Enhanced Trading Bot Architecture

Project Structure

```
trading_bot/
├── app.py                      # Main Flask application
├── fyers_api.py                # API integration
├── indicators/                 # Technical indicators module
│   ├── __init__.py
│   ├── basic.py                # RSI, MACD, BB
│   ├── advanced.py             # EMA, Parabolic SAR, SuperTrend
│   └── composite.py            # Combined indicator calculations
├── strategy/                   # Strategy logic
│   ├── __init__.py
│   ├── weights.py              # Weight management
│   ├── signals.py              # Signal generation
│   └── backtesting.py         # Strategy validation
├── ml_engine/                  # Separate ML project
│   ├── data_collection.py      # Historical data gathering
│   ├── feature_engineering.py # Indicator calculation at scale
│   ├── weight_optimization.py # ML models for weight finding
│   ├── model_training.py      # Training pipelines
│   └── model_serving.py       # API for trained models
├── models/                     # Trained ML models storage
│   ├── weights_nse_reliance.pkl
│   ├── weights_nse_tcs.pkl
│   └── default_weights.json
├── config/                     # Configuration management
│   ├── __init__.py
│   ├── trading_params.py
│   └── ml_params.py
└── utils/                      # Utility functions
    ├── data_utils.py
    └── validation.py
```

Implementation Phases

Phase 1: Enhanced Indicators (Immediate)

1. Add EMA, Parabolic SAR, SuperTrend to `indicators/advanced.py`
2. Create composite indicator in `indicators/composite.py`

3. Update strategy logic to use weighted scoring

Phase 2: ML Weight Optimization (2-3 months)

1. Build data collection pipeline for historical data
2. Implement basic ML model for weight optimization
3. Create model serving API
4. Integrate with main trading bot

Phase 3: Advanced ML Features (6+ months)

1. Multi-asset weight optimization
2. Real-time model updates
3. Market regime detection
4. Risk management integration

Technical Implementation

1. Enhanced Strategy Module

python

```python
# strategy/signals.py
class WeightedSignalGenerator:
    def __init__(self, weights_config):
        self.weights = weights_config

    def calculate_composite_score(self, indicators_df):
        score = 0
        for indicator, weight in self.weights.items():
            indicator_score = self._normalize_indicator(
                indicators_df[indicator].iloc[-1],
                indicator
            )
            score += indicator_score * weight
        return score

    def generate_signal(self, composite_score):
        if composite_score > self.weights['buy_threshold']:
            return 'BUY'
        elif composite_score < self.weights['sell_threshold']:
            return 'SELL'
        return 'NEUTRAL'
```

2. ML Integration Pattern

python

```python
# ml_engine/model_serving.py
class WeightOptimizer:
    def __init__(self, model_path):
        self.model = joblib.load(model_path)

    def get_optimal_weights(self, symbol, market_conditions):
        features = self._prepare_features(symbol, market_conditions)
        weights = self.model.predict(features)
        return self._format_weights(weights)

    def update_weights_periodically(self):
        # Retrain model with latest data
        pass
```

3. Data Pipeline

python

```python
# ml_engine/data_collection.py
class HistoricalDataCollector:
    def collect_multi_asset_data(self, symbols, years=5):
        # Collect historical data for multiple assets
        # Calculate all indicators
        # Prepare ML training dataset
        pass

    def calculate_performance_metrics(self, signals, prices):
        # Calculate returns, Sharpe ratio, max drawdown
        # This becomes your ML target variable
        pass
```

ML Model Recommendations

Initial Approach: Linear Models

python

```python
from sklearn.linear_model import Ridge, Lasso
from sklearn.ensemble import RandomForestRegressor

# Start with Ridge regression for interpretability
model = Ridge(alpha=1.0)
```

Advanced Approach: Ensemble Methods

python

```python
from sklearn.ensemble import VotingRegressor
from xgboost import XGBRegressor
from lightgbm import LGBMRegressor

# Combine multiple models
ensemble = VotingRegressor([
    ('rf', RandomForestRegressor()),
    ('xgb', XGBRegressor()),
    ('lgb', LGBMRegressor())
])
```

Integration Strategy

Option 1: File-based Integration (Simple)

- ML model outputs weights to JSON file
- Trading bot reads weights on startup/periodically
- Good for MVP and testing

Option 2: API-based Integration (Scalable)

- ML engine runs as separate service
- Trading bot calls API to get weights
- Enables real-time updates and A/B testing

Option 3: Embedded Integration (Performance)

- Load trained model directly in trading bot
- Fastest execution, but larger memory footprint
- Best for production deployment

Performance Considerations

1. **Caching Strategy**: Cache weights for multiple symbols
2. **Model Versioning**: Track model performance over time
3. **Fallback Mechanism**: Default weights if ML model fails
4. **Monitoring**: Track prediction accuracy and trading performance

Risk Management

1. **Position Sizing**: Integrate Kelly Criterion for optimal position sizing
2. **Stop Losses**: Dynamic stop-loss based on volatility
3. **Portfolio Constraints**: Maximum allocation per asset
4. **Drawdown Protection**: Reduce position sizes during losing streaks