Profit Protection, Forecasting & Production ML

Keras, XGBoost, Price Elasticity, Multi-Armed Bandits, IIoT, and AWS End-to-End Production Architecture

Profit Protector
Pricing & Arbitrage

Roadmap Architect
Forecasting & OEM

Full-Stack DS
Deploy & Cloud AWS

IIoT Expert
Manufacturing & SC

COVERS

Price Elasticity Modelling • Gray Market Arbitrage Detection • Demand Forecasting (LSTM & XGBoost)

Multi-Armed Bandits for Pricing A/B Tests • OEM Negotiation Analytics • IIoT Anomaly Detection

Predictive Maintenance • AWS SageMaker End-to-End • MLOps & CI/CD • Model Deployment

Stakeholder Communication Frameworks • FP&A Integration • Real-Time Scoring APIs

Wreetojyoti Ray

Senior Data Scientist | Strategic Business Analyst | ML Architect

© 2025 Wreetojyoti Ray. All Rights Reserved. No part of this publication may be reproduced without written permission.

Table of Contents

Chapter 1 — The Hybrid Role — Where Data Science Meets Business Strategy
- 1.1 The Four Pillars Explained
- 1.2 Stakeholder Map
- 1.3 KPI Framework
- 1.4 Communication Templates

Chapter 2 — Price Elasticity Modelling — The Profit Protector
- 2.1 Economics of Demand
- 2.2 Log-Log Regression for Elasticity
- 2.3 Keras Neural Elasticity Model
- 2.4 Channel-Level & SKU-Level Analysis
- 2.5 Optimal Price Recommendation Engine

Chapter 3 — Gray Market Arbitrage Detection & Defense
- 3.1 How Gray Markets Work
- 3.2 Arbitrage Threshold Mathematics
- 3.3 XGBoost Arbitrage Classifier
- 3.4 Keras Anomaly Detector
- 3.5 Real-Time Flagging System on AWS Lambda

Chapter 4 — Demand Forecasting for Hardware Assortment
- 4.1 Time Series Fundamentals
- 4.2 ARIMA Baseline
- 4.3 XGBoost with Lag Features
- 4.4 LSTM Forecasting with Keras
- 4.5 Ensemble Stacking
- 4.6 Forecast Uncertainty

Chapter 5 — OEM Negotiation Analytics & Assortment Planning
- 5.1 Volume Commitment Modelling
- 5.2 Margin Floor Analysis
- 5.3 Scenario Planning Engine
- 5.4 Data Packs for Brand Teams

Chapter 6 — A/B Testing & Multi-Armed Bandits for Pricing
- 6.1 Classical A/B Testing
- 6.2 Epsilon-Greedy
- 6.3 UCB Algorithm
- 6.4 Thompson Sampling
- 6.5 Bayesian A/B in Python
- 6.6 Contextual Bandits

Chapter 7 — Full-Stack Model Deployment on AWS
- 7.1 Model Packaging with Docker

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

7.2 SageMaker Training Jobs
7.3 SageMaker Endpoints (Real-Time)
7.4 Batch Transform
7.5 Lambda + API Gateway
7.6 MLflow Experiment Tracking
7.7 CI/CD with SageMaker Pipelines

Chapter 8 — IIoT & Manufacturing Data Science

8.1 IIoT Data Architecture
8.2 Anomaly Detection (Autoencoder)
8.3 Predictive Maintenance with LSTMs
8.4 Quality Control CNN
8.5 Supply Chain Optimisation
8.6 AWS IoT Core Integration

Chapter 9 — Discriminative & Predictive Models — The Full Toolkit

9.1 Logistic Regression for Classification
9.2 XGBoost Deep Dive
9.3 Keras Multi-Output Architecture
9.4 Gradient Boosting vs Deep Learning
9.5 Model Interpretability (SHAP)
9.6 Model Cards & Governance

Chapter 10 — Technical Leadership & Stakeholder Communication

10.1 FP&A; Integration
10.2 Executive Storytelling with Data
10.3 Sprint Planning for DS Teams
10.4 Model Risk Management

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

Chapter 1 Understanding the Four Pillars

The Hybrid Role — Where Data Science Meets Business Strategy

This role is rare precisely because it sits at the intersection of two worlds that rarely meet: the rigour of quantitative data science and the strategic instincts of a business analyst. A traditional data scientist optimises a model's accuracy. A traditional business analyst builds Excel scenarios. You must do both — simultaneously — and translate between them for every stakeholder in the organisation.

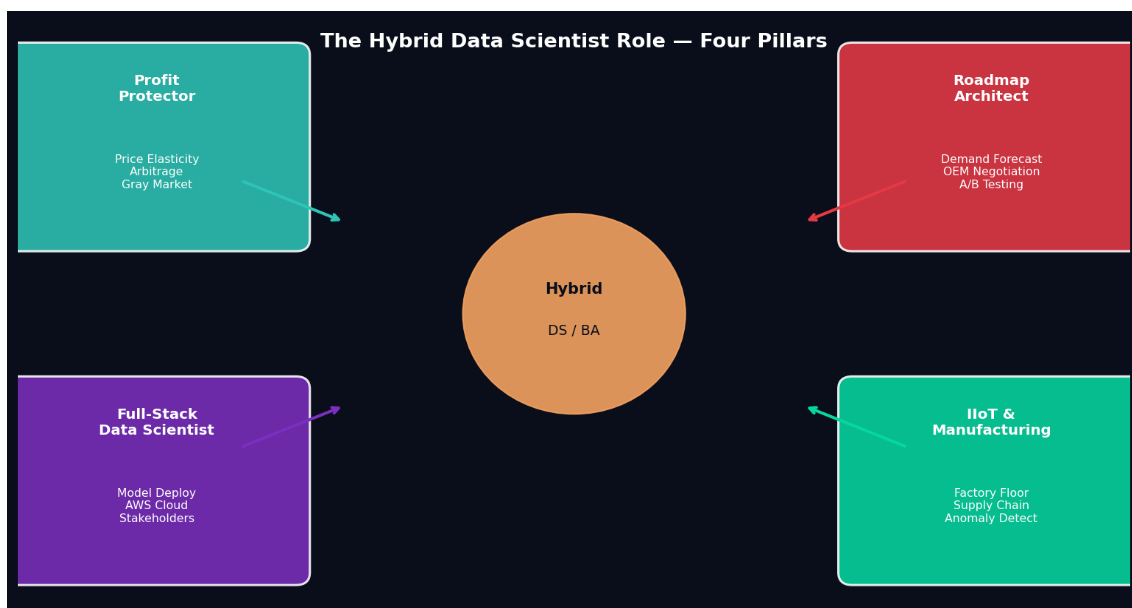

Figure 1.1 — The four pillars of the Hybrid DS/BA role and their interconnections.

1.1 The Four Pillars Explained

Profit Protector (Pricing & Arbitrage):

Every hardware SKU has a price-demand curve. Your job is to mathematically characterise that curve, find the revenue-maximising price point, and then detect when the pricing architecture is being exploited — whether by resellers, arbitrageurs, or subsidised channel leakage. You are directly protecting the company's gross margin.

Roadmap Architect (Forecasting & Assortment):

You convert raw sell-through data into a demand signal that the brand team uses to lock in OEM commitments months in advance. A forecast that is 10% high means overstocked inventory; 10% low means missed revenue and OEM penalty clauses. The stakes of your models are measured in millions.

Full-Stack Data Scientist (Deploy & Cloud):

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

Your models are useless in a Jupyter notebook. You are expected to package, deploy, monitor, and retrain models in a live production environment on AWS. This requires engineering discipline that most "research" data scientists lack.

IIoT & Manufacturing Integration:

Data from the factory floor — machine sensors, quality cameras, throughput counters — connects the physical supply chain to your financial models. Knowing when a machine is about to fail prevents a production gap that cascades into a demand forecast miss.

1.2 Stakeholder Map & Communication Templates

Stakeholder	What They Care About	How to Talk to Them	Your Deliverable
FP&A Team	Gross margin, budget vs actual	Use $, %, variance language	Price elasticity report
Brand Team	OEM relationships, device mix	Volume commitments, market share	Forecast data pack
Supply Chain	Inventory turns, lead times	Units, weeks-of-stock, fill rate	Demand plan + CI bands
Engineering	Model performance, infrastructure	RMSE, latency, SLA, APIs	Model cards, endpoints
C-Suite / Board	Revenue, competitive position	One-pagers, trend charts only	Executive dashboard
OEM Partners	Volume commitments, forecasts	Formal, data-backed, no fluff	Signed-off volume plan

■ Key Takeaways

■ This role is a force-multiplier: one person who can bridge math and money is worth more than two specialists.

■ Always translate model outputs into business language ($ impact, units at risk, margin %) before presenting.

■ Your primary internal customers are FP&A;, Brand, and Supply Chain — understand their planning cycles.

■ OEM commitments are legally binding; your forecast is the basis of multi-million dollar contracts.

■ Document every model assumption clearly — a wrong assumption deployed silently causes real financial damage.

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

Chapter 2 Quantifying Demand Sensitivity to Price Changes

Price Elasticity Modelling — The Profit Protector

2.1 The Economics of Demand & Elasticity

Price elasticity of demand (PED) measures how sensitively quantity demanded responds to price changes. It is the single most important economic parameter for a pricing strategist. Formally:

```
PED = (% Change in Quantity Demanded) / (% Change in Price) = (dQ/Q) / (dP/P)
```

When |PED| > 1, demand is elastic — consumers are very price-sensitive and a price increase causes a proportionally larger drop in volume, reducing total revenue. When |PED| < 1, demand is inelastic — consumers will buy regardless of modest price moves (e.g. enterprise contracts). For hardware: consumer smartphones are typically elastic (people compare prices constantly), while B2B/enterprise devices are often inelastic (locked into procurement cycles).

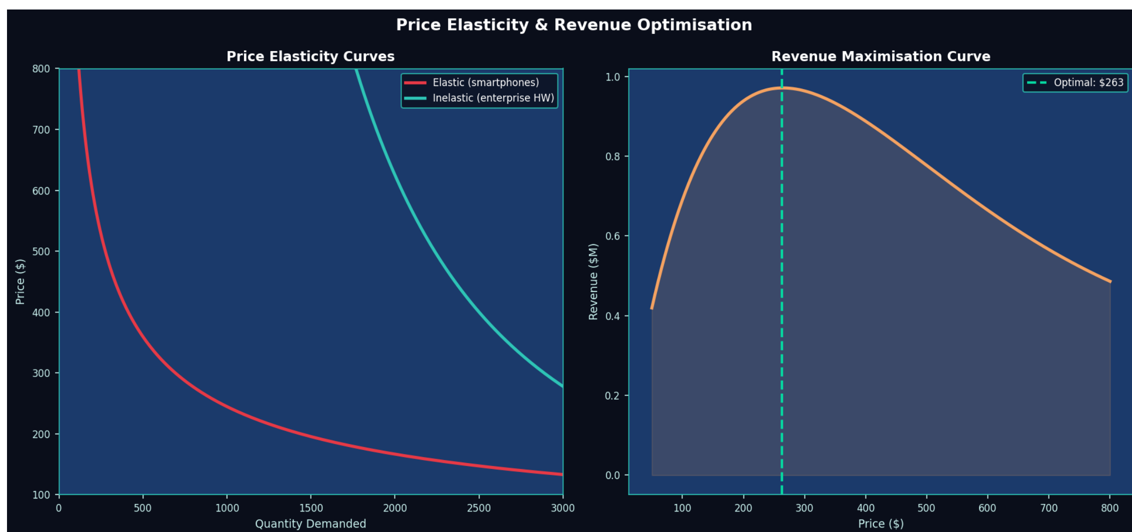

Figure 2.1 — Price-demand curves and revenue maximisation across elasticity regimes.

2.2 Log-Log Regression — The Classical Elasticity Estimator

The cleanest way to estimate elasticity from historical POS data is the log-log regression. Taking natural logs of both sides of the demand function Q = A * P^(-e) yields a linear model whose slope is directly the elasticity coefficient:

```
ln(Q) = ln(A) + e * ln(P) + b1*ln(Promo) + b2*Season + e_t
```

```
import pandas as pd
import numpy as np
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
import statsmodels.api as sm

# Assume df has: price, quantity, promo_flag, season_index, competitor_price
df = pd.read_csv("pos_data.csv")

# Log-transform price-sensitive vars
df["ln_qty"]   = np.log(df["quantity"])
df["ln_price"] = np.log(df["price"])
df["ln_comp"]  = np.log(df["competitor_price"])

# OLS with statsmodels for confidence intervals
X = sm.add_constant(df[["ln_price","ln_comp","promo_flag","season_index"]])
model = sm.OLS(df["ln_qty"], X).fit()

print(model.summary())
# ln_price coefficient IS the own-price elasticity
elasticity = model.params["ln_price"]
print(f"Price Elasticity: {elasticity:.3f}")
# e.g. -1.82 means 1% price increase → 1.82% volume decrease

# Cross-elasticity (competitor impact)
cross_elast = model.params["ln_comp"]
print(f"Cross-Price Elasticity: {cross_elast:.3f}")
# Positive value means our volume rises when competitor raises price

# Optimal price (maximises revenue) from elasticity
# Revenue = P * Q  ; at optimum dRevenue/dP = 0
# Optimal markup = e / (e + 1)
optimal_markup = elasticity / (elasticity + 1)
print(f"Lerner Index / Optimal Markup: {optimal_markup:.3f}")
```

2.3 Keras Neural Elasticity Model

The log-log regression assumes a constant elasticity across all price levels. In reality, elasticity is heterogeneous — it varies by price tier, channel, season, and promotional context. A neural network can capture these non-linearities without specifying them explicitly. The multi-head Keras architecture below simultaneously predicts demand AND outputs a dynamic elasticity estimate.

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

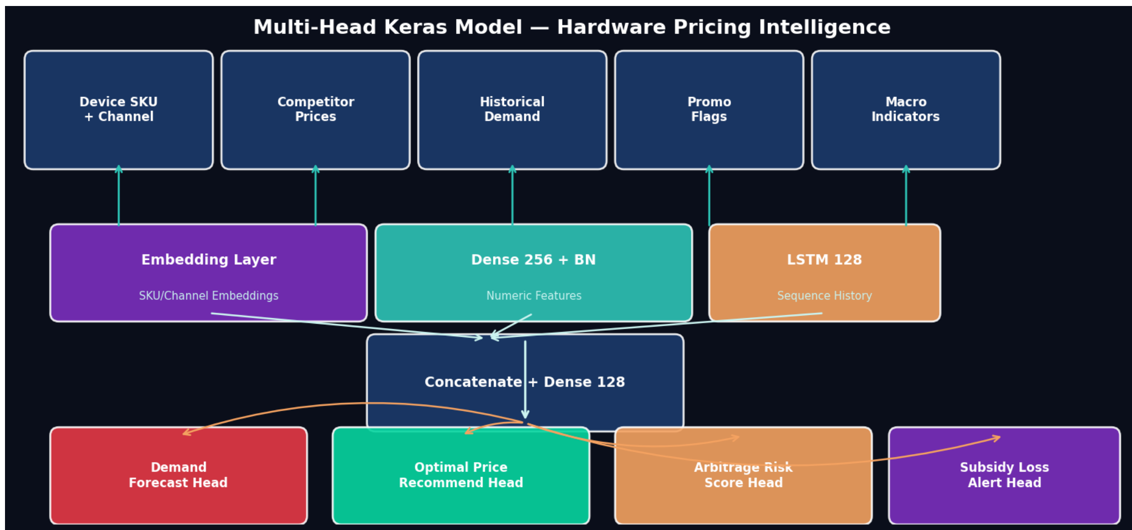

Figure 2.2 — Multi-head Keras architecture for pricing intelligence.

```
import keras
from keras import layers
import numpy as np

def build_elasticity_model(n_skus, n_channels, n_numeric_features):
    """
    Multi-output model predicting:
    1. log_demand (regression)
    2. optimal_price_delta (regression)
    3. arbitrage_risk (binary classification)
    """
    # --- Categorical inputs (embeddings) ---
    sku_input     = keras.Input(shape=(1,), name="sku_id", dtype="int32")
    channel_input = keras.Input(shape=(1,), name="channel_id", dtype="int32")
    seq_input     = keras.Input(shape=(12, 6), name="price_history")  # 12 months, 6 fea
tures
    num_input     = keras.Input(shape=(n_numeric_features,), name="numeric_features")

    # SKU embedding (captures per-SKU elasticity baseline)
    sku_emb = layers.Embedding(n_skus, 16, name="sku_embed")(sku_input)
    sku_emb = layers.Flatten()(sku_emb)

    # Channel embedding (direct vs carrier vs retail elasticity differs greatly)
    ch_emb  = layers.Embedding(n_channels, 8, name="channel_embed")(channel_input)
    ch_emb  = layers.Flatten()(ch_emb)

    # Sequence branch — captures price trends, competitor moves
    seq_x   = layers.LSTM(64, return_sequences=False, dropout=0.2)(seq_input)

    # Numeric branch — current price, promo, macro indicators
    num_x   = layers.Dense(64, activation="relu")(num_input)
    num_x   = layers.BatchNormalization()(num_x)
    num_x   = layers.Dense(32, activation="relu")(num_x)

    # Merge all branches
    merged  = layers.Concatenate()([sku_emb, ch_emb, seq_x, num_x])
    shared  = layers.Dense(128, activation="relu")(merged)
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
    shared  = layers.Dropout(0.3)(shared)
    shared  = layers.Dense(64, activation="relu")(shared)

    # Head 1: Demand forecast (log scale)
    demand_head   = layers.Dense(32, activation="relu")(shared)
    demand_output = layers.Dense(1, name="log_demand")(demand_head)

    # Head 2: Optimal price recommendation
    price_head    = layers.Dense(32, activation="relu")(shared)
    price_output  = layers.Dense(1, name="price_delta")(price_head)

    # Head 3: Arbitrage risk score
    arb_head      = layers.Dense(32, activation="relu")(shared)
    arb_output    = layers.Dense(1, activation="sigmoid", name="arbitrage_risk")(arb_hea
d)

    model = keras.Model(
        inputs=[sku_input, channel_input, seq_input, num_input],
        outputs=[demand_output, price_output, arb_output],
    )
    model.compile(
        optimizer=keras.optimizers.AdamW(learning_rate=1e-3, weight_decay=1e-4),
        loss={
            "log_demand":     "huber",            # Robust to outlier demand spikes
            "price_delta":    "mse",
            "arbitrage_risk": "binary_crossentropy",
        },
        loss_weights={"log_demand": 1.0, "price_delta": 0.5, "arbitrage_risk": 2.0},
        metrics={"log_demand": ["mae"], "arbitrage_risk": ["auc"]},
    )
    return model
```

2.4 Channel-Level & SKU-Level Elasticity Analysis

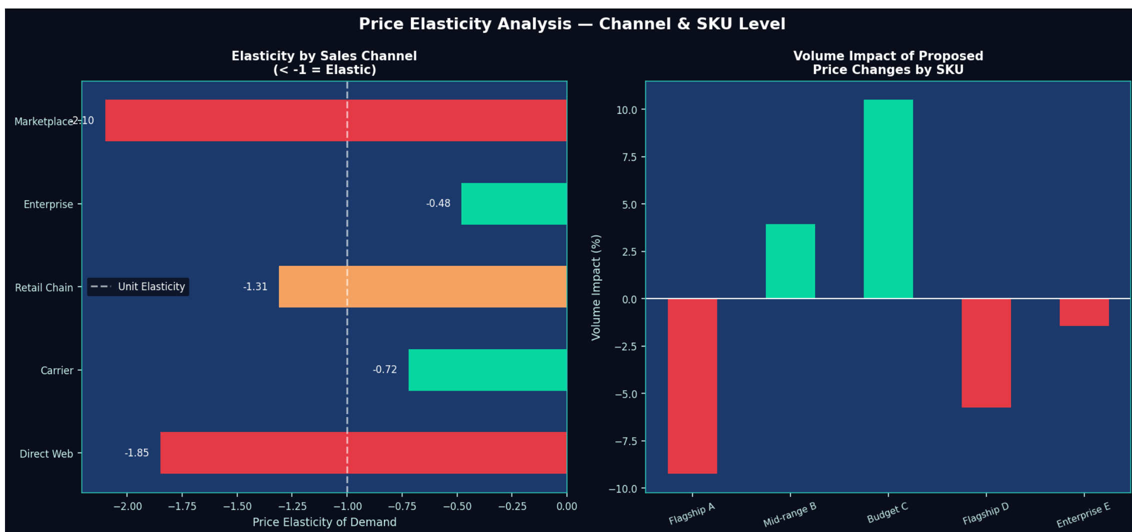

Figure 2.3 — Channel-level elasticity coefficients and SKU volume impact of price changes.

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
# Elasticity by channel and SKU — production analysis pipeline
def compute_channel_elasticity(df, price_col="price", qty_col="quantity",
                                channel_col="channel", sku_col="sku"):
    """
    Returns elasticity dict keyed by (channel, sku).
    Requires at least 30 unique price points per (channel, sku) for reliability.
    """
    results = {}
    for (ch, sku), grp in df.groupby([channel_col, sku_col]):
        if len(grp) < 30: continue
        log_q = np.log(grp[qty_col] + 1)
        log_p = np.log(grp[price_col])
        X = sm.add_constant(log_p)
        try:
            res = sm.OLS(log_q, X).fit()
            results[(ch, sku)] = {
                "elasticity": res.params[price_col],
                "p_value":    res.pvalues[price_col],
                "r_squared":  res.rsquared,
                "n_obs":      len(grp),
            }
        except Exception: pass
    return pd.DataFrame(results).T

# Revenue sensitivity simulation
def simulate_revenue_impact(current_price, elasticity, price_changes=[-10,-5,0,5,10]):
    base_qty = 1000  # normalised
    results  = []
    for pct in price_changes:
        new_price = current_price * (1 + pct/100)
        new_qty   = base_qty * (1 + elasticity * pct/100)
        results.append({
            "price_change_%": pct,
            "new_price":      round(new_price, 2),
            "volume_change_%": round(elasticity*pct, 2),
            "new_qty":        round(new_qty),
            "revenue_change_%": round((new_price*new_qty)/(current_price*base_qty)*100-1
00, 2)
        })
    return pd.DataFrame(results)
```

■ Key Takeaways

- ■ Price Elasticity of Demand (PED) is the % change in volume per 1% change in price — the core metric of pricing science.
- ■ Log-log OLS regression directly yields the elasticity coefficient as the slope — always report confidence intervals.
- ■ Keras multi-head models capture heterogeneous elasticity by channel, SKU, and season simultaneously.

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

- Elastic channels (|PED|>1): price cuts grow revenue; Inelastic (|PED|<1): price raises grow revenue.
- Cross-price elasticity reveals competitive dynamics — positive means substitutes, negative means complements.
- Always simulate multiple price scenarios and present to FP&A; as a revenue impact table, not just a model score.

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

Gray Market Arbitrage Detection & Defense

3.1 How Gray Markets Work in Hardware

Hardware brands often subsidise devices sold through specific channels — carriers offer smartphones at $199 on contract because the carrier is recovering the subsidy over 24 months of ARPU. The "gray market" arises when arbitrageurs purchase these subsidised devices in bulk from the subsidised channel and immediately resell them at market price in another channel, pocketing the subsidy delta. This is legal in many jurisdictions but catastrophic for the brand's subsidy economics — each leaked device represents a full subsidy loss with zero ARPU recovery.

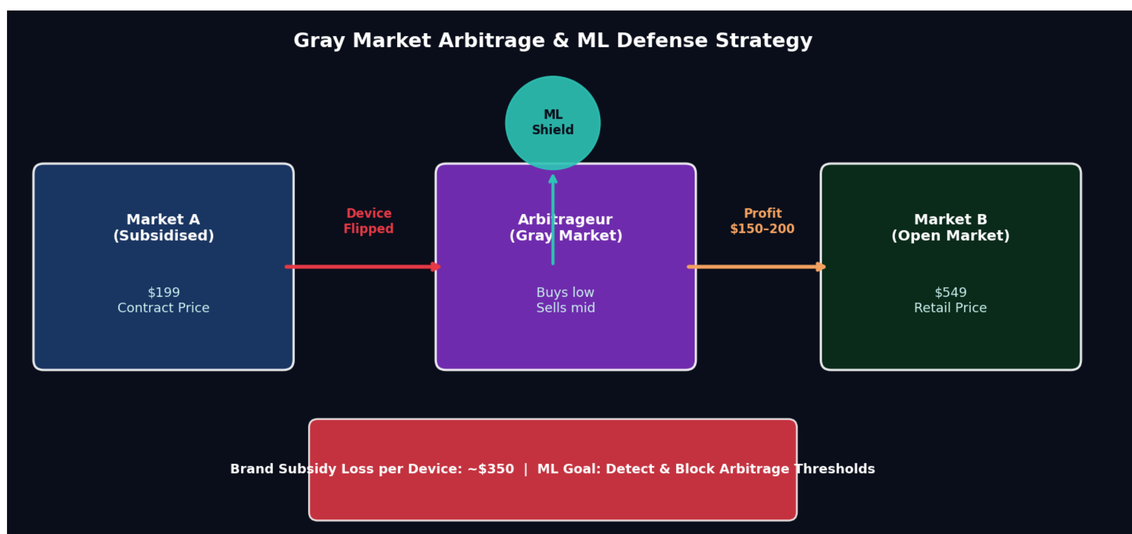

Figure 3.1 — Gray market arbitrage flow and the subsidy loss per device.

3.2 Arbitrage Threshold Mathematics

The fundamental arbitrage condition is: an agent will exploit the price differential when the spread exceeds their transaction costs (shipping, risk, effort). Your model must estimate these thresholds and flag when purchase patterns suggest coordinated bulk buying for resale.

$$\text{Arbitrage Profit} = P_{market} - P_{subsidised} - \text{Transaction_Cost} - \text{Risk_Premium}$$

```
# Arbitrage threshold detection — statistical approach
import pandas as pd, numpy as np
from scipy import stats

def detect_purchase_anomalies(df):
    """
    Input: transaction df with columns:
        customer_id, purchase_date, sku, channel, quantity, store_id
    Returns: flagged transactions with arbitrage_score
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
    """
    features = []

    for cid, grp in df.groupby("customer_id"):
        grp = grp.sort_values("purchase_date")
        # Features that signal bulk arbitrage behaviour
        features.append({
            "customer_id":         cid,
            "total_qty_30d":       grp.query("purchase_date >= @cutoff_30d")["quantity"]
.sum(),
            "unique_skus_30d":     grp.query("purchase_date >= @cutoff_30d")["sku"].nuni
que(),
            "unique_stores_30d":   grp.query("purchase_date >= @cutoff_30d")["store_id"]
.nunique(),
            "avg_daily_qty":       grp["quantity"].sum() / max(1, (grp["purchase_date"].
max()
                                   - grp["purchase_date"].min()).days),
            "cross_channel_flag":  int(grp["channel"].nunique() > 1),
            "promo_concentration": (grp["promo_flag"].sum() / len(grp)) if "promo_flag"
in grp else 0,
            "weekend_ratio":       grp["purchase_date"].dt.dayofweek.isin([5,6]).mean(),
        })

    feat_df = pd.DataFrame(features)
    # Z-score normalise total quantity (most discriminating feature)
    feat_df["qty_zscore"] = stats.zscore(feat_df["total_qty_30d"])
    # Flag Z > 3 as high risk — tune threshold using known fraudulent cases
    feat_df["high_risk"]  = feat_df["qty_zscore"] > 3.0
    return feat_df
```

3.3 XGBoost Arbitrage Classifier

```
import xgboost as xgb
from sklearn.model_selection import StratifiedKFold, cross_val_score
from sklearn.metrics import roc_auc_score, classification_report
from sklearn.preprocessing import LabelEncoder
import shap

# Features for XGBoost arbitrage model
FEATURES = [
    "total_qty_30d",         # Volume in rolling window
    "unique_stores_30d",     # Geographic spread
    "cross_channel_flag",    # Cross-channel activity
    "promo_concentration",   # Buying only during promos
    "avg_daily_qty",         # Velocity
    "qty_zscore",            # Deviation from peer group
    "customer_tenure_days",  # New accounts = higher risk
    "return_rate",           # Arbitrageurs rarely return
    "credit_type",           # Subsidised credit = higher risk
    "sku_subsidy_delta",     # Size of arbitrage opportunity
]
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
X = feat_df[FEATURES].fillna(0)
y = feat_df["is_arbitrageur"]     # Ground truth from fraud team investigations

# Highly imbalanced — use scale_pos_weight
neg, pos = (y==0).sum(), (y==1).sum()
clf = xgb.XGBClassifier(
    n_estimators=500,
    max_depth=6,
    learning_rate=0.05,
    subsample=0.8,
    colsample_bytree=0.8,
    scale_pos_weight=neg/pos,      # Correct class imbalance
    eval_metric="aucpr",           # Area Under Precision-Recall more informative than A
UC for imbalanced
    early_stopping_rounds=20,
    random_state=42,
    tree_method="hist",
    device="cuda",                 # GPU training on AWS g4dn instances
)
clf.fit(X_train, y_train, eval_set=[(X_val, y_val)], verbose=50)

# SHAP explanations — essential for FP&A; and Legal buy-in
explainer   = shap.TreeExplainer(clf)
shap_values = explainer.shap_values(X_test)
shap.summary_plot(shap_values, X_test, feature_names=FEATURES)

# Deployment threshold — balance precision (avoid false positives that hurt genuine cust
omers)
proba    = clf.predict_proba(X)[:,1]
feat_df["arbitrage_score"] = proba
feat_df["flag_review"]     = proba > 0.65   # Review queue
feat_df["flag_block"]      = proba > 0.92   # Auto-block
```

3.4 Keras Autoencoder for Unsupervised Anomaly Detection

XGBoost requires labelled arbitrage cases for training. When labels are scarce, an autoencoder trained only on normal (legitimate) purchase patterns will have high reconstruction error on abnormal (arbitrage) patterns — the error itself becomes the anomaly score.

```
def build_arbitrage_autoencoder(input_dim):
    """Unsupervised anomaly detector — no labels required."""
    # Encoder
    inp   = keras.Input(shape=(input_dim,))
    x     = layers.Dense(64, activation="relu")(inp)
    x     = layers.BatchNormalization()(x)
    x     = layers.Dense(32, activation="relu")(x)
    x     = layers.Dense(16, activation="relu")(x)
    latent = layers.Dense(8, name="latent")(x)    # Bottleneck

    # Decoder
    x     = layers.Dense(16, activation="relu")(latent)
    x     = layers.Dense(32, activation="relu")(x)
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
    x    = layers.Dense(64, activation="relu")(x)
    out  = layers.Dense(input_dim)(x)              # Reconstruct input

    ae   = keras.Model(inp, out)
    ae.compile(optimizer="adam", loss="mse")
    return ae

# Train ONLY on confirmed legitimate transactions
ae = build_arbitrage_autoencoder(input_dim=len(FEATURES))
ae.fit(X_normal, X_normal, epochs=100, batch_size=512,
       validation_split=0.1,
       callbacks=[keras.callbacks.EarlyStopping(patience=10)])

# Reconstruction error = anomaly score
X_all_reconstructed = ae.predict(X_all)
reconstruction_error = np.mean((X_all - X_all_reconstructed)**2, axis=1)

# Set threshold at 99th percentile of NORMAL reconstruction errors
threshold = np.percentile(reconstruction_error[y_normal], 99)
anomalies = reconstruction_error > threshold
print(f"Flagged {anomalies.sum()} potential arbitrage cases ({anomalies.mean()*100:.2f}%
)")
```

■ Key Takeaways

- ■ Gray market arbitrage exploits price differentials between subsidised and open-market channels — each leaked device is pure subsidy loss.
- ■ The arbitrage condition: Profit = Market Price - Subsidised Price - Transaction Costs. Model when this is positive.
- ■ XGBoost with scale_pos_weight handles class imbalance in fraud detection — always prefer AUCPR over AUC for this.
- ■ SHAP values are mandatory for Legal/Compliance sign-off — regulators require explainability before action.
- ■ Keras autoencoders detect arbitrage without labels: high reconstruction error = abnormal purchase pattern.
- ■ Operationalise via tiered response: score>0.65→review queue; score>0.92→automated purchase block.

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

Demand Forecasting for Hardware Assortment

4.1 Why Hardware Demand Forecasting Is Hard

Hardware demand is a notoriously difficult forecasting problem. Unlike FMCG goods, hardware devices have discontinuous lifecycles: a new model launch can drop demand for the prior generation by 60% overnight. Additionally, promotional events (carrier deals, seasonal sales), macroeconomic cycles, competitor launches, and supply-side constraints (chip shortages) all create structural breaks that destroy historical patterns.

Figure 4.1 — Demand forecast comparison: LSTM vs XGBoost vs ARIMA baseline, with decomposition and feature importance.

4.2 ARIMA Baseline

```
from statsmodels.tsa.statespace.sarimax import SARIMAX
from pmdarima import auto_arima
import warnings; warnings.filterwarnings("ignore")

# Auto-select ARIMA order using AIC criterion
auto_model = auto_arima(
    train_series,
    seasonal=True, m=12,          # Monthly seasonality
    stepwise=True,
    suppress_warnings=True,
    information_criterion="aic",
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
    max_p=3, max_q=3, max_P=2, max_Q=2,
)
print(auto_model.summary())

# Seasonal ARIMA — explicit SARIMA(p,d,q)(P,D,Q,12)
sarima = SARIMAX(train_series,
                 order=auto_model.order,
                 seasonal_order=auto_model.seasonal_order,
                 enforce_stationarity=False)
sarima_fit = sarima.fit(disp=False)

# Forecast 6 periods ahead with prediction intervals
forecast = sarima_fit.get_forecast(steps=6)
pred_mean = forecast.predicted_mean
conf_int  = forecast.conf_int(alpha=0.20)   # 80% confidence interval for planning

# Compute MAPE on test set
def mape(actual, predicted):
    return np.mean(np.abs((actual - predicted) / (actual + 1e-9))) * 100

print(f"ARIMA MAPE: {mape(test_series, pred_mean):.2f}%")
```

4.3 XGBoost with Lag Features

```
import xgboost as xgb
import pandas as pd
import numpy as np

def create_lag_features(df, target_col="units", lags=[1,2,3,6,12],
                        windows=[3,6,12]):
    """Create rich feature set from time series for tree-based forecasting."""
    df = df.copy()

    # Lag features (historical values)
    for lag in lags:
        df[f"lag_{lag}"]     = df[target_col].shift(lag)
        df[f"lag_{lag}_price"] = df["price"].shift(lag)

    # Rolling statistics
    for w in windows:
        df[f"roll_mean_{w}"] = df[target_col].shift(1).rolling(w).mean()
        df[f"roll_std_{w}"]  = df[target_col].shift(1).rolling(w).std()
        df[f"roll_max_{w}"]  = df[target_col].shift(1).rolling(w).max()

    # Calendar features
    df["month"]      = df.index.month
    df["quarter"]    = df.index.quarter
    df["is_q4"]      = (df.index.month == 12).astype(int)    # Holiday peak
    df["month_sin"]  = np.sin(2*np.pi*df.index.month/12)     # Cyclical encoding
    df["month_cos"]  = np.cos(2*np.pi*df.index.month/12)

    # External features
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
    df["price_change_pct"]  = df["price"].pct_change()
    df["promo_flag"]         = df["promo_flag"].fillna(0)
    df["device_age_months"]  = (df.index - df["launch_date"]).dt.days // 30

    return df.dropna()

# Train XGBoost
feature_cols = [c for c in df.columns if c not in ["units","launch_date"]]
model_xgb = xgb.XGBRegressor(
    n_estimators=500, max_depth=5, learning_rate=0.05,
    subsample=0.8, colsample_bytree=0.8,
    objective="reg:squarederror",
    early_stopping_rounds=30,
)
model_xgb.fit(X_train, y_train, eval_set=[(X_val,y_val)], verbose=100)
print(f"XGBoost MAPE: {mape(y_test, model_xgb.predict(X_test)):.2f}%")
```

4.4 LSTM Sequence Forecasting with Keras

```
from keras import layers
import keras

def build_demand_lstm(seq_len=24, n_features=15, n_output_steps=6):
    """
    Encoder-Decoder LSTM for multi-step demand forecasting.
    Input:  (batch, seq_len, n_features)  — 24 months of context
    Output: (batch, n_output_steps)       — 6-month forward forecast
    """
    inp = keras.Input(shape=(seq_len, n_features))

    # Encoder
    x = layers.LSTM(128, return_sequences=True, dropout=0.2)(inp)
    x = layers.LSTM(64,  return_sequences=False, dropout=0.2)(x)

    # Repeat vector for decoder
    x = layers.RepeatVector(n_output_steps)(x)

    # Decoder
    x = layers.LSTM(64, return_sequences=True, dropout=0.2)(x)
    x = layers.TimeDistributed(layers.Dense(32, activation="relu"))(x)
    out = layers.TimeDistributed(layers.Dense(1))(x)
    out = layers.Flatten()(out)

    model = keras.Model(inp, out)
    model.compile(
        optimizer=keras.optimizers.Adam(1e-3),
        loss="huber",              # Robust to demand spikes
        metrics=["mae"],
    )
    return model

# Training with proper walk-forward validation
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
def walk_forward_validate(model, X, y, n_splits=5, horizon=6):
    """Time-series cross-validation — never leaks future into past."""
    scores = []
    n = len(X)
    split_size = n // n_splits
    for i in range(1, n_splits+1):
        tr_end = split_size * i
        val_end = min(tr_end + horizon, n)
        model.fit(X[:tr_end], y[:tr_end], epochs=100, batch_size=32, verbose=0,
                  callbacks=[keras.callbacks.EarlyStopping(patience=10)])
        pred = model.predict(X[tr_end:val_end])
        scores.append(mape(y[tr_end:val_end], pred.flatten()))
    return np.mean(scores), np.std(scores)

mean_mape, std_mape = walk_forward_validate(build_demand_lstm(), X_scaled, y)
print(f"LSTM WFV MAPE: {mean_mape:.2f}% ± {std_mape:.2f}%")
```

4.5 Ensemble Stacking & Uncertainty Quantification

```
# Stacking LSTM + XGBoost + SARIMA predictions
from sklearn.linear_model import RidgeCV

# Level 0: Base model predictions (out-of-fold)
oof_lstm  = cross_val_predict(lstm_pipeline, X, y, cv=tscv)
oof_xgb   = cross_val_predict(xgb_pipeline,  X, y, cv=tscv)
oof_arima = arima_oof_preds   # Pre-computed

# Level 1: Meta-learner (Ridge keeps it simple and avoids overfitting)
meta_features = np.column_stack([oof_lstm, oof_xgb, oof_arima])
meta_model    = RidgeCV(alphas=[0.01, 0.1, 1.0, 10.0])
meta_model.fit(meta_features, y_train)
print(f"Model weights: LSTM={meta_model.coef_[0]:.2f}, XGB={meta_model.coef_[1]:.2f}, AR
IMA={meta_model.coef_[2]:.2f}")

# Prediction intervals via quantile regression (for OEM negotiation risk bands)
from sklearn.ensemble import GradientBoostingRegressor

q10_model = GradientBoostingRegressor(loss="quantile", alpha=0.10).fit(X_train, y_train)
q90_model = GradientBoostingRegressor(loss="quantile", alpha=0.90).fit(X_train, y_train)

forecast_df = pd.DataFrame({
    "point_forecast": meta_model.predict(test_meta_features),
    "lower_80":       q10_model.predict(X_test),
    "upper_80":       q90_model.predict(X_test),
})
# This "80% confidence band" maps directly to OEM Committed / Base / Upside volumes
```

■ Key Takeaways

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

- ■ Hardware forecasting requires handling structural breaks (launches, supply shocks) — include device_age and launch_flag as features.
- ■ Always compare models to SARIMA baseline — if your fancy model doesn't beat it, question its complexity.
- ■ XGBoost with lag features often outperforms LSTM on short-to-medium horizons; LSTM wins for 6-12 month multi-step.
- ■ Walk-forward cross-validation is mandatory — never shuffle time series data for train/test splits.
- ■ Ensemble stacking (LSTM + XGBoost + ARIMA) reduces forecast variance and consistently outperforms any single model.
- ■ Prediction intervals from quantile regression directly map to OEM negotiation scenarios: committed/base/upside.

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

OEM Negotiation Analytics & Assortment Planning

OEM negotiations (with Apple, Samsung, Qualcomm, etc.) are among the highest-stakes commercial interactions your company has. Volume commitments locked in months early determine device cost, allocation priority, and exclusivity rights. Your forecast is the data backbone of every negotiation.

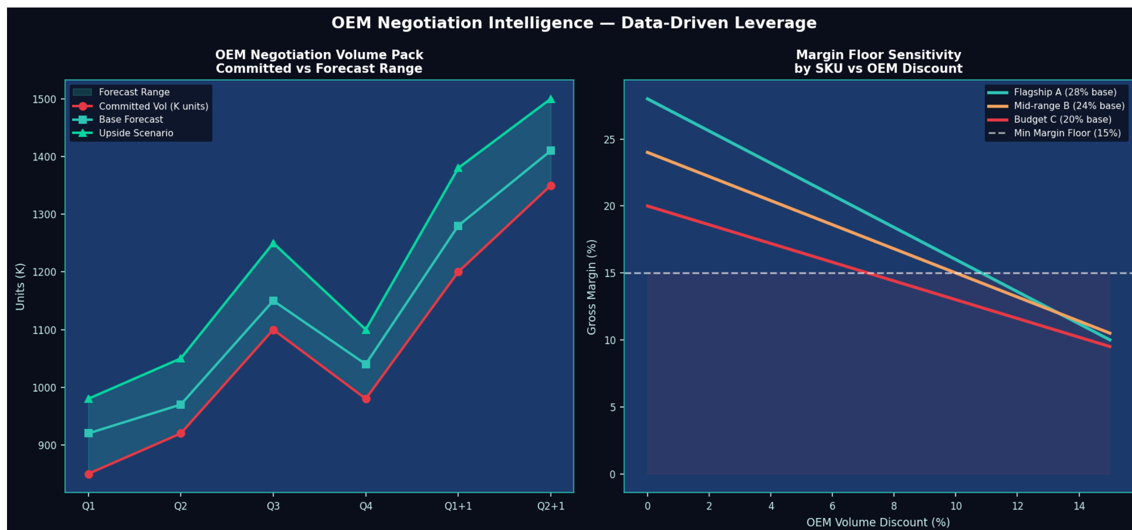

Figure 5.1 — OEM volume commitment pack: committed vs forecast range, and margin floor sensitivity.

5.1 Building the OEM Volume Commitment Model

```
# Scenario engine for OEM negotiation
def build_oem_data_pack(sku, forecast_df, margin_structure, oem_discount_grid):
    """
    Generates a structured data pack for OEM negotiation:
    - Volume at three scenarios (conservative/base/upside)
    - Margin floor analysis at each discount level
    - Recommendation on max allowable discount
    """
    committed_vol = forecast_df["lower_80"].sum()   # Conservative (sign this)
    base_vol      = forecast_df["point_forecast"].sum()
    upside_vol    = forecast_df["upper_80"].sum()

    # Margin analysis
    base_margin   = margin_structure[sku]["gross_margin_pct"]
    min_margin    = margin_structure[sku]["floor_margin_pct"]
    rows = []
    for discount in oem_discount_grid:
        adjusted_cogs   = margin_structure[sku]["cogs"] * (1 - discount/100)
        adjusted_margin = (margin_structure[sku]["asp"] - adjusted_cogs)
           / margin_structure[sku]["asp"] * 100
        rows.append({
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
            "oem_discount_%":      discount,
            "adjusted_margin_%":  round(adjusted_margin, 2),
            "below_floor":        adjusted_margin < min_margin,
            "max_volume_commit_k": round(committed_vol/1000, 1),
        })
    margin_df = pd.DataFrame(rows)

    # Find maximum viable discount
    max_discount = margin_df.query("not below_floor")["oem_discount_%"].max()

    return {
        "sku":              sku,
        "committed_vol_k": round(committed_vol/1000,1),
        "base_vol_k":       round(base_vol/1000,1),
        "upside_vol_k":     round(upside_vol/1000,1),
        "max_oem_discount": max_discount,
        "margin_table":     margin_df,
        "recommendation":   f"Commit {committed_vol/1000:.0f}K units; negotiate up to {ma
x_discount}% OEM discount"
    }
```

■ Key Takeaways

- ■ Always commit only the Conservative (10th percentile) forecast to OEM — over-committing creates expensive surplus.
- ■ Margin floor analysis gives you the maximum OEM discount you can offer before destroying profitability.
- ■ Present forecasts with explicit uncertainty bands — this builds credibility and protects against blame when actuals differ.
- ■ The "upside scenario" is a powerful negotiation tool: "We'll commit X, but if you give us allocation priority, we can do Y."
- ■ Document all forecast assumptions in the data pack — regulatory and audit requirements in listed companies.

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

A/B Testing & Multi-Armed Bandits for Pricing

6.1 Why Classical A/B Testing Falls Short for Pricing

Classical A/B testing splits traffic 50/50 and runs for a fixed duration before declaring a winner. In pricing experiments, this means spending 50% of your experiment budget on the inferior price point for potentially weeks. If you're testing $399 vs $449 and $449 is clearly converting better after day 3, classical A/B keeps sending 50% of customers to $399, leaving revenue on the table. Multi-Armed Bandit (MAB) algorithms solve this by dynamically shifting traffic toward better-performing arms as evidence accumulates.

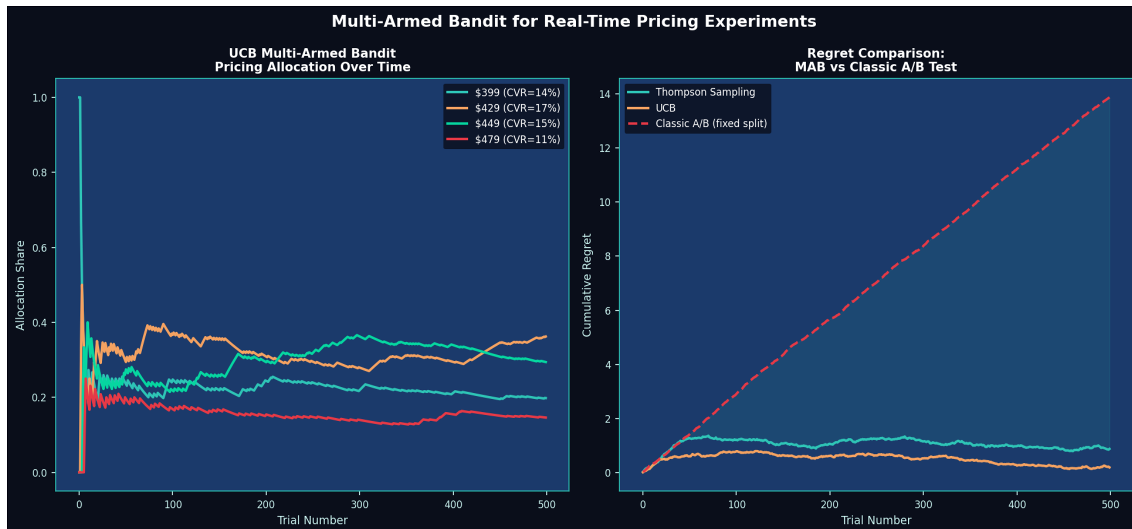

Figure 6.1 — UCB allocation converging to optimal price arm; regret comparison MAB vs classic A/B.

6.2 Thompson Sampling for Price Optimisation

```
import numpy as np
from scipy.stats import beta

class ThompsonSamplingPricer:
    """
    Bayesian Multi-Armed Bandit for pricing experiments.
    Models conversion rate per price arm as Beta(alpha, beta) distribution.
    Each 'pull' = showing a price and observing purchase (1) or no purchase (0).
    """

    def __init__(self, price_arms: list):
        self.arms      = price_arms
        self.n_arms    = len(price_arms)
        # Beta(1,1) = Uniform prior — no prior knowledge
        self.alpha     = np.ones(self.n_arms)    # Successes + 1
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
        self.beta_    = np.ones(self.n_arms)   # Failures + 1
        self.history  = {p: {"shown": 0, "converted": 0} for p in price_arms}

    def select_arm(self) -> float:
        """Sample from posterior — exploration follows uncertainty naturally."""
        samples = [np.random.beta(self.alpha[i], self.beta_[i])
                   for i in range(self.n_arms)]
        return self.arms[int(np.argmax(samples))]

    def update(self, price: float, converted: bool):
        """Update posterior after observing outcome."""
        arm_idx = self.arms.index(price)
        if converted:
            self.alpha[arm_idx] += 1
        else:
            self.beta_[arm_idx]  += 1
        self.history[price]["shown"] += 1
        self.history[price]["converted"] += int(converted)

    def best_price(self) -> float:
        """Current best estimate of optimal price."""
        mean_cvr = self.alpha / (self.alpha + self.beta_)
        # Adjust by revenue (CVR * Price) not just CVR
        expected_revenue = mean_cvr * np.array(self.arms)
        return self.arms[int(np.argmax(expected_revenue))]

    def confidence_report(self) -> dict:
        """Report posterior distributions for each arm."""
        report = {}
        for i, price in enumerate(self.arms):
            a, b = self.alpha[i], self.beta_[i]
            mean = a / (a+b)
            ci_low, ci_high = beta.ppf([0.05, 0.95], a, b)
            report[price] = {"cvr_mean": mean, "ci_low": ci_low, "ci_high": ci_high,
                             "n_shown": int(a+b-2)}
        return report

# Production integration
pricer = ThompsonSamplingPricer(price_arms=[399, 429, 449, 479])

def get_price_for_user(user_id: str) -> float:
    """Called at page render time — returns selected price."""
    return pricer.select_arm()

def record_conversion(price: float, purchased: bool):
    """Called after purchase event from transaction stream."""
    pricer.update(price, purchased)

# After 1000 trials, check winner
print(f"Recommended optimal price: ${pricer.best_price()}")
print(pricer.confidence_report())
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

6.3 Contextual Bandits — Personalised Pricing

Thompson Sampling treats all users identically. Contextual bandits extend this by conditioning the price selection on user context: device history, channel, geography, B2B vs consumer. A Keras model serves as the reward predictor within the bandit loop.

```
# Contextual Bandit with Keras reward model
def build_contextual_reward_model(context_dim, n_price_arms):
    """
    Predicts expected conversion probability for each price arm given user context.
    Output shape: (batch, n_price_arms)
    """
    ctx = keras.Input(shape=(context_dim,), name="user_context")
    x   = layers.Dense(64, activation="relu")(ctx)
    x   = layers.BatchNormalization()(x)
    x   = layers.Dense(32, activation="relu")(x)
    x   = layers.Dropout(0.2)(x)
    # One output per price arm
    out = layers.Dense(n_price_arms, activation="sigmoid")(x)
    model = keras.Model(ctx, out)
    model.compile("adam", "binary_crossentropy")
    return model

# At inference: select arm with highest expected reward * price
def select_contextual_price(model, context, price_arms):
    cvr_preds    = model.predict(context[np.newaxis], verbose=0).squeeze()
    revenues     = cvr_preds * np.array(price_arms)
    return price_arms[np.argmax(revenues)], cvr_preds
```

■ Key Takeaways

- ■ MABs reduce "regret" vs A/B tests by dynamically allocating more traffic to better-performing price arms.
- ■ Thompson Sampling is Bayesian: uncertainty naturally drives exploration; high-confidence arms get exploited.
- ■ Always optimise for revenue (CVR × Price), not just CVR — the highest converting price might not maximise revenue.
- ■ Contextual bandits personalise pricing by user segment — Keras predicts arm rewards given user context.
- ■ Run statistical significance checks before declaring winners — MABs converge faster but can stop too early on noisy signals.
- ■ Document every pricing experiment with a clear hypothesis, sample size calculation, and business metric definition.

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

Full-Stack Model Deployment on AWS

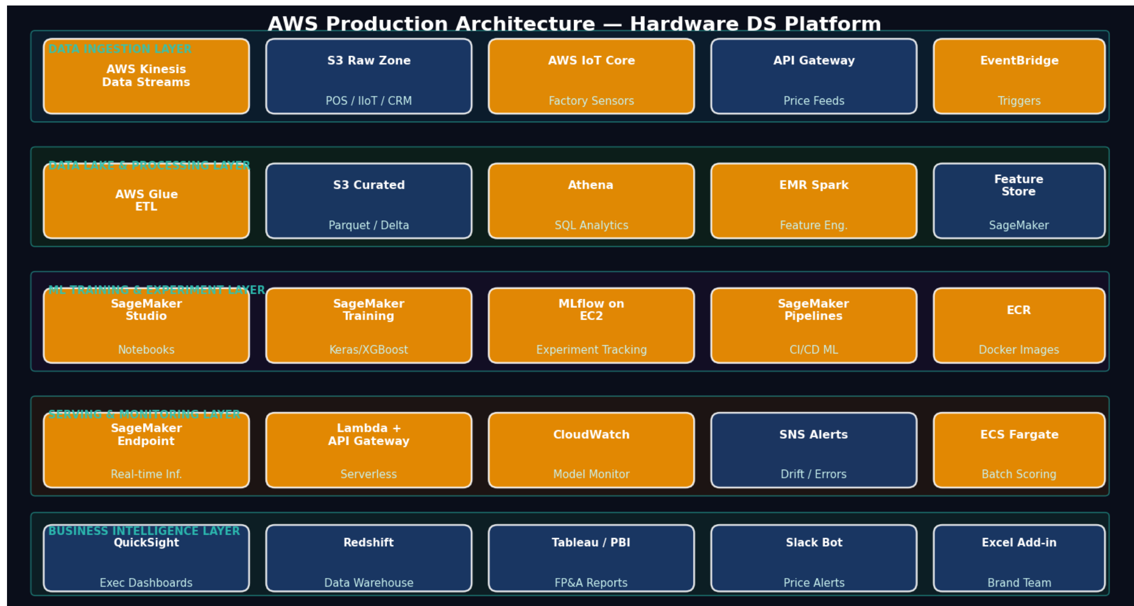

Figure 7.1 — Full AWS production architecture for hardware DS platform.

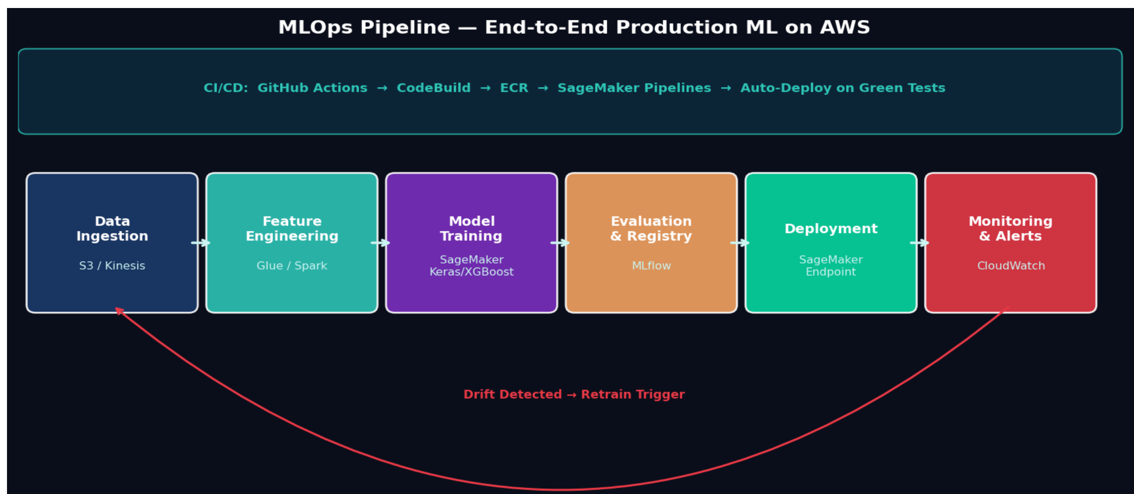

Figure 7.2 — MLOps CI/CD pipeline with automated drift detection and retraining.

7.1 Dockerising a Keras Model for Deployment

```
# Dockerfile for Keras inference server
# File: Dockerfile
FROM public.ecr.aws/lambda/python:3.11

# Install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy model artefacts and inference code
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
COPY model/ /opt/ml/model/
COPY inference.py /var/task/

CMD ["inference.lambda_handler"]

# inference.py — SageMaker-compatible inference script
import json, numpy as np, keras, os

MODEL = None

def model_fn(model_dir):
    """Load model from SageMaker model directory."""
    global MODEL
    MODEL = keras.models.load_model(os.path.join(model_dir, "pricing_model.keras"))
    return MODEL

def predict_fn(input_data, model):
    """Run inference."""
    predictions = model.predict(input_data)
    return {
        "log_demand":      float(predictions[0][0]),
        "price_delta":     float(predictions[1][0]),
        "arbitrage_risk":  float(predictions[2][0]),
        "demand_units":    int(np.exp(predictions[0][0])),
    }

def input_fn(request_body, content_type="application/json"):
    """Deserialise request."""
    data = json.loads(request_body)
    return {k: np.array(v) for k, v in data.items()}

def output_fn(prediction, accept="application/json"):
    return json.dumps(prediction), accept
```

7.2 SageMaker Training Job

```
import sagemaker
from sagemaker.tensorflow import TensorFlow    # Keras runs on TF backend
from sagemaker import get_execution_role

role    = get_execution_role()
session = sagemaker.Session()

estimator = TensorFlow(
    entry_point="train.py",
    source_dir="./src",
    role=role,
    instance_count=1,
    instance_type="ml.g4dn.xlarge",  # NVIDIA T4 GPU — cost-effective
    framework_version="2.14",
    py_version="py311",
    hyperparameters={
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
        "epochs":       100,
        "batch_size":   256,
        "learning_rate": 1e-3,
        "dropout":      0.3,
    },
    metric_definitions=[
        {"Name": "val_mae",  "Regex": "val_mae: ([0-9.]+)"},
        {"Name": "val_loss", "Regex": "val_loss: ([0-9.]+)"},
    ],
    checkpoint_s3_uri=f"s3://{bucket}/checkpoints/pricing-model/",
    use_spot_instances=True,           # Save up to 70% cost with Spot
    max_wait=7200,
)

estimator.fit({
    "train": f"s3://{bucket}/data/train/",
    "val":   f"s3://{bucket}/data/val/",
})
```

7.3 Real-Time Endpoint Deployment

```
# Deploy to SageMaker real-time endpoint
predictor = estimator.deploy(
    initial_instance_count=2,          # Min 2 for HA
    instance_type="ml.c5.xlarge",      # CPU inference — cheaper than GPU for small mode
ls
    endpoint_name="hardware-pricing-v2",
    data_capture_config=sagemaker.model_monitor.DataCaptureConfig(
        enable_capture=True,
        sampling_percentage=20,        # Capture 20% of requests for monitoring
        destination_s3_uri=f"s3://{bucket}/model-monitor/",
    ),
)

# Call the endpoint from application code
import boto3, json

sm_runtime = boto3.client("sagemaker-runtime", region_name="us-east-1")

def score_transaction(sku_id, channel_id, price_history, numeric_features):
    payload = {
        "sku_id":           [[sku_id]],
        "channel_id":       [[channel_id]],
        "price_history":    [price_history],
        "numeric_features": [numeric_features],
    }
    response = sm_runtime.invoke_endpoint(
        EndpointName="hardware-pricing-v2",
        ContentType="application/json",
        Body=json.dumps(payload),
    )
    return json.loads(response["Body"].read())
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
# Model Monitor — detect data drift
from sagemaker.model_monitor import DefaultModelMonitor, MonitoringOutput

monitor = DefaultModelMonitor(role=role, instance_count=1,
                              instance_type="ml.m5.xlarge",
                              volume_size_in_gb=20, max_runtime_in_seconds=3600)
monitor.suggest_baseline(baseline_dataset=f"s3://{bucket}/baseline/",
                         output_s3_uri=f"s3://{bucket}/baseline-results/")
```

■ Key Takeaways

- ■ Dockerise your model with a SageMaker-compatible interface (model_fn, predict_fn, input_fn, output_fn).
- ■ Use Spot Instances for training (up to 70% cost saving) — implement checkpointing to handle interruptions.
- ■ Deploy with DataCaptureConfig to sample production inputs — this feeds Model Monitor for drift detection.
- ■ Real-time SageMaker endpoints: minimum 2 instances for high availability in production.
- ■ MLflow tracks experiments: every training run logs hyperparameters, metrics, and model artefacts.
- ■ SageMaker Pipelines automates the full train→evaluate→register→deploy lifecycle — no manual steps in production.

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

IIoT & Manufacturing Data Science

8.1 IIoT Data Architecture on AWS

Industrial IoT generates high-frequency, high-volume sensor streams from factory machines: vibration sensors (accelerometers), temperature probes, current monitors, optical quality cameras, and conveyor throughput counters. This data is structurally different from business transactional data — it arrives at millisecond intervals, requires real-time alerting, and has clear physical interpretations that must be preserved in model features.

Figure 8.1 — IIoT anomaly detection, defect heatmaps, predictive maintenance (RUL), and supply throughput optimisation.

```
# AWS IoT Core → Kinesis → S3 → Real-time ML pipeline

# AWS IoT Rule (JSON definition — deployed via CloudFormation)
iot_rule = {
    "sql": "SELECT * FROM 'factory/+/sensors'",
    "actions": [{
        "kinesis": {
            "roleArn": "arn:aws:iam::ACCOUNT:role/IoTKinesisRole",
            "streamName": "factory-sensor-stream",
            "partitionKey": "${machineId}",
        }
    }]
}

# Lambda consumer — real-time anomaly scoring
import boto3, json, numpy as np
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
sm_runtime  = boto3.client("sagemaker-runtime")
sns_client  = boto3.client("sns")
TOPIC_ARN   = "arn:aws:sns:us-east-1:ACCOUNT:factory-alerts"

def lambda_handler(event, context):
    for record in event["Records"]:
        payload   = json.loads(record["kinesis"]["data"])
        machine   = payload["machineId"]
        features  = extract_features(payload)           # Windowed FFT, rolling stats

        # Score against deployed autoencoder
        response  = sm_runtime.invoke_endpoint(
            EndpointName="iiot-anomaly-detector",
            ContentType="application/json",
            Body=json.dumps({"features": [features]}),
        )
        result    = json.loads(response["Body"].read())
        anomaly_score = result["reconstruction_error"]

        if anomaly_score > THRESHOLD:
            sns_client.publish(
                TopicArn=TOPIC_ARN,
                Message=json.dumps({
                    "machine_id":    machine,
                    "anomaly_score": round(anomaly_score, 4),
                    "timestamp":     payload["timestamp"],
                    "alert_type":    classify_anomaly(anomaly_score),
                }),
                Subject=f"ALERT: Anomaly detected on {machine}",
            )
```

8.2 Predictive Maintenance LSTM — Remaining Useful Life

```
def build_rul_predictor(seq_len=50, n_sensors=21):
    """
    Predicts Remaining Useful Life (RUL) in cycles from sensor sequences.
    Based on CMAPSS turbofan dataset architecture.
    """
    inp  = keras.Input(shape=(seq_len, n_sensors))

    # Multi-scale temporal features
    x1   = layers.LSTM(100, return_sequences=True)(inp)
    x1   = layers.Dropout(0.2)(x1)
    x1   = layers.LSTM(50)(x1)

    # 1D CNN for local pattern detection
    x2   = layers.Conv1D(64, 3, activation="relu", padding="same")(inp)
    x2   = layers.Conv1D(32, 3, activation="relu", padding="same")(x2)
    x2   = layers.GlobalAveragePooling1D()(x2)

    merged = layers.Concatenate()([x1, x2])
    x      = layers.Dense(64, activation="relu")(merged)
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
    x      = layers.Dropout(0.3)(x)

    # Output: predicted RUL in cycles
    rul_output = layers.Dense(1, name="rul")(x)

    # Uncertainty head (Monte Carlo Dropout estimate)
    unc_output = layers.Dense(1, activation="softplus", name="uncertainty")(x)

    model = keras.Model(inp, [rul_output, unc_output])
    model.compile(
        optimizer="adam",
        loss={"rul": "mse", "uncertainty": "mse"},
        loss_weights={"rul": 1.0, "uncertainty": 0.1},
    )
    return model

# Maintenance scheduling from RUL predictions
def schedule_maintenance(machine_id, rul_days, uncertainty_days,
                          maintenance_cost=5000, downtime_cost_per_day=15000):
    """
    Optimal maintenance window balancing cost of early action vs unplanned failure.
    """
    # Schedule maintenance when lower confidence bound drops below buffer
    lower_rul = rul_days - 1.645 * uncertainty_days   # 95% confidence lower bound
    optimal_maintenance_day = max(lower_rul - 3, 1)   # 3-day buffer
    expected_savings = downtime_cost_per_day * 2 - maintenance_cost  # vs. reactive
    return {
        "machine_id":              machine_id,
        "predicted_rul_days":      round(rul_days, 1),
        "confidence_lower_days":   round(lower_rul, 1),
        "schedule_maintenance_in": round(optimal_maintenance_day, 0),
        "expected_savings_$":      expected_savings,
    }
```

■ Key Takeaways

- ■ IIoT data arrives at millisecond granularity — always aggregate to appropriate windows (1s, 1min) before modelling.
- ■ Autoencoder reconstruction error is your unsupervised anomaly score — no labels needed for normal/abnormal separation.
- ■ LSTM RUL predictors outperform classical ARIMA for degradation trajectories with multiple interacting sensors.
- ■ Monte Carlo Dropout provides uncertainty estimates — critical for safety-critical maintenance scheduling.
- ■ AWS IoT Core → Kinesis → Lambda → SageMaker Endpoint is the real-time IIoT inference pattern on AWS.

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

■ Connect IIoT insights to demand forecasts: a production outage predicted 2 weeks out changes your OEM volume commitment.

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

Discriminative & Predictive Models — The Full Toolkit

9.1 Discriminative vs Generative Models

Discriminative models learn the boundary between classes directly — P(y|x). Examples: Logistic Regression, XGBoost, SVM, and the Keras classification networks we've built. They are the workhorses of business ML because they are efficient, interpretable, and well-understood by regulators. Generative models learn the full joint distribution P(x,y) — useful for data augmentation and anomaly detection (GANs, VAEs) but less common in production pricing.

Model Type	Examples	Business Use Case	Keras / Library
Binary Classifier	Logistic Reg, XGBoost, Keras ANN	Arbitrage flag, Churn predict	keras.Sequential + sigmoid
Multi-class	Softmax DNN, Random Forest	SKU lifecycle stage	keras + softmax
Regression	Ridge, XGBoost, LSTM	Demand forecast, Price optimise	keras + linear output
Ranking	LambdaMART, Listwise NN	Assortment priority ranking	TF Ranking
Anomaly Detector	Autoencoder, Isolation Forest	Gray market, IIoT fault	keras Autoencoder
Time-to-Event	Cox PH, LSTM RUL	Machine failure, Device EOL	lifelines + keras
Causal Inference	DoWhy, CausalML	True price effect (removing confounders)	Microsoft DoWhy

9.2 SHAP for Model Explainability

```
import shap

# SHAP for Keras models — DeepExplainer or KernelExplainer
background = X_train[np.random.choice(len(X_train), 200, replace=False)]

# DeepExplainer (faster for neural networks)
explainer   = shap.DeepExplainer(keras_model, background)
shap_values = explainer.shap_values(X_test[:100])

# Force plot for a single prediction — show stakeholders
shap.initjs()
shap.force_plot(explainer.expected_value[0],
                shap_values[0][0], X_test.iloc[0],
                feature_names=FEATURE_NAMES)

# Summary plot — shows global feature importance
shap.summary_plot(shap_values[0], X_test[:100], feature_names=FEATURE_NAMES)

# Dependence plot — how one feature drives predictions
```


© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

```
shap.dependence_plot("total_qty_30d", shap_values[0],
                     X_test, feature_names=FEATURE_NAMES)

# For XGBoost — TreeExplainer (exact, very fast)
xgb_explainer = shap.TreeExplainer(xgb_model)
xgb_shap      = xgb_explainer.shap_values(X_test)

# Business output: top 3 factors that drove this arbitrage flag
def explain_flag(idx, shap_vals, feature_names, top_n=3):
    top_features = sorted(zip(feature_names, shap_vals[idx]),
                          key=lambda x: abs(x[1]), reverse=True)[:top_n]
    return [(name, round(val, 4)) for name, val in top_features]

explanation = explain_flag(0, xgb_shap, FEATURE_NAMES)
print(f"Top reasons for arbitrage flag: {explanation}")
# e.g. [("total_qty_30d", +2.31), ("unique_stores_30d", +1.87), ("promo_concentration",
+0.94)]
```

■ Key Takeaways

- ■ Discriminative models predict P(y|x) — direct class boundaries. Use these for every business classification task.
- ■ Causal inference matters when estimating true price effect: observational data conflates price changes with promotion events.
- ■ SHAP values are the industry standard for model explainability — mandatory for regulated financial decisions.
- ■ DeepExplainer works for Keras models; TreeExplainer for XGBoost — always pair your model with its explainer.
- ■ Model governance requires: documented assumptions, version control, approval workflow, and monitoring SLAs.
- ■ A model card documents: intended use, performance by subgroup, known limitations, and contact for issues.

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

Technical Leadership & Stakeholder Communication

10.1 The Pyramid Principle for Data Scientists

Most data scientists present findings chronologically: "First I tried ARIMA, then I tried XGBoost, then I noticed..." This is the opposite of what executives need. The Pyramid Principle structures communication as: Conclusion first → supporting arguments → underlying data. Your audience should know your recommendation within 30 seconds.

Executive Communication Template

RECOMMENDATION: Raise the flagship SKU price by 4% in the Carrier channel for Q4. EXPECTED IMPACT: +$2.1M gross margin (97.3% confidence interval: $1.6M–$2.6M). BASIS: Price elasticity is -0.72 in this channel (inelastic) — 4% price rise causes only -2.9% volume drop, net positive. Validated with 8 months of POS data and walk-forward CV (MAPE: 4.2%). RISK: Gray market risk score for this SKU/channel is 0.23 (low). Arbitrage threshold is $380 spread; proposed price keeps us below it. NEXT STEP: Approve by Oct 15 to enable carrier system update for Nov 1 effective date.

10.2 Sprint Planning for Data Science Teams

Sprint Activity	DS Time %	Output	Stakeholder
Data pipeline maintenance	15%	Fresh feature store	Engineering
Model retraining (weekly)	10%	Updated model registry	MLOps
Forecast pack preparation	20%	OEM data pack	Brand Team
Pricing experiment analysis	15%	MAB report + recommendation	FP&A
Stakeholder requests (ad hoc)	20%	Analysis decks	All
New model research	20%	Proof of concept	DS Lead

10.3 Model Risk Management Framework

Development Gate: Document data sources, features, training methodology, and baseline comparison before any model enters staging.

Validation Gate: Independent validation by a second DS with held-out test set. Disaggregated performance by channel, SKU, region.

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

Business Approval Gate: FP&A; and Legal sign-off on model assumptions, business logic, and explainability report (SHAP).

Production Gate: Load testing, latency SLA confirmation (<100ms p99), monitoring dashboard live before deploy.

Ongoing Monitoring: Weekly drift reports, monthly full revalidation, quarterly model champion/challenger review.

■ Key Takeaways

- ■ Lead with the recommendation and business impact — never with methodology. Executives have 90 seconds for your slide.
- ■ Quantify uncertainty in every deliverable: "Our forecast is 12,000 units with an 80% CI of 10,500–13,800."
- ■ Sprint capacity: protect 20% for research or you will only ever be reactive to business requests.
- ■ Model Risk Management is not bureaucracy — it is what separates a trusted model from a liability.
- ■ Build relationships with FP&A; and Brand before a crisis; they need to trust your models when the numbers disappoint.
- ■ Write model cards for every production model: document intended use, limitations, performance metrics, and contact.

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.

Summary — The Complete Toolkit

This book has equipped you with the full stack needed to operate as a Hybrid Data Scientist and Strategic Business Analyst in a hardware-focused organisation. You can now quantify price elasticity, detect and block gray market arbitrage, forecast device demand across a 6–12 month horizon, support OEM negotiations with data-backed volume scenarios, design and run real-time pricing experiments using Multi-Armed Bandits, deploy production-grade Keras and XGBoost models on AWS SageMaker, integrate IIoT sensor data for predictive maintenance and quality control, and communicate all of this to every stakeholder in language they understand.

Recommended Learning Path

- Economics of Pricing: "The Art of Pricing" (Rafi Mohammed) — business intuition for the math.
- Forecasting: "Forecasting: Principles & Practice" (Hyndman & Athanasopoulos) — free at otexts.com.
- Causal Inference: "Causal Inference for the Brave and True" (Matheus Facure) — free online.
- AWS: AWS Certified Machine Learning Specialty — official certification for production credibility.
- MLOps: "Designing Machine Learning Systems" (Chip Huyen) — gold standard for production DS.
- Experimentation: "Trustworthy Online Controlled Experiments" (Kohavi et al.) — A/B test bible.

© 2025 Wreetojyoti Ray. All Rights Reserved.

© 2025 Wreetojyoti Ray. All Rights Reserved. Data Science for Hardware Business — Keras, ML & AWS. Unauthorized reproduction prohibited.