ChatGPT

Análisis del Proyecto *env.model_0.7*

Estructura de archivos y componentes

El proyecto presenta varias carpetas principales, cada una con responsabilidades específicas. En la raíz (`env.model_0.7/`) hay archivos de configuración (por ejemplo, `logging.json`), scripts de Streamlit (como `streamlit_app.py`, `streamlit_train.py`, etc.), y versiones de modelos (`streamlit_model.pt`). También existen directorios como **data_manager**, **influx_service**, **tabs** y **utils**. El directorio **data_manager/** contiene subcarpetas *config*, *data_manager*, *data_sources*, *examples*, *exports* y *scripts*. Aquí hay archivos para configurar bases de datos (`schema.sql`), conectores de datos (por ejemplo, `ccxt_fetcher.py`, `influx_writer.py`, `kafka_consumer_sqlite.py`, `websocket_to_kafka.py`), ejemplos de productores/consumidores Kafka, y scripts auxiliares (p.ej. `verify_coherence.py`). El directorio **influx_service/** incluye scripts para interacción con InfluxDB (`consumer.py`, `producer.py`, etc.) y un `docker-compose.yml`. La carpeta **tabs/** agrupa módulos de interfaz (Backtest, Data, Evolve, Trading, Train, Utils). En **utils/** se agrupan utilidades generales: *binance_ws_ingest.py* (ingesta desde Binance), *data_cleaning.py*, *db_storage.py*, *io_utils.py*, *scaler_utils.py*, entre otros. A nivel global destacan además `backtest_tab.py` en la raíz (duplicado con el de tabs), `fetched_ohlcv.json` (datos iniciales de mercado), y scripts de entrenamiento como `fiboevo.py`.

Clases y funciones principales

Sin el código completo es difícil listar todas las clases, pero de los nombres se infiere lo siguiente: **DataManager** (en `data_manager/manager.py`) probablemente orquesta la conexión a la base de datos y gestiona lecturas/escrituras de mercado. Los módulos en *data_sources* definen clases o funciones para obtener datos: *CCXTFetcher* obtiene datos históricos vía la librería CCXT; *InfluxWriter/InfluxToSQL* manejan la persistencia en InfluxDB y su sincronización con SQL; *KafkaConsumerSQLite* y *WebsocketToKafka* consumen y producen datos en un sistema de mensajería Kafka. En **influx_service** habrá clases para leer/escribir a InfluxDB (producer/consumer). En la interfaz (carpeta *tabs*), cada archivo define una pestaña de la aplicación: es probable que contengan clases o funciones como `BacktestTab`, `DataTab`, `EvolveTab`, `TradingTab`, `TrainTab`, y `UtilsTab` que construyen componentes de UI con Streamlit. Asimismo, la raíz incluye scripts `streamlit_train.py`, `streamlit_predict.py`, etc., con flujos de entrenamiento y predicción, y `streamlit_app.py` que integra la aplicación. Las utilidades en **utils/** brindan funciones auxiliares: por ejemplo, `scaler_utils.py` seguramente contiene escaladores de datos, `data_cleaning.py` funciones para limpiar datasets, y `db_storage.py` métodos para guardar datos en base de datos.

Integraciones y partes pendientes

Varios componentes parecen aislados y faltan conexiones claras. Por ejemplo, existen scripts de ingesta de datos (CCXT, websockets) pero no hay un flujo maestro que ejecute periódicamente `WebsocketToKafka` → `KafkaConsumerSQLite` → `DataManager` → `Backtest/Trading`. Tampoco se evidencia un puente automático entre la interfaz gráfica (Streamlit) y los módulos de procesamiento: los archivos en *tabs/* podrían no estar todos enlazados en `streamlit_app.py`. Hay archivos duplicados (ej. `backtest_tab.py` en raíz y en `tabs/`) que sugieren refactor pendientes. El archivo `streamlit_model.py` y el modelo serializado `streamlit_model.pt` parecen inconclusos

(tal vez para inferencia), al igual que `fiboevo.py` y su copia, indicando un módulo de evolución de estrategias todavía en desarrollo. No hay evidencia de scripts que automaticen la normalización previa al backtest ni de validación de datos tras la ingesta; el script `verify_coherence.py` insinúa esa intención pero queda por integrar. En resumen, faltan orquestadores o pipelines que unan cada etapa de obtención, almacenamiento, preprocesamiento, entrenamiento y backtesting de datos.

Problemas de sintaxis y coherencia

Se han detectado varias inconsistencias en nomenclatura y archivos redundantes que pueden inducir errores en Python. Por ejemplo, el fichero `fiboevo - copia.py` contiene un espacio en el nombre; esto impide importarlo directamente y sugiere que es una copia temporal mal ubicada. La coexistencia de `utils.py` en la raíz y la carpeta `utils/` con otro `__init__.py` puede generar conflictos de paquetes y confusiones de importación. La presencia de `trading_tab.py.bak` indica un respaldo antiguo que debería eliminarse o integrarse. Varios scripts (`streamlit_*`) podrían solaparse funcionalmente y necesitar revisión para evitar redefiniciones (p.ej. `streamlit_app.py` vs. pestañas en *tabs*). No se aprecia un manejo consistente de excepciones o validación de entradas en los módulos de datos (p. ej. falta comprobación de valores nulos en los fetchers). El código de limpieza de datos (`data_cleaning.py`, `scaler_utils.py`) debe revisarse contra casos límite (NaNs, outliers). En general, hay que revisar indentaciones, nombres duplicados y posibles errores de encoding (por ejemplo, problemas con UTF-8 en archivos JSON o en nombres de archivo con espacios). A nivel de formato de datos, es clave manejar los NaN/NA apropiadamente: por ejemplo, Pandas provee `dropna()` para eliminar filas con datos faltantes y `fillna()` para rellenarlos [1] [2]; estas herramientas deberían emplearse de manera consistente en el proceso de limpieza de *utils*.

Manejo y normalización de datos

El proyecto abarca múltiples fuentes de datos: cotizaciones en tiempo real (via WebSocket de Binance), datos históricos (CCXT), flujos de mercado vía Kafka e InfluxDB. En principio, cada fuente debe integrar sus datos en un repositorio central (p.ej. SQLite o CSV). En los archivos disponibles se ve la intención de guardar datos en Influx (ideal para series de tiempo) y luego transferirlos a SQLite. No obstante, no hay un script único que llame a estos conversores en secuencia automática. Además, falta normalizar y limpiar esos datos antes del análisis: todo dataset debe pasarse por un preprocesamiento. Se supone que `scaler_utils.py` contiene funciones (posiblemente de scikit-learn) para escalar características numéricas. El escalado *min-max*, por ejemplo, ajusta los valores a un rango [0,1] [3], lo cual es buena práctica antes de alimentar un modelo de machine learning. Sin embargo, hay que verificar que dicho escalado se aplique consistentemente tanto en entrenamiento como en backtest (reaplicando el mismo transformador). El archivo `data_cleaning.py` debe implementar la eliminación o imputación de valores faltantes. Tal como señala la documentación de Pandas, usar `dropna()` puede eliminar filas con datos nulos [1], mientras que `fillna()` permite rellenarlos con valores predeterminados o estadísticas (media, mediana) [2]. Es crítico decidir estas estrategias de limpieza para evitar sesgos o pérdidas de datos útiles. Finalmente, los datos deben almacenarse de forma persistente: además del CSV de exportación (`ohlcv_2025-09-23.csv`), probablemente se use una base SQLite (`marketdata.db.bak`). Hay que comprobar la coherencia entre estos repositorios (p.ej. que el script *verify_coherence* valide que los datos de la base y los CSV coincidan).

Persistencia de configuraciones en la interfaz

La aplicación cuenta con múltiples parámetros ajustables por el usuario (símbolo a operar, rangos de fechas, hiperparámetros del agente, etc.). Es fundamental que estas configuraciones no se pierdan al

refrescar la página. Streamlit ofrece `st.session_state` para retener valores durante la sesión, pero al recargar la pestaña el estado se restablece [4]. En consecuencia, si no se implementa un mecanismo de guardado externo (por ejemplo, escribir las opciones en un archivo JSON al cerrar), cualquier ajuste del usuario se perderá. Hay archivos JSON de configuración (como `influx.json` en *data_manager*) pero no está claro si se vinculan con la UI. Se debería asegurar que, cuando el usuario modifique configuraciones en los formularios (mediante widgets con `key`), estas se guarden automáticamente (usando callbacks o persistiendo en un archivo de settings). En ausencia de esto, la interfaz podría olvidar parámetros entre sesiones, afectando la experiencia de uso. Además, los defaults deben cargarse desde los archivos de configuración, garantizando sincronía entre la UI y las fuentes de datos.

Logros actuales y estado del proyecto

Hasta el momento, el código incluye la estructura básica de un sistema integrado de trading algorítmico: - **Colección de datos:** Hay scripts para obtener datos de mercado en vivo (por WebSockets y Kafka) y de forma on-demand (CCXT). Se dispone de un esquema de base de datos (en `data_manager/schema.sql`) y archivos de ejemplo (`marketdata.db.bak`, CSV con OHLCV, JSONs con estadísticas procesadas) que indican que parte del pipeline de datos ha sido implementado.
- **Procesamiento de datos:** Existen utilidades para limpieza (`data_cleaning.py`) y escalado de características (`scaler_utils.py`), así como un gestor de datos genérico (`DataManager`) pensado para unificar las operaciones de lectura/escritura en la base de datos. Se han añadido scripts de ejemplo de productor/consumidor Kafka y de migración entre InfluxDB y SQLite, lo cual demuestra la intención de compatibilidad con sistemas de alto rendimiento.
- **Interfaz de usuario:** Se creó una app con Streamlit (p. ej. `streamlit_app.py`) que organiza la aplicación en pestañas (Backtest, Data, Evolve, Trading, Training, Utils). Cada tab tiene su propio módulo, lo que facilita la escalabilidad de la UI. También se prepararon flujos de entrenamiento (`streamlit_train.py`) y predicción (`streamlit_predict.py`), implicando que ya se puede entrenar o cargar modelos de reinforcement learning.
- **Modelado y estrategia:** El proyecto muestra archivos relacionados con evolución de estrategias (`fiboevo.py`) y entrenamiento de modelos (probablemente Stable Baselines u otro, dado el contexto). También existe un modelo preentrenado (`streamlit_model.pt`). Esto indica que se ha avanzado en la parte algorítmica del trading.

En conjunto, se ha esbozado un framework completo: adquisición de datos, almacenamiento, preprocesamiento, modelado, backtesting y una interfaz GUI. Sin embargo, muchas piezas aún no están interconectadas ni finalizadas.

Tareas pendientes

- **Críticas:**
- *Conexión de pipeline completo:* Implementar un controlador o scheduler que ejecute automáticamente la cadena de ingesta → almacenamiento → preprocesamiento → backtest/entrenamiento. Esto incluye enlazar los módulos de *data_sources* con la base de datos y hacer que la interfaz dispare el backtest con los datos actuales.
- *Persistencia de configuraciones:* Añadir lógica para guardar en disco (JSON u otro) las opciones seleccionadas en la UI, de modo que sobrevivan a recargas. Esto puede implicar usar callbacks de Streamlit o librerías de settings (por ejemplo, [30]).
- *Corrección de duplicados/nombres conflictivos:* Renombrar/eliminar archivos problemáticos (p.ej. quitar el espacio en `fiboevo - copia.py`, eliminar respaldos `.bak`, consolidar `utils.py`). Asegurarse de que los paquetes se importan correctamente sin ambigüedades.

- *Validación de datos:* Completar el script `verify_coherence.py` para comprobar la integridad de los datos (que no falten cotizaciones, formatos correctos, etc.), y usarlo después de cada ingesta.

- **Importantes:**

- *Limpieza y normalización robustas:* Mejorar `data_cleaning.py` para manejar casos especiales (NaN, outliers, duplicados). Establecer normas claras de escalado (p.ej., aplicar siempre MinMaxScaler a [0,1] [3]) y documentarlas.
- *Integración de alertas/logging:* Configurar el `logging.json` y asegurar que cada módulo registra eventos relevantes (errores de red, fallos de base de datos, etc.).
- *Refactorización de la UI:* Revisar que cada pestaña en *tabs/* esté correctamente importada y utilizada en la app de Streamlit, evitando código muerto. Garantizar uniformidad visual y experiencia de usuario.

- *Enlazar modelos de RL:* Asegurar que los algoritmos de entrenamiento funcionen hasta el final y sus pesos puedan exportarse/importarse sin error (por ejemplo, validando que `streamlit_model.pt` es utilizable en `streamlit_predict.py`).

- **Menores:**

- *Limpieza de código y documentación:* Escribir docstrings y comentarios aclaratorios en funciones clave (por ejemplo, describir qué hace cada tab). Eliminar imports no usados y archivos de ejemplo que ya no sirvan.
- *Actualizar requisitos:* Verificar que `requirements.txt` enumera todas las librerías necesarias (CCXT, kafka-python, pandas, etc.) y sus versiones compatibles.
- *UI Aesthetic y UX:* Detallar etiquetas y ayuda en la interfaz para que el usuario comprenda qué parámetros modificar.
- *Pruebas unitarias:* Agregar tests básicos para funciones de procesamiento de datos y módulos de gestión (garantizar que futuras modificaciones no rompan funcionalidades).

Con estos pasos, el proyecto pasará de un prototipo incompleto a una plataforma de trading algorítmico más sólida y coherente, con flujos de datos bien definidos y una interfaz fiable.

Referencias: Se han utilizado criterios estándar de procesamiento de datos (por ejemplo, el escalado de características con `MinMaxScaler` [3] y métodos de limpieza de Pandas como `dropna()` / `fillna()` [1] [2]). También se revisaron las notas de Streamlit sobre gestión de estado para subsanar problemas de persistencia [4] . Estos lineamientos guían las recomendaciones hechas.

---

[1] [2] Limpieza de datos en Pandas: Explicado con ejemplos
https://www.freecodecamp.org/espanol/news/limpieza-de-datos-en-pandas-explicado-con-ejemplos/

[3] MinMaxScaler | Interactive Chaos
https://interactivechaos.com/es/manual/tutorial-de-machine-learning/minmaxscaler

[4] Session State - Streamlit Docs
https://docs.streamlit.io/develop/api-reference/caching-and-state/st.session_state