # ====== Clean numerics ======
    if "Land_Area" in df.columns:
        df["Land_Area"] = _clean_area(df["Land_Area"])
        df["Land_Area"] = pd.to_numeric(df["Land_Area"], errors="coerce")
    if "Floor_Area" in df.columns:
        df["Floor_Area"] = _clean_area(df["Floor_Area"])
        df["Floor_Area"] = pd.to_numeric(df["Floor_Area"], errors="coerce")

    for c in ["Price_Gross","Bedrooms","Bathrooms","Floor_Area","Land_Area"]:
        if c in df.columns:
            df[c] = pd.to_numeric(df[c], errors="coerce")

    df = df[df["Price_Gross"] > 0]

    # ====== House age ======
    # ===== House age (v9.1 stable) =====
    # Robust build-year inference + v8-style reliable fallback

    sale_year = pd.to_numeric(df["Year"], errors="coerce")
```

```python
# 1) Build year from best-available source
build_year = pd.Series(np.nan, index=df.index, dtype="float64")

# Year_Built_Est (prefer)
if "Year_Built_Est" in df.columns:
    by = pd.to_numeric(df["Year_Built_Est"], errors="coerce")
    build_year = by.combine_first(build_year)

# LUD_Age may be '1998' (year) or '45' (age)
if "LUD_Age" in df.columns:
    tmp = pd.to_numeric(df["LUD_Age"], errors="coerce")
    if tmp.isna().all():
        tmp = pd.to_numeric(df["LUD_Age"].astype(str).str.extract(r"(\d{1,4}
    yr_max = int(np.nanmax(sale_year)) if np.isfinite(np.nanmax(sale_year))
    is_year = tmp.between(1850, yr_max)
    is_age  = tmp.between(0, 150)
    build_year = np.where(is_year, tmp, build_year)
    build_year = np.where(is_age, sale_year - tmp, build_year)
    build_year = pd.to_numeric(build_year, errors="coerce")

# 2) House_Age = sale_year - build_year, with basic sanity checks
house_age = pd.to_numeric(sale_year - build_year, errors="coerce")
house_age[(house_age < 0) | (house_age > 200)] = np.nan

# 3) Hierarchical fill (City -> Town -> global)
if "City" in df.columns:
    house_age = house_age.groupby(df["City"]).transform(lambda s: s.fillna(s
elif "Town" in df.columns:
    house_age = house_age.groupby(df["Town"]).transform(lambda s: s.fillna(s

# 4) v8-style reliable fallback (ensures no all-NaN)
if house_age.isna().all():
    house_age = pd.Series(40.0, index=df.index)  # conservative default
else:
    house_age = house_age.fillna(house_age.median())

# 5) Finalize
df["House_Age"] = house_age.clip(0, 120).astype("float32")


# ====== House type (grouped) ======
if "House_Type" in df.columns:
    df["House_Type"] = df["House_Type"].fillna("Unknown")
    df["House_Type_Grouped"] = (
        df["House_Type"]
          .replace({
              "Townhouse": "Townhouse/Unit",
              "Unit": "Townhouse/Unit",
              "Terrace Apartments": "Townhouse/Unit",
              "Apartment": "Apartment",
              "Flat": "Apartment",
          })
          .fillna("Unknown")
    )
else:
    df["House_Type_Grouped"] = "Unknown"

# ====== Year filter ======
df = df[df["Year"].between(year_range[0], year_range[1])]
```

```python
# ====== City inference & filter ======
if "City" not in df.columns:
    df["City"] = df.apply(_infer_city, axis=1)
df = df[df["City"].isin(cities)]

# ====== Normalize to latest year ======
year_med = df.groupby("Year")["Price_Gross"].transform("median")
base_year = df["Year"].max()
base_med = df.loc[df["Year"]==base_year, "Price_Gross"].median()
df["Price_Norm"] = df["Price_Gross"] / year_med * base_med

# ====== Winsorize ======
p1, p99 = df["Price_Norm"].quantile([0.01, 0.99])
df = df[(df["Price_Norm"] >= p1) & (df["Price_Norm"] <= p99)]

# ====== Areas to m² when needed ======
if "Land_Area" in df.columns:
    df["Land_Area"] = _maybe_to_m2(df["Land_Area"])
if "Floor_Area" in df.columns:
    df["Floor_Area"] = pd.to_numeric(df["Floor_Area"], errors="coerce")

# ====== PER-CITY: fit with dynamic/manual neighbors, predict, strict band =
feats = ["Bedrooms","Bathrooms","Floor_Area","Land_Area","House_Age","House_
near_list = []

for c in cities:
    sub = df[df["City"]==c].copy()
    if sub.empty:
        continue

    # Down-sample very large cities
    if len(sub) > max_city_rows:
        sub = sub.sample(max_city_rows, random_state=42)

    Xc = pd.get_dummies(sub[feats], columns=["House_Type_Grouped"], drop_fir
    yc = sub["Price_Norm"].values

    okc = Xc.isna().sum(axis=1) <= allow_missing
    Xc = Xc.loc[okc].fillna(0)
    yc = yc[okc.values]
    base_c = sub.loc[okc].reset_index(drop=True)
    if len(base_c) == 0:
        continue

    # dynamic/manual k
    if city_k and c in city_k:
        k_eff = int(city_k[c])
    else:
        k_eff = int(max(min_neighbors, int(len(base_c) * frac_neighbors)))
    k_eff = max(1, min(k_eff, len(base_c)))
    print(f"{c}: n_neighbors={k_eff}, Sample Size={len(base_c)}")

    scaler_c = StandardScaler()
    Xs_c = scaler_c.fit_transform(Xc)

    knn_c = KNeighborsRegressor(
        n_neighbors=k_eff,
        weights="distance",
        algorithm="auto",
        n_jobs=-1
```

```python
        )
        knn_c.fit(Xs_c, yc)

        base_c["Pred_Price"] = knn_c.predict(Xs_c)
        base_c["Gap_to_Budget"] = (base_c["Pred_Price"] - budget).abs()

        # STRICT band: include ALL within ± band
        sel = base_c[(base_c["Gap_to_Budget"] <= budget * band)].copy()
        sel["City"] = c
        near_list.append(sel)

    near = pd.concat(near_list, ignore_index=True) if near_list else df.head(0)

    # ===== City typical table =====
    def dom(s):
        vc = s.value_counts(normalize=True)
        return (vc.index[0], round(vc.iloc[0]*100,1)) if len(vc) else ("N/A", np

    rows = []
    for c in cities:
        sub = near[near["City"]==c]
        if len(sub)==0:
            rows.append([c, 0, np.nan, np.nan, np.nan, np.nan, np.nan, "N/A", np
            continue
        t_type, t_share = dom(sub["House_Type_Grouped"])
        rows.append([
            c,
            int(len(sub)),
            int(np.nanmedian(sub["Bedrooms"])),
            int(np.nanmedian(sub["Bathrooms"])),
            float(np.nanmedian(sub["Floor_Area"])),
            float(np.nanmedian(sub["Land_Area"])),
            float(np.nanmedian(sub["House_Age"])),
            t_type,
            t_share
        ])
    city_table = pd.DataFrame(rows, columns=[
        "City","N_in_band","Beds_med","Baths_med","Floor_m2_med","Land_m2_med","
    ])

    # ===== Price span table =====
    span_rows = []
    for c in cities:
        sub = near[near["City"]==c]["Pred_Price"]
        if len(sub)==0:
            span_rows.append([c, 0, np.nan, np.nan, np.nan, np.nan, np.nan, np.n
        else:
            span_rows.append([
                c,
                int(len(sub)),
                float(sub.min()),
                float(sub.quantile(0.25)),
                float(sub.median()),
                float(sub.quantile(0.75)),
                float(sub.max()),
                float(sub.median() - budget),
            ])
    price_span_table = pd.DataFrame(span_rows, columns=[
        "City","Count","Pred_min","Pred_p25","Pred_median","Pred_p75","Pred_max"
    ])
```

```python
    for col in ["Pred_min","Pred_p25","Pred_median","Pred_p75","Pred_max","Media
        price_span_table[col] = price_span_table[col].round(0).astype("Int64")

    headline = (
        f"Budget {budget:,.0f} at {base_year} level, per-city KNN "
        f"(K={max(min_neighbors,int(frac_neighbors*100))}% of city; min capped,
        f"{'manual k for ' + ', '.join(city_k.keys()) if city_k else 'dynamic k'
        f"strict ±{int(band*100)}% band; includes ALL matches per city.)"
    )

    return near, city_table, price_span_table, headline


# ========================= RUN =========================
# Option A: if your merged dataframe variable is named `df`, just run:
df = pd.read_csv("Combined_Residential_Property_Sale_Stats.csv")  # <- uncomment

near9, city_table9, price_span9, headline9 = knn_budget_band_v9(
    data=df,  # or replace with your variable holding the merged dataset
    budget=780_000,
    cities=("Auckland","Wellington","Dunedin"),
    year_range=(2018, 2024),
    allow_missing=2,
    band=0.02,                       # ±2%
    frac_neighbors=0.15,             # slightly smaller dynamic share
    min_neighbors=80,                # lower floor to reduce work
    city_k={"Auckland":1000,"Wellington":600,"Dunedin":400},  # manual k
    max_city_rows=120_000
)

print("Headline:", headline9)
print("\n=== City typical configuration (within ±2% band; medians) ===")
display(
    city_table9.set_index("City").rename(columns={
        "N_in_band":"N (within band)",
        "Beds_med":"Beds (med)",
        "Baths_med":"Baths (med)",
        "Floor_m2_med":"Floor (med, m²)",
        "Land_m2_med":"Land (med, m²)",
        "Age_med":"Age (med, yrs)",
        "Top_Type":"Top house type",
        "Top_Type_Share_%":"Top type share (%)"
    })
)

print("\n=== Price range of ALL homes within ±2% (per city) ===")
display(
    price_span9.set_index("City").rename(columns={
        "Count":"N",
        "Pred_min":"Pred min",
        "Pred_p25":"P25",
        "Pred_median":"Median",
        "Pred_p75":"P75",
        "Pred_max":"Pred max",
        "Median_minus_budget":"Median - budget"
    })
)
```

```
/opt/anaconda3/envs/civil763/lib/python3.13/site-packages/numpy/lib/_nanfunctions
_impl.py:1214: RuntimeWarning: Mean of empty slice
  return np.nanmean(a, axis, out=out, keepdims=keepdims)
/opt/anaconda3/envs/civil763/lib/python3.13/site-packages/numpy/lib/_nanfunctions
_impl.py:1214: RuntimeWarning: Mean of empty slice
  return np.nanmean(a, axis, out=out, keepdims=keepdims)
/opt/anaconda3/envs/civil763/lib/python3.13/site-packages/numpy/lib/_nanfunctions
_impl.py:1214: RuntimeWarning: Mean of empty slice
  return np.nanmean(a, axis, out=out, keepdims=keepdims)
/opt/anaconda3/envs/civil763/lib/python3.13/site-packages/numpy/lib/_nanfunctions
_impl.py:1214: RuntimeWarning: Mean of empty slice
  return np.nanmean(a, axis, out=out, keepdims=keepdims)
/opt/anaconda3/envs/civil763/lib/python3.13/site-packages/numpy/lib/_nanfunctions
_impl.py:1214: RuntimeWarning: Mean of empty slice
  return np.nanmean(a, axis, out=out, keepdims=keepdims)
/opt/anaconda3/envs/civil763/lib/python3.13/site-packages/numpy/lib/_nanfunctions
_impl.py:1214: RuntimeWarning: Mean of empty slice
  return np.nanmean(a, axis, out=out, keepdims=keepdims)
Auckland: n_neighbors=1000, Sample Size=119852
Wellington: n_neighbors=600, Sample Size=61568
Dunedin: n_neighbors=400, Sample Size=16945
Headline: Budget 780,000 at 2024 level, per-city KNN (K=80% of city; min capped,
manual k for Auckland, Wellington, Dunedin; strict ±2% band; includes ALL matches
per city.)

=== City typical configuration (within ±2% band; medians) ===
```

City	N (within band)	Beds (med)	Baths (med)	Floor (med, m²)	Land (med, m²)	Age (med, yrs)	Top house type	Top type share (%)
Auckland	5094	3	1	92.0	104.0	34.0	Bungalow (Post-war)	42.9
Wellington	2709	3	1	123.0	551.0	48.0	Bungalow (Post-war)	67.4
Dunedin	399	3	2	180.0	629.0	54.0	Bungalow (Post-war)	61.4

```
=== Price range of ALL homes within ±2% (per city) ===
```

City	N	Pred min	P25	Median	P75	Pred max	Median - budget
Auckland	5094	764487	773500	780764	787668	795594	764
Wellington	2709	764436	773588	780040	787101	795573	40
Dunedin	399	764552	772772	780640	788117	795572	640

Pattern 2 Final Output: TABLE 2, FIGURE 2, Geospatial Map

TABLE 2. Home Characteristics by City (NZD 780,000 Budget ± 2 % Price Band, 2018-2024)

In [22]:
```python
import pandas as pd

summary = (
    city_table9[["City","N_in_band","Beds_med","Baths_med","Floor_m2_med","Land_
    .assign(**{
        "Typical home": lambda df: df["Beds_med"].astype(str) + " bed / " + df["
    })
    .rename(columns={
        "N_in_band": "Matched homes",
        "Floor_m2_med": "Floor (m²)",
        "Land_m2_med": "Land (m²)",
        "Age_med": "Age (yrs)"
    })[["City","Matched homes","Typical home","Floor (m²)","Land (m²)","Age (yrs
    .style.format({
        "Matched homes": "{:,}",
        "Floor (m²)": "{:.0f}",
        "Land (m²)": "{:.0f}",
        "Age (yrs)": "{:.0f}"
    })
    .hide(axis="index")
    # caption on two lines
    .set_caption("🏠 Typical homes you can find with a $780,000 budget<br>(±2% b
    .set_table_styles([
        {"selector": "caption", "props": [
            ("text-align", "center"),
            ("font-size", "14px"),
            ("font-weight", "bold"),
            ("color", "#333"),
            ("padding", "10px")
        ]},
        {"selector": "th", "props": [
            ("background-color", "#F4F4F4"),
            ("font-weight", "bold"),
            ("text-align", "center"),
            ("border", "1px solid #DDD")
        ]},
        {"selector": "td", "props": [
            ("border", "1px solid #DDD"),
            ("text-align", "center"),
            ("padding", "6px")
        ]}
    ])
    # make city names bold
    .applymap(lambda v: "font-weight: bold;" if v in city_table9["City"].tolist(
)
summary
```

```
/var/folders/4x/6bbjp51n6j5dvvsfk6gjgrzw0000gp/T/ipykernel_12208/185610537.py:44:
FutureWarning: Styler.applymap has been deprecated. Use Styler.map instead.
  .applymap(lambda v: "font-weight: bold;" if v in city_table9["City"].tolist() e
lse "", subset=["City"])
```

Out[22]:

🏠 Typical homes you can find with a $780,000 budget (±2% band, 2018–2024 median level)

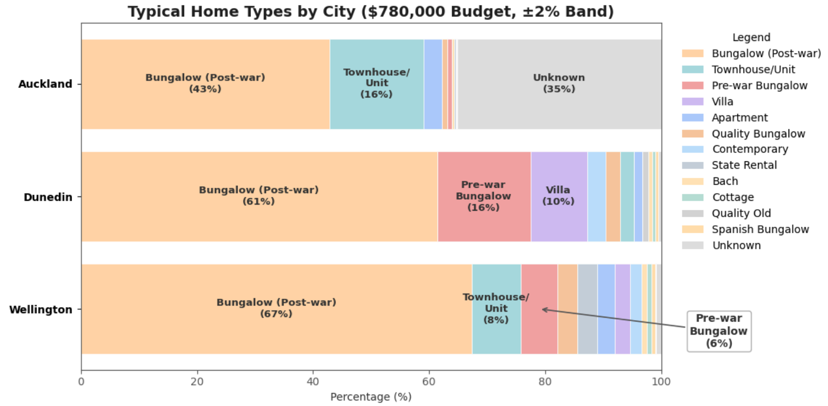
City	Matched homes	Typical home	Floor (m²)	Land (m²)	Age (yrs)
Auckland	5,094	3 bed / 1 bath	92	104	34
Wellington	2,709	3 bed / 1 bath	123	551	48
Dunedin	399	3 bed / 2 bath	180	629	54

FIGURE 2. Typical Home Types by City (NZD 780,000 Budget ± 2 % Price Band, 2018-2024)

In [23]:

```python
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.patches import Patch
from itertools import cycle

# ========= Percentage data =========
type_counts = (
    near9.groupby(["City","House_Type_Grouped"]).size().unstack(fill_value=0)
)
type_pct = type_counts.div(type_counts.sum(axis=1), axis=0) * 100
city_order = type_pct.max(axis=1).sort_values(ascending=False).index.tolist()
type_pct = type_pct.loc[city_order]

# ========= Softer pastel palette =========
force_unknown = "Unknown"
base_colors = {
    "Bungalow (Post-war)": "#FFD6A5",
    "Pre-war Bungalow":    "#F4A3A3",
    "Townhouse/Unit":      "#A8DADC",
    "Apartment":           "#AECBFA",
    "Villa":               "#D0BDF4",
    "State Rental":        "#C3CEDA",
    "Quality Bungalow":    "#F7C59F",
    "Quality Old":         "#D3D3D3",
    "Cottage":             "#B8E0D2",
    "Contemporary":        "#BDE0FE",
    "Spanish Bungalow":    "#FFE0AC",
    force_unknown:         "#E0E0E0"
}
pastel_cycle = cycle([
    "#C1E1C1","#FFE5B4","#FFCF9F","#E6D0E3","#BEE5EB","#FDE2E4","#F6EAC2","#C7E9
])
all_types = list(type_pct.columns)
color_map = {t: base_colors.get(t, next(pastel_cycle)) for t in all_types}

# ========= Plot settings =========
top_n = 4
label_threshold = 6
min_label_width = 3

fig, ax = plt.subplots(figsize=(11, 5.5))
ypos = range(len(type_pct.index))

for yi, city in zip(ypos, type_pct.index):
    row = type_pct.loc[city]
    known = row.drop(labels=[force_unknown], errors="ignore").sort_values(ascend
    tail = row[[force_unknown]] if force_unknown in row.index else pd.Series(dty
    ordered = pd.concat([known, tail])
    left = 0.0
    label_targets = known.head(top_n).index.tolist()

    for t, pct in ordered.items():
        if pct <= 0:
            continue
        ax.barh(
            yi, pct, left=left,
            color=color_map.get(t, "#DDDDDD"),
            edgecolor="white", linewidth=0.6
        )

        # ---------- label rules ----------
        label_text = None

        # Skip Wellington Pre-war Bungalow (for callout)
        if (city == "Wellington") and (t in {"Pre-war Bungalow"}):
            label_text = None
        elif t == "Townhouse/Unit" and pct >= label_threshold:
            label_text = f"Townhouse/\nUnit\n({pct:.0f}%)"
        elif t == "Pre-war Bungalow" and pct >= label_threshold:
            label_text = f"Pre-war\nBungalow\n({pct:.0f}%)"
        elif (t == force_unknown) and (city == "Auckland") and pct >= label_thre
            label_text = f"{t}\n({pct:.0f}%)"
        elif (t in label_targets) and pct >= label_threshold:
            label_text = f"{t}\n({pct:.0f}%)"

        if label_text:
            ax.text(
                left + pct/2, yi, label_text,
                ha="center", va="center",
                fontsize=9.5, weight="bold", color="#333333"
            )
        left += pct

# ========= Cosmetics =========
ax.set_yticks(list(ypos))
ax.set_yticklabels(type_pct.index, color="black", weight="bold")
ax.set_xlim(0, 100)
ax.set_xlabel("Percentage (%)", color="#333333")
ax.set_title("Typical Home Types by City ($780,000 Budget, ±2% Band)",
             fontsize=14, weight="bold", color="#222222")
ax.tick_params(axis='x', colors="#555555")
ax.tick_params(axis='y', colors="black")

for spine in ax.spines.values():
    spine.set_visible(True)
    spine.set_color("#555555")

# ========= Legend sorting =========
```

```python
# Sort by total percentage (descending), Unknown last
total_pct = type_pct.sum(axis=0).sort_values(ascending=False)
legend_order = [t for t in total_pct.index if t != force_unknown] + [force_unkno

handles = [Patch(facecolor=color_map[t], label=t) for t in legend_order if t in
ax.legend(
    handles=handles,
    title="Legend",
    bbox_to_anchor=(1.02, 1), loc="upper left", frameon=False
)

plt.tight_layout()

# ========= Callouts for Wellington =========
city_to_call = "Wellington"
y_idx = list(type_pct.index).index(city_to_call)
row = type_pct.loc[city_to_call]
known = row.drop(labels=[force_unknown], errors="ignore").sort_values(ascending=
ordered = pd.concat([known, row[[force_unknown]]]) if force_unknown in row.index

left = 0.0
x_center = {}
for t, pct in ordered.items():
    x_center[t] = left + pct/2
    left += pct

def callout(house_type, dy_offset, text=None):
    if house_type in x_center:
        txt = text or f"{house_type}\n({row[house_type]:.0f}%)"
        ax.annotate(
            txt,
            xy=(x_center[house_type], y_idx),
            xycoords="data",
            xytext=(102, y_idx + dy_offset),
            textcoords="data",
            ha="left", va="center",
            fontsize=10, weight="bold", color="#333333",
            bbox=dict(boxstyle="round,pad=0.25", fc="white", ec="#AAAAAA", alpha
            arrowprops=dict(arrowstyle="->", lw=1.2, color="#555555")
        )


# --- Pre-war Bungalow callout (centered, 3 rows) ---

if "Pre-war Bungalow" in x_center:
    txt = f"Pre-war\nBungalow\n({row['Pre-war Bungalow']:.0f}%)"
    ax.annotate(
        txt,
        xy=(x_center["Pre-war Bungalow"], y_idx),
        xycoords="data",
        xytext=(110, y_idx - 0.2),
        textcoords="data",
        ha="center", va="center",  # ✅ centered horizontally and vertically
        fontsize=10, weight="bold", color="#333333",
        bbox=dict(boxstyle="round,pad=0.25", fc="white", ec="#AAAAAA", alpha=0.9
        arrowprops=dict(arrowstyle="->", lw=1.2, color="#555555")
    )

plt.show()
```



Geospatial Map — Regional Median Prices (2018–2024)

FOR PRESENTATION ONLY, NOT USED FOR ANALYSIS

This map is mainly included to satisfy the geospatial component of the project and to provide a simple visual highlight for the presentation. Each hexagon shows the **median residential sale price (2018–2024)** — darker areas indicate lower values, while lighter areas show higher values. The figure itself was not adjusted or stretched by the image size setup; it simply helps to **highlight the main focus regions** — **Auckland, Wellington, and Dunedin** — that will be discussed in the presentation.

> *Source:* Stats NZ Data Service (2025). **Regional Council 2025 – clipped** and **Territorial Authority 2025 – clipped**.
> Retrieved from https://datafinder.stats.govt.nz/data/
> Licensed under CC BY 4.0 (https://creativecommons.org/licenses/by/4.0/).

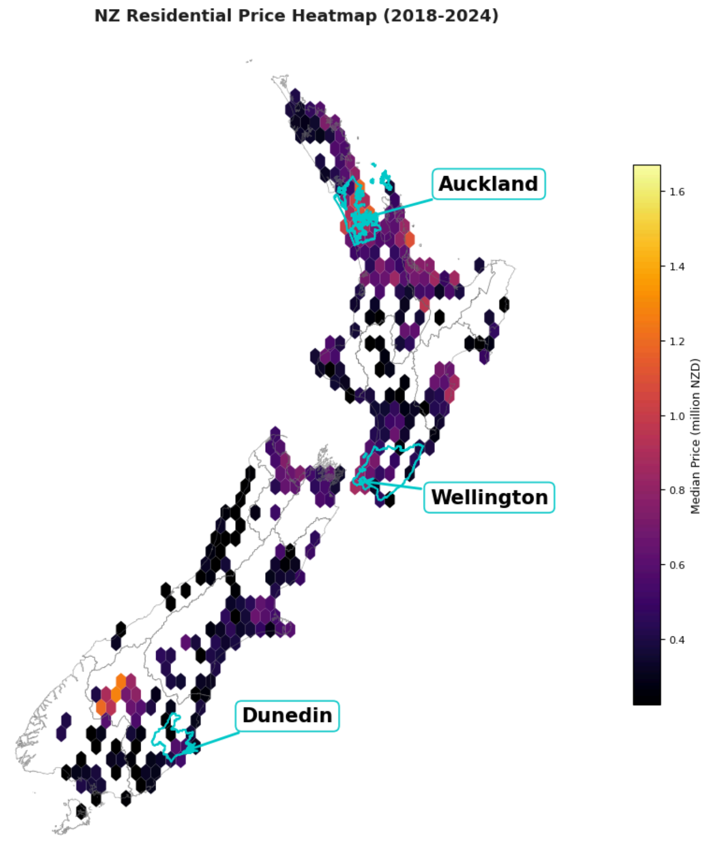
In [24]:
```python
import numpy as np, pandas as pd, matplotlib.pyplot as plt
import geopandas as gpd
from matplotlib.colors import Normalize
from matplotlib.ticker import FuncFormatter
from pathlib import Path

# ---------- user inputs ----------
csv_path = "Combined_Residential_Property_Sale_Stats.csv"
lat = "Latitude"; lon = "Longitude"; price = "Price_Gross"
date_col = "Sale_Date"  # fallback to CL_Sale_Date if missing
year_min, year_max = 2018, 2024

rc_path = "regional-council-2025-clipped.shp"      # Regional Councils
ta_path = "territorial-authority-2025-clipped.shp" # Territorial Authorities

extent = (165, 179.5, -48, -33)
gridsize = 60
mincnt = 20
```

```python
# ---------- helpers ----------
def resolve_shp(p):
    p = Path(p)
    if p.is_file() and p.suffix.lower()==".shp": return str(p)
    if p.is_dir():
        files = list(p.glob("*.shp"))
        if files: return str(files[0])
    raise FileNotFoundError(f"No .shp found at {p.resolve()}")

def find_name_col(gdf, kind):
    if kind=="rc":
        cands = ["REGC2025_1","REGC2024_1","REGC2023_1","REGC2025_NAME",
                 "REGC2024_NAME","REGC2023_NAME","REGC_NAME","REGCNAME","REGC202
    else:
        cands = ["TA2025_1","TA2024_1","TA2023_1","TA2025_NAME",
                 "TA2024_NAME","TA2023_NAME","TA_NAME","TANAME","TA2025_V"]
    for c in cands:
        if c in gdf.columns: return c
    # fallback to first object column
    txt = [c for c in gdf.columns if c!="geometry" and gdf[c].dtype==object]
    return txt[0] if txt else None

# ---------- 1) load sales + build hexbin ----------
df = pd.read_csv(csv_path, low_memory=False)

lat_s = df.get(lat, df.get("CL_Latitude"))
lon_s = df.get(lon, df.get("CL_Longitude"))
price_s = df.get(price, df.get("CL_Sale_Price_Gross"))
year_s = pd.to_datetime(df.get(date_col, df.get("CL_Sale_Date")), errors="coerce

mask = (~lat_s.isna()) & (~lon_s.isna()) & (~price_s.isna()) & (price_s>0) \
       & (year_s.between(year_min, year_max)) \
       & (lat_s.between(extent[2], extent[3])) & (lon_s.between(extent[0], exten

lat_v, lon_v, price_v = lat_s[mask].to_numpy(), lon_s[mask].to_numpy(), price_s[

# stronger contrast for colors
if price_v.size:
    vmin, vmax = np.percentile(price_v, [5, 95])
    norm = Normalize(vmin=vmin, vmax=vmax)
else:
    norm = None

fig, ax = plt.subplots(figsize=(9, 11))

# use a higher-contrast colormap (e.g., 'inferno' or 'magma' or 'turbo')
hb = ax.hexbin(
    lon_v, lat_v, C=price_v, gridsize=gridsize, reduce_C_function=np.median,
    extent=extent, mincnt=mincnt, cmap="inferno", norm=norm, linewidths=0.0, zor
)

# small, unobtrusive colorbar (in millions)
cb = plt.colorbar(hb, fraction=0.035, pad=0.015)
cb.ax.yaxis.set_major_formatter(FuncFormatter(lambda v, pos: f"{v/1e6:.1f}"))
cb.set_label("Median Price (million NZD)", fontsize=9)
cb.ax.tick_params(labelsize=8)

# remove axes, ticks, and outer frame
ax.set_xlim(extent[0], extent[1]); ax.set_ylim(extent[2], extent[3])
ax.set_axis_off()
```

```python
for spine in ax.spines.values():
    spine.set_visible(False)

# keep map proportion correct (avoid stretching)
ax.set_aspect('equal', adjustable='datalim')


# ---------- 2) overlay boundaries (robust) ----------
rc = gpd.read_file(resolve_shp(rc_path)).to_crs(4326)
ta = gpd.read_file(resolve_shp(ta_path)).to_crs(4326)
name_rc = find_name_col(rc, "rc"); name_ta = find_name_col(ta, "ta")

# thin national outline
rc.boundary.plot(ax=ax, color="#6e6e6e", linewidth=0.5, alpha=0.6, zorder=2)

# --- helper: normalized text for robust string match ---
def _norm(s: pd.Series) -> pd.Series:
    s = s.astype(str).str.strip().str.lower()
    try:
        s = s.str.normalize("NFKD").str.encode("ascii", "ignore").str.decode("as
    except Exception:
        pass
    return s.str.replace(r"[^a-z]+", " ", regex=True)

rc["_k"] = _norm(rc[name_rc]) if name_rc in rc.columns else ""
ta["_k"] = _norm(ta[name_ta]) if name_ta in ta.columns else ""

# --- Auckland & Wellington (name-based) ---
sel_akl_rc = rc["_k"].str.contains(r"\bauckland\b", na=False)
sel_wlg_rc = rc["_k"].str.contains(r"\bwellington\b", na=False)

# --- Dunedin City (name first, then geometry fallback) ---
sel_dud_ta = ta["_k"].str.contains(r"\bdunedin(\s+city)?\b", na=False)

if not sel_dud_ta.any():
    # geometry fallback: which TA polygon contains the Dunedin city point?
    dud_pt = gpd.GeoSeries([gpd.points_from_xy([170.5028], [-45.8788])[0]], crs=
    # fast spatial filter
    try:
        idx = ta.sindex.query(dud_pt.iloc[0], predicate="intersects")
        if len(idx) == 0:
            idx = ta.sindex.query(dud_pt.iloc[0], predicate="nearest")
        # refine by actual contains
        cand = ta.iloc[list(idx)]
        mask = cand.contains(dud_pt.iloc[0]).to_numpy()
        if mask.any():
            sel_dud_ta = ta.index.isin(cand[mask].index)
        else:
            # nearest as last resort
            sel_dud_ta = ta.index.isin([cand.index[0]])
    except Exception:
        pass

def safe_plot(gdf, mask, **kw):
    if bool(mask.any()):
        gdf.loc[mask].boundary.plot(ax=ax, **kw)

# highlight (ensure on top of hex layer)
rc.loc[sel_akl_rc].boundary.plot(ax=ax, color="#00C8C8", linewidth=1.5, zorder=4
rc.loc[sel_wlg_rc].boundary.plot(ax=ax, color="#00C8C8", linewidth=1.5, zorder=4
```

```python
ta.loc[sel_dud_ta].boundary.plot(ax=ax, color="#00C8C8", linewidth=1.5, zorder=4



# ---------- 3) city callouts (bigger text, on top of everything) ----------
cities = pd.DataFrame({
    "city": ["Auckland", "Wellington", "Dunedin"],
    "lat": [-36.8485, -41.2866, -45.8788],
    "lon": [174.7633, 174.7756, 170.5028],
})
gdf_pts = gpd.GeoDataFrame(cities, geometry=gpd.points_from_xy(cities["lon"], ci

def callout(ax, pt, text, dx, dy):
    ax.annotate(
        text,
        (pt.x, pt.y),                 # arrow tip (data coordinates)
        xytext=(dx, dy),              # move box relative to tip
        textcoords="offset points",   # <--- ensures dx/dy in pixels
        fontsize=15,
        fontweight="bold",
        bbox=dict(boxstyle="round,pad=0.3", fc="white", ec="#00C8C8", lw=1.2, al
        arrowprops=dict(arrowstyle="->", lw=2, color="#00C8C8"),
        ha="center", va="bottom",     # can change to 'left'/'right'/'top'
        annotation_clip=False
    )


akl_pt = gdf_pts.loc[gdf_pts["city"]=="Auckland","geometry"].iloc[0]
wlg_pt = gdf_pts.loc[gdf_pts["city"]=="Wellington","geometry"].iloc[0]
dud_pt = gdf_pts.loc[gdf_pts["city"]=="Dunedin","geometry"].iloc[0]

callout(ax, akl_pt, "Auckland", 100, 20)   # move box right & down
callout(ax, wlg_pt, "Wellington", 100, -20)
callout(ax, dud_pt, "Dunedin", 80, 20)

# compact title; no heavy frame
fig.suptitle(f"NZ Residential Price Heatmap (2018-2024)", fontsize=13, y=0.9, we
plt.tight_layout()

fig.patch.set_alpha(0)      # Clear figure background
ax.set_facecolor("none")    # Remove axes background

plt.show()
```

```
Ignoring fixed x limits to fulfill fixed data aspect with adjustable data limits.
/var/folders/4x/6bbjp51n6j5dvvsfk6gjgrzw0000gp/T/ipykernel_12208/3443888725.py:11
2: UserWarning: This pattern is interpreted as a regular expression, and has matc
h groups. To actually get the groups, use str.extract.
  sel_dud_ta = ta["_k"].str.contains(r"\bdunedin(\s+city)?\b", na=False)
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed x limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed x limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
```



Pattern 3 — Typical Home Price Projection & Volatility Estimate (1995 – 2024 → 2035)

This analysis expands upon the cleaned Combined Residential Property Sale Stats dataset to examine long-term growth patterns and future projection uncertainty for **Auckland, Wellington, and Dunedin**.

Data Preparation and Method

The study period covers 1995 – 2024 (inclusive) — a 30-year window that captures multiple housing cycles, interest-rate regimes, and structural market shifts.

Step 1 — Structural Tolerance Sensitivity Test To ensure that growth rates were not distorted by atypical property types or extreme observations, a ± 10 % structural tolerance was applied to floor area and land area. This window was selected after comparative testing of ± 5 %, ± 10 %, and ± 20 % ranges, balancing sample size stability with representativeness of “typical” homes. The ± 10 % setting retained sufficient observations while filtering out outliers that could bias median trends. For each city, annual medians were recalculated within this window, producing consistent year-to-year trajectories.

Step 2 — CAGR-Based Trend Modelling Using each city’s median price time series, the **Compound Annual Growth Rate (CAGR)** and Histogram-based Gradient Boosting Regressor (HGBR) Machine Learning was computed to capture the average yearly appreciation.

Step 3 — Model Verification (Accuracy Check) To validate that the CAGR model aligns with the real historical pattern, predicted values from the fitted trend line were compared with observed medians. The CAGR-Based Trend Modelling was chosed for its interpretability and robustness when data span long periods with compounding effects.Unlike polynomial or high-order regressions, CAGR avoids over-fitting and provides a clear baseline for forward extrapolation. The R^2 statistics exceeded 0.83 (Auckland), 0.97 (Wellington), and 0.94 (Dunedin), confirming strong agreement between modelled and actual data. This verification ensured the CAGR framework adequately represents long-term market movement before applying it to projections.

Typical Homes CAGR Prediction with ± 10 % Structural Tolerance was then selected.

- **Auckland (4.36 % CAGR, R^2 = 0.839)**
- **Wellington (5.48 % CAGR, R^2 = 0.971)**
- **Dunedin (5.52 % CAGR, R^2 = 0.935)**

Analysis and Interpretation of Patterns

Figure 3- Typical Home Price Projection & Volatility Estimate Using the 2024 median price of NZD 780 000 as a common baseline, each city’s future price path was projected to 2035 via its CAGR, with volatility bands illustrating historical price fluctuation.

Typical Home Price Projection & Volatility Estimate

City	2035 Projection (NZD)	Approx. Change	σ (log-return)
Auckland	1.25 M	+ 59.9 %	0.065
Wellington	1.40 M	+ 79.8 %	0.062
Dunedin	1.41 M	+ 80.6 %	0.079

Overall Interpretation

At a baseline of **NZD 780 000 (2024)**:

- Auckland homes are expected to grow modestly, reflecting a more mature, supply-constrained market.
- Wellington and Dunedin display stronger compounding potential, supported by historically higher volatility and faster median-price appreciation.
- By 2035, regional convergence emerges: despite different starting points, typical values cluster near NZD 1.4 million.

These findings reinforce that **growth potential and market stability vary geographically**, and historical volatility provides a quantitative window into each city’s long-term price behaviour.

Data Source

Combined Residential Property Sale Stats (2025 release), processed from `CSTDAT8700_DataDelivery_20250717.xlsx` and `CSTDAT8700_Output2_20250717.csv`. Analysis performed in Python (`pandas`, `matplotlib`, `numpy`) with city-level grouping, CAGR calculation, and log-return volatility estimation.

In [25]:
```
pip install prophet
```

```
Requirement already satisfied: prophet in /opt/anaconda3/envs/civil763/lib/python
3.13/site-packages (1.1.7)
Requirement already satisfied: cmdstanpy>=1.0.4 in /opt/anaconda3/envs/civil763/l
ib/python3.13/site-packages (from prophet) (1.2.5)
Requirement already satisfied: numpy>=1.15.4 in /opt/anaconda3/envs/civil763/lib/
python3.13/site-packages (from prophet) (2.3.1)
Requirement already satisfied: matplotlib>=2.0.0 in /opt/anaconda3/envs/civil763/
lib/python3.13/site-packages (from prophet) (3.10.6)
Requirement already satisfied: pandas>=1.0.4 in /opt/anaconda3/envs/civil763/lib/
python3.13/site-packages (from prophet) (2.3.1)
Requirement already satisfied: holidays<1,>=0.25 in /opt/anaconda3/envs/civil763/
lib/python3.13/site-packages (from prophet) (0.82)
Requirement already satisfied: tqdm>=4.36.1 in /opt/anaconda3/envs/civil763/lib/p
ython3.13/site-packages (from prophet) (4.67.1)
Requirement already satisfied: importlib_resources in /opt/anaconda3/envs/civil76
3/lib/python3.13/site-packages (from prophet) (6.5.2)
Requirement already satisfied: python-dateutil in /opt/anaconda3/envs/civil763/li
b/python3.13/site-packages (from holidays<1,>=0.25->prophet) (2.9.0.post0)
Requirement already satisfied: stanio<2.0.0,>=0.4.0 in /opt/anaconda3/envs/civil7
63/lib/python3.13/site-packages (from cmdstanpy>=1.0.4->prophet) (0.5.1)
Requirement already satisfied: contourpy>=1.0.1 in /opt/anaconda3/envs/civil763/l
ib/python3.13/site-packages (from matplotlib>=2.0.0->prophet) (1.3.2)
Requirement already satisfied: cycler>=0.10 in /opt/anaconda3/envs/civil763/lib/p
ython3.13/site-packages (from matplotlib>=2.0.0->prophet) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /opt/anaconda3/envs/civil763/
lib/python3.13/site-packages (from matplotlib>=2.0.0->prophet) (4.59.0)
Requirement already satisfied: kiwisolver>=1.3.1 in /opt/anaconda3/envs/civil763/
lib/python3.13/site-packages (from matplotlib>=2.0.0->prophet) (1.4.8)
Requirement already satisfied: packaging>=20.0 in /opt/anaconda3/envs/civil763/li
b/python3.13/site-packages (from matplotlib>=2.0.0->prophet) (25.0)
Requirement already satisfied: pillow>=8 in /opt/anaconda3/envs/civil763/lib/pyth
on3.13/site-packages (from matplotlib>=2.0.0->prophet) (11.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in /opt/anaconda3/envs/civil763/l
ib/python3.13/site-packages (from matplotlib>=2.0.0->prophet) (3.2.3)
Requirement already satisfied: pytz>=2020.1 in /opt/anaconda3/envs/civil763/lib/p
ython3.13/site-packages (from pandas>=1.0.4->prophet) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in /opt/anaconda3/envs/civil763/li
b/python3.13/site-packages (from pandas>=1.0.4->prophet) (2025.2)
Requirement already satisfied: six>=1.5 in /opt/anaconda3/envs/civil763/lib/pytho
n3.13/site-packages (from python-dateutil->holidays<1,>=0.25->prophet) (1.17.0)
Note: you may need to restart the kernel to use updated packages.
```

3.1 Structural Tolerence Sensitivity test - Impact on Sample Size

Purpose To define the most appropriate filtering bandwidth (±%) and time window for the **constant-structure** price-trend analysis (Pattern 3).

Testing Process We tested three tolerance levels (±5 %, ±10 %, ±20 %) around each city's 2024 *typical configuration* derived from Pattern 2 (≈ NZD 780 000 budget homes). For each bandwidth we evaluated yearly median prices and sample counts (`N`) to assess both **structural comparability** and **data continuity**.

Findings

- **±5 %:** Sample counts too low in several cities, leading to gaps and unstable medians.
- **±20 %:** Abundant samples but overly broad — mixing houses of different scale and density (e.g., 73 m^2 – 110 m^2 in Auckland are not conceptually comparable).
- **±10 %:** Balanced — sufficient yearly observations while keeping houses within a coherent structural range (≈ 82 – 101 m^2 for Auckland).

Final Decision

- Adopt a **±10 %** tolerance for all cities to maintain comparability and adequate sample sizes in long-term trend analysis.

Outcome This setup preserves internal consistency (same structure over time) while ensuring each city's trend is based on a realistic and data-rich period.
It also complements Pattern 2's budget-based view:

> Pattern 2 shows *what* NZD 780 k buys today; Pattern 3 shows *how* those same types of homes have changed in value over time.

In [26]:
```
df = pd.read_csv("Combined_Residential_Property_Sale_Stats.csv")
print(df.columns[:20])
```

```
Index(['QPID', 'Sale_ID', 'Building_ID', 'Situation_No', 'Street_Name',
       'Street_Suffix', 'Suburb', 'Town', 'Region_ID', 'Region_Name',
       'TA_Code', 'TA_Name', 'Meshblock', 'SAU', 'Sale_Tenure',
       'Price_Relationship', 'Sale_Date', 'Price_Net', 'Price_Chattels',
       'Price_Other'],
      dtype='object')
```

In [27]:
```
print(list(df.columns))
```

```
['QPID', 'Sale_ID', 'Building_ID', 'Situation_No', 'Street_Name', 'Street_Suffi
x', 'Suburb', 'Town', 'Region_ID', 'Region_Name', 'TA_Code', 'TA_Name', 'Meshbloc
k', 'SAU', 'Sale_Tenure', 'Price_Relationship', 'Sale_Date', 'Price_Net', 'Price_
Chattels', 'Price_Other', 'Price_Gross', 'CV_Capital_Value', 'LV_Land_Value', 'IV
_Improvements_Value', 'Revision_Date', 'Floor_Area', 'Site_Cover', 'Land_Area',
'Bldg_Construction', 'Bldg_Condition', 'Roof_Construction', 'Roof_Condition', 'Ca
tegory', 'LUD_Age', 'Land_Use_Desc', 'Surrounding_Improv_Class', 'Contour', 'Vie
w', 'Modernisation', 'House_Type', 'Deck', 'Driveway', 'No_Main_Roof_Garages', 'F
ree_Standing_Garages', 'Year_Built_Est', 'Landscaping_Quality', 'Lot_Position',
'School_Zone_1', 'School_Zone_2', 'Valuation_Ref', 'Latitude', 'Longitude', 'Bedr
ooms', 'Bathrooms']
```

In [28]:
```
print(df["Region_Name"].unique()[:20])
```

```
['Northland Region' 'Auckland (Unitary)' 'Gisborne (Unitary)'
 'Waikato Region' 'Bay of Plenty Region' 'Manawatu-Whanganui Region'
 'Hawkes Bay Region' 'Taranaki Region' 'Wellington Region'
 'Tasman Nelson Marlborough' 'West Coast Region' 'Canterbury Region'
 'Chathams' 'Otago Region' 'Southland Region']
```

In [29]:
```
import pandas as pd
import numpy as np
import re

# If df already exists, keep it; otherwise read
try:
```

```python
    df
except NameError:
    df = pd.read_csv("Combined_Residential_Property_Sale_Stats.csv")

# ---- A. Standardize column names ----
df.columns = (
    df.columns.astype(str)
      .str.strip()
      .str.replace(r"\s+", "_", regex=True)
)

# ---- B. Force numeric columns ----
def to_num(s):
    if pd.isna(s): return np.nan
    s = str(s).strip()
    if s in {"", "NA", "NaN", "None", "—", "-", "n/a"}:
        return np.nan
    s = re.sub(r"[,\s]", "", s)       # Remove commas/spaces
    s = re.sub(r"(m2|m²)$", "", s, flags=re.IGNORECASE)  # Remove m2/m² suffix
    try:
        return float(s)
    except:
        return np.nan

for col in ["Floor_Area", "Land_Area", "Price_Gross"]:
    if col in df.columns:
        df[col] = df[col].map(to_num)

# ---- C. Year column ----
if "Year" not in df.columns:
    df["Year"] = pd.to_datetime(df.get("Sale_Date"), errors="coerce").dt.year
```

In [30]:
```python
def city_mask(d, city):
    r = d.get("Region_Name", pd.Series(index=d.index)).fillna("")
    ta = d.get("TA_Name",     pd.Series(index=d.index)).fillna("")
    town = d.get("Town",      pd.Series(index=d.index)).fillna("")
    if city == "Auckland":
        return ta.str.contains("Auckland|Manukau|Waitakere|North Shore|Papakura|
                               case=False, regex=True) | r.str.contains("Acklan
    if city == "Wellington":
        pat = "Wellington|Lower Hutt|Upper Hutt|Porirua"
        return r.str.contains("Wellington", case=False) | ta.str.contains(pat, c
    if city == "Dunedin":
        return (r.str.contains("Otago", case=False)) & (ta.str.contains("Dunedin
    return r.str.contains(city, case=False)

def audit_city(d, city):
    m = city_mask(d, city)
    sub = d[m].copy()
    print(f"\n=== {city} ===")
    print("rows total:", len(sub))
    for col in ["Floor_Area", "Land_Area", "Price_Gross"]:
        if col in sub:
            ok = sub[col].notna() & (sub[col] > 0)
            print(f"{col}: non-null&>0 = {ok.sum()}  (dtype={sub[col].dtype})")
        else:
            print(f"{col}: MISSING COLUMN")

for c in ["Auckland", "Wellington", "Dunedin"]:
    audit_city(df, c)
```

```
=== Auckland ===
rows total: 1094357
Floor_Area: non-null&>0 = 989984  (dtype=float64)
Land_Area: non-null&>0 = 713864  (dtype=float64)
Price_Gross: non-null&>0 = 1094345  (dtype=float64)

=== Wellington ===
rows total: 365339
Floor_Area: non-null&>0 = 337184  (dtype=float64)
Land_Area: non-null&>0 = 283241  (dtype=float64)
Price_Gross: non-null&>0 = 365338  (dtype=float64)

=== Dunedin ===
rows total: 101226
Floor_Area: non-null&>0 = 94248  (dtype=float64)
Land_Area: non-null&>0 = 90817  (dtype=float64)
Price_Gross: non-null&>0 = 101226  (dtype=float64)
```

In [31]:
```python
# ==================== UNIVERSAL HEADER (run once before other scripts) =======
import re
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.ticker import StrMethodFormatter

# ---------------------------------------------------------------------
# 0) Load df if not already loaded + normalize column names
# ---------------------------------------------------------------------
try:
    df  # if already in memory, keep it
except NameError:
    df = pd.read_csv("Combined_Residential_Property_Sale_Stats.csv")

df.columns = df.columns.astype(str).str.strip().str.replace(r"\s+", "_", regex=T

# ---------------------------------------------------------------------
# 1) Robust numeric parsers & unit normalization to m² / NZD
# ---------------------------------------------------------------------
def _to_float_basic(x):
    if pd.isna(x): return np.nan
    s = str(x).strip()
    if s == "" or s.lower() in {"na","nan","none","n/a","—","-"}: return np.nan
    s = re.sub(r"[,\s]", "", s)
    # mark units
    s = re.sub(r"(m2|m²|㎡|sqm|sq\.?m)$", "M2UNIT", s, flags=re.I)
    s = re.sub(r"(ha|hectare|hectares)$", "HAUNIT", s, flags=re.I)
    s = re.sub(r"(acre|acres)$", "ACREUNIT", s, flags=re.I)
    try:
        if s.endswith("M2UNIT"):   return float(s[:-6])
        if s.endswith("HAUNIT"):   return float(s[:-6]) * 10000.0
        if s.endswith("ACREUNIT"): return float(s[:-8]) * 4046.8564224
        return float(s)
    except:
        return np.nan

def normalize_areas_prices(df,
                           floor_col="Floor_Area",
```

```python
                              land_col="Land_Area",
                              price_col="Price_Gross"):
    # floor
    if floor_col in df.columns:
        df[floor_col] = df[floor_col].map(_to_float_basic)
    # land
    if land_col in df.columns:
        df[land_col] = df[land_col].map(_to_float_basic)
        # Heuristic: if looks like hectares (q95 < 500 m²), multiply by 10,000
        land = pd.to_numeric(df[land_col], errors="coerce")
        if land.notna().any():
            q95 = np.nanpercentile(land.dropna(), 95)
            if np.isfinite(q95) and q95 < 500:
                df[land_col] = land * 10000.0
                print(f"[normalize] {land_col}: detected hectare-like values → c
    # price
    if price_col in df.columns:
        df[price_col] = df[price_col].map(_to_float_basic)
    # year
    if "Year" not in df.columns:
        df["Year"] = pd.to_datetime(df.get("Sale_Date"), errors="coerce").dt.yea
    # quick diagnostics
    def _qq(s):
        s = pd.to_numeric(s, errors="coerce").dropna()
        return ("nan","nan","nan") if s.empty else tuple(np.percentile(s, [5,50,
    print("[normalize] Floor_Area q5/50/95 (m²):", _qq(df.get("Floor_Area")))
    print("[normalize] Land_Area  q5/50/95 (m²):", _qq(df.get("Land_Area")))

normalize_areas_prices(df)

# ---------------------------------------------------------------------
# 2) City mask (robust across Region/TA/Town variants)
# ---------------------------------------------------------------------
def city_mask(d, city):
    region_col = next((c for c in d.columns if "region" in c.lower()), None)
    ta_col     = next((c for c in d.columns if ("ta" in c.lower()) or ("territor
    town_col   = next((c for c in d.columns if ("town" in c.lower()) or ("city"

    rn   = d.get(region_col, pd.Series("", index=d.index)).fillna("").astype(str
    ta   = d.get(ta_col,     pd.Series("", index=d.index)).fillna("").astype(str
    town = d.get(town_col,   pd.Series("", index=d.index)).fillna("").astype(str

    if city == "Auckland":
        return (rn.str.contains("Auckland", case=False, regex=False) |
                ta.str.contains("Auckland|Manukau|Waitakere|North Shore|Papakura
                                case=False, regex=True) |
                town.str.contains("Auckland", case=False, regex=False))
    if city == "Wellington":
        pat = "Wellington|Lower Hutt|Upper Hutt|Porirua"
        return (rn.str.contains("Wellington", case=False, regex=False) |
                ta.str.contains(pat, case=False, regex=True) |
                town.str.contains(pat, case=False, regex=True))
    if city == "Dunedin":
        pat_ta = r"Dunedin(\sCity|\sCity\sCouncil)?"
        pat_town = (r"Dunedin|Mosgiel|Port Chalmers|Waikouaiti|Middlemarch|Brigh
                    r"Waitati|Karitane|Aramoana|Seacliff|Ravensbourne|St\sKilda|
                    r"Green Island|Concord|Kaikorai|Caversham|Maori Hill|Opoho|N
        return ((rn.str.contains("Otago", case=False, regex=False) & ta.str.cont
                | town.str.contains(pat_town, case=False, regex=True))
```

```python
    # fallback: literal contains
    cols = [rn, ta, town]
    m = pd.Series(False, index=d.index)
    for col in cols:
        m |= col.str.contains(city, case=False, regex=False)
    return m

# ---------------------------------------------------------------------
# 3) Trendline utility: log-linear fit → CAGR & R², draw on given ax
# ---------------------------------------------------------------------
def fit_trend_and_annotate(ax, x_year, y_price, label="Trend", color=None,
                           annotate_xy=("right", "top")):
    """
    Fit ln(price) = a + b*year. Plot exp(a + b*year) on ax.
    Returns dict with slope, intercept, CAGR, R2.
    """
    s = pd.DataFrame({"x": x_year, "y": y_price}).dropna()
    if s.empty or (s["y"]<=0).all():
        return None
    X = s["x"].values
    Y = np.log(s["y"].values)  # log-space
    # simple OLS
    A = np.vstack([np.ones_like(X), X]).T
    coeff, *_ = np.linalg.lstsq(A, Y, rcond=None)
    a, b = coeff  # ln-price = a + b*year
    yhat = np.exp(a + b*X)
    ax.plot(X, yhat, linestyle="--", linewidth=2, label=label, color=color)

    # R² in log-space
    ss_res = np.sum((Y - (a + b*X))**2)
    ss_tot = np.sum((Y - Y.mean())**2)
    r2 = 1 - ss_res/ss_tot if ss_tot > 0 else np.nan
    cagr = np.exp(b) - 1.0

    # annotation
    tx = f"CAGR = {cagr*100:,.2f}%\n$R^2$ = {r2:.3f}"
    ha = "right" if annotate_xy[0]=="right" else "left"
    va = "top" if annotate_xy[1]=="top" else "bottom"
    ax.text(X.max() if ha=="right" else X.min(),
            max(yhat.min(), s['y'].min()) if va=="bottom" else max(yhat.max(), s
            tx, ha=ha, va=va, fontsize=10,
            bbox=dict(boxstyle="round,pad=0.3", fc="white", ec=color or "black",
    return {"intercept_ln": a, "slope_per_year_ln": b, "CAGR": cagr, "R2_log": r

# ---------------------------------------------------------------------
# 4) Band color/style presets (consistent across charts)
# ---------------------------------------------------------------------
BAND_COLORS = {0.05: "#1f77b4", 0.10: "#ff7f0e", 0.20: "#2ca02c"}  # blue / oran
def band_style(band):
    return dict(color=BAND_COLORS.get(band, None), linewidth=2)

# ---------------------------------------------------------------------
# 5) Quick helper: earliest year by city (sanity check)
# ---------------------------------------------------------------------
def print_earliest_years():
    for c in ["Auckland","Wellington","Dunedin"]:
        y = df[city_mask(df, c)]["Year"].min()
        print(f"Earliest {c} year in raw data: {y}")

print_earliest_years()
```

```
# Currency formatter (optional): apply to any price axis
def apply_nzd_formatter(ax):
    ax.yaxis.set_major_formatter(StrMethodFormatter('${x:,.0f}'))
# =================== END UNIVERSAL HEADER ===================
```

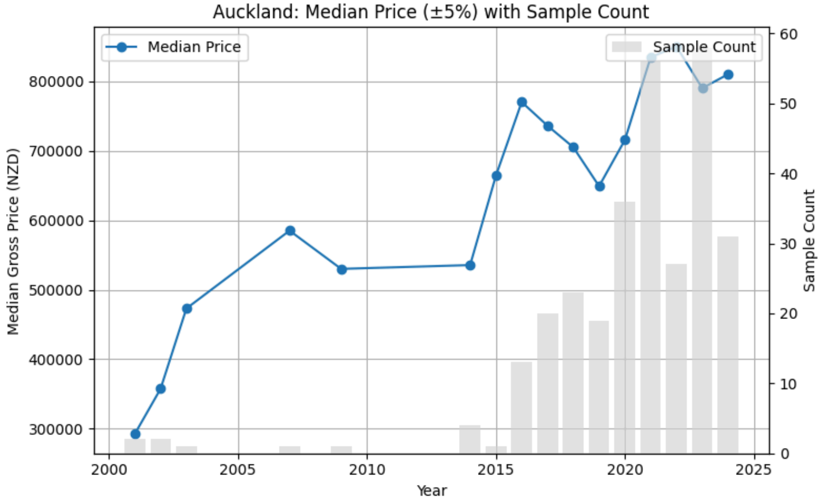
```
[normalize] Land_Area: detected hectare-like values → converted to m² (*10,000).
[normalize] Floor_Area q5/50/95 (m²): (np.float64(0.0), np.float64(110.0), np.flo
at64(250.0))
[normalize] Land_Area  q5/50/95 (m²): (np.float64(0.0), np.float64(627.0), np.flo
at64(1604.0))
Earliest Auckland year in raw data: 1990
Earliest Wellington year in raw data: 1990
/var/folders/4x/6bbjp51n6j5dvvsfk6gjgrzw0000gp/T/ipykernel_12208/1128587834.py:9
7: UserWarning: This pattern is interpreted as a regular expression, and has matc
h groups. To actually get the groups, use str.extract.
  return ((rn.str.contains("Otago", case=False, regex=False) & ta.str.contains(pa
t_ta, case=False, regex=True))
Earliest Dunedin year in raw data: 1990
```

3.1.1 Structural Tolerence ±5 %

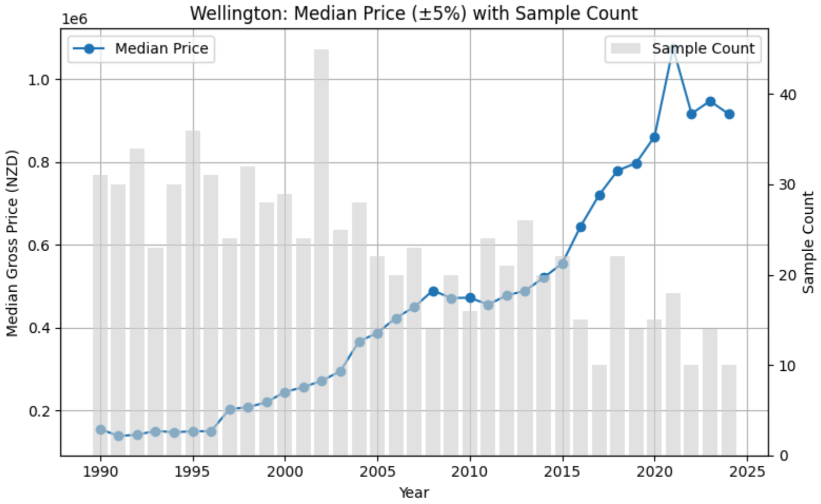
In [32]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# From Figure 2, the floor area and land area medians for each city
band = 0.05
config = {
    "Auckland":   {"floor": 92,  "land": 104},
    "Wellington": {"floor": 123, "land": 551},
    "Dunedin":    {"floor": 180, "land": 629},
}

def city_mask(df, city):
    rn = df["Region_Name"].fillna("")
    ta = df.get("TA_Name", pd.Series("", index=df.index)).fillna("")
    town = df.get("Town", pd.Series("", index=df.index)).fillna("")

    if city == "Auckland":
        return ta.str.contains("Auckland|Manukau|Waitakere|North Shore|Papakura|
                               case=False, regex=True)
    if city == "Wellington":
        return ta.str.contains("Wellington", case=False, regex=False)
    if city == "Dunedin":
        return (rn.str.contains("Otago", case=False, regex=False) &
                (ta.str.contains("Dunedin", case=False, regex=False) |
                 town.str.contains("Dunedin", case=False, regex=False)))


    # fallback: match anywhere
    return rn.str.contains(city, case=False, regex=False)

def plot_city(df, city, floor_med, land_med, band=0.1):
    m = city_mask(df, city)
    sub = df[m &
             df["Floor_Area"].between(floor_med*(1-band), floor_med*(1+band)) &
             df["Land_Area"].between(land_med*(1-band), land_med*(1+band))].copy

    # Analyzing subset
    print(f"{city}: rows={len(sub)}  "
          f"floor[{floor_med*(1-band):.1f},{floor_med*(1+band):.1f}], "
          f"land[{land_med*(1-band):.1f},{land_med*(1+band):.1f}]")
    if sub.empty:
        print("  ↳ Empty subset after filtering.")
        return

    counts = sub.groupby("Year")["Price_Gross"].size().reset_index(name="N")
    med = sub.groupby("Year")["Price_Gross"].median().reset_index(name="Median_G
    series = med.merge(counts, on="Year", how="left").sort_values("Year")

    fig, ax1 = plt.subplots(figsize=(8,5))
    ax2 = ax1.twinx()
    ax1.plot(series["Year"], series["Median_Gross"], marker="o", label="Median P
    ax2.bar(series["Year"], series["N"], alpha=0.6, color="lightgray", label="Sa

    ax1.set_title(f"{city}: Median Price (±5%) with Sample Count")
    ax1.set_xlabel("Year"); ax1.set_ylabel("Median Gross Price (NZD)")
    ax2.set_ylabel("Sample Count")
    ax1.legend(loc="upper left"); ax2.legend(loc="upper right")
    ax1.grid(True); fig.tight_layout(); plt.show()

# — Call the plotting function for each city —
for city, p in config.items():
    plot_city(df, city, p["floor"], p["land"], band=0.05)
```

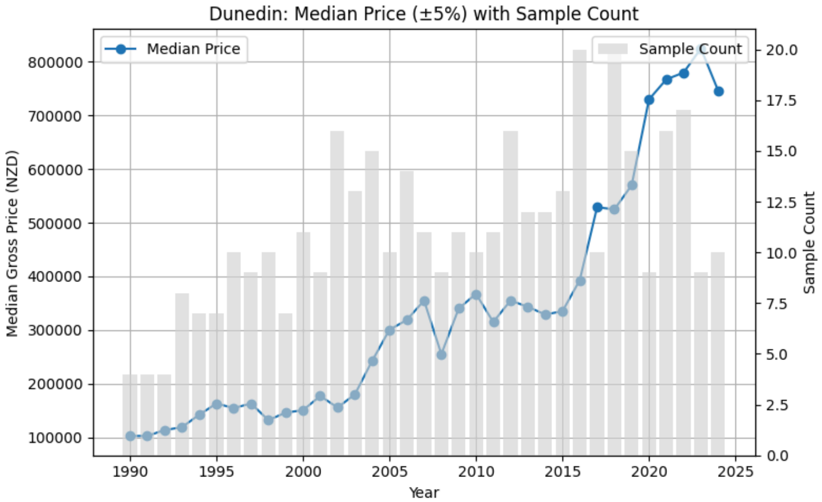
```
Auckland: rows=295  floor[87.4,96.6], land[98.8,109.2]
```




```
Wellington: rows=806  floor[116.8,129.2], land[523.4,578.6]
```




```
Dunedin: rows=389  floor[171.0,189.0], land[597.5,660.5]
```

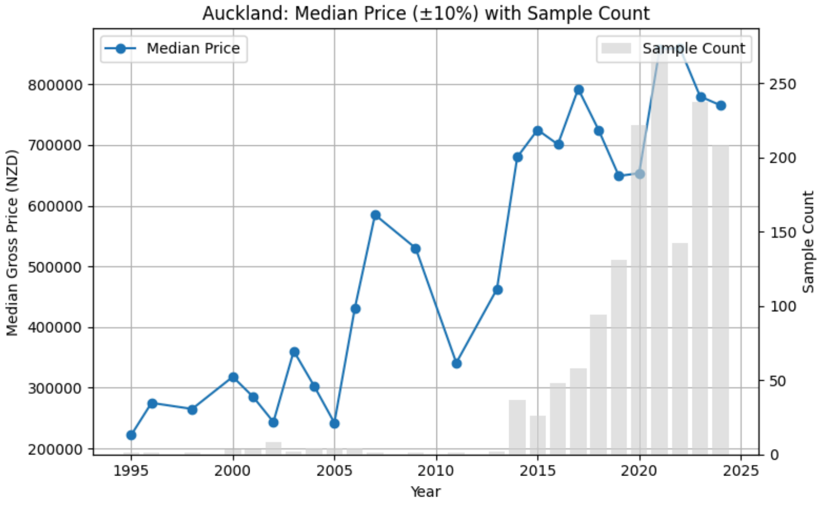


3.1.2 Structural Tolerence ±10 %

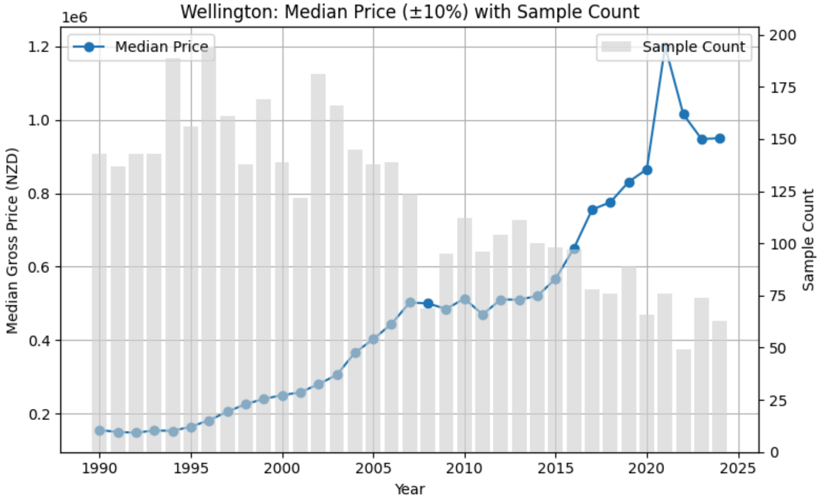
In [33]:

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# From Figure 2, the floor area and land area medians for each city
band = 0.1
config = {
    "Auckland":   {"floor": 92,  "land": 104},
    "Wellington": {"floor": 123, "land": 551},
    "Dunedin":    {"floor": 180, "land": 629},
}

def city_mask(df, city):
    rn = df["Region_Name"].fillna("")
    ta = df.get("TA_Name", pd.Series("", index=df.index)).fillna("")
    town = df.get("Town", pd.Series("", index=df.index)).fillna("")

    if city == "Auckland":
        return ta.str.contains("Auckland|Manukau|Waitakere|North Shore|Papakura|
                               case=False, regex=True)
    if city == "Wellington":
        return ta.str.contains("Wellington", case=False, regex=False)
    if city == "Dunedin":
        return (rn.str.contains("Otago", case=False, regex=False) &
                (ta.str.contains("Dunedin", case=False, regex=False) |
                 town.str.contains("Dunedin", case=False, regex=False)))


    # fallback: match anywhere
    return rn.str.contains(city, case=False, regex=False)

def plot_city(df, city, floor_med, land_med, band=0.1):
    m = city_mask(df, city)
    sub = df[m &
            df["Floor_Area"].between(floor_med*(1-band), floor_med*(1+band)) &
            df["Land_Area"].between(land_med*(1-band), land_med*(1+band))].copy

    # Analyzing subset
    print(f"{city}: rows={len(sub)}  "
          f"floor[{floor_med*(1-band):.1f},{floor_med*(1+band):.1f}], "
          f"land[{land_med*(1-band):.1f},{land_med*(1+band):.1f}]")
    if sub.empty:
        print("  ↳ Empty subset after filtering.")
        return

    counts = sub.groupby("Year")["Price_Gross"].size().reset_index(name="N")
    med = sub.groupby("Year")["Price_Gross"].median().reset_index(name="Median_G
    series = med.merge(counts, on="Year", how="left").sort_values("Year")

    fig, ax1 = plt.subplots(figsize=(8,5))
    ax2 = ax1.twinx()
    ax1.plot(series["Year"], series["Median_Gross"], marker="o", label="Median P
    ax2.bar(series["Year"], series["N"], alpha=0.6, color="lightgray", label="Sa

    ax1.set_title(f"{city}: Median Price (±10%) with Sample Count")
    ax1.set_xlabel("Year"); ax1.set_ylabel("Median Gross Price (NZD)")
    ax2.set_ylabel("Sample Count")
    ax1.legend(loc="upper left"); ax2.legend(loc="upper right")
    ax1.grid(True); fig.tight_layout(); plt.show()

# —— Call the plotting function for each city ——
for city, p in config.items():
    plot_city(df, city, p["floor"], p["land"], band=0.1)
```

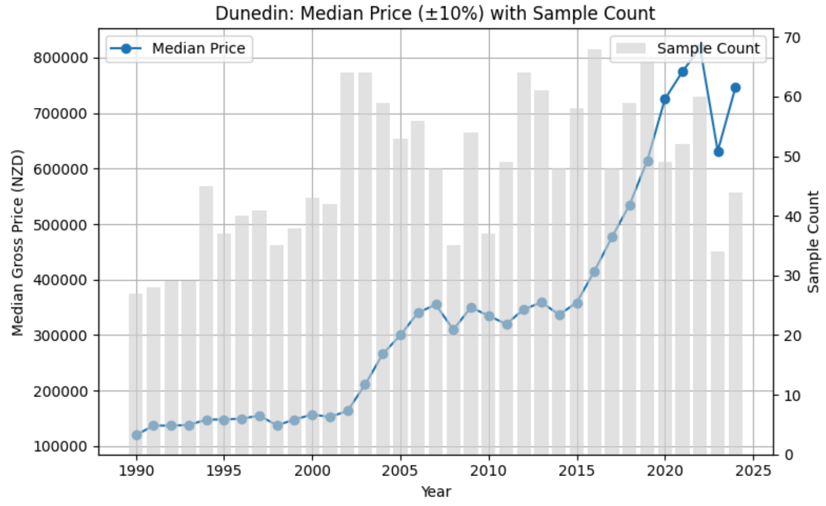
```
Auckland: rows=1512  floor[82.8,101.2], land[93.6,114.4]
```




```
Wellington: rows=4180  floor[110.7,135.3], land[495.9,606.1]
```




```
Dunedin: rows=1664  floor[162.0,198.0], land[566.1,691.9]
```



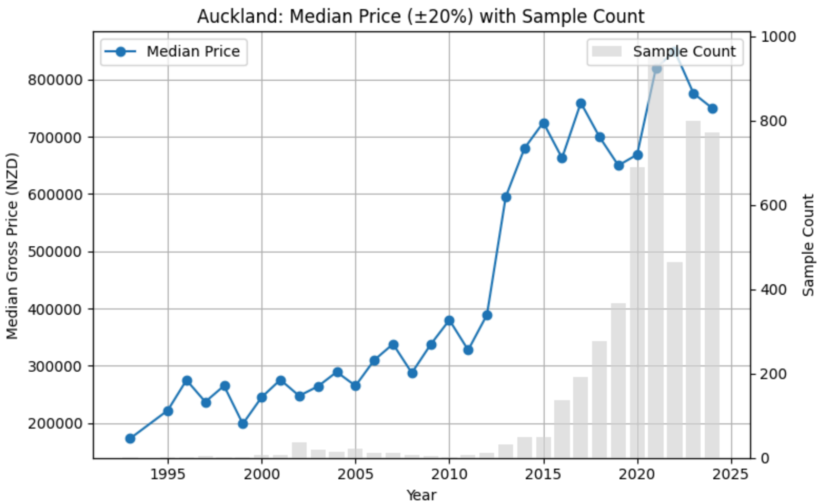
3.1.3 Structural Tolerence ±20 %

In [34]:

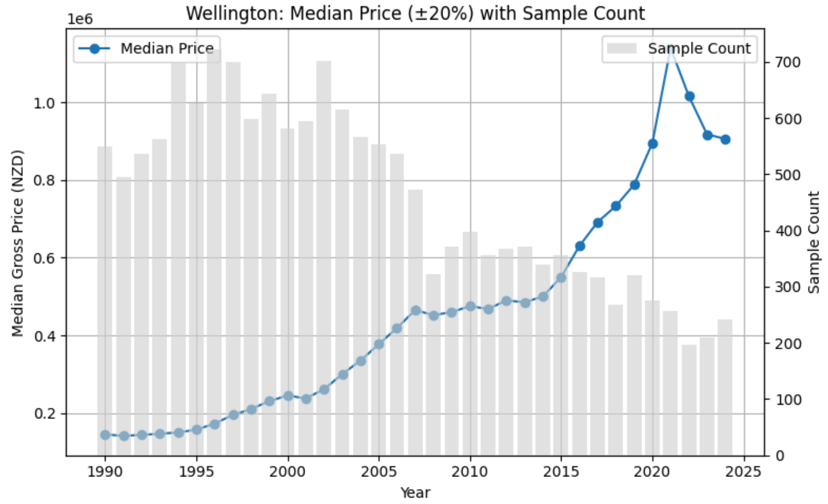
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# From Figure 2, the floor area and land area medians for each city
band = 0.05
config = {
    "Auckland":   {"floor": 92,  "land": 104},
    "Wellington": {"floor": 123, "land": 551},
    "Dunedin":    {"floor": 180, "land": 629},
}

def city_mask(df, city):
    rn = df["Region_Name"].fillna("")
    ta = df.get("TA_Name", pd.Series("", index=df.index)).fillna("")
    town = df.get("Town", pd.Series("", index=df.index)).fillna("")

    if city == "Auckland":
        return ta.str.contains("Auckland|Manukau|Waitakere|North Shore|Papakura|
                               case=False, regex=True)
    if city == "Wellington":
        return ta.str.contains("Wellington", case=False, regex=False)
    if city == "Dunedin":
        return (rn.str.contains("Otago", case=False, regex=False) &
                (ta.str.contains("Dunedin", case=False, regex=False) |
                 town.str.contains("Dunedin", case=False, regex=False)))


    # fallback: match anywhere
    return rn.str.contains(city, case=False, regex=False)

def plot_city(df, city, floor_med, land_med, band=0.2):
    m = city_mask(df, city)
    sub = df[m &
```

```python
            df["Floor_Area"].between(floor_med*(1-band), floor_med*(1+band)) &
            df["Land_Area"].between(land_med*(1-band), land_med*(1+band))].copy

    # Analyzing subset
    print(f"{city}: rows={len(sub)}  "
          f"floor[{floor_med*(1-band):.1f},{floor_med*(1+band):.1f}], "
          f"land[{land_med*(1-band):.1f},{land_med*(1+band):.1f}]")
    if sub.empty:
        print("  ↳ Empty subset after filtering.")
        return

    counts = sub.groupby("Year")["Price_Gross"].size().reset_index(name="N")
    med = sub.groupby("Year")["Price_Gross"].median().reset_index(name="Median_G
    series = med.merge(counts, on="Year", how="left").sort_values("Year")

    fig, ax1 = plt.subplots(figsize=(8,5))
    ax2 = ax1.twinx()
    ax1.plot(series["Year"], series["Median_Gross"], marker="o", label="Median P
    ax2.bar(series["Year"], series["N"], alpha=0.6, color="lightgray", label="Sa

    ax1.set_title(f"{city}: Median Price (±20%) with Sample Count")
    ax1.set_xlabel("Year"); ax1.set_ylabel("Median Gross Price (NZD)")
    ax2.set_ylabel("Sample Count")
    ax1.legend(loc="upper left"); ax2.legend(loc="upper right")
    ax1.grid(True); fig.tight_layout(); plt.show()

# —— Call the plotting function for each city ——
for city, p in config.items():
    plot_city(df, city, p["floor"], p["land"], band=0.2)
```

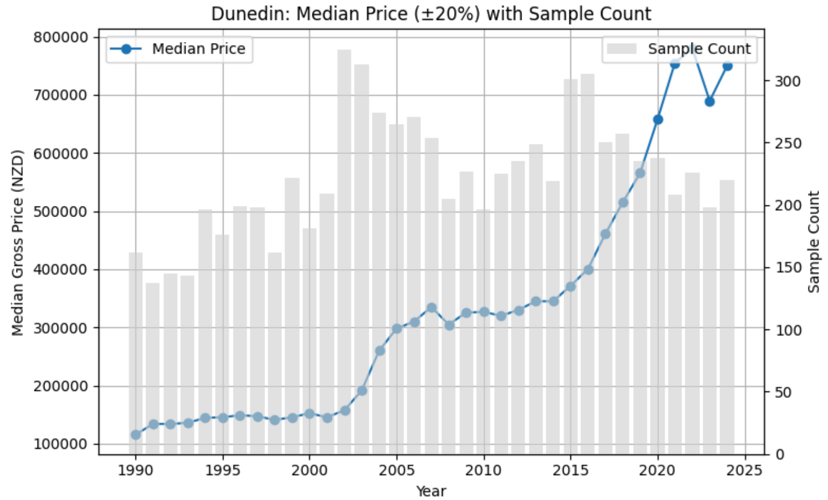
```
Auckland: rows=4964  floor[73.6,110.4], land[83.2,124.8]
```




```
Wellington: rows=16050  floor[98.4,147.6], land[440.8,661.2]
```




```
Dunedin: rows=7826  floor[144.0,216.0], land[503.2,754.8]
```



3.2 Time-Window Sensitivity Test (Auckland only)

Objective

After fixing the structural tolerance at **±10%**, this test evaluates how different time windows affect the **price-trend model** (log-linear) for Auckland's constant-structure homes.

Setup

- Dataset: Constant-structure homes filtered using each city's 2024 typical configuration (±10% band).
- City: **Auckland only** (Wellington and Dunedin remain unchanged — included for visual comparison only).
- Windows tested: `2014–2024` (short window) vs `1995–2024` (long window).
- Model: Annual median price fitted using
  Comparison focuses on fitted curve shape, residual trend, and predictive stability (CAGR consistency).

Observations

- **Short window (2014–2024)**:
  - The fitted line shows an almost **linear progression**, missing structural shifts and turning points before 2014.
  - Residuals display **systematic bias**, alternating over- and underestimation across years.
  - While the sample size is larger, the trend becomes **overly sensitive to end-year prices**, reducing model robustness.
- **Long window (1995–2024)**:
  - Produces a smoother slope and a **more stable CAGR**, reflecting the full market cycle.
  - Residuals are randomly distributed, indicating **better model fit**.
  - Early-year sample sizes are smaller, but the model captures long-term dynamics more accurately.

Conclusion

- The short window creates an artificial linear progression and weaker predictive reliability.
- Therefore, **Auckland's trend model is based on the 1990–2024 period**, despite smaller early samples, to ensure realistic long-term growth representation and comparability with other cities.

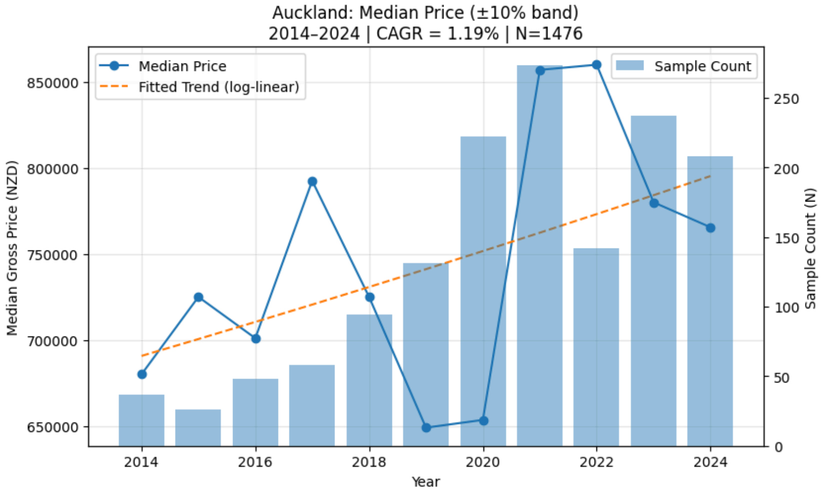
> Note: This sensitivity test modifies the **time window only** for Auckland; the **±10% structural tolerance** remains unchanged.

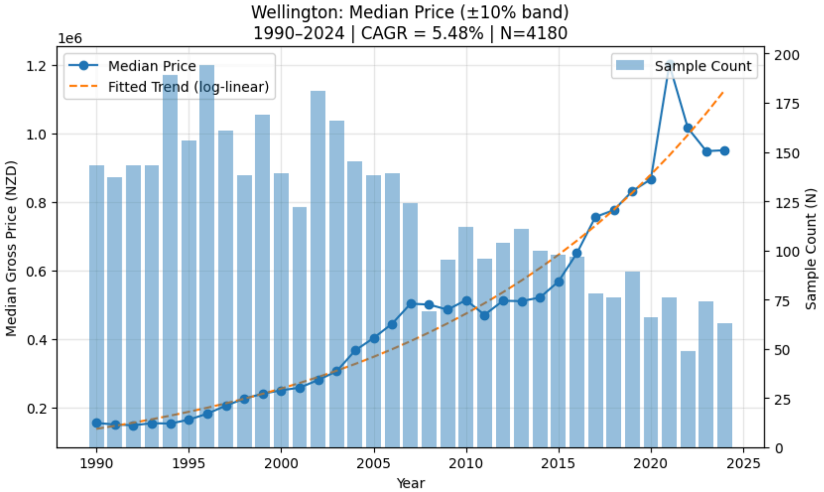
3.2.1 Auckland Time-Window = 2014-2024

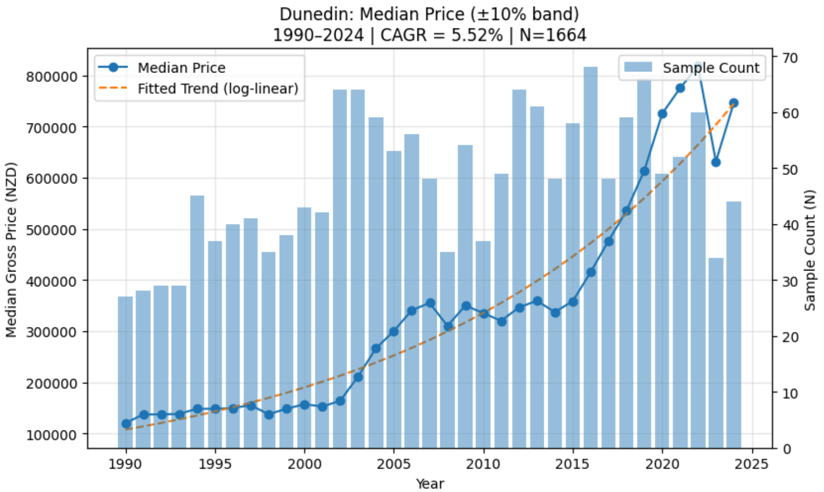
In [35]:

```
# ==============================================================
# Pattern 3 — ±10% structural tolerance + city-specific windows
# CAGR + log-linear price-trend per city, with inline plots only
# ==============================================================

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

# ---------- Config ----------
band = 0.10  # ±10%
config = {
    "Auckland":  {"floor": 92,  "land": 104},
    "Wellington": {"floor": 123, "land": 551},
    "Dunedin":   {"floor": 180, "land": 629},
}

# Time windows (you can switch Auckland to 1990–2024 as desired)
WINDOWS = {
    "Auckland":  (2014, 2024),
    "Wellington": (1990, 2024),
    "Dunedin":    (1990, 2024),
}

# ---------- Helpers ----------
def ensure_year(df):
    """Ensure Year column exists (int)."""
    if "Year" not in df.columns:
        df = df.copy()
        df["Year"] = pd.to_datetime(df["Sale_Date"], errors="coerce").dt.year
    return df

def city_mask(df, city):
    """Boolean mask selecting rows for a given city."""
    rn = df["Region_Name"].fillna("")
    ta = df.get("TA_Name", pd.Series("", index=df.index)).fillna("")
    town = df.get("Town", pd.Series("", index=df.index)).fillna("")

    if city == "Auckland":
        return ta.str.contains("Auckland|Manukau|Waitakere|North Shore|Papakura|
                               case=False, regex=True)
    if city == "Wellington":
        return ta.str.contains("Wellington", case=False, regex=False)
    if city == "Dunedin":
        return (rn.str.contains("Otago", case=False, regex=False) &
                (ta.str.contains("Dunedin", case=False, regex=False) |
                 town.str.contains("Dunedin", case=False, regex=False)))
    # fallback
    return rn.str.contains(city, case=False, regex=False)

def log_linear_fit(years, prices):
    """Fit ln(price) = a + b*year; return fitted prices and slope b."""
    x = np.asarray(years, dtype=float)
    y = np.log(np.asarray(prices, dtype=float))
    b, a = np.polyfit(x, y, 1)   # y ≈ a + b*x
    yhat = a + b * x
    return np.exp(yhat), b

def compute_cagr(p0, p1, y0, y1):
    n = (y1 - y0)
    if n <= 0 or p0 <= 0 or p1 <= 0:
        return np.nan
    return (p1 / p0) ** (1 / n) - 1

# ---------- Core plotting function ----------
def plot_city(df, city, floor_med, land_med, band=0.10):
    df = ensure_year(df)

    # Basic validity and constant-structure filter
    sub = df.loc[
        city_mask(df, city) &
        (df["Floor_Area"].astype(float) > 0) &
```

```python
        (df["Land_Area"].astype(float) > 0)
    ].copy()

    fa_low, fa_high = floor_med * (1 - band), floor_med * (1 + band)
    la_low, la_high = land_med * (1 - band), land_med * (1 + band)
    sub = sub[sub["Floor_Area"].between(fa_low, fa_high) &
              sub["Land_Area"].between(la_low, la_high)]

    if sub.empty:
        print(f"⚠️ {city}: empty subset after filtering.")
        return None

    # Time window
    y0, y1 = WINDOWS.get(city, (1990, 2024))
    sub = sub[(sub["Year"] >= y0) & (sub["Year"] <= y1)]

    # Yearly median price + sample counts
    yearly = (sub.groupby("Year")["Price_Gross"]
                .agg(MedianPrice="median", N="size")
                .reset_index()
                .sort_values("Year"))

    if yearly.empty:
        print(f"⚠️ {city}: no data inside {y0}-{y1}.")
        return None

    # CAGR using first/last available year inside the window
    p_start = yearly.iloc[0]["MedianPrice"]
    p_end   = yearly.iloc[-1]["MedianPrice"]
    cagr = compute_cagr(p_start, p_end,
                        int(yearly.iloc[0]["Year"]), int(yearly.iloc[-1]["Year"]
    cagr_pct = np.round(cagr * 100, 2) if pd.notna(cagr) else np.nan

    # Log-linear fitted trend
    fitted, slope = log_linear_fit(yearly["Year"], yearly["MedianPrice"])
    yearly["Fitted"] = fitted

    # ---------- Plot inline ----------
    fig, ax1 = plt.subplots(figsize=(8.5, 5.2))
    ax2 = ax1.twinx()

    ax1.plot(yearly["Year"], yearly["MedianPrice"], "o-", label="Median Price")
    ax1.plot(yearly["Year"], yearly["Fitted"], "--", label="Fitted Trend (log-li
    ax2.bar(yearly["Year"], yearly["N"], alpha=0.45, label="Sample Count")

    ax1.set_title(
        f"{city}: Median Price (±{int(band*100)}% band)\n"
        f"{y0}-{y1} | CAGR = {cagr_pct:.2f}% | N={int(yearly['N'].sum())}"
    )
    ax1.set_xlabel("Year")
    ax1.set_ylabel("Median Gross Price (NZD)")
    ax2.set_ylabel("Sample Count (N)")
    ax1.grid(True, alpha=0.3)
    ax1.legend(loc="upper left")
    ax2.legend(loc="upper right")
    plt.tight_layout()
    plt.show()

    return {
        "City": city,
        "Window": f"{y0}-{y1}",
        "CAGR_%": cagr_pct,
        "First_Year": int(yearly.iloc[0]["Year"]),
        "First_Median": float(p_start),
        "Last_Year": int(yearly.iloc[-1]["Year"]),
        "Last_Median": float(p_end),
        "Total_Samples": int(yearly["N"].sum())
    }

# ---------- Run for all cities & get a tidy summary ----------
summary_rows = []
for c, p in config.items():
    row = plot_city(df, c, floor_med=p["floor"], land_med=p["land"], band=band)
    if row is not None:
        summary_rows.append(row)

cagr_table = pd.DataFrame(summary_rows).sort_values("CAGR_%", ascending=False)
cagr_table
```







Out[35]:

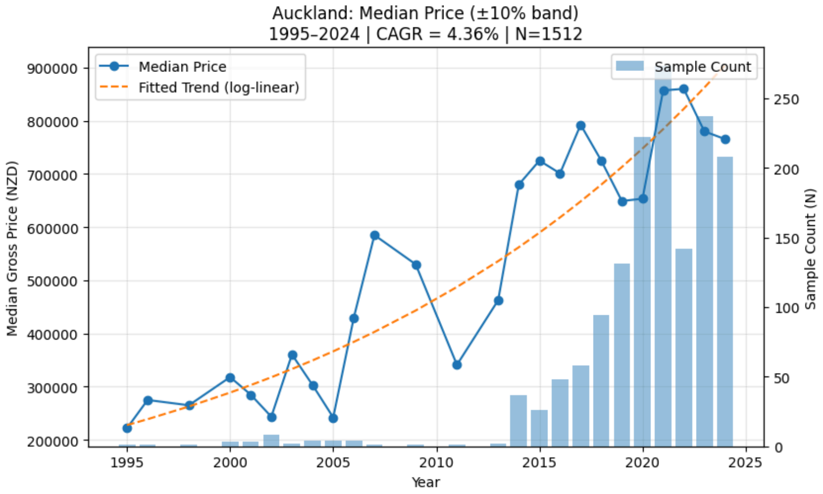
	City	Window	CAGR_%	First_Year	First_Median	Last_Year	Last_Median	Tota
2	Dunedin	1990-2024	5.52	1990	120000.0	2024	746000.0	
1	Wellington	1990-2024	5.48	1990	155000.0	2024	950000.0	
0	Auckland	2014-2024	1.19	2014	680000.0	2024	765500.0	

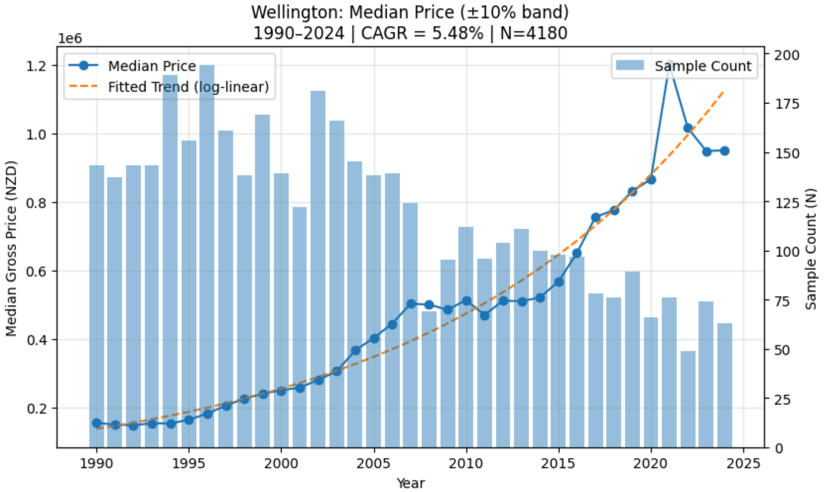
3.2.2 Auckland Time-Window = 1995-2024

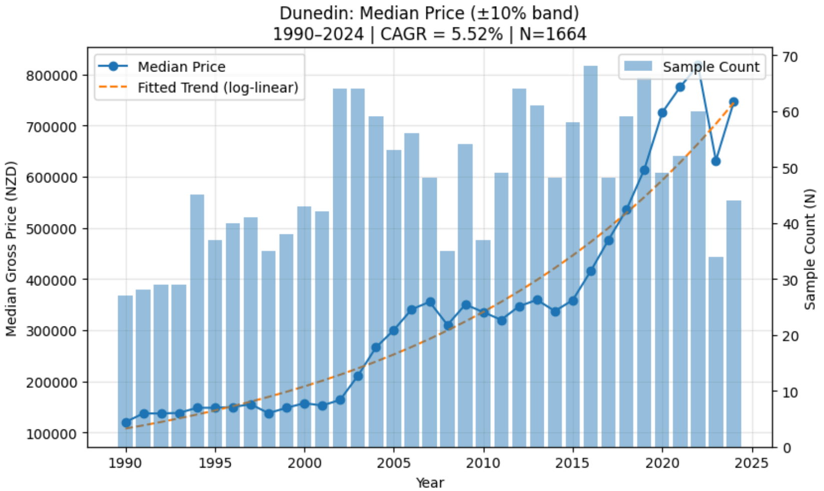
In [36]:

```python
# ==============================================================
# Pattern 3 — ±10% structural tolerance + city-specific windows
# CAGR + log-linear price-trend per city, with inline plots only
# ==============================================================

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

# ---------- Config ----------
band = 0.10  # ±10%
config = {
    "Auckland":  {"floor": 92,  "land": 104},
    "Wellington": {"floor": 123, "land": 551},
    "Dunedin":   {"floor": 180, "land": 629},
}

# Time windows (you can switch Auckland to 1990–2024 as desired)
WINDOWS = {
    "Auckland":  (1995, 2024),
    "Wellington": (1990, 2024),
    "Dunedin":    (1990, 2024),
}

# ---------- Helpers ----------
def ensure_year(df):
    """Ensure Year column exists (int)."""
    if "Year" not in df.columns:
        df = df.copy()
        df["Year"] = pd.to_datetime(df["Sale_Date"], errors="coerce").dt.year
    return df

def city_mask(df, city):
    """Boolean mask selecting rows for a given city."""
    rn = df["Region_Name"].fillna("")
    ta = df.get("TA_Name", pd.Series("", index=df.index)).fillna("")
    town = df.get("Town", pd.Series("", index=df.index)).fillna("")

    if city == "Auckland":
        return ta.str.contains("Auckland|Manukau|Waitakere|North Shore|Papakura|
                               case=False, regex=True)
    if city == "Wellington":
        return ta.str.contains("Wellington", case=False, regex=False)
    if city == "Dunedin":
        return (rn.str.contains("Otago", case=False, regex=False) &
                (ta.str.contains("Dunedin", case=False, regex=False) |
                 town.str.contains("Dunedin", case=False, regex=False)))
    # fallback
    return rn.str.contains(city, case=False, regex=False)

def log_linear_fit(years, prices):
    """Fit ln(price) = a + b*year; return fitted prices and slope b."""
    x = np.asarray(years, dtype=float)
    y = np.log(np.asarray(prices, dtype=float))
    b, a = np.polyfit(x, y, 1)   # y ≈ a + b*x
    yhat = a + b * x
    return np.exp(yhat), b

def compute_cagr(p0, p1, y0, y1):
```

```python
    n = (y1 - y0)
    if n <= 0 or p0 <= 0 or p1 <= 0:
        return np.nan
    return (p1 / p0) ** (1 / n) - 1

# ---------- Core plotting function ----------
def plot_city(df, city, floor_med, land_med, band=0.10):
    df = ensure_year(df)

    # Basic validity and constant-structure filter
    sub = df.loc[
        city_mask(df, city) &
        (df["Floor_Area"].astype(float) > 0) &
        (df["Land_Area"].astype(float) > 0)
    ].copy()

    fa_low, fa_high = floor_med * (1 - band), floor_med * (1 + band)
    la_low, la_high = land_med * (1 - band), land_med * (1 + band)
    sub = sub[sub["Floor_Area"].between(fa_low, fa_high) &
              sub["Land_Area"].between(la_low, la_high)]

    if sub.empty:
        print(f"⚠️ {city}: empty subset after filtering.")
        return None

    # Time window
    y0, y1 = WINDOWS.get(city, (1990, 2024))
    sub = sub[(sub["Year"] >= y0) & (sub["Year"] <= y1)]

    # Yearly median price + sample counts
    yearly = (sub.groupby("Year")["Price_Gross"]
                .agg(MedianPrice="median", N="size")
                .reset_index()
                .sort_values("Year"))

    if yearly.empty:
        print(f"⚠️ {city}: no data inside {y0}-{y1}.")
        return None

    # CAGR using first/last available year inside the window
    p_start = yearly.iloc[0]["MedianPrice"]
    p_end   = yearly.iloc[-1]["MedianPrice"]
    cagr = compute_cagr(p_start, p_end,
                        int(yearly.iloc[0]["Year"]), int(yearly.iloc[-1]["Year"]
    cagr_pct = np.round(cagr * 100, 2) if pd.notna(cagr) else np.nan

    # Log-linear fitted trend
    fitted, slope = log_linear_fit(yearly["Year"], yearly["MedianPrice"])
    yearly["Fitted"] = fitted

    # ---------- Plot inline ----------
    fig, ax1 = plt.subplots(figsize=(8.5, 5.2))
    ax2 = ax1.twinx()

    ax1.plot(yearly["Year"], yearly["MedianPrice"], "o-", label="Median Price")
    ax1.plot(yearly["Year"], yearly["Fitted"], "--", label="Fitted Trend (log-li
    ax2.bar(yearly["Year"], yearly["N"], alpha=0.45, label="Sample Count")

    ax1.set_title(
        f"{city}: Median Price (±{int(band*100)}% band)\n"
        f"{y0}-{y1} | CAGR = {cagr_pct:.2f}% | N={int(yearly['N'].sum())}"
    )
    ax1.set_xlabel("Year")
    ax1.set_ylabel("Median Gross Price (NZD)")
    ax2.set_ylabel("Sample Count (N)")
    ax1.grid(True, alpha=0.3)
    ax1.legend(loc="upper left")
    ax2.legend(loc="upper right")
    plt.tight_layout()
    plt.show()

    return {
        "City": city,
        "Window": f"{y0}-{y1}",
        "CAGR_%": cagr_pct,
        "First_Year": int(yearly.iloc[0]["Year"]),
        "First_Median": float(p_start),
        "Last_Year": int(yearly.iloc[-1]["Year"]),
        "Last_Median": float(p_end),
        "Total_Samples": int(yearly["N"].sum())
    }

# ---------- Run for all cities & get a tidy summary ----------
summary_rows = []
for c, p in config.items():
    row = plot_city(df, c, floor_med=p["floor"], land_med=p["land"], band=band)
    if row is not None:
        summary_rows.append(row)

cagr_table = pd.DataFrame(summary_rows).sort_values("CAGR_%", ascending=False)
cagr_table
```







Out[36]:

	City	Window	CAGR_%	First_Year	First_Median	Last_Year	Last_Median	Tota
2	Dunedin	1990-2024	5.52	1990	120000.0	2024	746000.0	
1	Wellington	1990-2024	5.48	1990	155000.0	2024	950000.0	
0	Auckland	1995-2024	4.36	1995	222000.0	2024	765500.0	

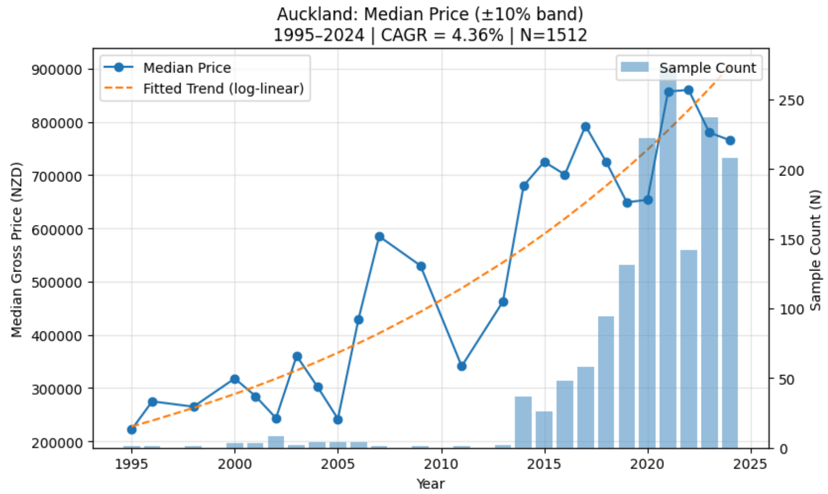
In [37]:

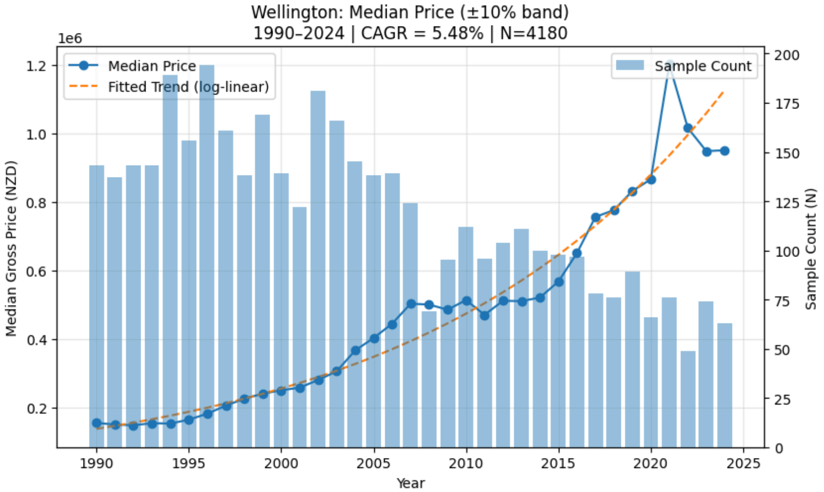
```
# ==============================================================
# Pattern 3 — ±10% structural tolerance + city-specific windows
# CAGR + log-linear price-trend per city, with inline plots only
# ==============================================================
```

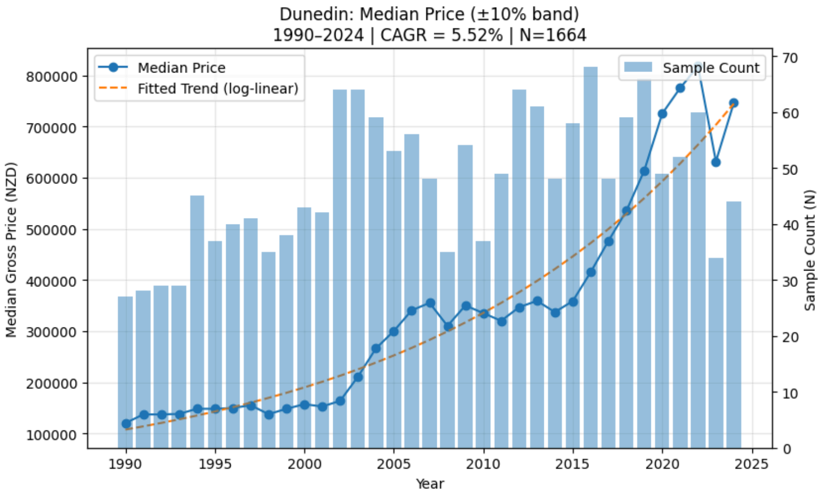
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# ---------- Config ----------
band = 0.10   # ±10%
config = {
    "Auckland":  {"floor": 92,  "land": 104},
    "Wellington": {"floor": 123, "land": 551},
    "Dunedin":   {"floor": 180, "land": 629},
}

# Time windows (you can switch Auckland to 1990–2024 as desired)
WINDOWS = {
    "Auckland":  (1995, 2024),
    "Wellington": (1990, 2024),
    "Dunedin":    (1990, 2024),
}

# ---------- Helpers ----------
def ensure_year(df):
    """Ensure Year column exists (int)."""
    if "Year" not in df.columns:
        df = df.copy()
        df["Year"] = pd.to_datetime(df["Sale_Date"], errors="coerce").dt.year
    return df

def city_mask(df, city):
    """Boolean mask selecting rows for a given city."""
    rn = df["Region_Name"].fillna("")
    ta = df.get("TA_Name", pd.Series("", index=df.index)).fillna("")
    town = df.get("Town", pd.Series("", index=df.index)).fillna("")

    if city == "Auckland":
        return ta.str.contains("Auckland|Manukau|Waitakere|North Shore|Papakura|
                               case=False, regex=True)
    if city == "Wellington":
        return ta.str.contains("Wellington", case=False, regex=False)
    if city == "Dunedin":
        return (rn.str.contains("Otago", case=False, regex=False) &
                (ta.str.contains("Dunedin", case=False, regex=False) |
                 town.str.contains("Dunedin", case=False, regex=False)))
    # fallback
    return rn.str.contains(city, case=False, regex=False)

def log_linear_fit(years, prices):
    """Fit ln(price) = a + b*year; return fitted prices and slope b."""
    x = np.asarray(years, dtype=float)
    y = np.log(np.asarray(prices, dtype=float))
    b, a = np.polyfit(x, y, 1)   # y ≈ a + b*x
    yhat = a + b * x
    return np.exp(yhat), b

def compute_cagr(p0, p1, y0, y1):
    n = (y1 - y0)
    if n <= 0 or p0 <= 0 or p1 <= 0:
        return np.nan
    return (p1 / p0) ** (1 / n) - 1
```

```python
# ---------- Core plotting function ----------
def plot_city(df, city, floor_med, land_med, band=0.10):
    df = ensure_year(df).copy()
    df = df.reset_index(drop=True)  # <--- add this line

    # Basic validity and constant-structure filter
    sub = df.loc[
        city_mask(df, city) &
        (df["Floor_Area"].astype(float) > 0) &
        (df["Land_Area"].astype(float) > 0)
    ].copy()

    fa_low, fa_high = floor_med * (1 - band), floor_med * (1 + band)
    la_low, la_high = land_med * (1 - band), land_med * (1 + band)
    sub = sub[sub["Floor_Area"].between(fa_low, fa_high) &
              sub["Land_Area"].between(la_low, la_high)]

    if sub.empty:
        print(f"⚠️ {city}: empty subset after filtering.")
        return None

    # Time window
    y0, y1 = WINDOWS.get(city, (1990, 2024))
    sub = sub[(sub["Year"] >= y0) & (sub["Year"] <= y1)]

    # Yearly median price + sample counts
    yearly = (sub.groupby("Year")["Price_Gross"]
                .agg(MedianPrice="median", N="size")
                .reset_index()
                .sort_values("Year"))

    if yearly.empty:
        print(f"⚠️ {city}: no data inside {y0}-{y1}.")
        return None

    # CAGR using first/last available year inside the window
    p_start = yearly.iloc[0]["MedianPrice"]
    p_end   = yearly.iloc[-1]["MedianPrice"]
    cagr = compute_cagr(p_start, p_end,
                        int(yearly.iloc[0]["Year"]), int(yearly.iloc[-1]["Year"]
    cagr_pct = np.round(cagr * 100, 2) if pd.notna(cagr) else np.nan

    # Log-linear fitted trend
    fitted, slope = log_linear_fit(yearly["Year"], yearly["MedianPrice"])
    yearly["Fitted"] = fitted

    # ---------- Plot inline ----------
    fig, ax1 = plt.subplots(figsize=(8.5, 5.2))
    ax2 = ax1.twinx()

    ax1.plot(yearly["Year"], yearly["MedianPrice"], "o-", label="Median Price")
    ax1.plot(yearly["Year"], yearly["Fitted"], "--", label="Fitted Trend (log-li
    ax2.bar(yearly["Year"], yearly["N"], alpha=0.45, label="Sample Count")

    ax1.set_title(
        f"{city}: Median Price (±{int(band*100)}% band)\n"
        f"{y0}-{y1} | CAGR = {cagr_pct:.2f}% | N={int(yearly['N'].sum())}"
    )
    ax1.set_xlabel("Year")
    ax1.set_ylabel("Median Gross Price (NZD)")
```

```python
    ax2.set_ylabel("Sample Count (N)")
    ax1.grid(True, alpha=0.3)
    ax1.legend(loc="upper left")
    ax2.legend(loc="upper right")
    plt.tight_layout()
    plt.show()

    return {
        "City": city,
        "Window": f"{y0}-{y1}",
        "CAGR_%": cagr_pct,
        "First_Year": int(yearly.iloc[0]["Year"]),
        "First_Median": float(p_start),
        "Last_Year": int(yearly.iloc[-1]["Year"]),
        "Last_Median": float(p_end),
        "Total_Samples": int(yearly["N"].sum())
    }

# ---------- Run for all cities & get a tidy summary ----------
summary_rows = []
for c, p in config.items():
    row = plot_city(df, c, floor_med=p["floor"], land_med=p["land"], band=band)
    if row is not None:
        summary_rows.append(row)

cagr_table = pd.DataFrame(summary_rows).sort_values("CAGR_%", ascending=False)
cagr_table
```







Out[37]:

	City	Window	CAGR_%	First_Year	First_Median	Last_Year	Last_Median	Tota
2	Dunedin	1990-2024	5.52	1990	120000.0	2024	746000.0	
1	Wellington	1990-2024	5.48	1990	155000.0	2024	950000.0	
0	Auckland	1995-2024	4.36	1995	222000.0	2024	765500.0	

3.3 Future Prediction Sensitivity Test

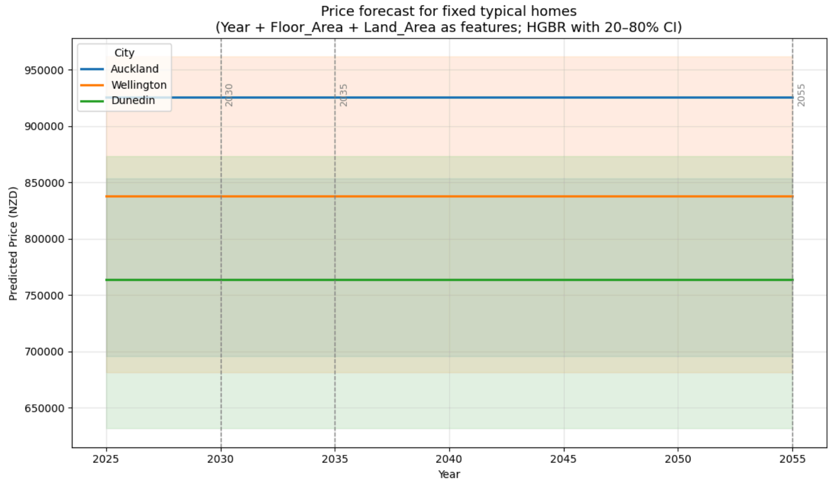
3.3.1 HGBR Machine Learning

In [38]:
```
%pip install xgboost
```

```
Requirement already satisfied: xgboost in /opt/anaconda3/envs/civil763/lib/python
3.13/site-packages (3.0.5)
Requirement already satisfied: numpy in /opt/anaconda3/envs/civil763/lib/python3.
13/site-packages (from xgboost) (2.3.1)
Requirement already satisfied: scipy in /opt/anaconda3/envs/civil763/lib/python3.
13/site-packages (from xgboost) (1.16.1)
Note: you may need to restart the kernel to use updated packages.
```

In [39]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.ensemble import HistGradientBoostingRegressor

# ========================
# Fixed Configuration
# ========================
TRAIN_YEARS = (2014, 2024)
FORECAST_YEARS = np.arange(2025, 2056)
CITIES = ["Auckland", "Wellington", "Dunedin"]
COLORS = {"Auckland":"#1f77b4","Wellington":"#ff7f0e","Dunedin":"#2ca02c"}  # Bl

# === Lock the typical home sizes ===
typicals = {
    "Auckland":  {"Floor_Area":92,  "Land_Area":104},
    "Wellington":{"Floor_Area":123, "Land_Area":551},
    "Dunedin":   {"Floor_Area":180, "Land_Area":629},
}

# ========================
# Data preparation
# ========================
df = df.copy()
if "Year" not in df.columns:
    df["Year"] = pd.to_datetime(df["Sale_Date"], errors="coerce").dt.year
df["Price_Gross"] = pd.to_numeric(df["Price_Gross"], errors="coerce")
df["Floor_Area"] = pd.to_numeric(df["Floor_Area"], errors="coerce")
df["Land_Area"]  = pd.to_numeric(df["Land_Area"], errors="coerce")

# Identify cities
def classify_city(row):
    t = str(row.get("Town","")).lower()
    ta = str(row.get("TA_Name","")).lower()
    r = str(row.get("Region_Name","")).lower()
    if "auckland" in (t+ta+r): return "Auckland"
    if any(k in (t+ta+r) for k in ["wellington","lower hutt","upper hutt","porir
    if "dunedin" in (t+ta+r): return "Dunedin"
    return np.nan

df["City"] = df.apply(classify_city, axis=1)
df = df[df["City"].isin(CITIES)]
df = df[(df["Year"].between(TRAIN_YEARS[0], TRAIN_YEARS[1])) & (df["Price_Gross"

# ========================
# Model training function
# ========================
```

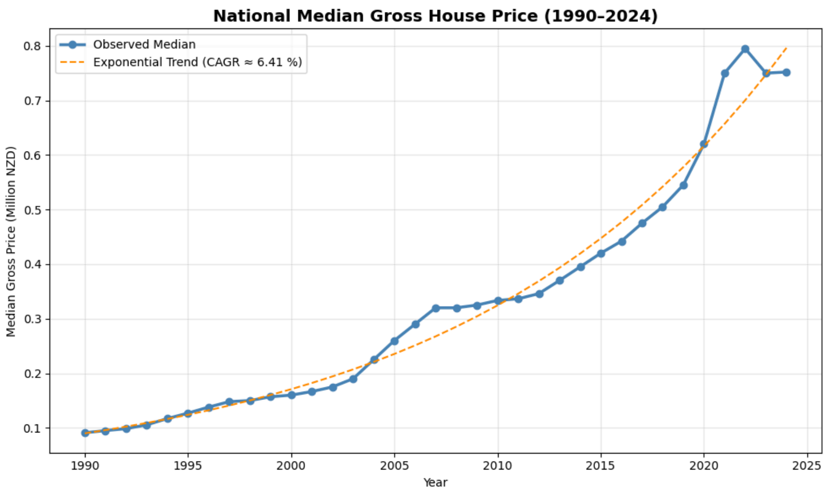
```python
FEATURES = ["Year","Floor_Area","Land_Area"]

def train_city_models(city):
    sub = df[df["City"]==city].dropna(subset=["Price_Gross"]+FEATURES)
    if len(sub)==0:
        print(f"[WARN] {city} has no valid rows, skip.")
        sub = pd.DataFrame({"Year":[2023,2024],"Floor_Area":[100,100],"Land_Area
    X = sub[FEATURES].astype(float)
    y = sub["Price_Gross"].astype(float)
    m_mean = HistGradientBoostingRegressor(loss="squared_error", max_iter=600, l
    m_lo   = HistGradientBoostingRegressor(loss="quantile", quantile=0.2, max_it
    m_hi   = HistGradientBoostingRegressor(loss="quantile", quantile=0.8, max_it
    for m in [m_mean,m_lo,m_hi]: m.fit(X,y)
    return {"mean":m_mean,"lo":m_lo,"hi":m_hi}

models = {c:train_city_models(c) for c in CITIES}

# ========================
# Forecast
# ========================
def forecast_city(city):
    cfg = typicals[city]
    future = pd.DataFrame({
        "Year": FORECAST_YEARS,
        "Floor_Area": float(cfg["Floor_Area"]),
        "Land_Area":  float(cfg["Land_Area"])
    })
    mean = models[city]["mean"].predict(future)
    lo   = models[city]["lo"].predict(future)
    hi   = models[city]["hi"].predict(future)
    return pd.DataFrame({"City":city,"Year":FORECAST_YEARS,"Pred_Mean":mean,"Pre

pred_all = pd.concat([forecast_city(c) for c in CITIES])

# ========================
# Plot
# ========================
plt.figure(figsize=(11,6.5))
for city in CITIES:
    sub = pred_all[pred_all["City"]==city]
    c = COLORS[city]
    plt.fill_between(sub["Year"], sub["Pred_Lo"], sub["Pred_Hi"], color=c, alpha
    plt.plot(sub["Year"], sub["Pred_Mean"], color=c, linewidth=2.2, label=city)

for yr in [2030,2035,2055]:
    plt.axvline(x=yr, color="gray", linestyle="--", linewidth=1)
    plt.text(yr+0.2, plt.gca().get_ylim()[1]*0.96, str(yr),
             rotation=90, va="top", ha="left", color="gray", fontsize=9)

plt.title("Price forecast for fixed typical homes\n(Year + Floor_Area + Land_Are
plt.xlabel("Year"); plt.ylabel("Predicted Price (NZD)")
plt.grid(True, alpha=0.3); plt.legend(title="City", loc="upper left")
plt.tight_layout(); plt.show()
```



3.3.2 Verify Model Accurancy using Nation Median Gross Price Trend

In [40]:
```python
# ==============================================================
# Pattern 3 — National Median Gross Sale Price Trend (1990–2024)
# ==============================================================

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

# ---------- Load and prepare ----------
df = pd.read_csv("Combined_Residential_Property_Sale_Stats.csv", low_memory=Fals

# Ensure correct types
df["Sale_Date"] = pd.to_datetime(df["Sale_Date"], errors="coerce")
df["Year"] = df["Sale_Date"].dt.year
df = df[df["Price_Gross"] > 0]

# ---------- Compute median by year ----------
annual = (
    df.groupby("Year")["Price_Gross"]
      .median()
      .reset_index()
      .query("1990 <= Year <= 2024")
)

# ---------- CAGR calculation ----------
start_price = annual.iloc[0]["Price_Gross"]
end_price   = annual.iloc[-1]["Price_Gross"]
years = annual.iloc[-1]["Year"] - annual.iloc[0]["Year"]
cagr = (end_price / start_price) ** (1 / years) - 1

# ---------- Plot ----------
plt.figure(figsize=(10,6))
plt.plot(annual["Year"], annual["Price_Gross"]/1e6, color="steelblue", linewidth
```

```python
# Fit and add trendline
z = np.polyfit(annual["Year"], np.log(annual["Price_Gross"]), 1)
plt.plot(
    annual["Year"],
    np.exp(np.polyval(z, annual["Year"])) / 1e6,
    linestyle="--", color="darkorange", label="Exponential Trend"
)

plt.title("National Median Gross House Price (1990–2024)", fontsize=14, weight="
plt.xlabel("Year")
plt.ylabel("Median Gross Price (Million NZD)")
plt.grid(alpha=0.3)
plt.legend(["Observed Median", f"Exponential Trend (CAGR ≈ {cagr*100:.2f} %)"])
plt.tight_layout()
plt.show()
```

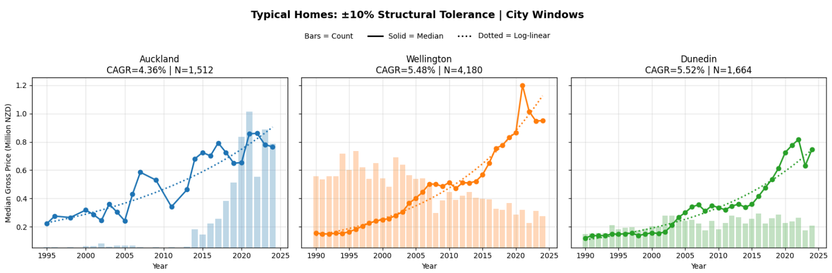


3.3.2 Selected Model - CAGR ±10% Structural Tolerance + City-Specific Windows (for Visualization Only)

In [41]:

```python
# ===============================================================
# Pattern 3 — ±10% structural tolerance + city-specific windows
# Three panels (AKL / WLG / DUN)
# Bars = Count | Solid = Median | Dotted = Log-linear fit
# Shared Y (price, M NZD) and shared RHS count scale
# ===============================================================

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
from matplotlib.ticker import FuncFormatter

# ---------- Config ----------
band = 0.10  # ±10%
config = {
    "Auckland":  {"floor": 92,  "land": 104},
    "Wellington": {"floor": 123, "land": 551},
    "Dunedin":   {"floor": 180, "land": 629},
}
WINDOWS = {
    "Auckland":  (1995, 2024),
    "Wellington": (1990, 2024),
    "Dunedin":    (1990, 2024),
}
city_colors = {"Auckland":"#1f77b4", "Wellington":"#ff7f0e", "Dunedin":"#2ca02c"

# ---------- Minimal helpers (your style, plus tiny robustness) ----------
def ensure_year(df):
    if "Year" not in df.columns and "Sale_Date" in df.columns:
        df = df.copy()
        df["Year"] = pd.to_datetime(df["Sale_Date"], errors="coerce").dt.year
    return df

def _coerce_numeric(s):
    return pd.to_numeric(
        s.astype(str).str.replace(r"[^\d\.\-]", "", regex=True).replace({"": Non
        errors="coerce"
    )

def city_mask(df, city):
    rn   = df.get("Region_Name", pd.Series("", index=df.index)).fillna("")
    ta   = df.get("TA_Name",     pd.Series("", index=df.index)).fillna("")
    town = df.get("Town",        pd.Series("", index=df.index)).fillna("")
    if city == "Auckland":
        return ta.str.contains("Auckland|Manukau|Waitakere|North Shore|Papakura|
                               case=False, regex=True)
    if city == "Wellington":
        return ta.str.contains("Wellington", case=False, regex=False)
    if city == "Dunedin":
        return (rn.str.contains("Otago", case=False, regex=False) &
                (ta.str.contains("Dunedin", case=False, regex=False) |
                 town.str.contains("Dunedin", case=False, regex=False)))
    return rn.str.contains(city, case=False, regex=False)

def log_linear_fit(years, prices):
    x = np.asarray(years, dtype=float)
    y = np.log(np.asarray(prices, dtype=float))
    b, a = np.polyfit(x, y, 1)
    yhat = a + b*x
    return np.exp(yhat)

def compute_cagr(p0, p1, y0, y1):
    n = (int(y1) - int(y0))
    if n <= 0 or p0 <= 0 or p1 <= 0:
        return np.nan
    return (p1 / p0) ** (1/n) - 1

# ---------- Build yearly data (once) ----------
def prepare_city(df, city, floor_med, land_med, band=0.10):
    df = ensure_year(df).copy()

    # Coerce numerics + ha→m² sanity (if areas look like hectares)
    for c in ["Price_Gross", "Land_Area"]:
        if c in df.columns:
            df[c] = _coerce_numeric(df[c])
    if "Land_Area" in df and df["Land_Area"].median(skipna=True) < 1000:
        df["Land_Area"] = df["Land_Area"] * 10_000
```

```python
    sub = df.loc[
        city_mask(df, city) &
        (df["Price_Gross"] > 0) & (df["Floor_Area"] > 0) & (df["Land_Area"] > 0)
    ].copy()
    if sub.empty:
        return None

    fa_low, fa_high = floor_med*(1-band), floor_med*(1+band)
    la_low, la_high = land_med *(1-band), land_med *(1+band)
    sub = sub[sub["Floor_Area"].between(fa_low, fa_high) &
              sub["Land_Area"].between(la_low, la_high)]
    if sub.empty:
        return None

    y0, y1 = WINDOWS.get(city, (1990, 2024))
    sub = sub[(sub["Year"] >= y0) & (sub["Year"] <= y1)]
    if sub.empty:
        return None

    yearly = (sub.groupby("Year")["Price_Gross"]
                .agg(MedianPrice="median", N="size")
                .reset_index()
                .sort_values("Year"))
    if yearly.empty:
        return None

    yearly["Fitted"] = log_linear_fit(yearly["Year"], yearly["MedianPrice"])

    p_start, p_end = yearly.iloc[0]["MedianPrice"], yearly.iloc[-1]["MedianPrice
    y_start, y_end = int(yearly.iloc[0]["Year"]), int(yearly.iloc[-1]["Year"])
    cagr = compute_cagr(p_start, p_end, y_start, y_end)

    return {
        "city": city,
        "yearly": yearly,
        "cagr_pct": None if pd.isna(cagr) else round(cagr*100, 2),
        "ywindow": (y0, y1),
        "total_N": int(yearly["N"].sum())
    }

# ---------- Side-by-side plot with shared axes ----------
def plot_pattern3_sbs(df, config, band=0.10):
    results = {city: prepare_city(df, city, p["floor"], p["land"], band)
               for city, p in config.items()}

    # Shared RHS (count) scale
    rhs_max = max((res["yearly"]["N"].max() for res in results.values() if res),

    fig, axes = plt.subplots(1, 3, figsize=(16, 5), sharey=True)
    plt.subplots_adjust(wspace=0.12)

    legend_elements = [
        Line2D([0],[0], color="#888", lw=0, label="Bars = Count"),
        Line2D([0],[0], color="#000", lw=2, label="Solid = Median"),
        Line2D([0],[0], color="#000", lw=2, ls=":", label="Dotted = Log-linear")
    ]

    for ax, city in zip(axes, ["Auckland","Wellington","Dunedin"]):
        res = results.get(city)
        color = city_colors.get(city, "#333")
        if not res:
            ax.text(0.5, 0.5, f"No data for {city}", ha="center", va="center")
            ax.set_xlabel("Year"); continue

        yearly = res["yearly"]

        # RHS bars (shared)
        ax2 = ax.twinx()
        ax2.bar(yearly["Year"], yearly["N"], alpha=0.28, color=color)
        ax2.set_ylim(0, rhs_max); ax2.set_yticks([])

        # LHS lines
        ax.plot(yearly["Year"], yearly["MedianPrice"]/1e6, "o-", lw=2, color=col
        ax.plot(yearly["Year"], yearly["Fitted"]/1e6, ":", lw=2, color=color)

        ax.set_xlabel("Year")
        ax.grid(alpha=0.3)
        ax.yaxis.set_major_formatter(FuncFormatter(lambda v, _: f"{v:,.1f}"))
        cagr = res["cagr_pct"]
        ax.set_title(f"{city}\nCAGR={cagr:.2f}% | N={res['total_N']:,}")

    axes[0].set_ylabel("Median Gross Price (Million NZD)")

    fig.legend(legend_elements, [e.get_label() for e in legend_elements],
               loc="upper center", ncol=3, frameon=False, bbox_to_anchor=(0.5, 0
    plt.suptitle(f"Typical Homes: ±{int(band*100)}% Structural Tolerance | City
                 fontsize=14, weight="bold", y=1.02)
    plt.tight_layout(rect=[0, 0, 1, 0.95])
    plt.show()

# ---------- Run ----------
plot_pattern3_sbs(df, config, band=band)
```



Pattern 3 Final Output: Figure 3

FIGURE 3 - Typical Homes Price Project & Voliality Estimate

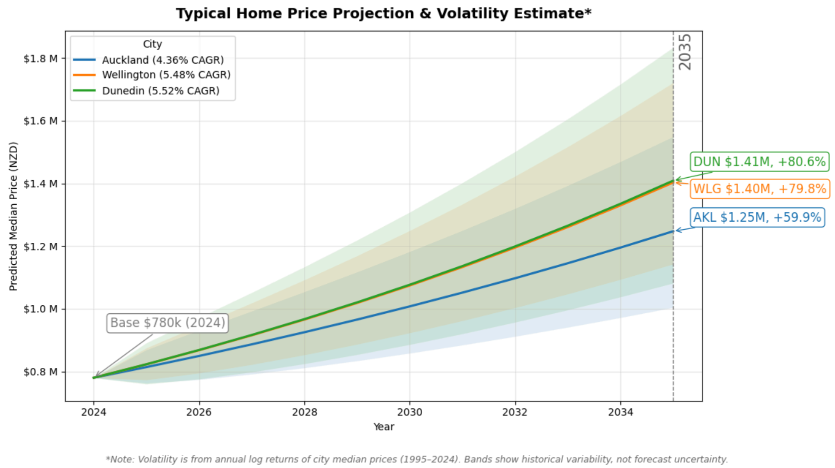
In [42]:

```python
# ==============================================================
# **Typical Home Price Projection (2024-2035)**
# Shaded = Historical Volatility (log returns 1990-2024)
```

```python
# ==============================================================

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter

# --------------------
# Inputs / parameters
# --------------------
CSV_PATH      = "Combined_Residential_Property_Sale_Stats.csv"  # <- update if di
TARGET_CITIES = ["Auckland", "Wellington", "Dunedin"]
CAGRS = {"Auckland": 4.36, "Wellington": 5.48, "Dunedin": 5.52}  # from your sum
BASE_YEAR      = 2024
BASE_PRICE     = 780_000
END_YEAR       = 2035
MILESTONES     = [2035]
VOL_WINDOW_YEARS = 30
MIN_YEARS_FOR_VOL = 6

# Colors
COLORS = {"Auckland": "#1f77b4", "Wellington": "#ff7f0e", "Dunedin": "#2ca02c"}

# --------------------
# Helper functions
# --------------------
def to_city(row):
    """Robustly map to the three target 'City' buckets from Town/Region/TA_Name.
    town = str(row.get("Town", "")).strip().lower()
    region = str(row.get("Region_Name", "")).strip().lower()
    ta = str(row.get("TA_Name", "")).strip().lower()

    if "auckland" in (town + region + ta):
        return "Auckland"
    if ("wellington" in (town + region + ta)) or ("lower hutt" in town) or ("upp
        return "Wellington"
    if ("dunedin" in (town + ta)) or ("otago" in region):
        return "Dunedin"
    return None


def project_path(base_price, cagr_pct, years, base_year):
    g = cagr_pct / 100.0
    years = np.asarray(years)
    return base_price * (1.0 + g) ** (years - base_year)


def millions_fmt(x, pos):
    return f"${x/1e6:.1f} M"


# --- generic callout helper (adjustable, supports fontsize) ---
def callout(ax, xy, text, offset=(8, 10), color="black", fontsize=9, lw=0.9):
    ax.annotate(
        text, xy=xy, xytext=offset, textcoords="offset points",
        fontsize=fontsize, color=color,
        bbox=dict(boxstyle="round,pad=0.25", fc="white", ec=color, lw=lw),
        arrowprops=dict(arrowstyle="->", lw=lw, color=color)
    )
```

```python
# --------------------
# Load & prepare
# --------------------
usecols = ["Sale_Date", "Price_Gross", "Town", "Region_Name", "TA_Name"]
df = pd.read_csv(CSV_PATH, usecols=usecols, low_memory=False)

df["Sale_Date"] = pd.to_datetime(df["Sale_Date"], errors="coerce")
df = df.dropna(subset=["Sale_Date", "Price_Gross"])
df["Year"] = df["Sale_Date"].dt.year
df = df[df["Price_Gross"] > 0]

df["City"] = df.apply(to_city, axis=1)
df = df[df["City"].isin(TARGET_CITIES)]

# Annual medians
annual = (df.groupby(["City", "Year"], as_index=False)["Price_Gross"]
            .median()
            .rename(columns={"Price_Gross": "MedianPrice"}))

# --------------------
# Estimate annual log-return volatility per city
# --------------------
vol_table = []
for city in TARGET_CITIES:
    s = (annual.loc[annual["City"] == city, ["Year", "MedianPrice"]]
                .dropna()
                .sort_values("Year")
                .set_index("Year")["MedianPrice"])
    r = np.log(s / s.shift(1)).dropna()

    if len(r) >= MIN_YEARS_FOR_VOL:
        r_tail = r.tail(VOL_WINDOW_YEARS) if len(r) > VOL_WINDOW_YEARS else r
        sigma = float(r_tail.std(ddof=1))
        n_obs = int(r_tail.shape[0])
    else:
        sigma = np.nan
        n_obs = int(len(r))

    vol_table.append({"City": city, "Sigma_logret": sigma, "N_years_used": n_obs

vol_df = pd.DataFrame(vol_table)

# --------------------
# Build projections & justified bands
# --------------------
years = np.arange(BASE_YEAR, END_YEAR + 1)
plt.figure(figsize=(11.5, 6.5))
ax = plt.gca()

for city in TARGET_CITIES:
    cagr = CAGRS[city]
    path = project_path(BASE_PRICE, cagr, years, BASE_YEAR)
    color = COLORS[city]
    plt.plot(years, path, lw=2.2, color=color, label=f"{city} ({cagr:.2f}% CAGR)

    sigma = float(vol_df.loc[vol_df["City"] == city, "Sigma_logret"].values[0])
    if np.isfinite(sigma) and sigma > 0:
        dt = (years - BASE_YEAR).astype(float)
        upper = path * np.exp(+sigma * np.sqrt(np.maximum(dt, 0.0)))
```

```
        lower = path * np.exp(-sigma * np.sqrt(np.maximum(dt, 0.0)))
        plt.fill_between(years, lower, upper, color=color, alpha=0.12, linewidth


# --- Milestones (keep dashed line; use callouts + rotated label) ---
abbr = {"Auckland": "AKL", "Wellington": "WLG", "Dunedin": "DUN"}

for y in MILESTONES:
    # Vertical dashed line
    ax.axvline(x=y, color="gray", linestyle="--", lw=1)

    # Rotated label at top
    ylim_top = ax.get_ylim()[1]
    ax.text(y + 0.1, ylim_top, f"{y}", rotation=90, va="top", ha="left",
            fontsize=15, color="0.35")

    # Callouts for each city's 2035 price & % increase
    for city in TARGET_CITIES:
        cagr = CAGRS[city]
        v = project_path(BASE_PRICE, cagr, [y], BASE_YEAR)[0]
        pct = (v / BASE_PRICE - 1) * 100
        txt = f"{abbr[city]} ${v/1e6:.2f}M, +{pct:.1f}%"

        offsets = {"Auckland": (20, 10), "Wellington": (20, -10), "Dunedin": (20
        callout(ax, (y, v), txt,
                offset=offsets.get(city, (8, 10)),
                color=COLORS[city],
                fontsize=12)


# --- Base price callout ---
callout(ax, (BASE_YEAR, BASE_PRICE), "Base $780k (2024)",
        offset=(16, 50), color="gray", fontsize=12)

# --------------------
# Styling
# --------------------

plt.title("Typical Home Price Projection & Volatility Estimate*",
          fontsize=14, fontweight="bold", pad=12)
plt.xlabel("Year")
plt.ylabel("Predicted Median Price (NZD)")
ax.yaxis.set_major_formatter(FuncFormatter(millions_fmt))
plt.grid(True, alpha=0.3)
plt.legend(title="City", loc="upper left")

# ---> Add footnote (keep everything else the same)
plt.figtext(
    0.5, 0.02,
    "*Note: Volatility is from annual log returns of city median prices (1995–20
    "Bands show historical variability, not forecast uncertainty.",
    ha="center", fontsize=9, style="italic", color="dimgray"
)

plt.tight_layout(rect=[0, 0.06, 1, 1])  # small bottom margin for footnote
plt.show()


# --------------------
# Table: projections for milestones
# --------------------
rows = []
for city in TARGET_CITIES:
    cagr = CAGRS[city]
    sigma = float(vol_df.loc[vol_df["City"] == city, "Sigma_logret"].values[0])
    for y in MILESTONES:
        base = project_path(BASE_PRICE, cagr, [y], BASE_YEAR)[0]
        if np.isfinite(sigma) and sigma > 0:
            dt = max(y - BASE_YEAR, 0)
            upper = base * np.exp(+sigma * np.sqrt(dt))
            lower = base * np.exp(-sigma * np.sqrt(dt))
        else:
            upper = np.nan
            lower = np.nan
        rows.append({
            "City": city,
            "Year": y,
            "Projection": base,
            "Band_Lower": lower,
            "Band_Upper": upper
        })
if sigma == 0 or np.isnan(sigma):
    print(f"⚠️ Warning: {city} has zero or missing volatility — check data cons:


proj_table = pd.DataFrame(rows)
with pd.option_context('display.float_format', lambda x: f"{x:,.4f}"):
    print("\nVolatility (log-return) estimates:")
    print(vol_df.to_string(index=False))
    print("\nMilestone projections (NZD):")
    print(proj_table.sort_values(["Year", "City"]).to_string(index=False))
```



```
Volatility (log-return) estimates:
      City  Sigma_logret  N_years_used
  Auckland        0.0654            30
Wellington        0.0617            30
   Dunedin        0.0795            30

Milestone projections (NZD):
      City  Year     Projection     Band_Lower     Band_Upper
  Auckland  2035 1,247,295.5629 1,004,224.5559 1,549,201.5328
   Dunedin  2035 1,408,566.0306 1,081,968.0344 1,833,749.4266
Wellington  2035 1,402,703.6757 1,142,997.9047 1,721,418.3803
```

In [58]:

```
!jupyter nbconvert --to html "/Users/Yuetian/Desktop/Civil 763/CIVIL 763 Project
  --output "2. CodeDoris_$(date +%Y%m%d_%H%M%S).html"
```

```
[NbConvertApp] Converting notebook /Users/Yuetian/Desktop/Civil 763/CIVIL 763 Pro
ject_Doris Zhao 2/2. Code_Doris Zhao.ipynb to html
[NbConvertApp] WARNING | Alternative text is missing on 28 image(s).
[NbConvertApp] Writing 3809192 bytes to /Users/Yuetian/Desktop/Civil 763/CIVIL 76
3 Project_Doris Zhao 2/2. CodeDoris_20251017_013919.html
```