User Manual & Design Philosophy
End-to-End Advanced ML Trading Framework V130

Chapter 1: Introduction & Core Philosophy

1.1 The Problem: Why Most Trading Strategies Fail

The financial markets are not a static puzzle to be solved once. They are a living, adaptive system that constantly shifts between different states or "regimes"—from strong trends to volatile sideways chop to quiet, range-bound periods. A strategy meticulously optimized on data from 2021 is almost guaranteed to fail in the market conditions of 2024. This is because a static algorithm with fixed rules is brittle; it will inevitably break when the market conditions it was designed for disappear. This framework is engineered from the ground up to solve this fundamental problem of **adaptation**.

1.2 The Solution: An Evolving, AI-Driven Research Partner

This script is not a simple backtester. It is a research platform for discovering and evolving adaptive trading systems. Its core philosophy is that a successful trading system must not be a static artifact, but a dynamic, learning entity capable of evolving its own understanding and approach as the market itself evolves.

It achieves this through a multi-layered approach:

- **Walk-Forward Optimization:** It constantly learns from recent market data, ensuring it is never "stale."
- **AI-Driven Strategy:** It uses a generative AI (Google's Gemini) to make high-level strategic decisions, acting as a virtual quant researcher.
- **Evolutionary Mechanics:** It is capable of inventing novel "hybrid" strategies by combining the best elements of its own successful history.

This manual will guide you through the setup, the framework's architecture, its evolutionary journey, and provide practical instructions for adapting it to your own research.

Chapter 2: Getting Started: Setup and Requirements

Before you can run the framework, your environment must be configured correctly. This is a critical first step.

2.1 Python Environment & Required Libraries

The framework is written for **Python 3.10 or newer**. You must install several key libraries that provide its core functionality.
Action: Open your terminal or command prompt and run the following command to install all necessary packages:
(you may need to use python 3.10)

pip install pandas numpy matplotlib shap xgboost optuna requests scikit-learn pydantic python-dotenv

2.2 Gemini API Key Configuration

The framework's intelligence relies on Google's Gemini API. You must provide a valid API key for it to function.

1. **Create the .env file:** In the same directory where you saved the Python script, create a new file named exactly .env (the initial dot is important as it's often a convention for environment files).
2. **Add Your Key:** Open the .env file with a text editor and add the following line, replacing "YOUR_API_KEY_HERE" with your actual, personal API key: GEMINI_API_KEY="YOUR_API_KEY_HERE"

Why it's designed this way: Using an environment file is a crucial security best practice that keeps your secret keys separate from the main script code. The python-dotenv library automatically finds this file and loads your key on startup. If the script cannot find the key, it will fall back and prompt you to paste it directly into the console.

2.3 Data File Structure and Configuration

The framework is engineered for multi-symbol, multi-timeframe analysis. This is a deliberate design choice to provide the model with essential market context.
The Default 12 Files: In the run_single_instance function, you will see a list named files_to_process which contains 12 default CSV files:

```
files_to_process=[
    "AUDUSD_Daily_202001060000_202506020000.csv", "AUDUSD_H1_202001060000_202506021800.csv", "AUDUSD_M15_202105170000_202506021830.csv",
    "EURUSD_Daily_202001060000_202506020000.csv", "EURUSD_H1_202001060000_202506021800.csv", "EURUSD_M15_202106020100_202506021830.csv",
    # ...and so on for GBPUSD and USDCAD
]
```

Why this structure? This is the heart of the contextual analysis. A model making a trade decision on the M15 (15-minute) timeframe for EURUSD is significantly more effective if it also knows the underlying trend on the H1 (hourly) and D1 (daily) charts. Taking a short-term "buy" signal is much riskier if the dominant daily trend is strongly bearish. The FeatureEngineer class is specifically designed to create contextual features (e.g., D1_ctx_Trend, H1_ctx_SMA) that feed this higher-timeframe information into the M15 model, making its decisions far more robust.

Data Requirements:

- **Location:** Place all your CSV data files in the **same directory** as the main Python script.
- **Format:** Files must contain OHLCV (Open, High, Low, Close, Volume) data.
- **Naming Convention:** The script parses the filename to identify the asset and timeframe. Your files **must** follow the SYMBOL_TIMEFRAME_*.csv format (e.g., XAUUSD_H1_2020_2024.csv).

Chapter 3: The Journey of Development - How We Got Here

The framework's current feature set is the result of solving a series of problems encountered during its development. Understanding this journey explains the purpose of each major component.

- **The Beginning (V90): The Flawed Backtester**
  - **Problem:** The initial script was a simple backtester with a single train/test split. Its results were unrealistic and prone to "curve-fitting."
  - **Solution:** The entire architecture was rebuilt around **Walk-Forward Analysis**. This method, implemented in the main for loop of run_single_instance, constantly slides a training window through time, ensuring the model is always tested on new, unseen data, which is a much more rigorous and realistic approach.
- **The Playbook & The Daemon (V112): Expanding the Horizon**

    - **Problem:** The framework could only test one hardcoded strategy. To research different ideas (e.g., trend-following vs. mean-reversion), a user had to manually edit the code.
    - **Solution:** The **Strategy Playbook** was born, externalizing strategy definitions into what is now the strategy_playbook.json file. Simultaneously, **Daemon Mode** was introduced (controlled by MAX_RUNS and CONTINUOUS_RUN_HOURS), allowing the framework to run autonomously for hours or days, generating a rich history of performance data.
- **Hybrid Synthesis (V119): The Creative Spark**
    - **Problem:** The AI could only pick from the predefined playbook. What if the best strategy was actually a combination of two existing ones?
    - **Solution: Hybrid Strategy Synthesis** was implemented. The check_and_create_hybrid function now prompts the AI to analyze its own history, identify the best-performing strategies, and invent a new hybrid by combining their features. The strategy_playbook.json file was moved to an external file so the AI could permanently add its new creations to the list of available options.
- **Strategic Review & Stagnation (V121): The Reality Check**
    - **Problem:** The framework could get "stuck," repeatedly trying a failing strategy just because it had been a champion in the distant past. It lacked long-term memory to recognize when a strategy was chronically failing.
    - **Solution:** The **Strategic Review** phase was created. Before each new run in daemon mode, the perform_strategic_review function now analyzes the entire historical_runs.jsonl file to calculate long-term health metrics like ChronicFailureRate and StagnationWarning. The AI is now explicitly instructed to penalize strategies with poor long-term health, forcing it to pivot.
- **Robust Memory (V124): The Fortress**
    - **Problem:** With long-running tests, a single crash during a file write could corrupt the entire JSON history file, wiping out days of research.
    - **Solution:** The memory system was re-architected. The single memory file was split into champion.json and historical_runs.jsonl. The .jsonl format saves each run on its own line, ensuring that a crash can only affect the last, incomplete entry. The load_memory function was taught to safely skip any malformed lines.
- **Readability (V130): The Finishing Touch**
    - **Problem:** The reports were cluttered with long, technical script version names, making them hard to compare at a glance.
    - **Solution:** The **Nickname System** was introduced. Each script version is now given a unique, AI-generated codename (e.g., 'Apollo', 'Vortex') which is stored in nickname_ledger.json. Reports now use these clean nicknames for display while providing a mapping key for full traceability.

Chapter 4: A Deep Dive into the V130 Architecture

This section connects the concepts from the development journey to the specific code implementation.

4.1 The AI's Command Center: GeminiAnalyzer

This class is the brain of the operation, containing several key methods:

- get_pre_flight_config(): This is the highest level of AI decision-making. It dynamically builds a prompt for the AI that includes the current champion, the strategic health report, and historical performance, asking it to select the single best strategy and initial parameters for the upcoming run.
- analyze_cycle_and_suggest_changes(): This is the tactical AI. After each training cycle within a run, it analyzes the most recent performance and suggests minor parameter tweaks for the next cycle.

- create_hybrid_strategy(): This is the creative AI. It is given the performance summaries of two successful but different strategies and prompted to synthesize a new one by blending their features.
- generate_nickname(): A small but useful utility to generate cool-sounding, unique codenames for reporting.

4.2 The Data Pipeline: DataLoader & FeatureEngineer

- DataLoader: This class is responsible for reading the 12 default CSV files, parsing their names for symbol/timeframe info, and merging them into three distinct DataFrames: D1, H1, and M15.
- FeatureEngineer: This is where the raw data is transformed into meaningful signals for the model. Its create_feature_stack method performs the crucial pd.merge_asof operation, which projects the daily and hourly features onto the 15-minute data. It also calculates dozens of technical indicators like RSI, ADX, Bollinger Bands, and candlestick patterns.

4.3 The Learning Engine: ModelTrainer

- This class orchestrates the entire machine learning process for a single cycle. Its train method first calls _optimize_hyperparameters to use the **Optuna** library, which intelligently tests dozens of model configurations to find the best one. It then trains the final model on all available data for that cycle and uses the **SHAP** library in _generate_shap_summary to determine which features were most important.

4.4 The Execution Engine: Backtester

- The run_backtest_chunk method takes the trained model and applies it to the out-of-sample data. It simulates trade execution, position management, and risk.
- **Circuit Breaker:** This method contains a critical safety feature: the MAX_DD_PER_CYCLE check. If the equity in the current cycle drops below a specified percentage from its peak, the backtester "trips a circuit breaker," closes all positions, and stops trading for the rest of that cycle. This simulates a risk management rule and provides valuable data to the AI about the strategy's failure modes.

Chapter 5: User's Guide to Customization & Operation

This framework is designed for your experimentation. Here’s how to tailor it to your needs.

5.1 Controlling the Run: Single vs. Daemon Mode

These are the primary controls at the top of the main() function:

```
# --- Daemon Mode Controls ---
CONTINUOUS_RUN_HOURS = 0
MAX_RUNS = 1 # Set > 1 to enable Daemon mode
# --- End Controls ---
```

- **For a single, one-off test:** Leave these settings as they are. The script will run once and exit.
- **To run for a set number of iterations:** Set MAX_RUNS = 5. The script will perform 5 complete end-to-end runs and then stop.
- **To run for a set amount of time:** Set CONTINUOUS_RUN_HOURS = 8. The script will run continuously for 8 hours.

5.2 Adjusting Core Backtesting Parameters

All primary settings are located in the fallback_config dictionary inside the main() function.

Example: To make the backtest run faster with less precision for a quick look, you could reduce the number of Optuna trials and use a shorter training window.

```
fallback_config={
    # ...
    "OPTUNA_TRIALS": 15,          // Reduced from 30 for faster, less thorough optimization
    "TRAINING_WINDOW": '180D',      // Reduced from 365D for faster training cycles
    "RETRAINING_FREQUENCY": '45D', // Retrain more often on a shorter test window
    # ...
}
```

5.3 Adding Your Own Strategy to the Playbook

You can add your own human-designed strategies for the AI to consider.

1. In your project folder, find and open the /Results/strategy_playbook.json file with a text editor.
2. Add a new top-level entry. The key is your new strategy's name. The value must contain a description and a list of features. The features you list must be valid column names produced by the FeatureEngineer class.

Example Entry:

```
"MyVolatilityStrategy": {
  "description": "A custom strategy that looks for breakouts when volatility expands.",
  "features": [
    "ATR",
    "bollinger_bandwidth",
    "hour",
    "day_of_week"
  ],
  "lookahead_range": [30, 90],
  "dd_range": [0.20, 0.35]
}
```

Save the file. On the next run, the AI will see "MyVolatilityStrategy" and can select it if it deems it appropriate.

Chapter 6: Understanding the Output Artifacts

The /Results directory is the framework's memory. Here is what each key file represents:

- champion.json: A snapshot of the single best run ever recorded, measured by Calmar Ratio. This is the benchmark that all new runs are compared against.

- historical_runs.jsonl: **The most important memory file.** Every completed run is recorded here, line by line. This file provides the raw data for the Strategic Review and Hybrid Synthesis, allowing the framework to learn from its entire operational history.
- strategy_playbook.json: The living list of all available strategies, both default and AI-generated. You can open this to see the hybrids the AI has invented.
- nickname_ledger.json: The simple dictionary mapping long script versions to their clean, readable codenames for reporting.
- /Results/[Codename]/ Folders: Each run gets its own dedicated folder named with its unique codename. Inside, you'll find the detailed text report, the equity curve image (...equity_curve.png), and the feature importance chart (...shap_summary.png).