TRANS System

Complete Technical Evaluation

Temporal Sequence Architecture v17

Production-Ready ML System for Pattern Detection

Document Version: 1.0

Generated: 2026-01-06

TRANS System: Complete Technical Evaluation

Document Version: 1.0

Generated: 2026-01-06

System Version: v17 (Production-Ready)

Classification: Technical Reference Documentation

Table of Contents

---

1. Executive Summary

1.1 System Purpose

TRANS (Temporal Sequence Architecture) is a production-ready machine learning system designed to detect consolidation patterns in micro/small-cap stocks with potential for 40%+ breakout gains. The system identifies stocks exhibiting specific technical characteristics (volatility contraction, volume dryup, trend coiling) that historically precede explosive price movements.

1.2 Key Innovation

The primary innovation is replacing static 92-feature snapshots with temporal sequences (14 features x 20 timesteps) that capture pattern evolution over time. This enables the model to learn:

- How volatility contracts before breakouts (BBW trajectory)
- Volume dryup patterns preceding explosive moves
- Price position dynamics within consolidation boundaries

1.3 Performance Metrics

Metric	EU Model	US Model
Optimal R-Config	+4R/-1.5R	+6R/-2R
Expected Value	+5.42	+1.12
Top-5 Win Rate	26.1%	~15%
Signal Rate (EV>3)	<3%	<3%

Key Finding: EU markets are approximately 5x more profitable than US markets for this strategy.

2. System Architecture Overview

2.1 Five-Stage Pipeline

```
Stage 00: Pattern Detection
    Input:  OHLCV + adj_close data
    Output: detected_patterns.parquet
    Engine: State Machine Scanner (sleeper_scanner_v17)

Stage 01: Sequence Generation
    Input:  Detected patterns + OHLCV data
    Output: sequences.npy (N x 20 x 14), labels.npy, metadata.parquet
    Engine: PathDependentLabelerV17 + VectorizedFeatureCalculator

Stage 02: Model Training
    Input:  Sequences + Labels + Metadata
    Output: best_model.pt
    Engine: HybridFeatureNetwork + AsymmetricLoss

Stage 03: Prediction
    Input:  Model + New Sequences
    Output: predictions.parquet (class probabilities + EV)
    Engine: Inference Pipeline

Stage 04: Evaluation
    Input:  Predictions + Actual Outcomes
    Output: metrics.json, drift_report.json
    Engine: Trading Performance Evaluator
```

2.2 Data Flow Diagram

```
OHLCV Data
    |
    v
+--------------------+
| State Machine      |  NONE -> QUALIFYING (10d) -> ACTIVE -> COMPLETED/FAILED
| Pattern Detector   |
+--------------------+
    |
    v
+--------------------+
| Path-Dependent     |  100-day outcome window
| Labeler (V17)      |  "First Event Wins" logic
+--------------------+
    |
    v
+--------------------+
| Feature Extractor |  14 features x 20 timesteps
| (Vectorized)       |  Price relativization, volume log-ratio
+--------------------+
    |
    v
+--------------------+
| Temporal Hybrid    |  LSTM + CNN + Attention
| Neural Network     |  ~77k parameters
+--------------------+
    |
    v
+--------------------+
| Expected Value     |  EV = P(K0)*(-1) + P(K1)*(-0.1) + P(K2)*(+5)
| Calculator         |
+--------------------+
    |
    v
Signal: TRADE (EV >= 3.0) or PASS (EV < 3.0)
```

3. Pattern Detection State Machine

3.1 State Definitions

```
NONE         -> No pattern detected
QUALIFYING   -> Days 1-10: All qualification criteria must hold
ACTIVE       -> Day 10+: Pattern boundaries established, monitoring
MATURE       -> Day 30+: Adjusted boundaries (optional)
COMPLETED    -> 2 consecutive closes above upper boundary (breakout)
FAILED       -> 2 consecutive closes below lower boundary (breakdown)
```

3.2 Qualification Criteria (All Must Hold Simultaneously)

Criterion	Threshold	Purpose
BBW	< 30th percentile	Volatility contraction
ADX	< 32	Low trend strength (coili
Volume	< 35% of 20-day avg	Supply drying up
Daily Range	< 65% of 20-day avg	Price tightening

3.3 Microstructure Quality Gates (V17)

Liquidity Gate:

```
daily_dollar_volume = close * volume
if daily_dollar_volume < 50000:
    reject_pattern()  # Insufficient liquidity
```

Ghost Detection:

```
zero_volume_pct = (volume == 0).sum() / len(volume)
if zero_volume_pct > 0.20:
    reject_pattern()  # "Ghost stock" - too illiquid
```

Intelligent Tightness:

- Wick Tightness: (high - low) / close - Target: 40-50%
- Body Tightness: abs(close - open) / close - Target: <15%

3.4 Boundary Establishment

Boundaries are established using 90th/10th percentile of High/Low during qualification period:

```
upper_boundary = period_data['high'].quantile(0.90)
lower_boundary = period_data['low'].quantile(0.10)
power_boundary = upper_boundary * 1.005  # 0.5% buffer
```

This creates a "Volatility Envelope" that ignores top 10% of manipulation spikes (scam wicks).

3.5 Iron Core Rule

Only CLOSES count as structural breaks. Intraday wicks are allowed.

```
if row['close'] > upper_boundary:
    consecutive_violations += 1
elif row['close'] < lower_boundary:
    consecutive_violations += 1
else:
    consecutive_violations = 0  # Reset on re-entry

if consecutive_violations >= 2:
    # Pattern completes or fails
```

---

4. V17 Path-Dependent Labeling System

4.1 R-Multiple Framework

Definition:

```
R = lower_boundary - stop_loss
Entry = upper_boundary (breakout price)
Stop Loss = lower_boundary * (1 - STOP_BUFFER_PERCENT)
```

Example Calculation:

```
Upper Boundary = $14.02
Lower Boundary = $12.26
Stop Buffer = 8%

Stop Loss = $12.26 * 0.92 = $11.28
R = $12.26 - $11.28 = $0.98

Target (+5R) = $14.02 + (5 * $0.98) = $18.92 (+35%)
Danger (-2R) = $14.02 - (2 * $0.98) = $12.06
```

4.2 Three-Class System

Class	Name	Strategic Value	Trigger Condition	Trading Meaning
0	DANGER	-1.0	Close < Stop Loss	Catastrophic failure
1	NOISE	-0.1	Neither target nor danger	No edge realized
2	TARGET	+5.0	Close >= +5R AND next_ope	Explosive breakout
-1	GREY	Excluded	Max close >= +2.5R but no	Ambiguous outcome

4.3 "First Event Wins" Algorithm

```
for day in range(100):  # 100-day outcome window
    current_price = df.loc[day, 'adj_close']  # Split-adjusted!

    # Priority 1: Check breakdown
    if current_price < stop_loss:
        return 0  # DANGER - immediate exit

    # Priority 2: Check target with confirmation
    if day < 99:  # Need next day for confirmation
        if current_price >= target_price:
            next_open = df.loc[day + 1, 'open']
            if next_open >= target_price:
                return 2  # TARGET - confirmed

# Neither triggered after 100 days
if max_close >= grey_threshold:
    return -1  # GREY ZONE - exclude from training
return 1  # NOISE - stayed in range
```

4.4 Regional R-Multiple Configurations

Region	Target R	Danger R	Stop Buffer	Rationale
EU	+4R	-1.5R	8%	Less algo competition, sh
US	+6R	-2R	8%	Higher volatility, more n

4.5 Critical Implementation Notes

1. Uses adj_close for outcome tracking (prevents reverse split artifacts)
2. Next-open confirmation required for targets (filters wicks)
3. Grey zones excluded from training (ambiguous patterns)
4. Breakdown checked on ALL days including last day of window
5. Target check invalid on last day (no next_open available)

5. Neural Network Architecture

5.1 HybridFeatureNetwork Overview

```
Input: (batch, 20 timesteps, 14 features)
              |
      +-------+--------+
      |                |
   Branch A         Branch B
   (Temporal)       (Context)
      |                |
   LSTM(32,2)      GRN(5->32)
   Bidirectional   [DISABLED]
      |
   CNN[3,5]
      |
   Attention(8)
      |
   Pool -> 64
      |
   Coil Encoder
      |
   +---------+
   |         |
   v         v
   Gated Fusion (when GRN enabled)
         |
   Linear(64->64)
         |
   Dropout(0.3)
         |
   Linear(64->3)
         |
   Output: [P(Danger), P(Noise), P(Target)]
```

5.2 Branch A: Temporal Pattern Detection

LSTM Path:

```
self.engineered_branch = nn.LSTM(
    input_size=14,
    hidden_size=32,
    num_layers=2,
    batch_first=True,
    dropout=0.2,
    bidirectional=False
)
# Output: (batch, 20, 32) -> take last timestep -> (batch, 32)
```

CNN Path:

```
self.conv_3 = nn.Conv1d(14, 32, kernel_size=3, padding=1)
self.conv_5 = nn.Conv1d(14, 64, kernel_size=5, padding=2)
# Output: (batch, 96, 20) -> attention -> (batch, 96)
```

Attention:

```
self.attention = nn.MultiheadAttention(
    embed_dim=96,
    num_heads=8,
    dropout=0.1,
    batch_first=True
)
# Output: (batch, 20, 96) -> mean pool -> (batch, 96)
```

Coil Encoder:

```
self.coil_encoder = nn.Sequential(
    nn.Linear(128, 64),  # LSTM(32) + CNN/Attn(96) = 128
    nn.LayerNorm(64),
    nn.ReLU(),
    nn.Dropout(0.2)
)
# Output: (batch, 64) - the "coil embedding"
```

5.3 Branch B: GRN Context (DISABLED)

Status: USE_GRN_CONTEXT = False in config/constants.py

When enabled, Branch B processes 5 static context features:

- Float Turnover
- Trend Position
- Base Duration
- Relative Volume
- Distance to High

The GRN generates a multiplicative gate that modulates the temporal signal:

```
gate = torch.sigmoid(self.context_gate_proj(context_embedding))
gated_coil = coil_embedding * gate
```

5.4 Parameter Count

```
LSTM:      14*32 + 32*32*4 = 4,544 (x2 layers)
CNN:       14*32*3 + 14*64*5 = 5,824
Attention: 96*96*3 = 27,648
Encoder:   128*64 = 8,192
Fusion:    64*64 + 64*3 = 4,288
-----------------------------------
Total:     ~77,000 parameters
```

6. Feature Engineering

6.1 14 Temporal Features

Index	Feature	Normalization	Description
0	open	Relative to day 0	(open_t / close_0) - 1
1	high	Relative to day 0	(high_t / close_0) - 1
2	low	Relative to day 0	(low_t / close_0) - 1
3	close	Relative to day 0	(close_t / close_0) - 1
4	volume	Log ratio to day 0	log(volume_t / volume_0)
5	bbw_20	Raw (bounded)	Bollinger Band Width
6	adx	Raw (0-100)	Average Directional Index
7	volume_ratio_20	Raw	volume / 20-day avg
8	vol_dryup_ratio	Window-normalized	mean(vol[-3:]) / mean(vol
9	var_score	Window-normalized	Volume Accumulation Rate
10	nes_score	Window-normalized	Normalized Energy Score
11	lpf_score	Window-normalized	Liquidity Flow Pressure
12	upper_boundary	Relative to day 0	(upper - close_0) / close
13	lower_boundary	Relative to day 0	(lower - close_0) / close

6.2 Price Relativization (Indices 0-3, 12-13)

Purpose: Make model price-agnostic (a $2 stock and $200 stock produce identical sequences for identical patterns)

```
# Reference: close at timestep 0
reference_close = windows[:, 0:1, 3:4]  # (n_windows, 1, 1)

# Relativize: (Price_t / Price_0) - 1
for idx in [0, 1, 2, 3, 12, 13]:
    windows[:, :, idx] = windows[:, :, idx] / reference_close - 1
```

Example:

- Day 0 close = $50
- Day 5 close = $55
- After: Day 5 = +0.10 (10% gain)

6.3 Volume Log-Ratio Normalization (Index 4)

Purpose: Prevent raw volume (20,000+) from dominating feature space

```
# Reference: volume at timestep 0
volume_0 = windows[:, 0, 4:5]

# Log ratio: log(volume_t / volume_0)
volume_normalized = np.log(volume_all / volume_0)

# Clip extreme values
volume_normalized = np.clip(volume_normalized, -4, 6)
```

Interpretation:

- 0.0 = same volume as day 0
- +0.69 = volume doubled
- -0.69 = volume halved
- +2.3 = 10x spike

6.4 Composite Score Window Normalization (Indices 8-11)

Purpose: Scale-invariant within each 20-day window

```
# Per-sequence, per-feature normalization
for feature in [8, 9, 10, 11]:
    mean = windows[:, :, feature].mean(axis=1, keepdims=True)
    std = windows[:, :, feature].std(axis=1, keepdims=True)
    windows[:, :, feature] = (windows[:, :, feature] - mean) / std
```

6.5 Vol_DryUp_Ratio (Primary Signal)

Formula:

```
vol_dryup_ratio = mean(volume[-3:]) / mean(volume[-20:])
```

Signal Interpretation:

Value	Interpretation
< 0.30	IMMINENT MOVE - volume se
0.30-0.50	Building pressure
0.50-0.80	Normal activity
> 0.80	Volume stable/increasing

6.6 ADX Warmup Fix (2026-01-06)

Problem: ADX requires 14-day warmup. Sequences starting at pattern day 0 had invalid ADX values (~96 instead of ~25-35).

Solution: Include INDICATOR_WARMUP_DAYS (30 days) of prefix data before pattern start.

```
warmup_start_idx = max(0, pattern.start_idx - INDICATOR_WARMUP_DAYS)
warmup_offset = pattern.start_idx - warmup_start_idx
pattern_data = df.iloc[warmup_start_idx:pattern.end_idx + 1]

# Windows start AFTER warmup_offset
windows = [
    feature_data[warmup_offset + i : warmup_offset + i + window_size]
    for i in range(n_windows)
]
```

7. Training Pipeline

7.1 Temporal Split (ENFORCED)

Critical: Random split is BLOCKED. Prevents window overlap leakage.

```
def temporal_train_test_split(dates, split_ratio=0.8):
    sorted_indices = dates.argsort()
    train_size = int(n_samples * split_ratio)

    train_indices = sorted_indices[:train_size]   # Earlier dates
    test_indices = sorted_indices[train_size:]    # Later dates

    # Validate no temporal leakage
    assert dates[train_indices].max() < dates[test_indices].min()
```

7.2 Pattern-Aware Splitting

Each pattern generates ~11 sequences (sliding windows). These MUST stay together.

```
# Split by pattern_id, not by sequence
unique_patterns = np.unique(pattern_ids)

train_patterns = patterns[pattern_date < train_cutoff]
val_patterns = patterns[(pattern_date >= train_cutoff) & (pattern_date < val_cutoff)]
test_patterns = patterns[pattern_date >= val_cutoff]

# Get sequences for each split
train_mask = np.isin(pattern_ids, train_patterns)
X_train = sequences[train_mask]
```

7.3 Feature Normalization (Training)

```
# Selective normalization - skip composite scores (already window-normalized)
features_to_normalize = [0, 1, 2, 3, 4, 5, 6, 7, 12, 13]  # Market + Technical + Boundaries
composite_features = [8, 9, 10, 11]  # SKIP - already normalized

# Compute mean/std from TRAIN set only
for feat_idx in features_to_normalize:
    train_mean[feat_idx] = X_train[:, :, feat_idx].mean()
    train_std[feat_idx] = X_train[:, :, feat_idx].std()

# Apply to all sets using TRAIN statistics
X_train = (X_train - train_mean) / train_std
X_val = (X_val - train_mean) / train_std
X_test = (X_test - train_mean) / train_std
```

7.4 AsymmetricLoss (Focal Loss Variant)

Purpose: Handle class imbalance by down-weighting easy examples

```
class AsymmetricLoss(nn.Module):
    def __init__(
        self,
        gamma_neg: float = 2.0,     # Focus on hard negatives
        gamma_pos: float = 1.0,     # Focus on hard positives
        label_smoothing: float = 0.01  # Prevent mode collapse
    ):
        ...

    def forward(self, logits, targets):
        # Stable log-probabilities
        log_probs = F.log_softmax(logits, dim=-1)

        # Focal weight: (1 - p_t)^gamma
        p_t = torch.exp(log_probs.gather(1, targets.unsqueeze(1)))
        focal_weight = (1.0 - p_t).pow(gamma_t)

        # Cross-entropy with label smoothing
        ce_loss = F.cross_entropy(logits, targets,
                                   label_smoothing=self.label_smoothing)

        return (focal_weight * ce_loss).mean()
```

7.5 Training Configuration

```
# Optimizer
optimizer = torch.optim.Adam(
    model.parameters(),
    lr=0.0001,
    weight_decay=1e-5
)

# Scheduler
scheduler = ReduceLROnPlateau(
    optimizer,
    mode='min',
    factor=0.5,
    patience=5
)

# Early stopping
early_stopping_patience = 15
```

7.6 Class Weights

```
CLASS_WEIGHTS = {
    0: 2.0,    # Danger - uncommon but important
    1: 1.0,    # Noise - common (base case)
    2: 10.0,   # Target - rare (what we want)
}
```

---

8. Expected Value System

8.1 Strategic Values

```
STRATEGIC_VALUES = {
    0: -1.0,   # DANGER: Costs 1R (Stop Loss hit)
    1: -0.1,   # NOISE: Opportunity cost / fees
    2: +5.0,   # TARGET: Average win (conservative)
}
```

8.2 EV Calculation

```
EV = P(Danger) * (-1.0) + P(Noise) * (-0.1) + P(Target) * (+5.0)
```

Example:

```
P(Danger) = 0.10  ->  0.10 * (-1.0)  = -0.10
P(Noise)  = 0.30  ->  0.30 * (-0.1)  = -0.03
P(Target) = 0.60  ->  0.60 * (+5.0)  = +3.00
                      -------------------
                      Total EV = +2.87
```

8.3 Signal Thresholds

Threshold	EV Range	Win Rate	Signal Rate	Use Case
STRONG	>= 5.0	90%+	<1%	Ultra-selective
GOOD	>= 3.0	60%+	~3%	**RECOMMENDED**
MODERATE	>= 1.0	27%+	~15%	Too noisy

AVOID	< 0	-	-	Negative expectancy

8.4 Trading Decision Logic

```
if predicted_ev >= 3.0 and prob_danger < 0.20:
    SIGNAL = "BUY"
    entry = pattern.upper_boundary
    stop = pattern.lower_boundary * 0.92  # 8% buffer
    target = entry + (5 * R)
else:
    SIGNAL = "PASS"
```

9. Evaluation Metrics

9.1 Primary Metric: Precision @ Top 15%

This is the only metric that matters for live trading.

```
def compute_precision_at_top_k(df, k_percent=15.0):
    # 1. Sort by predicted EV (descending)
    df_sorted = df.sort_values('predicted_ev', ascending=False)

    # 2. Take top k%
    k_count = int(len(df_sorted) * k_percent / 100)
    top_k = df_sorted.head(k_count)

    # 3. Calculate precision
    target_hits = (top_k['actual_label'] == 2).sum()
    precision = target_hits / k_count

    return precision  # Target: 40-60%
```

Why This Matters:

- Global accuracy includes patterns you'll never trade
- Precision @ Top 15% = your actual live trading win rate

9.2 Drift Monitoring (PSI)

Formula:

```
PSI = SUM[(current_% - reference_%) * ln(current_% / reference_%)]
```

Thresholds:

PSI	Interpretation	Action
< 0.1	No drift	Continue
0.1-0.25	Moderate drift	Monitor
> 0.25	Significant drift	**RETRAIN**

Monitored Features:

```
DRIFT_CRITICAL_FEATURES = [
    'vol_dryup_ratio',  # PRIMARY - micro-cap cycle indicator
    'bbw',              # Volatility contraction
]
```

---

10. Temporal Integrity Guarantees

10.1 No Look-Ahead Bias Mechanisms

1. State Machine: Can only see past and current day data
2. Temporal Split: Train on past, test on future (random blocked)
3. Feature Normalization: Relative to day 0 (no future data)
4. Indicator Warmup: 30-day prefix data before pattern start
5. 100-Day Outcome Window: Labels only after outcome completes

10.2 Temporal Validation

```
def validate_temporal_split(dates, train_indices, test_indices):
    train_dates = dates.iloc[train_indices]
    test_dates = dates.iloc[test_indices]

    max_train = train_dates.max()
    min_test = test_dates.min()

    if max_train > min_test:
        raise ValueError("Temporal leakage detected!")
```

11. Critical Files Reference

11.1 Tier 1: Must-Read (Core Understanding)

File	Location	Purpose
`temporal_hybrid.py`	`models/`	Neural network architectu
`path_dependent_labeler.p	`core/`	V17 3-class labeling
`sleeper_scanner_v17.py`	`core/`	Microstructure-aware dete
`constants.py`	`config/`	Strategic values, thresho
`02_train_temporal.py`	`pipeline/`	Training loop

11.2 Tier 2: Important (Deep Understanding)

File	Location	Purpose
`consolidation_tracker.py	`core/`	State machine implementat
`pattern_detector.py`	`core/`	Sequence generation
`vectorized_calculator.py	`features/`	14-feature extraction
`asymmetric_loss.py`	`models/`	Focal loss for imbalance
`evaluate_trading_perform	`pipeline/`	Trading metrics + drift

11.3 Tier 3: Supporting (Complete Understanding)

File	Location	Purpose
`temporal_split.py`	`utils/`	Temporal train/test split
`temporal_features.py`	`config/`	Feature configuration
`drift_monitor.py`	`ml/`	PSI calculation
`03_predict_temporal.py`	`pipeline/`	EV prediction
`data_loader.py`	`utils/`	GCS/local data loading

Appendix A: Complete Source Code

A.1 Configuration: constants.py

```python
"""
Centralized Constants for TRANS Temporal Architecture

Single source of truth for all constants used across the system.
Eliminates duplication and ensures consistency.
"""

from enum import Enum
from typing import Dict, Final


# ================================================================================
# OUTCOME CLASSES AND STRATEGIC VALUES (V17 Path-Dependent)
# ================================================================================

class OutcomeClass(Enum):
    """Path-dependent outcome classifications with risk-based thresholds

    3-class system using Risk Multiples (R) instead of fixed percentages.
    Values calibrated for realistic R-based trading:
    - Danger: -1R (stop loss hit)
    - Noise: -0.1R (opportunity cost / fees)
    - Target: +5R (conservative average win)
    """
    DANGER = (0, -1.0, "Stop Loss - Costs 1R")
    NOISE = (1, -0.1, "Base case - Opportunity cost / fees")
    TARGET = (2, 5.0, "Winner - Average win (conservative)")
    GREY_ZONE = (-1, None, "Ambiguous - Excluded from training")

    def __init__(self, class_id: int, strategic_value: float, description: str):
        self.class_id = class_id
        self.strategic_value = strategic_value
        self.description = description

# Strategic values for EV calculation
STRATEGIC_VALUES: Final[Dict[int, float]] = {
    0: -1.0,   # Danger: Costs 1R (Stop Loss)
    1: -0.1,   # Noise: Opportunity Cost / Fees
    2: 5.0,    # Target: Average Win (conservative estimate)
}

# ================================================================================
# SIGNAL THRESHOLDS
# ================================================================================

class SignalStrength(Enum):
    """Signal strength thresholds based on Expected Value"""
    STRONG = 5.0      # High confidence signal
    GOOD = 3.0        # Good confidence signal
    MODERATE = 1.0    # Moderate confidence signal
    WEAK = 0.0        # Weak/no signal
    AVOID = -1.0      # Negative EV, avoid

SIGNAL_THRESHOLDS: Final[Dict[str, float]] = {
    'STRONG': 5.0,
    'GOOD': 3.0,
    'MODERATE': 1.0,
    'WEAK': 0.0,
    'AVOID': -1.0
}
```

A.2 Neural Network: temporal_hybrid.py

```
"""
Multi-Modal Hybrid Temporal Model: LSTM + CNN + Attention + Gated Context

Branch A - Coil Shape Detector (Temporal):
1. LSTM Path: Captures sequential dependencies and long-term patterns
2. CNN Path: Discovers multi-scale temporal features (3, 5-day patterns)
3. Attention: Learns which timesteps and features are most important

Branch B - Potential Energy Detector (Context):
4. Gated Residual Network (GRN): Processes 5 static features with gating

Input:
  - Temporal: (batch_size, 20 timesteps, 14 features)
  - Context: (batch_size, 5 features) [Optional]
Output: (batch_size, 3 classes) - Danger, Noise, Target
"""

import torch
import torch.nn as nn
import torch.nn.functional as F
from typing import Optional


class GatedResidualNetwork(nn.Module):
    """Gated Residual Network (GRN) for Context Processing."""

    def __init__(self, input_dim: int, hidden_dim: int, dropout: float = 0.1):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.elu = nn.ELU()
        self.fc2 = nn.Linear(hidden_dim, hidden_dim * 2)
        self.glu = nn.GLU(dim=-1)
        self.dropout = nn.Dropout(dropout)
        self.norm = nn.LayerNorm(hidden_dim)

        if input_dim != hidden_dim:
            self.res_proj = nn.Linear(input_dim, hidden_dim)
        else:
            self.res_proj = nn.Identity()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        residual = self.res_proj(x)
        x = self.fc1(x)
        x = self.elu(x)
        x = self.fc2(x)
        x = self.dropout(x)
        x = self.glu(x)
        return self.norm(x + residual)


class HybridFeatureNetwork(nn.Module):
    """Multi-modal system with Gated Context Fusion."""

    def __init__(
        self,
        input_features: int = 14,
        sequence_length: int = 20,
        context_features: int = 5,
        lstm_hidden: int = 32,
        lstm_num_layers: int = 2,
```

A.3 Asymmetric Loss: asymmetric_loss.py

```python
"""
Production-Ready Asymmetric Focal Loss.

Improvements:
1. Numerical Stability: Uses Log-Softmax instead of Softmax -> Log
2. Label Smoothing: Prevents overfitting/mode collapse on noisy data
3. Hardware Safe: Uses buffers for auto-GPU handling
"""

import torch
import torch.nn as nn
import torch.nn.functional as F
from typing import Optional, List


class AsymmetricLoss(nn.Module):
    def __init__(
        self,
        gamma_neg: float = 2.0,
        gamma_pos: float = 1.0,
        gamma_per_class: Optional[List[float]] = None,
        class_weights: Optional[List[float]] = None,
        clip: float = 0.05,
        eps: float = 1e-8,
        disable_torch_grad_focal_loss: bool = True,
        label_smoothing: float = 0.01
    ):
        super().__init__()

        self.gamma_neg = gamma_neg
        self.gamma_pos = gamma_pos
        self.clip = clip
        self.eps = eps
        self.disable_torch_grad_focal_loss = disable_torch_grad_focal_loss
        self.label_smoothing = label_smoothing

        self.register_buffer('gamma_map', None)
        self.register_buffer('weight_map', None)

        self.gamma_config = gamma_per_class
        self.weights_config = class_weights

    def _init_buffers(self, num_classes, device):
        if self.gamma_config:
            gamma_t = torch.tensor(self.gamma_config, dtype=torch.float32, device=device)
        else:
            gamma_t = torch.ones(num_classes, device=device) * self.gamma_pos
            if num_classes > 1:
                gamma_t[1] = self.gamma_neg
        self.gamma_map = gamma_t

        if self.weights_config:
            weight_t = torch.tensor(self.weights_config, dtype=torch.float32, device=device)
        else:
            weight_t = torch.ones(num_classes, device=device)
        self.weight_map = weight_t

    def forward(self, logits: torch.Tensor, targets: torch.Tensor) -> torch.Tensor:
        if self.gamma_map is None:
            self._init_buffers(logits.size(1), logits.device)
```

A.4 Vectorized Feature Calculator: vectorized_calculator.py

```
"""
Vectorized Feature Calculator for Temporal Model

Optimized feature calculation using vectorized NumPy operations.
"""

import numpy as np
import pandas as pd
from typing import Tuple, Optional
import logging

logger = logging.getLogger(__name__)


class VectorizedFeatureCalculator:
    """Optimized feature calculation using vectorized operations"""

    def __init__(self):
        self.min_periods = 20

    def calculate_all_features(self, df: pd.DataFrame) -> pd.DataFrame:
        self._validate_input(df)
        df = self._calculate_technical_indicators(df)
        df = self._calculate_composite_scores(df)
        return df

    def _validate_input(self, df: pd.DataFrame) -> None:
        required = ['open', 'high', 'low', 'close', 'volume']
        missing = [col for col in required if col not in df.columns]
        if missing:
            raise ValueError(f"Missing required columns: {missing}")

    def _calculate_technical_indicators(self, df: pd.DataFrame) -> pd.DataFrame:
        df['bbw_20'] = self._vectorized_bbw(df['close'].values)
        df['adx'] = self._vectorized_adx(
            df['high'].values, df['low'].values, df['close'].values
        )
        df['volume_ratio_20'] = self._vectorized_volume_ratio(df['volume'].values)
        df['vol_dryup_ratio'] = self._vectorized_vol_dryup_ratio(df['volume'].values)
        return df

    def _calculate_composite_scores(self, df: pd.DataFrame) -> pd.DataFrame:
        df['cci_score'] = self._vectorized_cci(
            df['high'].values, df['low'].values, df['close'].values
        )
        df['var_score'] = self._vectorized_var(df['volume'].values, df['close'].values)
        df['nes_score'] = self._vectorized_nes(df['close'].values, df['volume'].values)
        df['lpf_score'] = self._vectorized_lpf(df['volume'].values, df['close'].values)
        return df

    def _vectorized_bbw(self, close: np.ndarray, period: int = 20) -> np.ndarray:
        close_series = pd.Series(close)
        sma = close_series.rolling(window=period, min_periods=1).mean()
        std = close_series.rolling(window=period, min_periods=2).std().bfill().fillna(0.0)

        upper_band = sma + (2 * std)
        lower_band = sma - (2 * std)

        bbw = np.where(sma != 0, (upper_band - lower_band) / sma, 0)
        return bbw
```

Document Information

File: TRANS_SYSTEM_COMPLETE_EVALUATION.md

Location: trans/

Size: ~2,500 lines

Format: PDF-optimized Markdown

Conversion to PDF:

```
# Using pandoc
pandoc TRANS_SYSTEM_COMPLETE_EVALUATION.md -o TRANS_EVALUATION.pdf \
    --pdf-engine=xelatex \
    --toc \
    --toc-depth=3 \
    -V geometry:margin=1in

# Using VSCode with Markdown PDF extension
# Right-click -> Markdown PDF: Export (pdf)
```

Document generated by Claude Code analysis of TRANS v17 codebase.