AWS Spot Instance Price Forecasting with PatchTST

Technical Documentation & Development Journey

Document Created: November 25, 2025

Owner : Nisha Chothe

Project: Spot ML - Time Series Transformer for Cloud Resource Optimization

Table of Contents

1. Executive Summary

This document details the development of an AWS Spot Instance price forecasting system using PatchTST (Patched Time Series Transformer). The project aims to predict spot instance prices 24 hours ahead and provide risk classifications (STABLE, CAUTION, MIGRATE) to optimize cloud resource costs and prevent workload interruptions.

The current implementation uses PatchTST but encounters gradient instability issues due to the non-stationary, heavy-tailed nature of AWS spot pricing data. After extensive debugging and attempted fixes (log transforms, gradient clipping, feature reduction), the fundamental architectural limitations of PatchTST for financial time series have been identified. Future work will transition to Temporal Fusion Transformer (TFT), which is specifically designed for real-world forecasting tasks.

Dataset Size	**3.3M training records (2023-2024), Expanding to 5M+**
Instance Pools	32 pools (4 instance types × 8 availability zones)
Forecast Horizon	24 hours ahead
Lookback Window	168 hours (7 days)
Current Model	PatchTST (experiencing gradient instability)
Features	9 engineered features
Training Period	2023 January - 2024 December (24 months)
Target Metric	MAPE < 3%, Risk Accuracy > 75%
Status	PatchTST implemented; TFT planned as future work

2. Project Overview

2.1 Problem Statement

AWS Spot Instances represent a fundamental trade-off in cloud computing: they offer substantial cost savings (up to 90% compared to On-Demand instances) but come with inherent uncertainty and potential interruptions. When AWS needs capacity back, spot instances can be terminated with only 2 minutes of warning, causing potential service disruptions and data loss.

The core challenge lies in the unpredictable nature of spot pricing. Prices fluctuate based on real-time supply and demand dynamics across availability zones, instance types, and regions. These fluctuations exhibit complex patterns:

- Time-of-day effects: Higher demand during business hours (9 AM - 5 PM)
- Day-of-week patterns: Lower usage on weekends, spikes on Mondays
- Seasonal variations: End-of-quarter compute-intensive workloads
- Event-driven spikes: AWS outages, major product launches, Black Friday
- Regional imbalances: Capacity constraints in specific availability zones
- Instance type competition: Popular instances (t3.medium) have higher volatility

Organizations running workloads on spot instances face several critical challenges:

- Cost unpredictability: Sudden price spikes can eliminate cost savings
- Workload interruptions: Unexpected terminations disrupt batch jobs, ML training, data processing
- Manual monitoring burden: DevOps teams must constantly watch pricing dashboards
- Opportunity cost: Conservative strategies leave savings on the table
- Resource allocation complexity: Deciding which workloads can tolerate interruptions

Current approaches to spot instance management are largely reactive. Teams either accept interruptions as inevitable or overpay by using On-Demand instances. What's needed is a proactive, predictive system that can forecast price movements and risk levels hours in advance, enabling intelligent workload placement and migration decisions.

2.2 Objectives

Primary Objective: Accurate Price Forecasting

Build a time series forecasting model that predicts spot instance prices 24 hours into the future with Mean Absolute Percentage Error (MAPE) below 3%. This accuracy threshold is critical because:

- Sub-3% error enables reliable cost projections for budget planning
- Allows confident decision-making for workload scheduling
- Minimizes false alarms that would cause unnecessary migrations
- Provides actionable intelligence for bid optimization strategies

Secondary Objective: Risk Classification

Develop a multi-class risk prediction system that categorizes each pool into three risk levels with >75% accuracy:

- STABLE (Target: 70% of predictions): Pools where prices are predictable and unlikely to spike. Safe for long-running batch jobs, ML training, and stateful workloads.
- CAUTION (Target: 20% of predictions): Pools with moderate volatility. Suitable for checkpointed workloads that can tolerate occasional interruptions. Requires monitoring but not immediate action.
- MIGRATE (Target: 10% of predictions): High-risk pools with imminent price spikes or high volatility. Recommendation is to migrate workloads to stable pools or temporarily use On-Demand instances.

Tertiary Objective: Production-Ready System

Engineer a scalable, maintainable pipeline capable of:

- Processing 20M+ records efficiently (memory-optimized, chunked loading)
- Training on diverse datasets (2023-2024 historical data, expanding to larger corpus)
- Handling 32+ instance pools with per-pool modeling
- Real-time inference (<50ms per pool prediction)
- Automated retraining as new data arrives
- Comprehensive logging and experiment tracking
- Model versioning and checkpoint management

3. Data Architecture

3.1 Dataset Description

Training Dataset: aws_2023_2024_complete_24months.csv

- Records: 3,305,408
- Time Range: January 2023 - December 2024 (24 months)
- Temporal Coverage: Complete 24-month period for robust seasonal pattern learning
- Pools: 32 (t3.medium, t4g.small, t4g.medium, c5.large across 8 AZs)
- Regions: us-east-1 (4 AZs), ap-south-1 (4 AZs)
- Sampling Frequency: Every 10 minutes
- Data Points per Day: 144 observations per pool (24 hours × 6 samples/hour)
- Total Pool-Hours: 275,040 hours of pricing history

Test Dataset: mumbai_spot_data_sorted_asc(1-2-3-25).csv

- Records: 20,683,580
- Time Range: January 2025 - March 2025 (3 months)
- Pools: 1,627 pools (comprehensive coverage)
- Filtered Pools: 32 pools matching trained instance types
- Purpose: Real-world validation on completely unseen future data
- Sampling Frequency: Every 10 minutes
- Use Case: Validates model generalization to new time period

Dataset Expansion (In Progress)

Status: Actively expanding training corpus to improve model robustness

- Objective: Increase temporal coverage and pool diversity
- Target: 5M+ training records across 50+ pools
- Additional regions: Expanding beyond us-east-1 and ap-south-1
- Historical depth: Extending back to 2022 for longer trend learning

- Benefit: Better capture of rare events, seasonal variations, and regime changes

3.2 Data Schema

Column	Type	Description	Example
timestamp	datetime	Timestamp (10-min intervals)	2023-01-01 00:00:00
InstanceType	string	EC2 instance type	t3.medium
AZ	string	Availability zone	aps1-az1
SpotPrice	float	Spot price in USD	0.0134
OndemandPrice	float	On-demand price	0.0448
Region	string	AWS region	ap-south-1

3.3 Data Characteristics & Challenges

Key Challenge: Non-Stationary Data with Extreme Regime Changes

- Volatility: Prices can spike 50-100x during capacity crunches
- Heavy-tailed distribution: 5% of records have extreme values ($>3\sigma$)
- Pool heterogeneity: Stable pools vs highly volatile pools
- Seasonal patterns: Daily cycles (hour-of-day), weekly cycles (weekday vs weekend)
- Event-driven spikes: AWS outages, holidays, major releases
- Non-stationarity: Statistical properties change over time
- Missing data: Occasional gaps due to AWS API rate limits

4. Model Architecture: PatchTST

4.1 Architecture Overview

PatchTST (Patched Time Series Transformer) treats time series like images by breaking sequences into patches. This approach captures local patterns within patches and long-range dependencies between patches using self-attention.

Architecture Flowchart:

```
    INPUT DATA

  [Batch, 9 Features, 168 Timesteps]


          |

          ▼


|      PATCH EMBEDDING                 |

|  • Break into 13 patches (24h each)        |

|  • Stride: 12 hours (50% overlap)          |

|  • Each patch: 9 × 24 = 216 values         |

|  • Linear projection: 216 → 128 dim        |


          |

          ▼

|    POSITIONAL ENCODING                |

|  Add learnable position embeddings          |

└────────────────────┬──────────────────────────────┘

          |

          ▼
```

```
|   TRANSFORMER ENCODER (2 layers)               |

|  • Self-attention (4 heads)                   |

|  • Feed-forward: 128 → 512 → 128              |

|  • Layer normalization                        |


              |

              ▼


|       FLATTEN                         |

|  13 patches × 128 dim = 1,664-dim vector      |


              |

        ┌──────────┴──────────┐

        ▼            ▼


|  FORECAST HEAD   |  |   RISK HEAD     |

|  1664→256→128→24 |  |   1664→64→32→3  |

|  (24h prices)    |  |   (Risk classes)|

└──────────────────┘  └─────────────────┘
```

4.2 Feature Engineering Deep Dive

Feature engineering is critical for time series forecasting. Each feature serves a specific purpose in capturing different aspects of price dynamics, temporal patterns, and external influences.

Feature 1: SpotPrice (Primary Signal)

Type: Observed (real-time)

Calculation: Raw spot price from AWS API

Why it's generated: This is the target variable we're predicting. Including historical SpotPrice as a feature allows the model to learn autoregressive patterns - the best predictor of tomorrow's price is often today's price plus a trend. The model learns price momentum, mean reversion, and level shifts.

Example: If SpotPrice has been stable at $0.013 for 10 days, the model learns it will likely remain near $0.013 tomorrow absent other signals.

Feature 2: ratio (Price Efficiency)

Type: Observed (derived)

Calculation: ratio = SpotPrice / (OndemandPrice + 1e-6)

Why it's generated: Raw prices vary dramatically across instance types (t4g.small: $0.01, c5.large: $0.08). The ratio normalizes prices to a 0-1 scale, where 0.5 means spot is 50% of on-demand cost. This captures "relative expensiveness" independent of absolute price levels. When ratio approaches 0.9, it signals spot is losing its cost advantage.

Example: If ratio jumps from 0.3 to 0.7 in 6 hours, it indicates capacity crunch and potential price spike continuation.

Feature 3: ratio_velocity (Price Momentum)

Type: Observed (derived)

Calculation: ratio_velocity = diff(ratio) = ratio[t] - ratio[t-1]

Why it's generated: Captures the rate of change in pricing. A positive velocity means prices are accelerating upward; negative means they're declining. This first derivative helps the model anticipate trend continuation or reversal. It's particularly valuable during rapid transitions (stable → spike) where the level (ratio) might still look acceptable but the velocity signals danger.

Example: ratio_velocity = +0.05 per hour for 3 consecutive hours warns of an ongoing price surge, even if absolute ratio is still only 0.5.

Feature 4: spot_mean_24h (Daily Baseline)

Type: Observed (derived)

Calculation: rolling_mean(SpotPrice, window=24h).shift(1)

Why it's generated: Provides a smoothed daily average price, filtering out hourly noise. The shift(1) is critical - it ensures the 24h mean uses only past data, preventing data leakage. This feature helps the model distinguish normal daily fluctuations from genuine trend changes. If current price is far above spot_mean_24h, it indicates an anomalous spike rather than a new baseline.

Example: If SpotPrice = $0.05 but spot_mean_24h = $0.02, the model learns this is likely a temporary spike that will revert to mean.

Feature 5: spot_mean_168h (Weekly Baseline)

Type: Observed (derived)

Calculation: rolling_mean(SpotPrice, window=168h).shift(1)

Why it's generated: Captures weekly trend and seasonal baseline. A 7-day window encompasses full weekly cycles (weekday vs weekend patterns). This longer-term average helps identify regime changes: if spot_mean_168h has been steadily increasing for weeks, it signals a fundamental shift in capacity/demand, not just daily variation. Used in conjunction with spot_mean_24h to separate short-term vs long-term trends.

Example: If spot_mean_24h > spot_mean_168h, it indicates recent prices are elevated relative to the weekly norm.

Feature 6: hour (Time-of-Day Pattern)

Type: Known (future values known)

Calculation: hour = timestamp.hour (0-23)

Why it's generated: Spot prices exhibit strong diurnal patterns. Business hours (9 AM - 5 PM) have higher demand as companies run batch jobs, ML training, data pipelines. Overnight (1 AM - 6 AM) sees lower usage. This cyclical pattern repeats daily. By encoding hour, the model learns "at 10 AM, expect 20% higher prices than 3 AM on average." This is a "known future" feature because when predicting 24h ahead, we know future hours.

Example: Model learns to predict a price bump every day at 9 AM UTC (Europe business hours) for us-east-1 pools.

Feature 7: day_of_week (Weekly Pattern)

Type: Known (future values known)

Calculation: day_of_week = timestamp.dayofweek (0=Monday, 6=Sunday)

Why it's generated: Weekdays (Mon-Fri) have consistently higher demand than weekends. Companies run heavy workloads during the work week; usage drops Friday evening through Sunday. Mondays often see spikes as accumulated weekend jobs launch. This weekly seasonality is predictable and helps the model adjust baselines by day. Tuesday 10 AM ≠ Saturday 10 AM in pricing behavior.

Example: Model predicts $0.015 for Tuesday but $0.011 for Saturday, same hour, based on learned weekly patterns.

Feature 8: is_weekend (Weekend Indicator)

Type: Known (future values known)

Calculation: is_weekend = 1 if day_of_week >= 5 else 0

Why it's generated: While day_of_week captures granular daily effects, is_weekend provides a binary signal that explicitly separates weekdays from weekends. Some patterns are best learned as "weekday behavior" vs "weekend behavior" rather than 7 separate days. This binary feature helps the model generalize: if it learns weekends are 30% cheaper, it applies to both Saturday and Sunday without needing separate parameters.

Example: Model uses is_weekend=1 to immediately lower its baseline prediction by 25%, regardless of other features.

Feature 9: event_impact (External Stress Events)

Type: Known (from event calendar)

Calculation: Merge with aws_stress_events_2023_2025.csv based on date ranges

Why it's generated: Certain events (AWS outages, Black Friday, Prime Day, major product launches) cause predictable demand surges or capacity constraints. These events are known in advance (holidays) or rapidly disseminated (outage announcements). By encoding event impact (0-10 scale), the model learns "during high-impact events, prices spike 50% above normal." This feature prevents the model from treating event-driven spikes as random noise. Events have pre-days and post-days (e.g., Black Friday: 3 days before, 2 days after).

Example: event_impact = 8 on Dec 24-26 (Christmas) signals to model: expect low demand and cheap prices despite it being weekdays.

4.3 Risk Classification Framework

The risk classification system provides actionable intelligence beyond raw price predictions. It translates forecasts into operational recommendations: STABLE (safe to run workloads), CAUTION (monitor closely), or MIGRATE (move workloads immediately). This three-tier system is designed to match real-world decision thresholds used by DevOps teams managing spot instances.

Risk Classification Logic (Detailed Explanation)

Risk labels are computed on REAL (inverse-transformed) prices, not scaled values. This is critical because our thresholds (5% volatility, 3% price change) are meaningful in dollar terms but meaningless on a [-3, 3] scaled range. The classification process works as follows:

Step 1: Extract Ground Truth Future Prices

For each training window, we look at the 24-hour forecast horizon - the actual prices that occurred after the lookback window. These 24 future prices are initially in scaled form (post-normalization). We inverse transform them back to real dollar values using the per-pool scaler fitted only on training data. This gives us the true price trajectory the model should predict.

Step 2: Extract Current Price

We identify the last price in the lookback window (the "current" price at prediction time). This represents the baseline against which we measure future changes. Like future prices, we inverse transform this to real dollars. This current price is what a DevOps engineer would see on their dashboard when deciding whether to keep workloads on spot.

Step 3: Compute Volatility

Volatility measures price unpredictability over the forecast horizon. We calculate the coefficient of variation: standard deviation divided by mean of the 24 future prices. If volatility is high, prices are bouncing around erratically, making it risky to commit long-running jobs. Low volatility means prices are stable and predictable. We use a small epsilon (1e-6) in the denominator to avoid division by zero if prices are uniformly zero (rare).

Formula: volatility = std(future_prices) / (mean(future_prices) + 1e-6)

Step 4: Compute Price Change Percentage

Price change measures trend: is the price going up or down, and by how much? We compute the absolute percentage difference between the final future price (24 hours ahead) and the current price. If this is large, it signals a significant trend (spike or drop). Small price changes indicate stability. We use absolute value because we care about magnitude of change, not direction - a 50% drop is as concerning as a 50% spike for risk management.

Formula: price_change = |final_price - current_price| / (current_price + 1e-6)

Step 5: Apply Thresholds for Classification

We classify into three risk levels based on empirically determined thresholds:

- STABLE (Class 0): volatility < 5% AND price_change < 3%
- Interpretation: Prices vary less than 5% on average and end within 3% of starting point.
- Operational Meaning: Safe to run long batch jobs, ML training, stateful workloads.
- Example: Price stays at $0.013 ± $0.0006 for 24 hours.

- CAUTION (Class 1): volatility < 15% AND price_change < 10%
- Interpretation: Moderate variability but still within reasonable bounds.
- Operational Meaning: Suitable for checkpointed workloads; monitor every 6 hours.
- Example: Price fluctuates between $0.012 and $0.015, ends at $0.014.

- MIGRATE (Class 2): volatility >= 15% OR price_change >= 10%
- Interpretation: High unpredictability or large trend shift.
- Operational Meaning: Immediate action required - migrate to stable pool or on-demand.
- Example: Price spikes from $0.013 to $0.025, or oscillates wildly between $0.01 and $0.04.

Why These Thresholds?

The 5%/3% and 15%/10% thresholds were chosen based on cloud cost management best practices:

- 3% price change: Within budget variance tolerance for most organizations
- 5% volatility: Allows for normal hourly fluctuations without false alarms
- 10% price change: Exceeds single-day budget limits; requires intervention
- 15% volatility: Indicates unstable pool; risk of cascading price spikes
- These align with AWS Savings Plans discount rates (20-70% off on-demand)

Risk Classification Flowchart:

```
┌─────────────────────────────────────────┐
│  INPUT: Future 24h prices (real $)      │
│         Current price (real $)          │
└────────────────┬────────────────────────┘
                 │
                 ▼
┌─────────────────────────────────────────┐
│  COMPUTE VOLATILITY                     │
│  vol = std(future) / mean(future)       │
└────────────────┬────────────────────────┘
                 │
                 ▼
┌─────────────────────────────────────────┐
│  COMPUTE PRICE CHANGE                   │
│  chg = |final - current| / current      │
└────────────────┬────────────────────────┘
                 │
        ┌────────┴────────┐
        ▼                 │
  vol < 5% ?              │
  chg < 3% ?              │
        │ YES             │ NO
        ▼                 ▼
   ┌─────────┐      ┌─────────┴─────────┐
```

```
| STABLE   |      |  vol < 15% ?     |
| Class 0  |      |  chg < 10% ?     |
└──────────┘    └────────┬─────────┘
                │
           ┌────────┴────────┐
           ▼ YES      NO ▼
         ┌──────────┐  ┌──────────┐
         | CAUTION  |  | MIGRATE  |
         | Class 1  |  | Class 2  |
         └──────────┘  └──────────┘
```

5. Development Journey: Bugs & Fixes

5.1 Initial Implementation

The project started with a Prophet-based approach (2.py) that had several limitations:

- Heavy reliance on Prophet (struggles with high-frequency, regime-shifting data)
- Hardcoded configuration (no YAML/CLI args)
- No model persistence or checkpointing
- Missing structured logging
- Data leakage risk (smoothing before splitting)

5.2 Critical Bugs Discovered

BUG #1: Data Leakage in Scaling

Severity: CRITICAL

Problem:

```
# WRONG: Fitting scaler on ENTIRE dataset before splitting
df_scaled = scaler.fit_transform(df_full)  # Sees validation & test data!
train, val, test = split_temporal(df_scaled)

# This causes:
# - Validation metrics overly optimistic
# - Real-world performance worse than expected
# - Model "cheats" by knowing future statistics
```

Solution:

```
# CORRECT: Split FIRST, then fit scaler only on training
train, val, test = split_temporal(df_raw)  # Split before any processing

preprocessor = CausalPreprocessor()
preprocessor.fit(train)  # Only fits on training data

train_processed = preprocessor.transform(train)
val_processed = preprocessor.transform(val)
test_processed = preprocessor.transform(test)
```

BUG #2: Risk Labels on Scaled Data

Severity: CRITICAL

Problem:

```
# WRONG: Computing volatility on scaled values
future_scaled = [0.5, 1.2, 0.8, ...]  # After RobustScaler
volatility = np.std(future_scaled) / np.mean(future_scaled)

# Risk thresholds (5%, 15%) designed for REAL prices
# But applied to scaled [-5, 5] range = meaningless!
if volatility < 0.05:  # This never works on scaled data
   risk = STABLE
```

Solution:

```
# CORRECT: Inverse transform to real prices first
def compute_risk_label(self, future_scaled, current_scaled, pool_id):
   # Convert back to real dollar values
   future_real = self.preprocessor.inverse_transform_target(
      future_scaled, pool_id
   )
   current_real = self.preprocessor.inverse_transform_target(
      np.array([current_scaled]), pool_id
   )[0]

   # Now compute volatility on REAL prices
   volatility = np.std(future_real) / (np.mean(future_real) + 1e-6)
   price_change = abs((future_real[-1] - current_real) / (current_real + 1e-6))

   # Apply thresholds
   if volatility < 0.05 and price_change < 0.03:
      return 0  # STABLE
   elif volatility < 0.15 and price_change < 0.10:
      return 1  # CAUTION
   else:
      return 2  # MIGRATE
```

BUG #3: Memory Explosion

Severity: HIGH

Problem:

```
# WRONG: Loading entire dataset at once
df = pd.read_csv('file.csv')  # 2.3M records × 13 features = ~1.5GB

# With copies during processing:
df_clean = clean(df)        # Another 1.5GB
df_featured = engineer(df_clean)  # Another 1.5GB
Total: 4.5GB for one operation!

# Won't run on 16GB RAM systems with other processes
```

Solution:

```
# CORRECT: Chunked loading with progress tracking
class MemoryEfficientDataHandler:
    @staticmethod
    def load_csv_chunked(path, chunk_size=100000):
        chunks = []
        total_rows = 0

        for chunk in pd.read_csv(path, chunksize=chunk_size):
            chunk = process_chunk(chunk)  # Process incrementally
            chunks.append(chunk)
            total_rows += len(chunk)

            if total_rows % 500000 == 0:
                logger.info(f"Loaded {total_rows:,} rows...")

        return pd.concat(chunks, ignore_index=True)

# Memory usage: ~400MB max (down from 4.5GB!)
```

BUG #4: Inefficient Dataset (50-100x Slowdown)

Severity: HIGH

Problem:

```
# WRONG: DataFrame filtering on every __getitem__ call
class SlowDataset(Dataset):
    def __getitem__(self, idx):
        pool_id, start_idx = self.indices[idx]

        # O(n) operation on EVERY access!
        pool_df = self.df[self.df['Pool_ID'] == pool_id]

        # Extract window
        X = pool_df.iloc[start_idx:start_idx+168]

        return X

# With 45,000 windows: 45,000 × O(n) = extremely slow!
# Epoch time: ~5 minutes
```

Solution:

```
# CORRECT: Pre-compute numpy arrays ONCE
class OptimizedDataset(Dataset):
    def __init__(self, df, ...):
        # Pre-split data by pool (O(n) once)
        self.pool_data = {}
        for pool_id in df['Pool_ID'].unique():
            pool_df = df[df['Pool_ID'] == pool_id]
            self.pool_data[pool_id] = {
                'features': pool_df[cols].values.astype(np.float32),
                'target': pool_df[['SpotPrice']].values.astype(np.float32)
            }

    def __getitem__(self, idx):
        pool_id, start_idx = self.indices[idx]
        data = self.pool_data[pool_id]

        # Direct numpy slicing: O(1)!
        X = data['features'][start_idx:start_idx+168].T

        return torch.from_numpy(X).float()

# Epoch time: ~30 seconds (10x speedup!)
```

6. Code Implementation

6.1 Complete Preprocessing Pipeline

```
class CausalPreprocessor:
    """No data leakage - split before fitting"""

    def __init__(self, config: Config):
        self.config = config
        self.scalers = defaultdict(dict)
        self.feature_cols = []
        self.pool_ids = []

    def fit(self, df: pd.DataFrame, df_events: pd.DataFrame = None):
        """Fit ONLY on training data"""
        logger.info("🔧 Fitting preprocessor (training data only)")

        df = self._resample_hourly(df)
        df = self._minimal_cleaning(df)
        df = self._engineer_causal_features(df, df_events)

        if config.per_pool_scaling:
            self._fit_per_pool_scalers(df)

        self.pool_ids = df['Pool_ID'].unique().tolist()
        return self

    def transform(self, df: pd.DataFrame, df_events: pd.DataFrame = None):
        """Transform using fitted scalers"""
        df = self._resample_hourly(df)
        df = self._minimal_cleaning(df)
        df = self._engineer_causal_features(df, df_events)
        df = self._apply_scaling(df)
        return df

    def _engineer_causal_features(self, df, df_events=None):
        """All features prevent future leakage"""
        # Price features
        df['ratio'] = df['SpotPrice'] / (df['OndemandPrice'] + 1e-6)
        df['ratio_velocity'] = df.groupby('Pool_ID')['ratio'].shift(1).diff()

        # Rolling features (with shift to prevent leakage!)
        for window in [24, 168]:
            df[f'spot_mean_{window}h'] = df.groupby('Pool_ID')['SpotPrice'].shift(1).transform(
                lambda x: x.rolling(window=window, min_periods=1).mean()
            )

        # Time features
        df['hour'] = df['timestamp'].dt.hour
        df['day_of_week'] = df['timestamp'].dt.dayofweek
        df['is_weekend'] = (df['day_of_week'] >= 5).astype(int)

        # Event impact
        if df_events is not None:
            df['event_impact'] = 0.0
            for _, event in df_events.iterrows():
                event_date = pd.to_datetime(event['Date'])
                event_start = event_date - pd.Timedelta(days=event['PreDays'])
                event_end = event_date + pd.Timedelta(days=event['PostDays'])
                mask = (df['timestamp'] >= event_start) & (df['timestamp'] <= event_end)
                df.loc[mask, 'event_impact'] += event['ImpactLevel']

        self.feature_cols = [
            'SpotPrice', 'ratio', 'ratio_velocity',
            'spot_mean_24h', 'spot_mean_168h',
            'hour', 'day_of_week', 'is_weekend', 'event_impact'
        ]

        return df
```

6.2 PatchTST Model Architecture

```
class PatchTSTWithRiskHead(nn.Module):
    def __init__(self, input_dim, patch_len, stride, d_model, n_heads,
                 num_layers, dropout, lookback, horizon, num_risk_classes=3):
        super().__init__()

        # Patch embedding
        self.patch_embed = PatchEmbedding(patch_len, stride, input_dim, d_model)

        # Positional encoding
        num_patches = (lookback - patch_len) // stride + 1
        self.pos_embed = nn.Parameter(torch.randn(1, num_patches, d_model) * 0.02)

        # Transformer encoder
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=d_model,
            nhead=n_heads,
            dim_feedforward=d_model * 4,
            dropout=dropout,
            activation='gelu',
            batch_first=True,
            norm_first=True
        )
        self.transformer = nn.TransformerEncoder(encoder_layer, num_layers=num_layers)

        # Forecast head
        self.forecast_head = nn.Sequential(
            nn.Flatten(),
            nn.Linear(num_patches * d_model, d_model * 2),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(d_model * 2, d_model),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(d_model, horizon)
        )

        # Risk head
        self.risk_head = nn.Sequential(
            nn.Linear(num_patches * d_model, 64),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(64, 32),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(32, num_risk_classes)
        )

        self._reset_parameters()

    def _reset_parameters(self):
        """Xavier initialization with small gain"""
        for name, param in self.named_parameters():
            if 'weight' in name and param.dim() >= 2:
                nn.init.xavier_uniform_(param, gain=0.1)
            elif 'bias' in name:
                nn.init.zeros_(param)

    def forward(self, x, return_risk=False):
        # x: [batch, features, seq_len]
        x = self.patch_embed(x)
        x = x + self.pos_embed
        x_encoded = self.transformer(x)

        forecast = self.forecast_head(x_encoded).unsqueeze(1)
        forecast = torch.clamp(forecast, min=-10.0, max=10.0)

        if return_risk:
            risk_logits = self.risk_head(x_encoded.flatten(1))
            return forecast, risk_logits

        return forecast
```

6.3 Training Loop with Gradient Monitoring

```
class ProductionTrainer:
  def train_epoch(self, epoch: int):
    self.model.train()
    total_metrics = defaultdict(float)

    pbar = tqdm(self.train_loader, desc=f"Epoch {epoch+1}")

    for batch_idx, (X, y, y_risk, _) in enumerate(pbar):
      X = X.to(self.config.device)
      y = y.to(self.config.device)
      y_risk = y_risk.to(self.config.device)

      # Input sanity checks
      if torch.isnan(X).any() or torch.isinf(X).any():
        logger.error(f"Invalid input at batch {batch_idx}")
        continue

      if X.abs().max() > 20:
        logger.warning(f"Large input: {X.abs().max():.2f}")
        continue

      # Forward pass
      forecast_pred, risk_pred = self.model(X, return_risk=True)
      loss, metrics = self.criterion(forecast_pred, y, risk_pred, y_risk)

      # Loss checks
      if torch.isnan(loss):
        logger.error(f"NaN loss at batch {batch_idx}!")
        raise RuntimeError("Training diverged")

      if loss.item() > 10:
        logger.error(f"Loss explosion: {loss.item():.2f}")
        raise RuntimeError("Loss explosion - stopping")

      # Backward pass
      self.optimizer.zero_grad()
      loss.backward()

      # Gradient clipping with monitoring
      grad_norm = torch.nn.utils.clip_grad_norm_(
        self.model.parameters(), self.config.gradient_clip
      )
    if grad_norm > 10:
        logger.error(f"Gradient explosion: {grad_norm:.2f}")
        continue  # Skip batch

    if grad_norm < 1e-8:
        logger.warning(f"Vanishing gradients: {grad_norm:.2e}")

    self.optimizer.step()

    pbar.set_postfix({'loss': loss.item(), 'grad': f'{grad_norm:.2f}'})

return {k: v / len(self.train_loader) for k, v in total_metrics.items()}
```

7. Challenges & Solutions Summary

Challenge	Impact	Solution
Data Leakage	Validation metrics 5-10% too optimistic	Split data BEFORE fitting any scalers or processors
Memory Constraints	Cannot load 20M records on 16GB RAM	Chunked loading (100K rows at a time) + pre-computed numpy arrays
Gradient Explosion	Training crashes after 20-30 batches	ONGOING: Tried log transform + clipping; planning TFT migration
Risk on Scaled Data	Risk classifications 100% wrong	Compute risk on inverse-transformed real prices
Dataset Inefficiency	Epoch time: 5 min (too slow)	OptimizedDataset with pre-split numpy arrays → 30 sec/epoch

8. Current Results & Training Issues

8.1 Training Configuration

Model	**PatchTST (2 layers, 4 heads, d_model=128)**
Parameters	~1M (996,955)
Batch Size	16
Learning Rate	1e-4 (reduced from 5e-4)
Optimizer	AdamW (weight_decay=1e-4)
Scheduler	CosineAnnealingLR
Gradient Clipping	0.5 (reduced from 1.0)
Epochs	50 (with early stopping, patience=15)
Loss	70% Forecast + 30% Risk
Device	Apple M1 (MPS)
Status	Gradient instability - training fails at epoch 1

8.2 Gradient Explosion Analysis

Observed Behavior:

- Training consistently crashes after 20-30 batches
- Gradient norms: 1791.98, 1793.77, 8998.88 (should be <2.0)
- Loss explosion: 11.73 (should be <1.0 for normalized data)
- Input range: [-1.13, 4.00] post-clipping (still too large)
- No NaN/Inf values in inputs, but gradients explode during backprop

Attempted Fixes (All Failed):

- Fix #1: Removed volatility features (spot_std, volatility) → Gradient = 1791 (still exploded)
- Fix #2: Aggressive clipping to [-5, 5] → Loss = 11.73 (still diverged)
- Fix #3: Log transform + tighter clipping [-3, 3] → Gradients remained unstable
- Fix #4: Reduced model size (d_model=128, 2 layers, 4 heads) → Marginal improvement only
- Fix #5: Lower learning rate (1e-4) + higher gradient clip (0.5) → Training stalled
- Fix #6: Batch normalization after patch embedding → No significant improvement

- Fix #7: Warm-up scheduler (5 epochs) → Still crashed in warm-up phase

8.3 Why PatchTST Fails on Spot Pricing Data

After extensive debugging and literature review, fundamental architectural incompatibilities between PatchTST and AWS spot pricing have been identified:

1. Architecture Designed for Smooth, Stationary Data

PatchTST was originally developed for weather forecasting, traffic prediction, and electricity load - domains where time series are relatively smooth with gradual changes. The patch mechanism assumes local smoothness: that adjacent timesteps within a 24-hour patch are correlated. AWS spot pricing violates this assumption with sudden regime shifts (stable \$0.013 → spike \$0.05 in 10 minutes). When patches contain these discontinuities, the linear projection layer amplifies the variance.

2. Self-Attention Amplifies Non-Stationarity

Transformer self-attention computes attention scores as dot products of query-key pairs. With non-stationary data, some patches have extremely high activation magnitudes while others are near zero. This creates imbalanced attention weights (some patches get 90% attention, others 1%), leading to gradient flow issues. During backpropagation, high-attention patches receive massive gradients while low-attention patches vanish. Financial time series literature shows this is a known problem with vanilla transformers on non-stationary data.

3. Patch Embedding Cannot Handle Heavy Tails

The patch embedding layer is a linear transformation that projects 216 input values (9 features × 24 hours) to 128 dimensions. With AWS pricing, even after RobustScaler normalization, 5% of windows contain extreme values (>3σ from median). These outliers cause the weight matrix to have very large singular values, making the transformation ill-conditioned. Small input perturbations lead to large output variations, destabilizing gradients. This is exacerbated by the ratio and ratio_velocity features, which create unbounded values when prices spike.

4. No Built-in Mechanism for Regime Detection

PatchTST treats all time periods uniformly through its positional encoding. It has no explicit mechanism to detect regime changes (stable → volatile) and adapt its behavior accordingly. AWS spot pricing has distinct regimes with different statistical properties: stable periods (70% of time, σ=0.002) vs spike periods (5% of time, σ=0.05). A fixed architecture cannot handle both without either underfitting stable periods or overfitting spikes. TFT addresses this with variable selection networks that adaptively weight features.

5. Initialization Scheme Mismatched to Data Scale

Xavier initialization assumes inputs are roughly [-1, 1] with Gaussian distribution. Even with clipping to [-3, 3], spot pricing features are heavily skewed and have extreme kurtosis. The initialization sets weight variance based on layer dimensions (gain / sqrt(fan_in)), but this doesn't account for input distribution properties. With heavy-tailed inputs, the optimal weight initialization should have lower variance. He initialization (designed for ReLU) doesn't help either, as we're using GELU activation.

6. Quantile Prediction Not Native

PatchTST produces point estimates through its forecast head. To get prediction intervals, we'd need to train multiple models or use Monte Carlo dropout. For risk classification in production, we need calibrated uncertainty estimates. Financial forecasting requires quantile predictions (10th, 50th, 90th percentile) to assess downside risk. TFT provides this natively through quantile regression loss, which is more stable than training separate models for each quantile.

7. Limited Interpretability for Stakeholder Trust

PatchTST's attention weights are difficult to interpret: they show which patches attend to which other patches, but patches are 24-hour chunks, not individual features. For DevOps teams to trust predictions, they need to know "price spike predicted because event_impact=8 and ratio_velocity=+0.05", not "patch 7 attended to patch 3". TFT provides variable importance scores at each timestep, making it explainable to non-ML stakeholders.

9. Future Work

9.1 Transition to Temporal Fusion Transformer (TFT)

Based on the PatchTST failure analysis, the next phase of the project will implement Temporal Fusion Transformer. TFT is specifically designed for real-world forecasting tasks with the exact characteristics present in AWS spot pricing data.

TFT Architecture Overview:

```
┌─────────────────────────────────────────────────────┐
│          INPUT FEATURES                             │
│  • Static: InstanceType, Region, AZ                 │
│  • Known: hour, day_of_week, is_weekend             │
│  • Observed: SpotPrice, ratio, event_impact         │
└───────────────────────┬─────────────────────────────┘
                        │
                        ▼
┌─────────────────────────────────────────────────────┐
│      VARIABLE SELECTION NETWORKS                    │
│ GRN (Gated Residual Network) for each variable type │
│  • Learns importance weights                        │
│  • Adaptive feature selection                       │
└───────────────────────┬─────────────────────────────┘
                        │
                        ▼
┌─────────────────────────────────────────────────────┐
│    LSTM ENCODER-DECODER                             │
│  • Encoder: Process historical context (168h)       │
│  • Decoder: Generate future representations (24h)   │
│  • Captures sequential dependencies              │
└──────────────────────┬──────────────────────────┘
                       │
                       ▼
┌─────────────────────────────────────────────────┐
│   TEMPORAL SELF-ATTENTION                       │
│  • Multi-head attention across time             │
│  • Learns long-range dependencies               │
│  • Interpretable attention weights              │
└──────────────────────┬──────────────────────────┘
                       │
                       ▼
┌─────────────────────────────────────────────────┐
│   QUANTILE OUTPUT LAYER                         │
│  • Predicts multiple quantiles (10%, 50%, 90%)  │
│  • Provides prediction intervals                │
│  • Uncertainty quantification                   │
└──────────────────────┬──────────────────────────┘
                       │
              ┌────────┴────────┐
              ▼                 ▼
    ┌─────────────────┐ ┌─────────────────┐
    │ PRICE FORECAST  │ │  RISK HEAD      │
    │ (24h quantiles) │ │  (3 classes)    │
    └─────────────────┘ └─────────────────┘
```

9.2 Why TFT Over PatchTST (Summary)

Criterion	PatchTST (Current)	TFT (Planned)
Gradient Stability	Poor (explosion)	Excellent (LSTM gating)
Non-Stationary Data	Struggles	Designed for regime changes
Heavy-Tailed Distribution	Fails	Robust via quantile loss
Interpretability	Black box	Feature importance + attention
Uncertainty	Point estimates only	Native quantile predictions
Multi-Horizon	Adapted from vision	Designed for forecasting
Feature Types	Uniform treatment	Static/Known/Observed separation
Training Stability	Requires extensive tuning	Stable out-of-the-box

9.3 TFT Implementation Plan

- Step 1: Use PyTorch Forecasting library (mature TFT implementation)
- Step 2: Define TimeSeriesDataSet with proper variable categorization
- Step 3: Train on 2023 Jan - 2024 Dec (24-month focused training)
- Step 4: Temporal split: 70% train / 15% val / 15% test
- Step 5: Hyperparameter tuning: hidden_size, attention heads, dropout
- Step 6: Add custom risk classification head on top of TFT encoder
- Step 7: Validate on 2025 Jan-Mar test set (completely unseen data)
- Step 8: Compare metrics against PatchTST baseline (if it trains)
- Step 9: Deploy best model with monitoring and retraining pipeline

Expected Outcomes:

- Training Stability: Full 50 epochs without gradient explosion
- Forecast MAPE: < 3% (improved from PatchTST which couldn't train)
- Risk Accuracy: > 75% (with calibrated probability estimates)
- Inference Speed: < 100ms per pool (acceptable for production)
- Interpretability: Variable importance scores for each prediction

- Uncertainty: Prediction intervals for risk management

9.4 Project Structure Evolution

Current Structure (PatchTST):

```
spot_ml_project/
 ├── data/
 |   ├── raw/
 |   |   ├── aws_2023_2024_complete_24months.csv (3.3M records)
 |   |   ├── mumbai_spot_data_sorted_asc.csv (20.7M records)
 |   |   └── aws_stress_events_2023_2025.csv (78 events)
 |   └── processed/
 ├── experiments/
 |   └── patchtst_v1/
 |       ├── checkpoints/ (empty - training failed)
 |       ├── logs/
 |       └── config.yaml
 ├── notebooks/
 |   └── patchtst_training.ipynb (debugging notebook)
 └── logs/
     └── patchtst_gradient_issues.log
```

Planned Structure (TFT):

```
spot_ml_project/
 ├── data/
 |   ├── raw/
 |   |   ├── aws_2023_2024_complete_24months.csv
 |   |   ├── mumbai_spot_data_sorted_asc.csv
 |   |   ├── aws_stress_events_2023_2025.csv
 |   |   └── expanded_dataset_2022_2024.csv (NEW - 5M+ records)
 |   └── processed/
 |       ├── train_2023_2024.parquet
 |       ├── val_2023_2024.parquet
 |       ├── test_2025.parquet
 |       └── scalers/ (per-pool RobustScalers)
 ├── experiments/
 |   ├── patchtst_v1/ (archived)
 |   └── tft_v1/
 |       ├── YYYYMMDD_HHMMSS/
 |       |   ├── config.yaml
 |       |   ├── checkpoints/
 |       |   |   ├── best_model.ckpt
 |       |   |   └── epoch_N.ckpt
 |       |   ├── metrics.json
 |       |   ├── training_history.png
 |       |   ├── variable_importance.csv
 |       |   ├── attention_weights.png
 |       |   └── visualizations/
 |       |       ├── forecast_pool_1.png
 |       |       ├── risk_confusion_matrix.png
 |       |       └── quantile_calibration.png
 ├── src/
 |   ├── preprocessing/
 |   |   ├── causal_preprocessor.py
 |   |   └── data_loader.py
 |   ├── models/
 |   |   ├── tft_wrapper.py (custom TFT with risk head)
 |   |   └── patchtst.py (archived)
 |   ├── training/
 |   |   ├── trainer.py
 |   |   └── callbacks.py
 |   └── evaluation/
 |       ├── metrics.py
 |       └── visualizations.py
 ├── notebooks/
 |   ├── tft_training.ipynb (main training pipeline)
 |   ├── tft_analysis.ipynb (results exploration)
 |   └── patchtst_postmortem.ipynb (failure analysis)
 ├── configs/
 |   ├── tft_base.yaml
 |   └── tft_tuned.yaml
 ├── logs/
```

```
|   └── tft_YYYYMMDD_HHMMSS.log
├── tests/
|    ├── test_preprocessing.py
|    ├── test_data_leakage.py
|   └── test_risk_labels.py
├── requirements.txt
├── setup.py
└── README.md
```

10. Lessons Learned

1. Data Leakage is Subtle and Pervasive

Even experienced practitioners can introduce leakage. Always split data BEFORE any processing, even "innocent" operations like scaling, outlier removal, or feature engineering. The shift(1) trick for rolling features is essential but easy to forget.

2. Architecture Choice Matters More Than Hyperparameter Tuning

Spent significant time tuning PatchTST (learning rate, model size, clipping) when the fundamental architecture was incompatible with the data. Should have recognized architectural limitations earlier and pivoted to TFT. No amount of tuning can fix a structural mismatch.

3. Financial/Pricing Data Needs Specialized Handling

Transformers designed for smooth, stationary data (weather, traffic) fail on heavy-tailed, non-stationary financial data. Always check if the model architecture has been validated on similar data distributions. TFT's success on electricity pricing and stock markets is a strong signal.

4. Gradient Monitoring Should Be First-Class

Implemented gradient norm logging as an afterthought. Should have had comprehensive gradient tracking from day 1: per-layer norms, histogram logging, attention weight variance. Early gradient instability is a clear signal to stop hyperparameter tuning and reconsider architecture.

5. Risk Labels Must Use Domain-Appropriate Scale

Computing risk thresholds (5%, 15%) on scaled values was obviously wrong in hindsight. Always ask: "Do these thresholds make sense on real dollar values?" Domain knowledge (cost management practices) should drive metric design, not ML convenience.

6. Optimization for Speed Pays Off Dramatically

Pre-computing numpy arrays gave 10x speedup (5min → 30sec per epoch). Memory-efficient chunked loading enabled processing 20M records. These optimizations enabled rapid iteration during debugging. Performance engineering is not optional for production ML.

7. Interpretability Is Critical for Stakeholder Adoption

DevOps teams won't trust black-box predictions. TFT's variable importance and attention weights provide the explainability needed for production deployment. Should have prioritized interpretability from the start.

8. Document Failures as Thoroughly as Successes

This technical document captures the PatchTST failure in detail. These lessons prevent repeating mistakes and guide future researchers. Negative results are valuable when properly documented.

11. Appendix: Current Project Structure

```
spot_ml_project/
 ├── data/
|    ├── raw/
|    |    ├── aws_2023_2024_complete_24months.csv (3.3M records, 24 months)
|    |    ├── mumbai_spot_data_sorted_asc(1-2-3-25).csv (20.7M records, Jan-Mar 2025)
|    |   └── aws_stress_events_2023_2025.csv (78 events)
|   └── processed/
|       └── (generated during training)
|
 ├── experiments/
|   └── patchtst_v1/
|       ├── logs/
|       |   └── gradient_explosion_analysis.log
|       └── config.yaml
|
 ├── notebooks/
|   └── patchtst_debugging.ipynb
|
 ├── logs/
|   └── patchtst_training_YYYYMMDD.log
|
└── README.md
```

Conclusion

This project documents the development of an AWS Spot Instance price forecasting system using PatchTST, including the comprehensive debugging process that revealed fundamental architectural incompatibilities with spot pricing data. Despite extensive efforts to stabilize training through log transforms, gradient clipping, feature reduction, and architectural modifications, PatchTST consistently fails due to gradient explosion caused by the non-stationary, heavy-tailed characteristics of financial time series data.

The failures documented here provide valuable insights for future work. The transition to Temporal Fusion Transformer represents a data-driven decision based on empirical evidence and architectural analysis. TFT's LSTM backbone, variable selection networks, quantile regression, and proven performance on financial datasets make it the appropriate choice for this forecasting task. The PatchTST implementation serves as a baseline and a cautionary tale about the importance of matching model architecture to data characteristics.

Future work will implement TFT on the same 2023-2024 training data with the goal of achieving <3% MAPE for price forecasting and >75% accuracy for risk classification, enabling production deployment for cost-optimized cloud workload management.