TradeNectics Trading Bot - Complete Technical Documentation

Table of Contents

Executive Summary

TradeNectics is an automated cryptocurrency trading bot built with C# and .NET 8.0. It combines technical analysis, machine learning, and risk management to execute trades on the Binance exchange automatically.

Key Features:

- **24/7 Automated Trading**: Runs as a background service
- **Machine Learning Predictions**: Uses ML.NET for trade decisions
- **Technical Analysis**: RSI, MACD, Bollinger Bands, Moving Averages
- **Risk Management**: Position sizing, stop-loss, drawdown protection
- **Paper Trading Mode**: Test strategies without real money
- **PostgreSQL Database**: Stores all market data and trade history

---

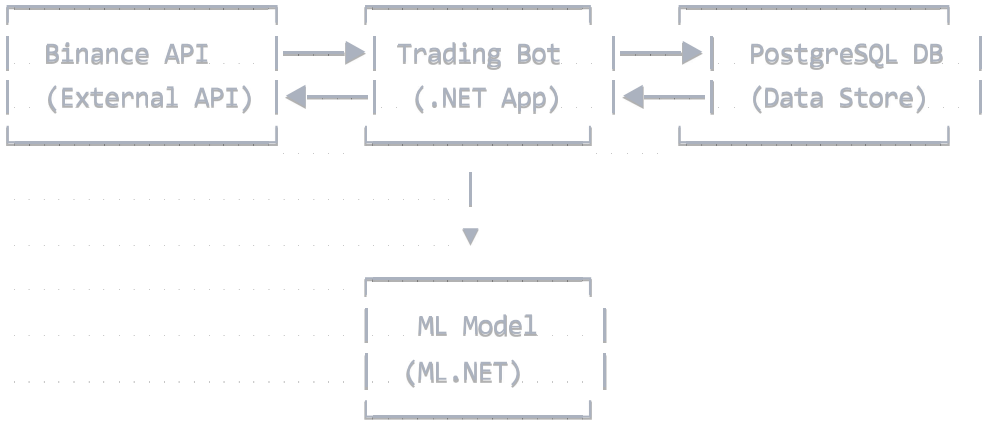
System Overview

What This Software Does

The TradeNectics bot is an automated trading system that:

1. **Connects to Binance Exchange** via REST API
2. **Analyzes Market Data** using technical indicators
3. **Predicts Trading Signals** using machine learning
4. **Executes Trades** automatically based on predictions
5. **Manages Risk** through position sizing and stop-losses
6. **Tracks Performance** in a PostgreSQL database

System Architecture Diagram



Architecture & Components

Technology Stack

- **Language**: C# 12.0
- **Framework**: .NET 8.0
- **ML Framework**: ML.NET 3.0
- **Database**: PostgreSQL with Entity Framework Core
- **HTTP Client**: System.Net.Http with Polly for resilience
- **Logging**: Serilog with file and console sinks
- **Background Processing**: IHostedService

Project Structure

```
TradeNectics/
├── Configuration/
│   └── TradingConfiguration.cs    # All settings
├── Models/
│   ├── Database/                  # EF Core entities
│   ├── API/                       # Binance API models
│   └── ML/                        # ML.NET models
├── Services/
│   ├── CryptoTraderService.cs     # API integration
│   ├── RiskManager.cs             # Risk management
│   ├── PortfolioManager.cs        # Portfolio tracking
│   └── TradingBotService.cs       # Main bot logic
├── Infrastructure/
│   ├── TradingContext.cs          # EF Core context
│   └── TechnicalAnalysis.cs       # Indicator calculations
└── Program.cs                     # Entry point
```

Core Components Explained

1. Configuration Management

csharp

```csharp
public class TradingConfiguration
{
    public string ApiKey { get; set; } = "";
    public string ApiSecret { get; set; } = "";
    public string BaseApiUrl { get; set; } = "https://api.binance.com";
    public Dictionary<string, decimal> SymbolQuantities { get; set; } = new();
    public decimal MinConfidenceScore { get; set; } = 0.7m;
    public bool PaperTradingMode { get; set; } = true;
    public TimeSpan ModelRetrainingInterval { get; set; } = TimeSpan.FromDays(7);
    public decimal MaxPositionSize { get; set; } = 0.02m; // 2% of portfolio
    public decimal StopLossPercent { get; set; } = 0.05m; // 5% stop loss
    public decimal MaxDailyLoss { get; set; } = 0.10m; // 10% daily loss limit
    public string[] TradingSymbols { get; set; } = { "BTCUSDT", "ETHUSDT", "ADAUSDT" };
}
```

Purpose: Centralizes all configuration settings

- Similar to configuration files in `/etc/` on Linux systems
- API credentials are loaded from environment variables for security

- Paper trading mode allows testing without real money

2. Database Models

The system uses Entity Framework Core with these main entities:

MarketData

```csharp
public class MarketData
{
    public int Id { get; set; }
    public string Symbol { get; set; } = "";
    public DateTime Timestamp { get; set; }
    public decimal Open { get; set; }
    public decimal High { get; set; }
    public decimal Low { get; set; }
    public decimal Close { get; set; }
    public decimal Volume { get; set; }
    public float RSI { get; set; }
    public float MovingAverage5 { get; set; }
    public float MovingAverage20 { get; set; }
    public float BollingerUpper { get; set; }
    public float BollingerLower { get; set; }
    public float MACD { get; set; }
    public float Signal { get; set; }
    public float VolumeRatio { get; set; }
    public decimal PriceChange24h { get; set; }
    public decimal VolumeChange24h { get; set; }
}
```

TradeRecord

csharp

```csharp
public class TradeRecord
{
    public int Id { get; set; }
    public string Symbol { get; set; } = "";
    public string Side { get; set; } = "";  // "BUY" or "SELL"
    public decimal Quantity { get; set; }
    public decimal Price { get; set; }
    public DateTime ExecutedAt { get; set; }
    public string MLPrediction { get; set; } = "";
    public decimal PnL { get; set; }
    public decimal PortfolioValueBefore { get; set; }
    public decimal PortfolioValueAfter { get; set; }
    public string OrderId { get; set; } = "";
    public bool IsPaperTrade { get; set; }
    public float ConfidenceScore { get; set; }
}
```

3. Service Interfaces

The system uses interfaces for dependency injection and testability:

csharp

```csharp
public interface ICryptoTraderService
{
    Task<TickerPrice?> GetPriceAsync(string symbol);
    Task<Ticker24hr?> Get24hrTickerAsync(string symbol);
    Task<List<KlineData>> GetKlineDataAsync(string symbol, string interval = "1h", int limit =
    Task<OrderResponse?> PlaceOrderAsync(OrderRequest orderRequest);
    Task<AccountBalance[]?> GetAccountBalancesAsync();
    Task<string> GetMLPredictionAsync(string symbol);
}

public interface IRiskManager
{
    bool CanPlaceOrder(OrderRequest order, decimal portfolioValue);
    decimal CalculatePositionSize(string symbol, decimal confidence, decimal portfolioValue);
    bool IsStopLossTriggered(string symbol, decimal currentPrice, decimal entryPrice, string si
}

public interface IPortfolioManager
{
    Task<Portfolio> GetCurrentPortfolioAsync();
    Task SavePortfolioSnapshotAsync();
    Task<decimal> CalculatePnLAsync();
}

public interface IMLTradingModel
{
    Task TrainModelAsync(List<TrainingData> trainingData);
    TradingPrediction PredictAction(CryptoFeatures features);
    Task SaveModelAsync(string filePath);
    Task LoadModelAsync(string filePath);
}
```

Detailed Workflow

Program Startup Sequence

1. **Main Entry Point**

csharp

```csharp
public static async Task Main(string[] args)
{
    var host = CreateHostBuilder(args).Build();

    // Ensure database exists
    using (var scope = host.Services.CreateScope())
    {
        var context = scope.ServiceProvider.GetRequiredService<TradingContext>();
        await context.Database.EnsureCreatedAsync();
    }

    await host.RunAsync();
}
```

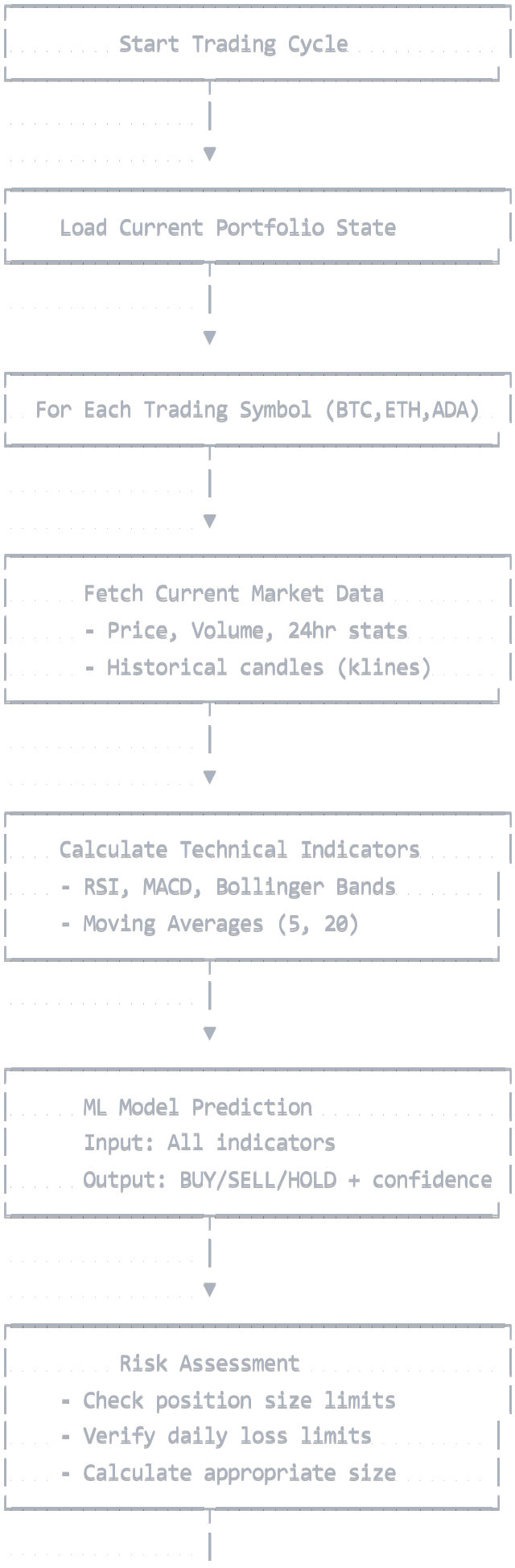
2. **Dependency Injection Setup**

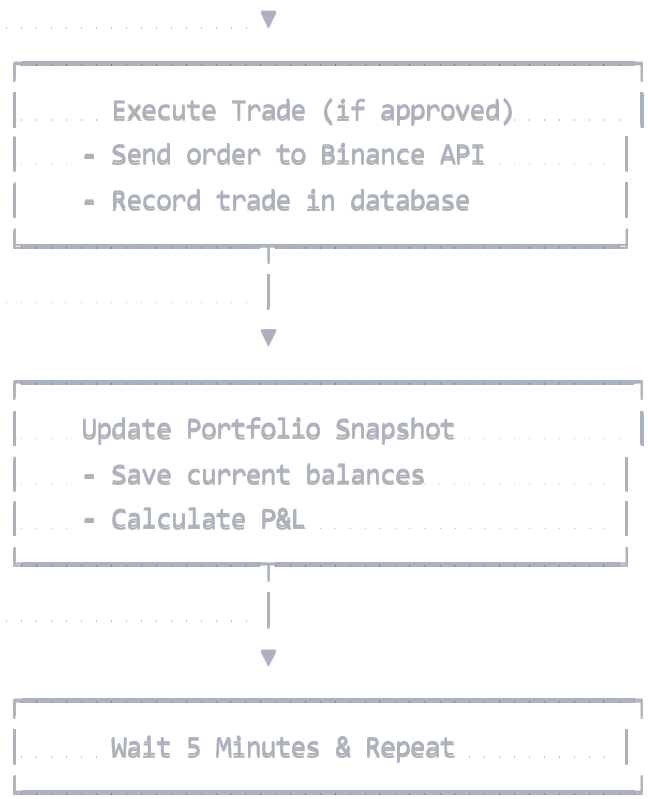
csharp

```csharp
.ConfigureServices((context, services) =>
{
    // Configuration
    services.AddSingleton(tradingConfig);

    // Database
    services.AddDbContext<TradingContext>(options =>
        options.UseNpgsql(connectionString));

    // Services
    services.AddScoped<ICryptoTraderService, CryptoTraderService>();
    services.AddScoped<IRiskManager, RiskManager>();
    services.AddScoped<IPortfolioManager, PortfolioManager>();
    services.AddSingleton<IMLTradingModel, MLTradingModel>();

    // Background service
    services.AddHostedService<TradingBotService>();
});
```

Trading Cycle Flow

The main trading loop executes every 5 minutes:


Start Trading Cycle
Load Current Portfolio State
For Each Trading Symbol (BTC,ETH,ADA)
Fetch Current Market Data
- Price, Volume, 24hr stats
- Historical candles (klines)
Calculate Technical Indicators
- RSI, MACD, Bollinger Bands
- Moving Averages (5, 20)
ML Model Prediction
Input: All indicators
Output: BUY/SELL/HOLD + confidence
Risk Assessment
- Check position size limits
- Verify daily loss limits
- Calculate appropriate size



Service Implementation

CryptoTraderService - API Integration

This service handles all communication with the Binance API:

csharp

```csharp
public class CryptoTraderService : ICryptoTraderService
{
    private static readonly HttpClient _httpClient = new HttpClient();
    private readonly TradingConfiguration _config;
    private readonly IMLTradingModel _mlModel;
    private readonly IMarketDataRepository _marketDataRepository;
    private readonly ILogger<CryptoTraderService> _logger;
    private readonly IAsyncPolicy _retryPolicy;

    // Constructor sets up HTTP client with API key
    public CryptoTraderService(...)
    {
        _httpClient.BaseAddress = new Uri(_config.BaseApiUrl);
        _httpClient.DefaultRequestHeaders.Add("X-MBX-APIKEY", _config.ApiKey);

        // Retry policy with exponential backoff
        _retryPolicy = Policy
            .Handle<HttpRequestException>()
            .WaitAndRetryAsync(
                retryCount: 3,
                sleepDurationProvider: retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAt
            );
    }
}
```

Key Methods:

1. **GetPriceAsync** - Fetches current price
2. **Get24hrTickerAsync** - Gets 24-hour statistics
3. **GetKlineDataAsync** - Retrieves historical candlestick data
4. **PlaceOrderAsync** - Executes buy/sell orders
5. **GetMLPredictionAsync** - Combines data fetching with ML prediction

RiskManager - Capital Protection

The risk manager ensures trades don't exceed safe limits:

csharp

```csharp
public class RiskManager : IRiskManager
{
    public bool CanPlaceOrder(OrderRequest order, decimal portfolioValue)
    {
        var positionValue = order.Quantity * (order.Price ?? 0);
        var positionSizePercent = positionValue / portfolioValue;

        if (positionSizePercent > _config.MaxPositionSize)
        {
            _logger.LogWarning("Order rejected: Position size {Size}% exceeds maximum {Max}%",
                positionSizePercent * 100, _config.MaxPositionSize * 100);
            return false;
        }

        return true;
    }

    public decimal CalculatePositionSize(string symbol, decimal confidence, decimal portfolioVa
    {
        // Kelly Criterion-based position sizing
        var baseSize = _config.MaxPositionSize * portfolioValue;
        var adjustedSize = baseSize * confidence;

        return Math.Min(adjustedSize, _config.MaxPositionSize * portfolioValue);
    }
}
```

Technical Analysis Module

Calculates all technical indicators:

csharp

```csharp
public static class TechnicalAnalysis
{
    public static float CalculateRSI(List<decimal> prices, int period = 14)
    {
        // Relative Strength Index calculation
        // RSI = 100 - (100 / (1 + RS))
        // Where RS = Average Gain / Average Loss
    }

    public static (float upper, float lower) CalculateBollingerBands(
        List<decimal> prices, int period = 20, float multiplier = 2f)
    {
        // Bollinger Bands = SMA ± (Standard Deviation × Multiplier)
    }

    public static (float macd, float signal) CalculateMACD(
        List<decimal> prices, int fastPeriod = 12, int slowPeriod = 26, int signalPeriod = 9)
    {
        // MACD = 12-day EMA - 26-day EMA
        // Signal = 9-day EMA of MACD
    }
}
```

Database Operations

Entity Framework Context

csharp

```csharp
public class TradingContext : DbContext
{
    public DbSet<MarketData> MarketData { get; set; }
    public DbSet<TradeRecord> TradeRecords { get; set; }
    public DbSet<ModelPerformance> ModelPerformances { get; set; }
    public DbSet<PortfolioSnapshot> PortfolioSnapshots { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // Composite index for efficient queries
        modelBuilder.Entity<MarketData>()
            .HasIndex(m => new { m.Symbol, m.Timestamp })
            .IsUnique();

        // Decimal precision for financial data
        modelBuilder.Entity<MarketData>()
            .Property(m => m.Close)
            .HasPrecision(18, 8);
    }
}
```

Common Database Operations

csharp

```csharp
// Save market data
await _context.MarketData.Add(marketData);
await _context.SaveChangesAsync();

// Query recent trades
var trades = await _context.TradeRecords
    .Where(t => t.ExecutedAt >= DateTime.UtcNow.AddDays(-30))
    .OrderBy(t => t.ExecutedAt)
    .ToListAsync();

// Calculate P&L
var totalPnL = trades.Sum(t => t.PnL);
```

Machine Learning Integration

ML Model Training

The system uses ML.NET for multiclass classification:

```csharp
public async Task TrainModelAsync(List<TrainingData> trainingData)
{
    var dataView = _mlContext.Data.LoadFromEnumerable(trainingData);

    var pipeline = _mlContext.Transforms.Conversion
        .MapValueToKey("LabelKey", "Label")
        .Append(_mlContext.Transforms.Concatenate("Features",
            nameof(CryptoFeatures.Price),
            nameof(CryptoFeatures.Volume),
            nameof(CryptoFeatures.RSI),
            // ... all other features
        ))
        .Append(_mlContext.MulticlassClassification.Trainers.SdcaMaximumEntropy())
        .Append(_mlContext.Transforms.Conversion.MapKeyToValue("PredictedLabel"));

    _model = pipeline.Fit(dataView);
}
```

Feature Engineering

The model uses these features:

- Current price and volume
- Price change (24h)
- RSI (Relative Strength Index)
- Moving averages (5-day, 20-day)
- Bollinger Bands (upper/lower)
- MACD and Signal line
- Volume ratio

Prediction Process

csharp

```csharp
public TradingPrediction PredictAction(CryptoFeatures features)
{
    var predictionEngine = _mlContext.Model
        .CreatePredictionEngine<TrainingData, TradingPrediction>(_model);

    return predictionEngine.Predict(trainingData);
    // Returns: { PredictedAction: "BUY", Confidence: [0.85, 0.10, 0.05] }
}
```

Risk Management

Position Sizing Strategy

The bot uses the Kelly Criterion for optimal position sizing:

```
Position Size = (Confidence × Base Size) × Portfolio Value

Where:
- Base Size = 2% (maximum per trade)
- Confidence = ML model confidence (0-1)
- Portfolio Value = Total account value in USDT
```

Stop Loss Implementation

csharp

```csharp
public bool IsStopLossTriggered(string symbol, decimal currentPrice, decimal entryPrice, string
{
    if (side.ToUpper() == "BUY")
    {
        var lossPercent = (entryPrice - currentPrice) / entryPrice;
        return lossPercent >= _config.StopLossPercent; // Default: 5%
    }
    else if (side.ToUpper() == "SELL")
    {
        var lossPercent = (currentPrice - entryPrice) / entryPrice;
        return lossPercent >= _config.StopLossPercent;
    }
    return false;
}
```

Daily Loss Limits

The system tracks daily P&L and stops trading if losses exceed 10% (configurable).

Error Handling & Resilience

Retry Policy with Polly

```csharp
_retryPolicy = Policy
    .Handle<HttpRequestException>()
    .WaitAndRetryAsync(
        retryCount: 3,
        sleepDurationProvider: retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)),
        onRetry: (outcome, timespan, retryCount, context) =>
        {
            _logger.LogWarning("Retry {RetryCount} after {Delay}ms", retryCount, timespan.Total
        });
```

Retry delays: 2s → 4s → 8s (exponential backoff)

Error Recovery in Trading Loop

```csharp
while (!stoppingToken.IsCancellationRequested)
{
    try
    {
        await RunTradingCycleAsync();
        await Task.Delay(TimeSpan.FromMinutes(5), stoppingToken);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error in trading cycle");
        await Task.Delay(TimeSpan.FromMinutes(1), stoppingToken); // Shorter retry on error
    }
}
```

Deployment Guide

Prerequisites

1. .NET 8.0 Runtime

bash

```bash
# Ubuntu/Debian
wget https://packages.microsoft.com/config/ubuntu/22.04/packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
sudo apt update
sudo apt install dotnet-runtime-8.0
```

2. PostgreSQL Database

bash

```bash
# Install PostgreSQL
sudo apt install postgresql postgresql-contrib

# Create database
sudo -u postgres psql
CREATE DATABASE tradenectics;
CREATE USER tradenectics WITH PASSWORD 'your_password';
GRANT ALL PRIVILEGES ON DATABASE tradenectics TO tradenectics;
```

3. Environment Variables

bash

```bash
# Create .env file
echo "BINANCE_API_KEY=your_api_key" >> .env
echo "BINANCE_API_SECRET=your_api_secret" >> .env
```

Configuration Files

appsettings.json

json

```json
{
  "ConnectionStrings": {
    "DefaultConnection": "Host=localhost;Database=tradenectics;Username=tradenectics;Password=y
  },
  "Tradi
```