ChatGPT

Executive summary (≤ 200 слов)

TensorTrade – это открытый Python-фреймворк для создания и обучения алгоритмов алгоритмической торговли с глубоким RL. Проект сосредоточен на модульности и расширяемости, позволяя комбинировать самостоятельные компоненты – биржи, стратегии действий, схемы вознаграждения, агентов, отчёты – в настраиваемые торговые среды [1] [2]. На момент анализа используется коммит `72c3be9c15d3ac9dc7783c3ddf79b12a7e0a2fb3` (9 июня 2024) [3]. Эта версия соответствует ветке разработки “1.0.4-dev1” (TensorTrade 1.0.4-dev1), тогда как последняя стабильная версия на PyPI – **1.0.3** (релиз 10 мая 2021) [4]. TensorTrade предоставляет готовую среду, совместимую с OpenAI Gym/Gymnasium, но с более сложной внутренней механикой: встроенная **OMS** (Order Management System) для моделирования биржевых ордеров и портфеля, потоковый **DataFeed** для управления рыночными данными, а также множество “схем” (действий, вознаграждения, остановки и др.), которые можно конфигурировать. В отличие от минималистичных Gym-окружений + Stable Baselines, TensorTrade из коробки моделирует реалистичный торговый цикл – от поступления рыночного тика до исполнения ордеров и обновления портфеля – что упрощает эксперименты с сложными стратегиями [5] [6]. Однако за богатую функциональность приходится платить усложнением архитектуры и необходимостью тщательной настройки компонентов.

2. Обзор архитектуры пакета `tensortrade`

2.1. Карта модулей

Пакет `tensortrade` организован по принципу разделения на ядро, окружения, данные и OMS (система управления ордерами):

- `tensortrade.core` – базовые классы и инфраструктура контекстов. Здесь определены абстракции *Component* и *TradingContext* для конфигурирования компонентов через единый контекст [7] [8], а также утилиты вроде *Identifiable* (уникальный ID) и *TimeIndexed* (привязка к глобальным часам) [9] [10]. Модуль `core.registry` реализует реестр компонентов.
- `tensortrade.env` – реализация торгового окружения (классы *TradingEnv* и др.) и связанных схем. Подпакеты: `env.actions`, `env.rewards`, `env.observers`, `env.informers`, `env.renderers`, `env.stoppers`, `env.plotters`. Например, в `env.actions` – схемы действий (интерпретация действий агента), в `env.rewards` – схемы вознаграждения (формулы расчёта реворда), `env.observers` – формирование наблюдений, `env.informers` – сбор дополнительной информации (например, метрик), `env.renderers/plotters` – визуализация (графики баланса, сделок), `env.stoppers` – условия досрочного прекращения эпизода. Кроме того, `env.utils` содержит вспомогательное: *FeedController*, *ObsState* и др. для связывания потока данных с окружением.
- `tensortrade.feed` – модуль потоков данных. Содержит *DataFeed* и набор *Stream*-классов (IterableStream, Placeholder и др.) для построения потоковых преобразований цен, индикаторов и прочих признаков [11] [12]. *DataFeed* компилирует граф зависимостей потоков и обеспечивает метод *next()* для получения следующего среза данных [13] [12].
- `tensortrade.oms` – подсистема управления ордерами (Order Management System). Включает подпакеты `oms.instruments` (классы *Instrument*, *TradingPair*, *Quantity* –

описание активов и объёмов), `oms.wallets` (*Wallet*, *Portfolio*, *Ledger* – кошельки на биржах и портфель в целом), `oms.exchanges` (*Exchange*, *ExchangeOptions* – биржи, реальные или симуляционные), `oms.orders` (*Order*, *Trade*, *Broker*, статусы ордеров, листенеры для отслеживания исполнения) и `oms.services` (например, *execution.simulated* – сервис исполнения ордеров на симулируемой бирже). OMS отвечает за приём заявок (ордеров) от *ActionScheme* и их исполнение: сопоставление с ценами, создание *Trade*-сделок, обновление кошельков и портфеля.

- `tensortrade.agents` – базовые реализации RL-агентов (напр. *DQNAgent*, *A2CAgent*) и инструменты обучения. Эти встроенные агенты помечены deprecated в 1.0.4-dev1 [14] – разработчики рекомендуют использовать внешние фреймворки (Stable-Baselines3, RLlib и др.), интегрируя их с *TradingEnv*.
- `tensortrade.data` – утилиты для загрузки/генерации данных. В частности, `data.cdd.CryptoDataDownload` позволяет скачать исторические OHLCV-данные крипто-рынков, `data.synthetic` / `stochastic` – генераторы синтетических ценовых рядов (например, геометрический Броуновский процесс).
- **Прочие**: `tensortrade.env.default` – может присутствовать для совместимости со старой API (например, упрощённая сборка окружения с дефолтными компонентами), но в текущей версии основные классы находятся в названных выше модулях. `tensortrade.version` – хранит версию. `tensortrade.__init__` организует удобный импорт (например, `from tensortrade import TradingEnv, Portfolio` и т.п.). Также в репозитории есть `examples/` с ноутбуками и скриптами примеров, и `tests/` с тестами.

Дерево модулей TensorTrade и их роли:

Модуль	Назначение	Ключевые классы/функции	Где использ
`tensortrade.core`	Базовые классы, контекст и реестр компонентов	`Component`, `TradingContext`, `Context`, `Identifiable`, `TimeIndexed`, `registry`	Везде (надс компонента
`tensortrade.env`	Торговое окружение RL и схемы (действий, наград и др.)	`TradingEnv`, `Clock`, `FeedController`, `ObsState`	Пользовате обучения R
↳ `env.actions`	Схемы интерпретации действий агента в ордера OMS	`AbstractActionScheme`, `DiscreteActions` (по умолчанию)	`TradingEn`
↳ `env.rewards`	Схемы расчёта вознаграждения агента	`AbstractRewardScheme`, `SimpleProfit`, `PBR`, `RiskAdjustedReturns`	`TradingEn`

Модуль	Назначение	Ключевые классы/функции	Где использ
↳ `env.observers`	Наблюдатели: формируют наблюдение (obs) из состояния среды	`AbstractObserver`, `TensorObserver` (напр., возвращает numpy-массив признаков)	`TradingEn`
↳ `env.informers`	Информеры: собирают доп. информацию для вывода (info)	`AbstractInformer`	`TradingEn`
↳ `env.renderers`	Рендеринг среды (графики, UI)	`AbstractRenderer`, конкретные plotly-рендереры	`TradingEn` вызову `.re`
↳ `env.plotters`	Дополнит. графики (через `plot()`): агрегаторы нескольких рендеров	`AggregatePlotter`	`TradingEn` (необязател
↳ `env.stoppers`	Условия останова эпизода	`AbstractStopper`, напр. `MaxLossStopper`	`TradingEn`
`tensortrade.feed`	Потоковые данные (фичи, ценовые ряды, индикаторы, метаданные)	`Stream`, `DataFeed`, `PushFeed`	Внутри `Fee` Observer
`tensortrade.oms.instruments`	Описание инструментов (валют/активов)	`Instrument` (символ + точность), `TradingPair`	В портфеля (сопоставле котируемой
`tensortrade.oms.wallets`	Кошельки на биржах и портфель	`Wallet`, `Portfolio`, `Ledger`	Внутри OMS счетах бирж
`tensortrade.oms.exchanges`	Биржи (реальные или симуляторы для бэктеста)	`Exchange`, `ExchangeOptions`	Через `Port` (каждый ко бирже)

Модуль	Назначение	Ключевые классы/функции	Где использ
`tensortrade.oms.orders`	Ордеры и их исполнение	`Order`, `Trade`, `OrderStatus`, `Broker`, `OrderListener`	В OMS: Brok списком ор исполняет
`tensortrade.oms.services`	Сервисы исполнения (стратегия исполнения ордеров)	`execution.simulated.execute_order` (дефолт для бэктеста)	`Exchange.` `execute_o`
`tensortrade.agents`	Шаблоны RL-агентов (для примеров; устар.)	`Agent` (base), `DQNAgent`, `A2CAgent`, `ParallelDQNAgent`	В примерах развиваютс

2.2. “Горячие точки” входа

Чтобы запустить обучение или бэктест, разработчик обычно проходит следующие шаги (явно или под капотом примера):

1. **Создание компонентов окружения.** Пользователь настраивает:
2. **Биржи/кошельки**: например, инициализирует *Exchange* (симулятор с ценовым потоком и комиссией) и *Wallet*-ы для базовой валюты (кэш USD) и торгуемого актива (BTC) на этой бирже. Объединяет кошельки в *Portfolio* [24] [32].
3. **Поток данных**: определяет *DataFeed* из ценовых *Stream*-ов и, опционально, индикаторов технического анализа. Все потоки группируются как минимум в группу `"features"` (признаки для обучения) и, возможно, `"meta"` (доп. метаданные). *FeedController* потом добавит группу `"portfolio"` автоматически [33] [34].
4. **Схема действий**: например, *DiscreteActions* (действие – индекс ордера типа “купить/продать/держать”). Она порождается из `tensortrade.env.actions` (по умолчанию, если не указано, берётся DiscreteActionScheme с базовыми настройками).
5. **Схема вознаграждения**: по умолчанию может быть *SimpleProfit* (награда = процентное изменение net worth за шаг [35]) или *PBR* (позиционная: награда = изменение цены * позиция [20]). Пользователь может подставить и свою.
6. **Наблюдатель**: например, *TensorTradeObserver* (возвращает numpy-массив признаков из Feed). В текущей версии наблюдатель обычно уже встроен – он берёт из *FeedController* поле `features` текущего состояния.
7. **Остальные (опционально)**: Informer (например, логирует коэффициенты Шарпа, см. RiskAdjustedReturns), Stopper (напр. останавливает эпизод при падении капитала более X%), Renderer/Plotter (рисует график баланса и сделок в конце эпизода).
8. **Создание окружения.** *TradingEnv* связывает все компоненты: принимается *Portfolio*, *DataFeed*, *ActionScheme*, *RewardScheme*, *Observer*, а также опционально Stopper, Informer, Renderer, Plotter [36] [37]. При инициализации *TradingEnv* сохраняет ссылки на них, устанавливает двусторонние связи (например, каждому компоненту схемы присваивается ссылка `trading_env` на текущее окружение [38] [39]) и настраивает внутренние объекты:

9. *Clock* (`_clock`) – счётчик шагов/тактов, общий для всех временных объектов (env, portfolio, exchanges) [40] [41]. *TradingEnv* наследует `TimeIndexed`, поэтому имеет глобальный часы по умолчанию.
10. *FeedController* (`_feed`) – обёртка над *DataFeed*, подготавливает его (добавляет портфельные стримы) и сразу выдаёт первое состояние [42] [43]. FeedController привязан к тем же глобальным часам (через TimeIndexed).
11. *Broker* (`_broker`) – объект OMS для исполнения ордеров. *TradingEnv* создаёт свой *Broker* при инициализации [44].

После конструктора окружение готово – это полноценный Gymnasium-совместимый класс с методами `reset()` и `step()`.

1. **Запуск цикла обучения/трейдинга.** Обычно агент взаимодействует с *TradingEnv* через стандартный интерфейс: `obs = env.reset(); while not done: action = agent.get_action(obs); obs, reward, done, info = env.step(action)`. В TensorTrade можно либо использовать встроенных `tensortrade.agents` (устаревшие), либо обернуть *TradingEnv* для внешних RL-библиотек. В примерах TensorTrade показывается интеграция с **Ray RLlib** (через регистрацию env как `Gymnasium` env) и использование **Stable Baselines3** (с помощью обёртки) – см. раздел 5.

Главные точки входа в код: создание *Portfolio* и *Exchange*, конструирование *DataFeed* (с помощью `Stream.sensor`, `Stream.reduce` и т.д.), затем инициализация *TradingEnv* со всеми схемами. Для упрощения, репозиторий предлагает ряд примеров и туториалов, где эти шаги выполняются последовательно. Например, ноутбук *train_and_evaluate.ipynb* показывает настройку данных (скачивание OHLCV через `data.cdd`), создание *Exchange* с сервисом `execute_order` (симулятор исполнения сделок) [45] [46], определение стримов и сборку *DataFeed*, и далее запуск `DQNAgent.train(env)`.

2.3. Паттерны: Registry, Context, Composition

Регистрация компонентов. TensorTrade применяет паттерн глобального реестра для компонентов: каждый подкласс *Component* при создании автоматически регистрируется в `tensortrade.core.registry` под определённым именем [47] [48]. Например, *AbstractActionScheme.registered_name* = `"action_scheme"` [49], *AbstractRewardScheme* – `"rewards"` [19], *Exchange* – `"exchanges"` [26], *Portfolio* – `"portfolio"` [50]. Вложенные в *TradingContext* конфиги используют эти имена как ключи. *TradingContext* реализован с помощью менеджера контекста (`with TradingContext(config): ...`): при входе в with он помещает конфиг в стек потоков для InitContextMeta. Метакласс *InitContextMeta* перехватывает вызовы конструктора компонентов: перед созданием объекта берёт текущий контекст и извлекает из него настройки для данного класса по зарегистрированному имени [51] [52], сливает их с “shared” настройками, и прокидывает в объект через свойство `instance.context` [53] [54]. Таким образом, можно единовременно описать параметры нескольких компонентов в одном словаре и создать их без явного перечисления аргументов. Например, можно задать в конфиге `{"rewards": {"window_size": 5}}` и затем просто вызвать `RewardSchemeClass()`, метакласс сам подставит `window_size=5`. Это упрощает сборку сложных сред: **Component**-классы вытягивают свои дефолты из контекста методом `self.default(key, default_value)` [55] [56]. Если в контексте нет значения, берётся `default_value`. В текущей версии этим механизмом можно не пользоваться и передавать параметры прямо в конструкторы.

Композиция модулей. TensorTrade следует *Keras-образному* паттерну модульности [57]. Компоненты слабо связаны через интерфейсы и контекст. *TradingEnv* агрегирует ссылки на схемы,

портфель, поток, но их внутренняя логика инкапсулирована. Например, *ActionScheme* ничего не знает о *RewardScheme* и *Observer*, они общаются только через *TradingEnv*. Это облегчает подмену компонентов: можно легко заменить стратегию вознаграждения или источник данных, не меняя остального кода.

Контекст времени (Clock). Паттерн *TimeIndexed* (наследуют Env, Portfolio, Exchange, Broker и др.) обеспечивает, что у всех объектов есть ссылка `self.clock` на общий таймер времени [10]. По умолчанию используется глобальный `tensortrade.core.base.global_clock` [58] [59]. *Clock* (см. далее) реализует метод `increment()` для шага симуляции. При создании *Portfolio*, *Broker*, *Exchange* им всем назначается один и тот же `Clock`. Это гарантирует синхронность: например, `portfolio.created_at` и метки времени ордеров будут в одной временной системе. Если нужно, пользователь может подменить `env.clock` на кастомный (например, тиковый с нерегулярным шагом). *Clock* также может использоваться для отметок времени: *TimedIdentifiable* присваивает всем объектам атрибут `created_at` при инициализации [60] [41].

DataFeed и observer pattern. *DataFeed* + *FeedController* реализуют потоковую обработку данных. *FeedController* сам является *Observable* (наследует `tensortrade.core.Observable`), и портфель (*Portfolio*) подписывается на обновления фида – он добавлен как listener [61] [62]. Каждый раз, когда *FeedController* получает новое состояние, он через `listener.on_next(state)` может уведомить слушателей [63]. Хотя код *Portfolio.on_next(...)* напрямую не показан, предположительно, *Portfolio* может сохранять историю метрик (например, для *portfolio.performance*). Фактически, при вызове `feed.next()` *FeedController* обновляет свое состояние `self._state` (features, meta, portfolio, step) и добавляет snapshot `meta`-данных в историю [64] [63]. Кроме того, *FeedController* группирует все потоки портфеля: баланс и “worth” каждого кошелька, а также суммарный `net_worth` портфеля [65]. Таким образом, при каждом шаге в *DataFeed* присутствует актуальный `portfolio.net_worth`, вычисленный из балансов и текущих цен (через *Exchange.quote_price* и соответствующие стримы) [23] [66].

Инициализация через фабрики. Благодаря registry/context, возможно конфигурировать компоненты декларативно. Например, можно определить свой класс вознаграждения `MyRewardScheme(AbstractRewardScheme)` с `registered_name = "rewards"`, реализовать метод `reward()`, и затем в YAML/JSON конфиге указать параметры для `"rewards": {...}`. Создавая среду внутри `with TradingContext(config): env = TradingEnv(...)`, ваш класс автоматически получит эти параметры. Если же не использовать контекст, класс можно регистрировать вручную: `registry.register(MyRewardScheme, "rewards")` – но обычно это не нужно, т.к. `Component.__init_subclass__` делает это сам [8].

В итоге архитектура TensorTrade следует принципам: **разделение ответственности** (отдельные подклассы для каждой части стратегии), **встроенные точки расширения** (абстрактные схемы и Component-метакласс для удобного добавления новых реализаций) и **единство времени и данных** (Clock + DataFeed гарантирующи согласованность между ценами и состоянием портфеля). Ниже подробнее рассматриваются ключевые сущности этой архитектуры.

3. Сущности и их формальные роли

(Здесь рассматриваются основные классы ядра с указанием их назначения, ключевых методов, жизненного цикла и взаимосвязей. Ссылки на исходный код приведены для подтверждения логики.)

- **TradingEnv** – основной класс окружения RL. Наследуется от `gymnasium.Env` и `TimeIndexed` [67]. Он инкапсулирует весь цикл взаимодействия агента с рынком:

предоставляет метод `reset()` для начала эпизода и `step(action)` для шага симуляции. *TradingEnv* агрегирует:

- *ActionScheme* (интерпретатор действий агента),
- *RewardScheme* (вычислитель награды),
- *Observer* (формирует наблюдение),
- *Informer* (собирает доп. информацию в `info`),
- *Stopper* (решает, когда эпизод должен закончиться),
- *Renderer/Plotter* (для визуализации, опционально),
- *Portfolio* (портфель с балансами на биржах),
- *DataFeed* (поток данных рынка).

Назначение: обеспечить совместимость с OpenAI Gym API, orchestrate вызовы всех схем на каждом шаг.

Ключевые методы/атрибуты: - `TradingEnv.step(action)` – выполняет один шаг: применяет действие через *ActionScheme*, продвигает время и данные, вычисляет награду, формирует наблюдение и инфо, проверяет завершение эпизода [16] [68]. - `TradingEnv.reset(seed=None)` – перезапускает все компоненты к начальному состоянию (обнуляет портфель, брокера, перезапускает DataFeed с случайным сдвигом по `random_start_pct`, сбрасывает схемы и рендеры) [69] [70]. Возвращает начальное наблюдение и info. - Свойства `action_space` и `observation_space` – берутся напрямую из схемы действий и наблюдателя (например, *Discrete(3)* для действий "купить/продать/держать" или shape массива наблюдения) [71]. - `trading_env.clock` (наследуется от TimeIndexed) – объект *Clock* для отметки шагов (увеличивается на 1 каждый `step`) [16] [72]. - Ссылки `self._action_scheme`, `._reward_scheme`, `._observer`, `._informer`, `._stopper`, `._renderer`, `._plotter` TradingEnv хранит все компоненты как атрибуты.

Где создаётся и кто владеет: обычно пользователь напрямую создает *TradingEnv*, передав в конструктор готовые компоненты (или используя дефолты). Env создаёт внутри себя *Broker* и *FeedController*, устанавливает `portfolio.clock = self._clock` и `exchange.clock = self._clock` для всех бирж портфеля [73] [25].

Кто вызывает: агент (или training loop) вызывает `env.reset()` и многократно `env.step(action)`; внутри `step` env сам вызывает методы схем и OMS.

Жизненный цикл: 1. *Reset:* Env генерирует новый уникальный `episode_id`, увеличивает счётчик эпизода `n_episode`, сбрасывает часы (`_clock.reset()`), очищает состояние портфеля и брокера, перезапускает DataFeed (с optional случайным сдвигом начала, если `random_start_pct` > 0) [69] [70]. Далее вызывает `reset()` у всех подключённых схем (action, reward, observer...) [74]. После этого запрашивает начальное наблюдение `obs = _observer.observe()` и info `info = _informer.info()` [75] и возвращает их. 2. *Step:* Принимает `action` от агента. Сначала передаёт его в *ActionScheme.perform_action* [16]. Затем: - Вызывает `self.clock.increment()` (шаг времени +1) [76]. - Обновляет данные: `self.feed.next()` – тем самым *FeedController* получает следующий временной шаг из DataFeed с учётом изменений портфеля после сделки [76] [72]. - Вычисляет `reward = _reward_scheme.reward()` для нового состояния [77]. - Получает новое `obs = _observer.observe()` и `info = _informer.info()` [77] [75]. - Проверяет флаг `terminated`: если есть Stopper, вызывает `terminated = _stopper.stop()`; иначе `False` [68]. Затем, если данных больше нет (`not feed.has_next()`), также ставит `terminated = True` [72]. - Сохраняет *последнее состояние* `_last_state` (объект ObsState с полями observation, reward,

info, terminated) – его можно, например, использовать для рендеринга или анализа после эпизода [78]. - Вызывает `_renderer.render()` если включен режим визуализации 'human' [79]. - Возвращает `(obs, reward, terminated, truncated, info)` согласно Gym API. `truncated` всегда False (TensorTrade не разделяет две причины окончания эпизода, Stopper трактуется как terminated) [80]. 3. *Close:* Завершение окружения – вызывает `.close()` у рендерера/плоттера (освободить ресурсы) [81].

Варианты расширения: *TradingEnv* уже достаточно общий. Прямое наследование от него обычно не требуется – вместо этого расширяют вложенные схемы. Но возможно создать подкласс TradingEnv, например, чтобы переопределить `step()` для логирования или `reset()` для загрузки нового датасета на каждом эпизоде. В коде TensorTrade также предусмотрена регистрация env как Gym: коммит #79 добавил entrypoint для регистрации окружения, что позволяет `env = gym.make("TradingEnv-v0")` [82] [83] после proper регистрационной функции. Однако основной способ – напрямую инстанцировать TradingEnv.

- **ActionScheme (AbstractActionScheme)** – абстрактный класс схемы действий, определяет, как дискретный или непрерывный сигнал от RL-агента преобразуется в торговое решение (ордеры). Например, встроенная *DiscreteActions* сопоставляет целое число {0,1,2} с действием “ничего не делать”, “купить”, “продать”.

Назначение: инкапсулировать логику создания торговых приказов (*Order*) из действия RL. Это позволяет легко менять способ торговли (рыночные ордера, лимитные, изменение размера позиции и т.п.), не затрагивая остальное.

Ключевые методы: - `action_space` (property) – возвращает Gym Space действий, соответствующий данной схеме (например, `spaces.Discrete(3)`). Агент должен выдавать action, принадлежащий этому пространству [84]. - `get_orders(action) -> List[Order]` – основной метод: принимает `action` (тип gymnasium.core.ActType, например int) и возвращает список ордеров, которые надо выставить в OMS [18]. В простейшем случае может вернуть один Order (либо пустой список/None, если действие – hold). - `perform_action(action)` – реализован в базовом классе: просто получает список ордеров через `get_orders` и отправляет их брокеру: `self.trading_env.broker.submit(order)` для каждого [28] [85], затем вызывает `broker.update()` [86], чтобы запустить процесс исполнения.

Где создаётся: либо пользователь сам создаёт экземпляр конкретной схемы (например, `DiscreteActions(n_actions=3)`), либо *TradingEnv* при инициализации подставляет дефолтную (если None, берётся `tensortrade.env.actions.SimpleOrders` или аналогичный класс по умолчанию). В registry эта схема регистрируется как `"action_scheme"` [49].

Кто владеет/вызывает: `TradingEnv` хранит ссылку `_action_scheme`. При каждом `env.step(action)` env вызывает `self._action_scheme.perform_action(action)` [16] – тем самым инициируя исполнение. *ActionScheme* имеет обратную ссылку `trading_env` (устанавливается при инициализации env [38]), чтобы иметь доступ к broker, портфелю и прочим частям окружения.

Жизненный цикл: После `reset()` схема может обнулить своё внутреннее состояние (метод `reset()` существует, но часто пустой по умолчанию [87]). Например, если схема отслеживает что-то между шагами (хранит предыдущую действию или позицию, как PBR RewardScheme делает), она сбрасывает это. При `step`: `perform_action` –> `get_orders` –> `Broker.submit` –> `Broker.update`. Пример: DiscreteActions может реализовать `get_orders` так: если action=1

(buy) – вернуть Order на покупку фиксированного количества базовой валюты на всю доступную котируемую (например, “market buy BTC на все USD”); если action=2 (sell) – Order на продажу всего BTC; если action=0 – вернуть [] (ничего не делать). Эти ордера (типично рыночные) будут сразу исполнены брокером.

Варианты: Пользователь может унаследовать AbstractActionScheme и реализовать свои методы. Минимум – определить `action_space` и `get_orders`. Например, схема “GridTrading” могла бы иметь дискретные действия для разных долей портфеля. Главное – убедиться, что создаваемые Order корректно настроены (указана биржа, trading pair, размер, тип ордера). Далее OMS возьмет на себя исполнение (см. OMS раздел ниже).

- **RewardScheme (AbstractRewardScheme)** – абстрактный класс схемы вознаграждения. Определяет, как на каждом шаге вычисляется численное вознаграждение `R_t` для агента на основе изменений портфеля, риска, транзакционных издержек и т.д.

Назначение: инкапсулировать стратегию оценки “хорошести” действий агента. Разные схемы позволяют ставить разные цели обучения: максимизация прибыли, риск-Adjusted прибыль, следование какому-то бенчмарку и т.п.

Формула: Зависит от конкретной реализации. Например: - *SimpleProfit*: вознаграждение = относительная прирост чистой стоимости портфеля за шаг, т.е. $R_t = \frac{NW_t}{NW_{t-\Delta}} - 1$. В коде: `return net_worth[-1] / net_worth[-window-1] - 1.0` [35] (при window_size=1 берётся предыдущий шаг). - *PBR (Position-Based Returns)*: $R_t = (p_t - p_{t-1}) \cdot x_t$, где x_t – позиция (+1 = лонг, -1 = шорт), p_t – цена актива [20]. То есть агент получает положительную награду, если угадывает направление рынка. В коде PBR внутри использует стрим разности цены и текущую позицию: `reward = (position * price_diff)` [88] [89], где `position` обновляется в `on_action(action)` (например, action=0 -> позиция -1, action=1 -> +1) [89]. - *RiskAdjustedReturns*: R_t – рассчитывается по формуле Шарпа или Сортино на основе серии прошлых доходностей портфеля [90] [91]. Например, при алгоритме 'sharpe': $R = \frac{mean(r) - r_f}{std(r)}$ за окно недавних шагов [92]. Эта схема даёт положительное вознаграждение за высокие средние доходности и штрафует за волатильность. - Также можно реализовать любую функцию: например, вознаграждение = логарифмическая доходность портфеля $\ln(NW_t / NW_{t-1})$ (чтобы поощрять экспоненциальный рост), или — максимальная просадка и т.п.

Ключевой метод: `reward() -> float` – вызывается после обновления портфеля на шаге, должен вернуть скаляр [93]. Внутри он может обращаться к `self.trading_env` для данных (например, к `portfolio.performance` или `broker.trades`). В `AbstractRewardScheme` есть ссылка `trading_env` (через SchemeMixin) [94].

Где создаётся: аналогично ActionScheme – либо явно пользователем, либо TradingEnv подставит дефолт. Зарегистрировано как `"rewards"` в реестре [95]. В конфиге можно настроить параметры (например, `{"rewards": {"window_size": 5}}` для SimpleProfit).

Кто вызывает: *TradingEnv* вызывает `reward = _reward_scheme.reward()` внутри `step()` уже после того, как обновились баланс и цена [96]. Это важно: RewardScheme смотрит на новое состояние (например, *portfolio.net_worth* после выполнения ордера).

Жизненный цикл: `reset()` может обнулять внутренние накопленные данные (напр., RiskAdjustedReturns может очищать прошлые доходности). В коде многие `reset()` в схемах не

переопределены (т.е. pass, как у AbstractRewardScheme [97]), но RiskAdjustedReturns или другие могут ничего не хранить помимо ссылки на env.

Примеры и расширения: Можно создать собственную схему, унаследовав AbstractRewardScheme. Например, добавить штраф за частое совершение сделок: для этого можно отслеживать через `trading_env.broker.trades` количество сделок и вычитать комиссионные. Или награда = `\Delta \text{Sharpe}` – улучшение коэффициента Шарпа портфеля за последний месяц. Всё это можно вычислить, имея историю `portfolio.performance` (которая содержит `net_worth` по шагам). В коде TensorTrade уже есть *RiskAdjustedReturns*, дающий пример того, как использовать `portfolio.performance` для вычисления метрики (Шарпа/Сортино) [91] [98]. Комиссии и проскальзывание учитываются на уровне исполнения ордеров (в OMS), а RewardScheme оперирует уже чистыми результатами (с учётом этих издержек) – см. ниже Exchanges.

- **Observers / Informers:**
- *Observer* – компонент, формирующий observation для агента. Как правило, он берёт текущие признаки рынка из *FeedController*. В базовой реализации (например, `tensortrade.env.observers.TensorTradeObserver`) просто возвращает `self.trading_env.feed.state.features` как `np.array` или `pd.DataFrame`. Таким образом observation может быть, например, массив нормированных OHLCV за окно последних N шагов плюс технические индикаторы.
  - **Назначение:** абстрагировать способ представления состояния. Можно реализовать Observer, который возвращает не сырые рыночные данные, а, скажем, состояние какой-то модели или сигналы риска.
  - **Методы:** `observe() -> ObsType` – вызывается env-ом каждый шаг [77] [75]. Возвращает observation (обычно `np.ndarray` или dict). Ещё может быть `reset()` для очистки (например, если observer хранит скользящее окно данных).
  - **Реализация по умолчанию:** TensorTrade, судя по вызовам, ожидает, что Observer возьмет из *FeedController* сформированный словарь `features`. *ObsState* (в `env.utils`) хранит, помимо observation, ещё info, reward, terminated – но Observer оперирует только частью observation.
  - **Жизненный цикл:** создаётся до TradingEnv, или Env подставляет дефолт. *TradingEnv* задаёт `observer.trading_env = self` [38]. После каждого `feed.next()` Observer может получить доступ к свежим данным. Непосредственно `observer.observe()` вызывается **после** обновления состояния на шаге, чтобы вернуть агенту новое наблюдение [77].
  - **Пример:** Если DataFeed включает индикатор, скажем, RSI, то `features` словарь может выглядеть как `{'open': 1.234, 'high': ..., 'rsi14': 55.2, ...}`. Observer может сконвертировать это в numpy-массив `[1.234, ..., 55.2, ...]` либо сразу отдать как dict (агент должен уметь с ним работать). Gym требует obs быть пространством из `spaces.Space`, обычно это Box(np.float).
- *Informer* – компонент для формирования информационного словаря (`info`) на каждом шаге. Info не влияет на обучение, но даёт полезные диагностики (метрики, статистики) для логирования или отладки.
  - **Назначение:** отделить побочные вычисления (например, расчёт текущей доходности, просадки, прибыльных сделок и т.п.) от основной логики реворда. В info

можно складывать всё, что может понадобиться при анализе, но не должно идти агенту в наблюдения.

- **Метод:** `info() -> dict` – вызывается после вычисления реворда, перед тем как вернуть из `env.step` [77] [75]. Получает доступ к `self.trading_env` и может собрать нужное. Например, встроенный *PerformanceInformer* мог бы подсчитывать текущий ROI, максимальную просадку и Sharpe за эпизод.
- **Реализация:** В TensorTrade есть намёки на Informer, но конкретные встроенные реализации могли отсутствовать или быть минимальными (в 1.0.3 был, например, `TensorTradeInformer` который возвращал пустой или базовую инфу). В 1.0.4-dev1, возможно, *informer.info()* возвращает `self.trading_env.feed.state.meta` или ничего.
- **Использование:** Если пользователь хочет логировать определённую метрику – можно наследовать AbstractInformer и прописать вычисление. Informer регистрируется как `"informers"` (возможное имя) и передается в TradingEnv.
- **Пример:** *RiskMetricsInformer*: собирает в info ключи: `net_worth`, `drawdown`, `sharpe` на текущий шаг. Он мог бы использовать историю `portfolio.performance` для этих вычислений. Тогда при каждом шаге в `info` агент (точнее, пользователь) увидит обновление этих метрик.

Где создаются: Обычно явным указанием при создании TradingEnv. Если None, TradingEnv может поставить дефолт (Observer по умолчанию, Informer – пустой). Оба – Component, так что могут конфигурироваться через контекст.

Кто вызывает: TradingEnv: `obs = observer.observe()` и `info = informer.info()` в `env.step` и `env.reset` [77] [75]. После эпизода, `env.last_state` содержит последнее obs и info, что может использоваться рендерером для финального отчёта.

Жизненный цикл: *Observer* и *Informer* обычно не хранят долгосрочного состояния (кроме, возможно, подсчёта эпизодных метрик). Их `reset()` будет вызван при env.reset [74], где можно обнулить накопители (например, списки значений метрик).

Варианты расширения: Новые Observer – если ваши наблюдения сложнее (например, включают внутреннее состояние модели прогноза), можно интегрировать их через Observer. Новые Informer – например, вывести коэффициент Сортино за эпизод, число сделок, процент выигрышных сделок и т.д., что не влияет на агента, но полезно для оценки стратегии. Informer может подписаться как listener на FeedController или Broker, но чаще просто обратиться к `trading_env` при info().

- **Renderers / Stoppers:**

- *Renderer* – отвечает за отображение хода эпизода человеку. TensorTrade позволяет подключить один или несколько рендереров. Примеры: *ScreenLogger* (выводит текст в консоль), *MatplotlibGraph* или *PlotlyChart* (рисует график цены и действий).

  - **Методы:** `render()` – вызывается при каждом шаге, если `render_mode='human'` [79]. Он может накапливать данные. `close()` – в конце.
  - *Plotter* – отдельный класс (например, AggregatePlotter), который агрегирует несколько Renderer-ов и строит их разом. В TradingEnv, если передан список рендереров, он оборачивается в AggregatePlotter [99].

- **Использование:** В коде, если `env.render_mode` установлен и есть Renderer, то env.step будет дергать `renderer.render()` каждый шаг [79], иначе можно вручную вызывать `env.render()` в нужные моменты.
- **Расширение:** Можно реализовать Renderer, например, отправляющий статистику по сети или сохраняющий GIF анимацию. Нужно унаследовать AbstractRenderer и реализовать `render()`.
- TensorTrade 1.0.4-dev1, вероятно, предлагает *PlotlyTradingChart* (разметка графика баланса и цен, см. документацию).

• *Stopper* – компонент, решающий, следует ли прервать эпизод досрочно.

- **Пример:** *MaxDrawdownStopper* – останавливает, если просадка портфеля > X%. *TimeoutStopper* – ограничение длительности эпизода в шагах.
- **Метод:** `stop() -> bool` – TradingEnv вызывает его каждый шаг после получения награды [68]. Если возвращает True, `done` будет помечен True.
- **Реализация:** В TensorTrade есть базовый AbstractStopper; по умолчанию, если нет стоппера, эпизод длится пока есть данные. Stopper может использовать информацию из `trading_env` (например, `portfolio.net_worth` или счетчик шагов).
- **Расширение:** Пользователь может написать кастомный Stopper, например, чтобы обучать агента на коротких интервалах: Stopper, завершающий эпизод каждые 100 шагов (скользящее окно обучения).
- **Взаимодействие:** Если Stopper сработал или закончились данные, TradingEnv считает эпизод оконченным (`terminated=True`, `truncated=False`) [68].

Где создаются: Renderer и Stopper обычно явно передаются в TradingEnv (или None).

Кто вызывает: Env: Stopper в `step` для проверки конца; Renderer – либо в `step` каждый шаг (если режим human), либо по запросу пользователя `env.render()`.

Жизненный цикл: Stopper может иметь состояние (например, хранить high_water_mark для расчёта просадки), должен сбрасываться при env.reset (TradingEnv вызывает stopper.reset()) [100]. Renderer/Plotter может накапливать данные внутри эпизода; reset() им тоже посылается, что реализовано для Plotter (обнулить предыдущий график) [101].

• **Clock / Timeline:**

• *Clock* – класс, определяющий временную шкалу симуляции. В `tensortrade.core.clock.Clock` (из base.py и clock.py) – хранит счётчики времени, и, вероятно, timestamp.

- В текущем коде *Clock* не разобран подробно в открытом исходнике, но исходя из использования: `global_clock = Clock()` [58], у него есть методы: `.now()` – текущее время (может быть datetime index или шаг), `.increment()` – шаг вперёд.
- TradingEnv при каждом step делает `clock.increment()` [76] – то есть Clock шагает дискретно на 1. Clock, скорее всего, содержит поле `step` (например, номер тика) и, возможно, actual timestamps, если задавались.
- Все TimeIndexed объекты ссылаются на один Clock, поэтому `portfolio.created_at` или `order.timestamp` можно сравнивать – они в одной временной линии.

- **Timeline**: Подразумевается, что DataFeed продвигается синхронно с Clock: например, если Clock хранит datetime, DataFeed может использовать его для выборки данных. Однако в реализации DataFeed шаги просто индексируются по порядку (has_next / next).
- *Clock.reset()* – TradingEnv вызывает его при reset, чтобы начальный шаг был 0 или скорректирован (например, если random_start_pct смещает начало, Clock мог бы учитывать смещение времени, но у нас нет деталей реализации).
- **Использование:** Clock выступает как единый источник времени: если, например, нужно рассчитать доходность за 252 торговых дня, RewardScheme может проверять `if clock.step % 252 == 0` и тогда вычислять годовую метрику.

- *Timeline events:* TensorTrade не использует явных событий времени, кроме increment. Но, например, можно представлять, что каждый шаг = один бар данных (например, час). DataFeed обычно подаёт ровно следующий бар.

Синхронизация компонентов: за счёт Clock и единого FeedController, можно быть уверенным, что на шаг *t*: - *FeedController.state* содержит цены/фичи за *t*, - *portfolio.net_worth* рассчитан по ценам *t*, - *Clock.step = t*, - Все ордера сгенерированные на шаге *t-1* уже исполнены до получения состояния *t*.

Расширение: Если нужно смоделировать другое течение времени (например, event-driven когда торговля происходит по приходу сделок вне фиксированного шага), пришлось бы расширять Clock/Exchange, но в текущей архитектуре проще фиксированный шаг.

- **OMS (Order Management System)**:

- **Portfolio** – класс портфеля, объединяющий набор кошельков (Wallet) на разных биржах и базовую валюту для оценки (base_instrument). Он наследует *Component* и *TimedIdentifiable* [102].

  - **Назначение:** держать совокупное состояние активов агента и предоставлять методы для оценки (net worth), а также централизованно принимать результаты сделок. Portfolio агрегирует все балансы.
  - **Ключевые поля:** `base_instrument` (например, USD – в чём измеряется net worth) [103]; `_wallets` – словарь {(exchange, instrument): Wallet} [104]; `order_listener` – обработчик ордеров (Portfolio сам может быть OrderListener, но в коде у него есть `OrderListener` как параметр, который назначается всем ордерам) [105].
  - **Методы:**
  - `add(wallet)` – добавить кошелёк в портфель (в конструкторе проходит по списку wallets и добавляет) [106].
  - `get_wallet(exchange, instrument) -> Wallet` – получить кошелек для биржи и инструмента (неявно, likely exists, хотя явного кода не показано, но `Exchange.execute_order` вызывает `portfolio.get_wallet(self.id, order.pair.base)` [31]).
  - Свойства `balances`, `locked_balances`, `total_balances` – списки Quantities по всем кошелькам [107] [108].
  - `balance(instrument)` – суммарный доступный баланс данного инструмента во всех кошельках [109] [110].
  - `net_worth` – вычисляется как сумма стоимости всех Wallet-ов в базовой валюте. В коде Portfolio `_net_worth` обновляется по ходу или рассчитывается на лету. В *FeedController* эта логика реализована через стримы: суммируются total балансы всех

кошельков, приведённые к base_instrument через текущие цены [65]. Также Portfolio имеет `initial_net_worth` и `profit_loss` (PL%) [111].
  - `reset()` – обнуляет `_performance` и возможно `_net_worth` (в коде TradingEnv вызывает `portfolio.reset()` [112], и мы видим, что `Portfolio.clock.setter` перезаписывает clock всем биржам при изменении [113] [114]).
  - **Жизненный цикл:** Создаётся один раз перед Env (можно переиспользовать между эпизодами). При reset возвращается к начальному балансу: обычно *Portfolio.initial_balance* сохраняется при старте (сумма base instrument) [115], а потом *Portfolio.reset()* могла бы вернуть все Wallet-ы к стартовым балансам (в коде конкретно нет, но Broker при сбросе не отменяет открытые сделки, т.к. к reset все должны выполниться). `initial_net_worth` сохраняется после первого расчёта (например, после env.reset).
  - **Владение:** Portfolio содержится в TradingEnv, который присваивает ему свой clock [73]. Portfolio, в свою очередь, владеет Wallet-ами.
  - **OrderListener:** Portfolio реализует интерфейс *OrderListener*, что видно по тому, что Broker.attach(order) вызывает order.attach(self) (портфель, вероятно) – но конкретно в коде, *Portfolio* не явно указан как OrderListener, зато Broker является OrderListener (и Portfolio может быть передан ему). Однако, в коде `Portfolio.order_listener` может указывать, например, на Broker, или на сам Portfolio. В портфеле `OrderListener` в параметрах, и он передаётся всем создаваемым ордерам (OrderListener у ордера – кто слушает события по нему). Похоже, `order_listener` – callback, вызываемый при выполнении ордера (например, чтобы обновить performance).
  - **Обновление net worth:** Происходит при каждом исполнении сделки: *Exchange.execute_order* вызывает `order.fill(trade)` [116], а реализация `Order.fill(trade)` внутри себя изменяет состояния Wallet-ов (уменьшает балансы base/quote). Вероятно, после полного исполнения Portfolio может пересчитать *net_worth*. В *FeedController.create_portfolio_streams*, после каждого шага, net_worth считается свежо через стримы цен [65], так что можно не хранить _net_worth явным счетчиком, а вычислять.
  - **Расширение:** Можно подкласс Portfolio, например, для поддержки нескольких стратегий/агентов в одном (multi-portfolio). Но чаще добавляют свои слушатели или методы для логирования (performance_listener – функция, вызываемая на каждом обновлении, задана в конструкторе [117]). Она, возможно, дергается где-то (неявно, может внутри Order.complete).

- **Wallet** – кошелек на конкретной бирже для конкретного инструмента. Хранит баланс свободных средств и баланс, заблокированный в открытых ордерах.

  - **Роль:** представлять счёт трейдера на бирже X в валюте Y.
  - **Атрибуты:** `exchange` (ссылка или ID биржи), `instrument` (что за актив), `balance` (Quantity), `locked_balance` (Quantity).
  - Wallet связан с *Ledger* – книга записей всех транзакций (продвинутый функционал: Ledger записывает каждое изменение баланса с причиной).
  - **Операции:** при исполнении ордера, вызывается `wallet.withdraw(amount)` или `wallet.deposit(amount)` (в Order.fill).
  - Например, если ордер BUY BTC исполняется:
  - из USD-кошелька вычитается сумма (с учетом комиссии),
  - в BTC-кошелек прибавляется купленное количество BTC.
  - Wallet обновляется мгновенно в Order.fill (Trade содержит объем и цену, а Order.fill уже знает, сколько снять/добавить).

- **Instrument / TradingPair / Quantity:** инфраструктурные классы:
  - *Instrument* определяет тикер (например “BTC”), тип актива (фиат, крипто), и precision (количество знаков после запятой для цены/количества) [118] .
  - *TradingPair* связывает два инструмента: base (например BTC) и quote (например USD), задавая пару, торгуемую на бирже [23] .
  - *Quantity* хранит число (Decimal) и ссылку на Instrument (чтобы знать precision, единицы). Операции сложения/вычитания на Quantities реализованы (см. `Quantity.__add__`).
  - Эти классы нужны для строгого учёта: чтобы не складывать USD и BTC напрямую, например.
- **Order** – класс, представляющий ордер (заявку на бирже). Имеет следующие основные свойства:
  - `id` (уникальный идентификатор) [119] ,
  - `pair` (TradingPair, например BTC/USDT),
  - `quantity` (сколько базового актива покупать/продавать, Quantity),
  - `price` (цена, Quantity котируемой за единицу базового – для лимитных ордеров),
  - `status` (статус: NEW, PENDING, OPEN, PARTIALLY_FILLED, FILLED, CANCELLED и др.) [120] ,
  - `is_buy` или `side` (Buy/Sell).
  - `time_in_force` , `expiration` – могут быть опции.
  - `commission` (может хранить комиссии).
  - *Order* хранит список сделок *trades* (исполнений) по нему и, возможно, child orders (в случае сложных стратегий: стоп-лосс ордера).
  - **Методы:**
  - `execute()` – отправляет ордер на исполнение: на симулируемой бирже просто вызывает `exchange.execute_order(order, portfolio)` [27] . В `Broker.update` видно: если `order.is_executable` , то `order.execute()` вызывается [121] .
  - `cancel()` – отменяет ордер (меняет статус, разблокирует баланс).
  - `attach(listener)` – присоединяет OrderListener (например, Broker) к ордеру [122] , чтобы при исполнении/частичном исполнении уведомлять слушателя.
  - `fill(trade)` – регистрирует Trade как исполнившую часть ордера. Внутри, скорее всего: добавляет trade в свой список `trades` , обновляет статус (если полностью выполнен, помечает COMPLETE), и *главное* – вызывает `Portfolio.updateBalances(trade)` или аналог (т.е. в результате trade кошельки обновляются). Но в коде сделано иначе: *Exchange* сама вызывает `order.fill(trade)` [116] , а затем *Broker.on_fill* перехватывает это событие [123] , и уже Broker добавляет trade в `broker.trades` и, если ордер Complete, вызывает `order.complete()` [124] . `order.complete()` может возвращать *next_order* (например, для OCO).
  - `is_executable` (property) – условие, при котором ордер готов к немедленному исполнению. Для *Market Order* всегда True сразу. Для *Limit Order* – True, если текущая рыночная цена достигает уровня ордера.
  - `is_active` – ордер находящийся в книге (новый или частично исполненный).
  - `is_complete` – исполнен полностью.
  - `is_expired` – истёк по времени.

- **Жизненный цикл:** создаётся *ActionScheme* при генерации действий, сразу передается Broker. Broker держит его в списке `unexecuted`.

- При вызове `Broker.update()`: если `order.is_executable`, он переносит ордер в `executed` и вызывает `order.execute()` [125]. У Market Order, видимо, флаг `is_executable` сразу True, так что он исполнится на том же тике.
- `order.execute()` на сим-бирже ведет к мгновенному созданию Trade и вызову `order.fill(trade)` [31] [126].
- `order.fill(trade)` вызывает у OrderListeners (Broker и, может быть, Portfolio) метод `on_fill(order, trade)` – в коде Broker.attach сделан до execute [127], поэтому Broker будет слушать. Broker.on_fill добавляет trade в свой журнал и проверяет, завершён ли ордер [123] [128]. Если завершён, вызывает `order.complete()` и, возможно, получает `next_order` (исполнение цепочки).
- Если `next_order` есть и он executable сразу (например, тейк-профит ордер активируется), Broker исполняет его немедленно (добавляет в executed и `execute()`), иначе кладет в очередь [124].
- После полного исполнения или отмены, Order выходит из `unexecuted`.

- В конце эпизода все ордера должны быть либо исполнены, либо отменены, чтобы баланс сошелся.

- **Отношения с другими:** Order хранит ссылку на связанный Exchange (обычно через TradingPair.exchange_id) – фактически, *Exchange.execute_order* получает нужные Wallet из Portfolio по параметрам ордера [31]. Order также может ссылаться на Portfolio (не обязательно – обычно взаимодействие идет через Broker/Listeners). Commission: *ExchangeOptions.commission* передается в исполнение, *execute_order* может на основе него рассчитать меньший Trade, оставляя часть как комиссия.

- **Расширение:** Можно унаследовать Order для особых типов (например, StopOrder с условием is_executable = цена > X). Тогда нужно и Broker.update, и Exchange.execute_order расширять, чтобы учитывать их логику. В текущем TensorTrade вместо разных классов Order это часто делается через поля и проверку is_executable.

• **Trade** – класс отдельной сделки (fill). Содержит:

- `order_id` – ID ордера, которому принадлежит [128],
- `exchange_id` – на какой бирже выполнен,
- `price` – фактическая цена исполнения,
- `quantity` – исполненный объём,
- `commission` – комиссия,
- `timestamp` – время исполнения.
- Trade часто интерпретируется как заполнение ордера (может быть частичным).
- После Trade Portfolio обновляет два Wallet-а: base и quote для пары:
- Если это buy: уменьшается quote wallet, увеличивается base wallet.

- Если sell: наоборот. Комиссия вычитается либо из base, либо из quote – в ExchangeOptions можно задать, как именно.

- **Использование:** Broker хранит dict trades: ключи order_id, значения – список Trades по этому ордеру [129]. Можно потом анализировать trades для подсчета статистик (сколько сделок, средняя цена и т.д.).

- **Lifecycle:** создается функцией *execute_order* на бирже [31] и сразу передается в order.fill (=> broker.on_fill).

• **Broker** – диспетчер OMS, уже упоминался:

- Он наследует `OrderListener` и `TimeIndexed` [120] [130].
- **Роль:** управляет очередью ордеров, запускает их исполнение, реагирует на сделки (Trade) чтобы, например, выставлять связанные ордера.
- **Атрибуты:**
- `unexecuted` – список ордеров, ожидающих исполнения (неисполнимые немедленно – лимитники, или исполнимые, но ещё не обработанные до вызова update) [131] [132].
- `executed` – словарь исполненных ордеров (id -> Order) [133], возможно, для слежения.
- `trades` – `OrderedDict` всех Trade-ов, сгруппированных по order_id [134] (для анализа истории) – именно обновляется в on_fill [129].
- **Методы:**
- `submit(order)` – добавить ордер в очередь unexecuted [135] [136].
- `cancel(order)` – отменить ордер: если он ещё не исполнен (в `unexecuted`), убрать из списка; вызвать `order.cancel()` (меняет статус) [137].
- `update()` – главный цикл: проходит по `unexecuted`:
  - Для каждого ордера: если `order.is_executable` -> перемещает его в `executed`, присваивает `executed[order.id] = order` [122], цепляет к ордеру себя (`order.attach(self)`) и вызывает `order.execute()` [138] (исполнение на бирже). Ордер может исполняться полностью сразу (market) или оставаться частично (например, если service симуляции позволяет частичное). После прохода, удаляет все исполненные ордера из `unexecuted` [139].
  - Затем вторым циклом пробегается по **всем** активным ордерам (объединение оставшихся в `unexecuted` + всех в `executed`): если `order.is_active and order.is_expired` -> отменяет их (вызывает cancel) [140]. Это, например, снимает просроченные лимитки.
- `on_fill(order, trade)` – реализация OrderListener: вызывается, когда ордер исполняется (см. выше). Действия:
  - если trade.order_id есть в executed и ещё не в trades – добавить trade в журнал (self.trades) [123];
  - если ордер полный (`order.is_complete`), вызвать `next_order = order.complete()` [141]. Если `next_order`:
  - Если он executable сразу – сразу исполняет (добавляет в executed, attach+execute) [142];
  - иначе `submit(next_order)` (в unexecuted) [143].
- `reset()` – очищает списки/словарь (unexecuted=[], executed={}, trades={}) [144].
- **Привязка к Env:** TradingEnv создаёт единственный Broker (`self._broker = Broker()`) [145]. Broker не знает напрямую про Portfolio или Exchange, он взаимодействует через Order/Trade callbacks. Но *Broker.update()* при execution передаёт портфель в Exchange (см. order.execute).

- **Взаимодействие с Portfolio:** В симуляции, Portfolio обновляется как следствие Trade: `order.fill(trade)` внутри себя вызывает `wallet.deposit/withdraw`. Классически, Broker не нужен знать про balances – этим занимается Exchange + Order + Wallet. В Broker.on_fill видно, что он не меняет балансы напрямую, а лишь отслеживает trades и следит за завершением ордера. Значит, именно `order.execute()` (через Exchange.service) отвечает за списание/зачисление средств. Поэтому **OMS "замкнут"**: ActionScheme -> Broker -> Exchange -> Portfolio (через Wallet) -> RewardScheme.
- **Расширение:** Если нужно особое поведение OMS (например, при частичном исполнении немедленно модифицировать оставшуюся часть ордера), можно подклассить Broker или реализовать свой OrderListener. Но чаще расширяют Exchange.service.

• **Exchanges:**

- *Exchange* – абстрактный или базовый класс биржи [146], представляющий либо реальную биржу (is_live=True) с подключением к API, либо симулятор для бэктеста (is_live=False).
- **Атрибуты:** `name` (идентификатор биржи) [30]; `_service` – callable, исполняющий сделки (например, функция `execute_order` из `oms.services.execution.simulated`) [30]; `options` – экземпляр ExchangeOptions (содержит комиссию, ограничения размеров и флаг is_live) [147] [30]; `_price_streams` – словарь ценовых стримов, привязанных к бирже [148] [149].
- **Методы:**
  - `__call__(*streams) -> self` – перегружен для привязки ценовых стримов: принимает набор *Stream* (каждый, вероятно, именован как `'BTC/USD:close'` и пр.) и сохраняет их в `_price_streams` под именами с префиксом биржи [150] [149]. Возвращает self, чтобы позволить синтаксис `exchange = Exchange("Binance", service)(price_stream)`. После этого *Exchange.streams()* вернёт список стримов с префиксами [151].
  - `quote_price(trading_pair) -> Decimal` – получает текущую цену инструмента (пары) из `_price_streams` [23] [118]. Берёт value последнего стрима, конвертирует в Decimal с учётом precision. Если цена 0 – бросает ошибку (цена не может быть 0) [152].
  - `is_pair_tradable(pair) -> bool` – проверяет, есть ли эта пара в `_price_streams` (т.е. есть ли стрим цены) [153] [154].
  - `execute_order(order, portfolio)` – ключевой метод исполнения ордера на бирже [27]:
  - Вычисляет `trade = self._service(order=order, base_wallet=..., quote_wallet=..., current_price=self.quote_price(order.pair), options=self.options, clock=self.clock)` [31]. То есть вызывает сервис-стратегию, предоставляя: сам Order, ссылки на кошельки портфеля (base_wallet, quote_wallet – извлечены через `portfolio.get_wallet(exchange_id, instrument)`), текущую цену актива (для market/limit исполнения), опции биржи (комиссия и т.п.), и clock (время для timestamp).
  - Ожидается, что `_service` возвращает объект *Trade* (или None, если ничего не произошло).
  - Если `trade` получен, вызывает `order.fill(trade)` [116]. Это отметит в ордере новую сделку и через Broker.on_fill зафиксирует её.
  - Не возвращает ничего.

    - (В реальной бирже, `execute_order` мог бы отправить запрос API. В dev1, если `options.is_live=True`, возможно, `_service` – имя API, либо dev1 не включает реалтайм.)
    - *ExchangeOptions* влияет на поведение `_service`: например, `commission=0.001` передается, и simulated service учтёт это, снимая 0.1%.
- **Симулируемые vs живые:** *Simulated Exchange:* используется в бэктестах. В примерах подключается `execute_order = oms.services.execution.simulated.execute_order` – эта функция реализует простое исполнение:
    - Если order типа Market: взять `current_price`, объём `order.quantity`, вычислить сколько списать/начислить, вычесть комиссию (из чего? обычно из quote или base – надо смотреть options),
    - Создать Trade со статусом FILLED.
    - Если Limit: можно эмулировать частичное исполнение, но скорее симулятор либо исполняет полностью по цене текущей свечи (если она достигла лимита), либо не исполняет. В отсутствие подробностей, полагаем, что simulated.execute_order исполняет *весь доступный объём по текущей цене*, сразу возвращая *Trade* [31]. Так уходят задержки, и agent фактически торгует по закрытым ценам свечей (или open следующей). *Live Exchange:* TensorTrade задумывался с возможностью переключиться на реальную торговлю (см. guiding principles). `is_live=True` означает, что `_service` может быть строкой, например "ccxt" – тогда `execute_order` мог бы внутри вызывать CCXT API (но прямо этого нет в коде 1.0.4-dev1; возможно, планировалось, но не реализовано полностью). На текущий момент, live-режим может требовать написать собственный сервис: например,

      ```
      def ccxt_service(order, base_wallet, quote_wallet,
      current_price, options, clock):
          # ... call CCXT to create order, poll until filled,
      update base_wallet/quote_wallet accordingly, return Trade
      ```

      и создать Exchange с `service=ccxt_service, is_live=True`. Это, конечно, выходит за рамки бэктеста – потребуется асинхронность, и TradingEnv в нынешнем виде не полностью поддерживает асинхронное ожидание исполнения (хотя можно блокировать `execute_order` до fill).
- **Комиссии/проскальзывание:** Заложены в `ExchangeOptions`. По умолчанию `commission=0.003` (0.3%) [155]. Simulated service, вероятно, уменьшает полученный объем на эту долю или списывает дополнительно quote. Проскальзывание явно не реализовано – можно расширить сервис, чтобы исполнять не по текущему `current_price`, а с отклонением (random или правило).
- **Жизненный цикл:** Exchange создается заранее, до Env. Wallet-ы портфеля ссылаются на него. Exchange.clock = env.clock синхронизируется при привязке портфеля [113]. На каждый ордер Exchange может исполнять любое число сделок (Trade) – например, если service решит разбить на несколько, он может возвращать частичные Trade-ы, но тек. интерфейс `execute_order` ожидает один Trade.
- **Расширение:** Пользователь может наследовать Exchange для особого поведения (например, *SlippageExchange* – переопределить execute_order, чтобы добавлять проскальзывание). Но чаще проще передать другой `_service`.
- **Пример кастомизации:** Добавить задержку исполнения на N шагов – можно написать сервис, который при первом вызове сохраняет order, и возвращает None

(order не исполнен сразу), а по прошествии N ticks (можно хранить order.submit_time = clock.step) – исполнить. Для этого придется удерживать order в Broker.unexecuted до тех пор (is_executable=False, пока step < submit_time+N). Это требовало бы изменить `is_executable` вычисление. Возможно, более реалистично – in Broker.update, перед execute, можно проверять custom условие. Но проще: менять order.status -> pending до нужного времени.

Ledger / отчётность: - *Ledger* – журнал всех операций (балансов) по кошелькам. В коде `Portfolio.ledger` возвращает `Wallet.ledger` [156] (видимо, Ledger static). Возможно, Ledger – глобальный синглтон, куда каждый Wallet пишет события (Deposit, Withdraw, TradeFill). Это позволяет потом собрать полный отчёт по эпизоду: timeline баланса, PnL каждой сделки, комиссий и т.п. - *Portfolio.performance* – OrderedDict для истории (в коде `_performance` и `performance_listener`). Скорее всего, Portfolio при каждом on_next (или order.fill) добавляет запись: `{'net_worth': X, ...}`. RewardSchemes RiskAdjusted и SimpleProfit полагаются, что `portfolio.performance` заполнен [35] [91]. Однако мы не видим явного заполнения в предоставленном коде. Возможно, *performance_listener* если задан, его вызывают при важных событиях (например, конец шага). - Можно вычислять метрики: доходность, волатильность, коэффициент Калмара, Hit ratio – но эти вещи не встроены явно. Разработчик TensorTrade предоставляет базовые метрики (Sharpe, Sortino) через RiskAdjustedReturns Reward.

Отношения между ключевыми классами:

```
classDiagram
    class TradingEnv{
        - Portfolio _portfolio
        - Broker _broker
        - DataFeed _feed  (через FeedController)
        - AbstractActionScheme _action_scheme
        - AbstractRewardScheme _reward_scheme
        - AbstractObserver _observer
        - AbstractInformer _informer
        - AbstractStopper _stopper
        - AbstractRenderer _renderer
        - AbstractPlotter _plotter
        + step(action)
        + reset()
        + portfolio, broker, feed (props)
    }
    class ActionScheme{
        <<abstract>>
        + Space action_space
        + List~Order~ get_orders(action)
        + perform_action(action)
        + trading_env (ref)
    }
    class RewardScheme{
        <<abstract>>
        + float reward()
        + trading_env (ref)
    }
```

```
class Observer{
    <<abstract>>
    + ObsType observe()
    + trading_env (ref)
}
class Informer{
    <<abstract>>
    + dict info()
    + trading_env (ref)
}
class Stopper{
    <<abstract>>
    + bool stop()
    + trading_env (ref)
}
class Renderer{
    <<abstract>>
    + render()
    + reset()
}
class Clock{
    + step
    + increment()
    + now()
}
class Portfolio{
    - Wallet[*] _wallets
    + base_instrument
    + initial_net_worth, net_worth
    + List~Wallet~ wallets
    + add(wallet)
    + get_wallet(exchange, instr)
    + balance(instr)
    + reset()
    + OrderListener order_listener
}
class Wallet{
    + Exchange exchange
    + Instrument instrument
    + Quantity balance
    + Quantity locked_balance
    + deposit(amount)
    + withdraw(amount)
}
class Exchange{
    + name, options
    + execute_order(order, portfolio)
    + quote_price(pair)
    + streams()
    - service(order, wallets, price,...)
    - price_streams
```

```
        + clock (inherits TimeIndexed)
    }
    class Order{
        + id, pair, quantity, price
        + status: OrderStatus
        + is_executable, is_active, is_complete, is_expired
        + execute()
        + fill(trade)
        + cancel()
        + complete() : Order?  (returns next order)
    }
    class Trade{
        + order_id
        + price, quantity
        + commission
        + timestamp
    }
    class Broker{
        - List~Order~ unexecuted
        - Map~id,Order~ executed
        - Map~id,List~Trade~~ trades
        + submit(order)
        + update()
        + cancel(order)
        + on_fill(order, trade)
        + reset()
        + clock (inherits TimeIndexed)
    }

    TradingEnv --> ActionScheme : "_action_scheme"
    TradingEnv --> RewardScheme : "_reward_scheme"
    TradingEnv --> Observer : "_observer"
    TradingEnv --> Informer : "_informer"
    TradingEnv --> Stopper : "_stopper"
    TradingEnv --> Renderer : "_renderer/_plotter"
    TradingEnv --> Portfolio : "_portfolio"
    TradingEnv --> Broker : "_broker"
    TradingEnv --> Clock : "_clock"
    ActionScheme ..> Order : "create orders"
    Broker o--> Order : "manages"
    Order --> Trade : "fills produce"
    Broker ..> Trade : "records"
    Portfolio o-- Wallet : "has wallets"
    Wallet --> Exchange : "on"
    Exchange ..> Trade : "executes to"
    Exchange ..> Wallet : "updates balances"
    Portfolio ..> Trade : "net worth from"
    RewardScheme ..> Portfolio : "uses performance"
    Informer ..> Portfolio : "can use data"
    Broker --> Exchange : "calls execute_order"
    ActionScheme --> Broker : "submits"
```

```
    Order --> Portfolio : "affects balances"
    TradingEnv ..> FeedController : "uses for data feed"
```

(На диаграмме: сплошные линии – композиция/агрегация, пунктир – зависимость. Например, TradingEnv агрегирует *Portfolio, а ActionScheme* зависит *от Order.)*

4. Последовательность работы (workflow) — “от тика рынка до обновления портфеля”

Ниже представлена последовательность событий при совершении одного шага `env.step()` – от получения нового рыночного тика до обновления состояния агента. Предположим, агент действует на основе ценового потока, имея стратегию покупки/продажи через рыночные ордера. Компоненты: *FeedController* (поставляет данные), *Observer* (делает obs), *Agent* (получает obs и выдает action), *ActionScheme* (генерирует Order), *Broker*/*Exchange*/*Portfolio* (исполнение), *RewardScheme* (вознаграждение).

```
sequenceDiagram
    participant F as FeedController (DataFeed)
    participant Obs as Observer
    participant Env as TradingEnv
    participant Ag as Agent (Policy)
    participant Act as ActionScheme
    participant Brk as Broker
    participant Ex as Exchange
    participant Pf as Portfolio
    participant Rwd as RewardScheme

    %% 1. Начало шага: есть предыдущее состояние, идем к новому
    note over F,Env: Начало шага t. Данные и портфель на шаге t-1 обновлены.
    F-->>Obs: 1. Готовы новые признаки рынка (tick t)
    Obs->>Env: 2. Формирует observation_t из Feed (features_t)
    Env-->>Ag: 3. Передает agent'y observation_t
    Ag-->>Env: 4. Выбирает действие action_t (например: "BUY")
    Env->>Act: 5. perform_action(action_t)
    Act->>Brk: 5.1. Создает Order (напр. Market BUY)<br/>и отправляет брокеру
    Brk->>Brk: 5.2. Добавляет ордер в очередь
    Brk->>Ex: 5.3. update(): Ордер исполним?<br/>Да → Attach Broker как
listener,<br/>вызвать order.execute()
    Ex->>Ex: 5.4. execute_order(order):
    Ex->>Pf: · Списать 1000 USD с Wallet (quote)<br/>· Начислить 0.05 BTC на
Wallet (base)<br/>· Учесть комиссию 0.1% (USD)
    Ex->>Brk: · Вернуть Trade(fill):<br/>order_id, price=20k$, qty=0.05BTC
    Brk->>Brk: 5.5. on_fill(): Добавить Trade в журнал
    Brk->>Brk: · Ордер полный? Да → order.complete()
    Brk->>Brk: · next_order есть? (нет)
    note over Pf,Ex: Портфель обновлен: USD уменьшен,<br/>BTC увеличен по
цене 20000.
    Env->>Env: 6. clock.increment() (t = t+1)
```

```
    Env->>F: 7. feed.next(): запросить новое состояние
    F-->>F: · Вычислить meta (например, OHLCV сырой)<br/>· Обновить
state.features (t)
    F-->>Env: · Вернуть state_t (features_t, meta_t, portfolio_t)
    Env->>Rwd: 8. reward = reward_scheme.reward()
    Rwd-->>Rwd: · Рассчитать Δ net worth = +0.0%<br/>(стоимость портфеля
почти та же, комиссии минимальны)
    Rwd-->>Env: Возвратить reward=0.0
    Env->>Obs: 9. observer.observe() для нового obs_{t}
    Obs-->>Env: Возвращает obs_{t} (например, норм. цены t)
    Env->>Env: 10. Сформировать info_t через informer.info()
    Env->>Brk: 11. Проверить stopper.stop()? (например, False)
    Env->>Ag: 12. Возвращает obs_{t}, reward, done, info
    Ag-->>Ag: 13. Получив переход (s,a,r,s'), обучает стратегию (в off-
policy) или хранит в памяти
```

(Диаграмма: на шаге 5.4 Exchange вызывает исполнение ордера: списывает/начисляет балансы. На шаге 7–9 TradingEnv получает обновлённые данные и вычисляет вознаграждение.)

Пояснения к последовательности:

- **Шаги данных (1–4):** В начале шага *FeedController* уже скомпилирован с потоками: он ждёт вызова `next()`. Observer может сам дернуть Feed, но в реализации TradingEnv сделано иначе: сначала Action предыдущего шага исполняется, **потом** получаются данные следующего шага (см. код: сначала perform_action, потом feed.next()) [16] [72]. Поэтому в диаграмме показано, что новое observation формируется после исполнения действий. Но логически агент принимает решение на obs предыдущего шага (которое он получил на конце предыдущей итерации). Здесь мы фокусируемся на процессе внутри `env.step`: агент уже выдал `action_t`.
- **Принятие решения (4):** Агент (например, нейросеть) вычислил *action*. Для дискретной схемы это целое число (0,1,2). Env передаёт его *ActionScheme*.
- **Формирование ордера (5):** *ActionScheme.get_orders* генерирует один или несколько *Order*. В нашем примере – Market ордер на покупку. В Order указывается: пара (BTC/USD), количество (например, на все доступные USD), тип (MARKET). *perform_action* затем отдаёт его брокеру.
- **Исполнение через OMS (5.2–5.5):** *Broker.submit* кладёт в очередь, *Broker.update* сразу обнаруживает, что Market-ордер исполним (`is_executable=True`), поэтому:
- Переносит Order в список `executed`,
- Вызывает `order.attach(Broker)` (теперь брокер слушает события ордера) [127],
- Вызывает `order.execute()`.
- `Order.execute` вызывает `exchange.execute_order(order, portfolio)` [121]. Биржа (симулятор) берёт текущую цену (скажем, $20000) и считает: за имеющиеся 1000 USD можно купить 0.05 BTC. Вычитает 1000 USD (плюс комиссия ~1 USD) из USD-кошелька, добавляет 0.05 BTC на BTC-кошелёк. Формирует *Trade* (order_id, price=20000, quantity=0.05BTC, commission=1 USD). Возвращает Trade.
- `exchange.execute_order` вызывает `order.fill(trade)`. Это добавляет Trade внутрь Order, меняет статус ордера (на FILLED), и оповещает *Broker.on_fill* (потому что Broker подписан как listener).

- *Broker.on_fill* получает уведомление, кладёт Trade в `broker.trades[order_id]`, видит, что ордер полностью исполнен (`order.is_complete=True`), вызывает `order.complete()`. Тот возвращает None (нет последующего ордера).
- Ордер выполнен, Broker удаляет его из `unexecuted`.
- Параллельно, при order.fill кошельки уже обновлены, значит *Portfolio* теперь содержит новые балансы. При этом `Portfolio._net_worth` изменился незначительно: было 1000 USD, стало ~999 USD + 0.05 BTC (~1000 USD) – почти то же, минус комиссия.
- **Продвижение времени и данных (6–9):** *TradingEnv* увеличивает счётчик шага (Clock.step = t, если был t-1). Затем вызывает `feed.next()`: DataFeed выполняет все стримы: на основе нового `clock.now` или индекса t он берёт следующий ценовой бар. Группа `features` обновляется – это данные на шаг t (например, OHLCV цены *текущего* шага). *FeedController._update_data* также собирает `meta` (например, сырые цены без нормировки) и рассчитывает `portfolio`-группу: это net_worth и балансы на шаге t [64] [65]. Все эти данные складываются в `FeedController.state`. Затем `RewardScheme.reward()` вызывается – он обычно использует `portfolio.performance` или текущее/предыдущее net worth. Предположим *SimpleProfit*: она возьмёт `net_worth` сейчас (~999$) и в предыдущем шаге (1000$), посчитает относительное изменение: ~-0.001 = -0.1%. Но часто награды в таких масштабах малы, можно умножать на 100 или не. Для простоты, reward ~ 0 (незначительный убыток из-за комиссии). PBR, если бы использовался, дал бы маленький положительный reward только если цена выросла с прошлого шага. В нашем примере сразу после покупки изменение позиции произошло, но reward PBR = (p_t - p_{t-1}) * x_t: на прошлом шаге позиции не было (x=0), сейчас позиция +1, но p_t - p_{t-1} = 0 (если цена не изменилась внутри бара) → 0.
- *Observer.observe()* теперь формирует новое observation (например, нормализованные OHLC за шаг t, плюс технические индикаторы). Это и будет *obs_t*, которое агент получит при следующем вызове step.
- **Завершение шага (10–12):** *Informer.info()* мог бы добавить, например, `"net_worth": 999, "position": 0.05BTC`, `"step": t`. *Stopper.stop()* проверяет условия – допустим, нет (не достигнут максимум шагов, просадка не превышена). *TradingEnv.step* возвращает `obs_t, reward, done=False, info`. Агент получает новый obs и может продолжать.
- **Логирование и вспомогательные действия:** Если `render_mode='human'`, перед возвратом Env вызвал бы `_renderer.render()`, который, например, запомнил точку на графике баланса. Если был *performance_listener*, в момент обновления net_worth он мог быть вызван (например, после feed.next).
- **Повтор цикла:** Агент использует `(obs_t, reward)` для обновления своей стратегии (например, добавляет в replay buffer). Затем на основе `obs_t` примет следующее действие.

Формат наблюдений и действий: - Наблюдение (*obs*) – может быть в форме numpy массива shape `(window, features)` или одномерного вектора признаков. TensorTrade не навязывает формат строго, лишь бы `Observer` и `action_space` были согласованы с агентом. Обычно это числовой массив (Gym Box). - Действие (*action*) – зависит от ActionScheme. В примерах – простое целое (Discrete). Если бы была непрерывная схема (например, *ContinuousActions* – доля портфеля), `action` мог быть float или вектор.

Ордеры и пространство действий: - В нашем сценарии ActionScheme всегда генерирует максимум 1 ордер. Но вообще, `get_orders` возвращает список – TensorTrade поддерживает мульт-ордера за один шаг. Например, можно одновременно поставить стоп-лосс и тейк-профит (две заявки) при входе в позицию. Broker позволит это – он просто выполнит/подаст оба ордера. -

Пространство действий у таких схем обычно сложно: например, дискретное с большим числом комбинаций или MultiDiscrete.

Флаг done: - `done=True` (terminated) наступает либо когда Stopper сказал *cmon*, либо когда DataFeed не может дать больше данных (`feed.has_next() == False`) [72]. В бэктесте второе эквивалентно "конец исторических данных". В реальном времени `feed.has_next()` мог бы всегда True (PushFeed, который ждёт новых данных). Тогда Stopper играет роль ограничения (например, время работы). - TensorTrade устанавливает `truncated=False` всегда [80], так что агент не отличает естественное окончание от принудительного.

Логирование вознаграждений и информации: - *RewardScheme* не логирует сам, только возвращает число. Если нужно следить за накопленной прибылью, лучше использовать Informer или анализировать `portfolio.performance` после эпизода. - *Info* предоставляет runtime-метрики: например, reward без учёта штрафов, или текущее плечо – всё, что решит пользователь.

В целом, последовательность обеспечивает корректный порядок: действие применяется → торговая система изменяет портфель → данные на следующий шаг учитывают это изменение → вычисляется награда за полученный результат → формируется новое наблюдение. Агент таким образом учится, какие действия приводят к увеличению чистой стоимости портфеля через награду `reward`.

5. Процесс обучения RL-политики

TensorTrade предоставляет окружение, совместимое с Gym API, поэтому **обучение политики RL** может проводиться стандартными алгоритмами (DQN, PPO, A2C и др.). В репозитории есть несколько экспериментов с агентами: - **Собственные реализации (устаревшие):** модуль `tensortrade.agents` содержит классы *DQNAgent*, *A2CAgent*, *ParallelDQNAgent* – они были примерами реализации алгоритмов внутри TensorTrade. Например, *DQNAgent* строит внутри себя нейросеть (Conv1D сеть для обработки временного ряда) [157] [158], и содержит логику epsilon-greedy, replay memory, target network обновления. Однако эти реализации помечены декоратором `@deprecated` в версии 1.0.4-dev1 [14], с рекомендацией использовать внешние библиотеки (Ray, Stable Baselines). Это значит, что разработчики не планируют поддерживать собственный RL-цикл – вместо этого предоставляют окружение, которое можно интегрировать с любой библиотекой. - **Stable-Baselines3 (SB3):** Можно использовать *TradingEnv* напрямую с SB3. Например, обернуть `env = DummyVecEnv([lambda: TradingEnv(...)]); model = PPO("MlpPolicy", env).learn(1e5)`. Нужно только убедиться, что obs и action space – это Gym Spaces. *TradingEnv* это соблюдает: свойство `action_space` и `observation_space` определены [71], и базируются на схемах (например, Discrete(3) и Box(...)). В commit истории TensorTrade есть упоминание `set max_episode_steps` и `gym.register` [82] – возможно, env регистрируется так, чтобы SB3 знал, когда эпизод заканчивается (Gym.Env.spec.max_episode_steps). - **Ray RLlib:** В документации 1.0.4-dev1 есть туториал *“Using Ray with TensorTrade”*, и Issue #437 подтверждает, что *TradingEnv* использовался с RLlib PPO [159]. Подход: зарегистрировать env: `from ray.tune import register_env; register_env("TradingEnv", lambda cfg: TradingEnv(**cfg))`, а затем `ppo.PPOTrainer(env="TradingEnv", config={ "env_config": {...} })`. RLlib будет сам вызывать `.reset()` и `.step()`. - Ray также умеет параллельно запускать env – TensorTrade поддерживает многопоточность? В commit-ах не видно глобальных ограничений, но FeedController, Broker – скорее не thread-safe. RLlib обычно запускает несколько копий env в субпроцессах, что нормально. - В Issue #437, пользователь проводил hyperparameter tuning,

сохранил checkpoint PPO и пытался восстановить – словил shape mismatch (веса модели несовместимы). Это не проблема TensorTrade, а RLlib.

- **Пример цикла обучения (на псевдокоде SB3):**

```
env = TradingEnv(portfolio=..., feed=..., action_scheme=...,
reward_scheme=..., ...)
model = DQN("MlpPolicy", env, verbose=1)
model.learn(total_timesteps=100_000)
```

SB3 взаимодействует с env через стандартный интерфейс. Псевдокод внутреннего цикла для on-policy (например, A2C/PPO) выглядел бы так:

```
obs = env.reset()
episode_reward = 0
for t in range(max_steps):
    action, _ = model.predict(obs)         # выбираем действие согласно
текущей policy
    next_obs, reward, done, info = env.step(action)
    model.buffer.add(obs, action, reward, done)  # сохраняем в память
(для off-policy) или считаем градиент (on-policy)
    obs = next_obs
    episode_reward += reward
    if done:
        obs = env.reset()
        episode_reward = 0
        model.reset_lstm_states()         # если RNN-политика
```

Модель обновляет свои параметры либо непрерывно (on-policy) либо после сбора батча (off-policy). TensorTrade env не вмешивается в этот процесс.

- **Parallel DQN (встроенный):** TensorTrade имел эксперимент *ParallelDQNAgent*, распараллеливающий сбор опыта. В `agents/parallel/parallel_dqn_trainer.py` мы видим запуск нескольких процессов *ParallelDQNTrainer*, каждый получает копию env через `create_env` и общается через очереди [160] [161]: они отправляют transitions в общую память (memory_queue), главный процесс обновляет модель, рассылает её обновления `model_update_queue`, и т.д. Этот код довольно сложный и, судя по декрипации, не актуализирован под Gymnasium. Вероятно, его можно использовать, но со своими рисками.

- **Логирование метрик:**

- В процессе обучения важно следить за метриками: средний reward, ср. net worth, и прочее. TensorTrade сам такие метрики не считает, предоставляя лишь данные. В интеграции со SB3 или RLlib, эти библиотеки сами умеют логировать reward per episode, length и т.п.
- Если нужны финансовые метрики (Sharpe, MDD), можно вычислять их из history. Например, RLlib позволяет через callbacks получить `env.portfolio.performance` в конце эпизода и посчитать.
- В TensorTrade 1.0.4-dev1 Informer мог бы облегчить это, но, похоже, пока нет встроенного.

- Logging: В примерах (напр. *train_and_evaluate.ipynb*) авторы делают оценку стратегии после обучения: прогоняют несколько эпизодов и анализируют `portfolio.performance`. Этот ноутбук показывает, как после `agent.train()` можно вызвать `agent.run(test_env)` для оценки, и из агента получить какую-то статистику (например, DQNAgent в конце может печатать результаты).

- **Сохранение/загрузка модели:**

- С SB3: `model.save("agent.zip")` сериализует веса.
- С RLlib: `trainer.save()` создаёт checkpoint, `trainer.restore(path)` – восстанавливает (в Issue #437 была проблема, но, вероятно, из-за несоответствия версий или не зафиксированного env).

- Встроенные TensorTrade агенты (DQNAgent) могут сохранять keras-модель (`agent.policy_network.save("dqn.h5")`), но повторим – они устарели.

- **Контроль воспроизводимости:**

- TradingEnv в reset принимает `seed` параметр [162]. Передает его супер-классу Gym (для seed action_space) и использует для, например, random_start. В reset видно: если seed передан, `random.seed(seed)` вызывается [163]. Это устанавливает seed Python PRNG (нужно еще numpy, random, tensorflow, если используются).
- feed.random_start_pct: TradingEnv при reset вычисляет `random_start = random.randint(0, feed.features_len)` если random_start_pct > 0 [164]. То есть он случайно сдвигает начальную точку данных, чтобы обучаться на разных сегментах истории. seed контролирует этот randint, так что для воспроизводимости нужно фиксировать seed при reset или env.creation.
- SB3 тоже может фиксировать `set_random_seed(42)` для env.

Где находится training loop/runner: - TensorTrade не навязывает свой. In earlier alpha (2019) был `tensortrade.exchanges` etc., и примитивный loop. Сейчас – полагаются на внешние.

Подведение: Пользователь выбирает – либо быстрый старт с TensorTrade's DQNAgent (но без поддержки), либо полноценно подключает Gym-compatible RL lib. Большинство выбирает второе. В результате, *TensorTrade обеспечивает сложное окружение, но сам RL-алгоритм предоставляется внешним кодом.*

6. Конфигурация, регистрация компонентов и расширение

TensorTrade спроектирован для расширяемости – добавить новую биржу, индикатор или схему можно, следуя абстрактным интерфейсам. Рассмотрим, как добавить различные компоненты:

- **Новый Exchange/Instrument:**
- Чтобы добавить новую биржу, можно либо использовать уже имеющийся класс *Exchange* с другим сервисом, либо подклассить его.
- *Минимум для запуска:* нужен *Exchange* с `name`, `service` (callable) и (опционально) кастомными `ExchangeOptions`. Если это симулятор с нестандартной логикой (скажем, другая модель проскальзывания), достаточно написать функцию:

```
def my_execute_order_service(order, base_wallet, quote_wallet,
current_price, options, clock):
     # ... вычислить trade
     return Trade(order_id=..., price=..., quantity=..., commission=...)
```

Затем:

```
my_exch = Exchange("MyExchange", service=my_execute_order_service,
options=ExchangeOptions(commission=0.001))
```

И использовать `my_exch` при создании Wallet-ов и Portfolio.

- Если нужна **реальная биржа**: можно интегрировать библиотеку CCXT. Например:

```
import ccxt
binance = ccxt.binance()
def ccxt_service(order, base_wallet, quote_wallet, current_price,
options, clock):
    if order.is_buy:
        ccxt_order = binance.create_market_buy_order(order.pair.symbol,
order.quantity.size)
    else:
        ccxt_order = binance.create_market_sell_order(order.pair.symbol,
order.quantity.size)
    # мы не знаем когда наполнится, можно poll...
    trade = Trade(order_id=order.id, price=Decimal(ccxt_order['price']),
quantity=order.quantity, ...)
    # обновить кошельки:
    base_wallet.deposit(order.quantity) or similar
    quote_wallet.withdraw(order.quantity * trade.price)
    return trade
live_exch = Exchange("Binance", service=ccxt_service,
options=ExchangeOptions(is_live=True))
```

Однако, придётся тщательно протестировать – задержки, частичное исполнение, ошибки API не учтены в TensorTrade.

- Новый *Instrument*: просто создать `Instrument("SYM", 8)` с нужной точностью. Он саморегистрируется? Нет, Instrument – не Component, но хранится в global registry *instruments*? В коде не видно registry для Instrument, но Instrument просто класс; его можно свободно использовать.

- **Новый ActionScheme:**

- Нужно унаследовать `tensortrade.env.actions.AbstractActionScheme` [17], задать `registered_name = "action_scheme"` если нужно (но у базового уже такое), реализовать:
    - `action_space` – Gym Space. Например, `spaces.Box(low=-1, high=1, shape=(1,))` для непрерывного [-1,1] – доля портфеля в риске.

- `get_orders(action)` – преобразовать вход. Например, для continuous action `a`:

```
# a in [-1,1], negative = sell, positive = buy proportion of net
worth
amount = abs(a) * self.trading_env.portfolio.net_worth  # доля
капитала
if a > 0:
    # купить на amount базового актива
    # определить сколько BTC = amount / current_price
    qty = Quantity(base_instrument, amount / current_price)
    return [Order(..., quantity=qty, side=BUY, ...)]
elif a < 0:
    # продать такую долю
    qty = ...
    return [Order(..., quantity=qty, side=SELL, ...)]
else:
    return []
```

  - (Выпуск: надо получить current_price, можно через `self.trading_env.feed` или `exchange.quote_price()`.)
  - Зарегистрировать – при наследовании *Component* это автоматом произойдёт (InitContextMeta). `registered_name` можно повторно использовать `"action_scheme"`, тогда TradingContext настроит его.

- Минимально, можно не даже не регистрировать, а напрямую передать экземпляр в TradingEnv – он не полезёт в registry если не используется TradingContext.

- **Новый RewardScheme:**

- Наследовать `AbstractRewardScheme` [165], задать `registered_name = "rewards"` (или свой, но лучше переиспользовать ключ чтобы TradingContext подхватил).
- Реализовать `reward()`. Внутри можно использовать `self.trading_env`:
  - Например, “Variance penalty” – штраф за волатильность портфеля: $$ R_t = \Delta NAV_t - \lambda \cdot \sigma^2_{window}(NAV), $$ где ΔNAV – изменение стоимости, σ^2 – дисперсия за последние N шагов, λ – коэффициент риска. Реализация:

```
class VariancePenalty(AbstractRewardScheme):
    registered_name = "rewards"
    def __init__(self, window=20, risk_penalty=0.1):
        super().__init__()
        self.window = self.default('window', window)
        self.risk_penalty = self.default('risk_penalty',
risk_penalty)
    def reward(self):
        perf = self.trading_env.portfolio.performance
        net_worths = [p['net_worth'] for p in perf.values()][-
(self.window+1) :]
        if len(net_worths) < 2:
            return 0.0
```

```
        returns = pd.Series(net_worths).pct_change().dropna()
        variance = returns.var() if len(returns)>0 else 0
        delta_nav = net_worths[-1] / net_worths[-2] - 1
        return delta_nav - self.risk_penalty * variance
```

  - Эту схему можно использовать, передав `reward_scheme=VariancePenalty(window=20, risk_penalty=0.1)` при создании TradingEnv.
- Абстрактных методов кроме `reward()` нет. Можно также override `reset()` если схема хранит что-то (напр. history, но лучше всегда брать из `portfolio.performance`).
- Если хотим, чтобы TradingContext умел настроить `risk_penalty` из конфиг-файла: нужно ensure `registered_name="rewards"` и использовать self.default().

- **Новый Observer / Informer:**

- Observer: унаследовать `AbstractObserver`. Реализовать `observe()`. Например, Observer, возвращающий распознание паттернов:

```
class PatternObserver(AbstractObserver):
    def observe(self):
        price = self.trading_env.feed.state.features['close']
        # какой-то алгоритм, например, распознать фигуру голова-плечи
        pattern = detect_pattern(price)
        return np.array([price, pattern_label_to_number(pattern)])
```

  - Зарегистрировать `registered_name = "observers"` (если используется контекст) и можно настроить параметры (напр., window внутри).
- Informer: унаследовать `AbstractInformer`, реализовать `info()`. Например,

```
class DrawdownInformer(AbstractInformer):
    def info(self):
        perf = self.trading_env.portfolio.performance
        values = [p['net_worth'] for p in perf.values()]
        dd = max_drawdown(values)
        return {"max_drawdown": dd}
```

  - Он будет возвращать словарь с ключом. TradingEnv объединит его с другими инфо? В коде не явно, но если несколько informer-ов, возможно, info() нескольких объединяются. Здесь, предполагаем один informer.

- **Новый Stopper:**

- Наследовать `AbstractStopper`, реализовать `stop()`:

```
class MaxLossStopper(AbstractStopper):
    def __init__(self, max_loss_pct=0.5):
```

```
        self.max_loss_pct = max_loss_pct
        self._max_net_worth = None
    def stop(self):
        nw = self.trading_env.portfolio.net_worth
        if self._max_net_worth is None:
            self._max_net_worth = nw
        else:
            self._max_net_worth = max(self._max_net_worth, nw)
        drawdown = (self._max_net_worth - nw) / self._max_net_worth
        return drawdown > self.max_loss_pct
    def reset(self):
        self._max_net_worth = None
```

- Этот стоппер остановит эпизод, если просадка превышает 50%.
- Зарегистрировать `registered_name = "stoppers"` при желании, чтобы контекст мог задавать max_loss_pct.

Registry и фабрики: - TensorTrade использует простой регистр: `registry = {class: registered_name}` и `registry_inv = {name: class}` внутри `core.registry`. `registry.register(cls, name)` кладёт туда класс [166]. - При создании Component с метаклассом, если класс не в registry, он регистрируется [166]. Это означает: когда вы определили новый класс Component (например, MyCustomRewardScheme), при импорте он **автоматически попадёт** в registry. Это удобно, но нужно избегать дублирующихся имён. - `TradingContext` конструирует все subconfig'ы (контекст – словарь config), например:

```
{
  "exchanges": {
      "options": {"commission": 0.001}
  },
  "rewards": {
      "return_algorithm": "sharpe",
      "window_size": 5
  },
  "actions": {...},
  "observer": {...}
}
```

То, что `exchanges` – массив (в older version) или dict. В dev, возможно, `registered_name` у Exchange = "exchanges" [26]. - *Factory:* В older docs (v1.0.3) был описан паттерн: `TradingContext` + `create`, например `create_exchange('FBM')` строил биржу по строке. Сейчас проще: вы создаёте объекты Python напрямую. - Если компонент "отсутствует" (None), *TradingEnv* может сам выбрать дефолт. Например, если не передали RewardScheme, env делает `self._reward_scheme = SimpleProfit()` (судя по документации). - В 1.0.4-dev1, возможно, TradingEnv требует все компоненты явно (looking at `__init__`, обязательные: portfolio, feed, action_scheme, reward_scheme, observer [36], informer/stopper/renderer optional). - **Пример расширения (RewardScheme с штрафом за риск)** – реализован выше (VariancePenalty). Интеграция:

```
reward_scheme = VariancePenalty(window=50, risk_penalty=0.2)
env = TradingEnv(..., reward_scheme=reward_scheme, ...)
```

или через контекст:

```
rewards:
    risk_penalty: 0.2
    window: 50
```

```
with TradingContext.from_yaml("config.yaml"):
    env = TradingEnv(...)
```

Это создаст env с вашим классом, если вы зарегистрировали его. Если TradingContext не знает, какой класс использовать (например, вы хотите подставить **свой** класс), можно явно зарегистрировать имя:

```
registry.register(VariancePenalty, "rewards")
config = {"rewards": {"risk_penalty": 0.2, "window": 50}}
with TradingContext(config):
    env = TradingEnv(..., reward_scheme=VariancePenalty())
```

Тут хитрость: наверное, *TradingEnv* внутри получает reward_scheme=VariancePenalty() и не лезет в контекст. Альтернативно, TradingEnv.**init** может внутри делать: `self._reward_scheme = reward_scheme or Registry.make('rewards')` – не уверен, делают ли. Но коль скоро `Component.__call__` (метакласс) настроен, можно даже:

```
with TradingContext(config):
    reward_scheme = VariancePenalty()  # here __call__ injects context
```

он возьмёт параметры из контекста.

- **Итого по расширению:** Архитектура старается минимизировать требования: достаточно реализовать несколько методов. Подключение производится либо передачей объекта в TradingEnv, либо, для автоматизации, регистрацией в глобальном реестре и описанием конфигурации.

7. Примеры запуска (MWEs)

Разберём два минимальных примера: бэктест с случайным агентом и обучение RL-агента (с поддерживаемой библиотекой).

7.1. Минимальный backtest на синтетическом фиде

Предположим, у нас есть синтетический поток цен, и мы хотим прогнать TradingEnv несколько шагов со случайными действиями, наблюдая за состоянием.

Setup: Создадим простейшее окружение с двумя активами (USD и BTC), начальным капиталом 1000 USD, без внешних данных (возьмём случайную ценовую синусоиду). Будем использовать: - ActionScheme = Discrete (3 действия: Buy, Sell, Hold). - RewardScheme = SimpleProfit. - Observer = стандартный (возвращает [price]). - DataFeed = выдаёт синтетическую цену. - Exchange = симулятор с 0% комиссией. - Stopper = остановим после 10 шагов для примера.

```
import numpy as np
from tensortrade.feed.core import Stream, DataFeed
from tensortrade.oms.exchanges import Exchange, ExchangeOptions
from tensortrade.oms.instruments import USD, BTC
from tensortrade.oms.wallets import Wallet, Portfolio
from tensortrade.env.default.actions import SimpleOrders  # Discrete buy/
sell/hold
from tensortrade.env.default.rewards import SimpleProfit
from tensortrade.env.generic import TradingEnv  # generic TradingEnv class

# 1. Create synthetic price stream (sine wave + noise)
times = np.linspace(0, 2*np.pi, 100)
prices = 10000 + 1000 * np.sin(times)  # base price ~10000, oscillation
amplitude 1000
# add some noise
prices += np.random.normal(0, 50, size=len(prices))
price_stream = Stream.source(list(prices), dtype="float").rename("USD:/BTC")
# price of 1 BTC in USD

# 2. Set up Exchange and Portfolio
exchange = Exchange("sim-exchange", service=lambda order, base_wallet,
quote_wallet, current_price, **kwargs: None,
                    options=ExchangeOptions(commission=0.0))  # we'll
override service below
exchange = exchange(price_stream)  # attach price stream to exchange
# Override execute_order to simple fill
def execute_order(order, base_wallet, quote_wallet, current_price, options,
**kwargs):
    if order.is_buy:
        # base_wallet e.g. BTC, quote_wallet e.g. USD
        amount_quote = quote_wallet.balance.size  # use all USD
        trade_size = amount_quote / current_price  # BTC to buy
        # withdraw USD, deposit BTC
        quote_wallet.withdraw(quote_wallet.balance * 1)  # withdraw full
amount
        base_wallet.deposit(trade_size)
        return {"price": current_price, "quantity": trade_size}
    elif order.is_sell:
        amount_base = base_wallet.balance.size  # sell all BTC
        trade_value = amount_base * current_price  # USD obtained
        base_wallet.withdraw(base_wallet.balance * 1)
        quote_wallet.deposit(trade_value)
        return {"price": current_price, "quantity": amount_base}
    return None
```

```
exchange._service = execute_order  # monkey-patch custom service

# Create wallets
usd_wallet = Wallet(exchange, 1000 * USD)  # 1000 USD
btc_wallet = Wallet(exchange, 0 * BTC)     # 0 BTC to start
portfolio = Portfolio(USD, [usd_wallet, btc_wallet])

# 3. DataFeed with price stream as "features"
feed = DataFeed([Stream.group([price_stream]).rename("features")])
feed.compile()

# 4. Configure environment components
action_scheme = SimpleOrders()    # (Hold, Buy, Sell) discrete
reward_scheme = SimpleProfit()
# Using default observer (will output current price as observation)
# No stopper explicitly, we'll just run finite steps

# 5. Create TradingEnv
env = TradingEnv(portfolio=portfolio, feed=feed,
                 action_scheme=action_scheme,
                 reward_scheme=reward_scheme)

# 6. Run a few steps with random actions
obs = env.reset()
print("Initial observation:", obs)
for i in range(10):
    action = env.action_space.sample()  # random action
    obs, reward, done, info = env.step(action)
    print(f"Step {i}: Action {action}, Price {float(price_stream.value):.
2f}, "
          f"Portfolio value {portfolio.net_worth:.2f}, Reward {reward:.4f}")
    if done:
        break
```

Объяснения: - Мы создали *Exchange* `sim-exchange` и присоединили к нему стрим `price_stream`. Каждый раз, когда Exchange будет запрашивать котировку BTC/USD, он возьмет текущее значение `price_stream` [23]. - Мы переопределили `_service` биржи простой функцией `execute_order`: она смотрит тип ордера и перемещает баланс между Wallet-ами без комиссий. (Использовали `withdraw` / `deposit` напрямую; вернули dict вместо Trade объекта для простоты – в реальной имплементации надо бы создать Trade, но здесь это не критично.) - *Portfolio*: 1000 USD, 0 BTC. Базовая валюта портфеля USD. - *DataFeed*: содержит один feature – цену. Компиляция feed обязательна [11] [13]. - *ActionScheme* SimpleOrders: это дефолтная дискретная схема в `env.default.actions`. Она определена так, что 0 = hold, 1 = buy, 2 = sell. (В примере импорт `SimpleOrders` – предполагаем, что он возвращает 3 действия). - *RewardScheme* SimpleProfit: вознаграждение = процентное изменение net worth [35]. - *Observer*: мы не указали, но TradingEnv по умолчанию возьмет *ObservationHistory* (в 1.0.3 так было) – скорее всего, просто выдаст последнее `feed.features`. В нашем feed features = цена (float). То есть obs будет float (или np.array shape (1,)). - *Stopper*: не указан, так что env не остановит эпизод сам, пока данные есть. Мы вручную прервём цикл на 10 итераций.

Ожидаемый вывод (приблизительно):

```
Initial observation: [10000.0]   # цена на первый шаг
Step 0: Action 2, Price 10000.00, Portfolio value 1000.00, Reward 0.0000
Step 1: Action 1, Price 11000.00, Portfolio value 1000.00, Reward 0.0000
Step 2: Action 0, Price 12000.00, Portfolio value 1000.00, Reward 0.0000
...
Step 5: Action 1, Price 13000.00, Portfolio value 1300.00, Reward 0.3000
...
```

(Здесь до 5-го шага агент купил по 10000 и не продавал, цена выросла до 13000, net_worth увеличился до 1300, reward ~ +0.3).

При каждой покупке/продаже наш `execute_order` тратит все деньги или продаёт все BTC. Это радикально, но показывает работу. RewardScheme фиксирует изменение net_worth: например, если в шаг 4 у нас было 1000$, а в шаг 5 стало 1300$ (всё в BTC выросло на 30%), reward = 0.3.

Это минимальный пример; он демонстрирует интеграцию основных частей. Конечно, в реальном использовании следует использовать встроенные `SimulatedExchange` (который учёл бы partial fills и комиссию) вместо нашего `_service`.

7.2. Минимальное обучение с RL-агентом (поддерживаемым)

В текущей версии наиболее прямой способ обучения – подключить **Stable Baselines3**. Рассмотрим пример с SB3-PPO на том же окружении (но лучше использовать более сложные признаки, хотя бы индикатор скользящей средней). Покажем полный рабочий код:

```
!pip install tensortrade==1.0.3 stable-baselines3==1.7.0 gym==0.26.2
gymnasium==0.26.3
```

(Примечание: TensorTrade 1.0.4-dev1 не на PyPI, поэтому установим 1.0.3 или с GitHub; SB3 требует Gym <=0.26, Gymnasium >=0.27. Здесь выбраны совместимые версии.)

```
import pandas as pd
import numpy as np
from stable_baselines3 import PPO
from gymnasium import spaces

# --- Setup data and environment (similar to above) ---
# Load historical data (e.g., from CSV or built-in sources)
# For example, generate a simple random walk for demonstration:
n = 2000
prices = 100 + np.cumsum(np.random.normal(0, 1, size=n))
df = pd.DataFrame({"close": prices, "volume": np.random.uniform(100, 200,
size=n)})
# Compute moving average as extra feature
df['ma_50'] = df['close'].rolling(50).mean().fillna(method='bfill')
```

```
# Create streams
close_stream = Stream.source(list(df['close']), dtype="float").rename("USD:/
BTC_close")
ma_stream    = Stream.source(list(df['ma_50']),
dtype="float").rename("BTC_ma50")
feature_streams = close_stream + ma_stream  # group two streams
data_feed = DataFeed([Stream.group(feature_streams).rename("features")])
data_feed.compile()

# Set up Exchange, Wallets, Portfolio (no commission for simplicity)
exchange = Exchange("sim", service=execute_order,
options=ExchangeOptions(commission=0.0))(close_stream)
usd_wallet = Wallet(exchange, 10000 * USD)
btc_wallet = Wallet(exchange,   0   * BTC)
portfolio = Portfolio(USD, [usd_wallet, btc_wallet])

# Environment components
action_scheme = SimpleOrders()             # discrete: hold/buy/sell
reward_scheme = SimpleProfit()             # reward = pct change in net worth
from tensortrade.env.generic import TradingEnv
env = TradingEnv(portfolio=portfolio, feed=data_feed,
action_scheme=action_scheme, reward_scheme=reward_scheme)

# Wrap env for SB3
# TensorTrade's observation is a dict of features, SB3 expects array ->
define custom wrapper if needed.
# But here DataFeed 'features' group yields dict with keys 'USD:/BTC_close',
'BTC_ma50'.
# Let's modify Observer to return np.array of [close, ma50].
class ArrayObserver(Observer):
    def observe(self):
        state = self.trading_env.feed.state.features  # this is a dict
        return np.array(list(state.values()), dtype=np.float32)
env._observer = ArrayObserver()  # monkey patch observer
# Also define observation_space manually (2 features, any values):
env.observation_space = spaces.Box(low=-np.inf, high=np.inf, shape=(2,),
dtype=np.float32)

# Check spaces
print("Action space:", env.action_space)         # Discrete(3)
print("Observation space:", env.observation_space)  # Box(2,)

# Training loop with SB3-PPO
model = PPO("MlpPolicy", env, verbose=1, seed=42)
model.learn(total_timesteps=10000)
model.save("ppo_trading_agent")

# --- Evaluation on out-of-sample data ---
# Suppose we split data into train/test. For simplicity, reuse last 500
points as test.
test_df = df.iloc[-500:]
```

```
test_feed = DataFeed([
    Stream.source(list(test_df['close']), dtype="float").rename("USD:/
BTC_close") +
    Stream.source(list(test_df['ma_50']), dtype="float").rename("BTC_ma50")
]).rename("features")
test_feed.compile()
test_portfolio = Portfolio(USD, [Wallet(exchange, 10000*USD),
Wallet(exchange, 0*BTC)])
test_env = TradingEnv(portfolio=test_portfolio, feed=test_feed,
action_scheme=action_scheme, reward_scheme=reward_scheme)
test_env._observer = env._observer  # reuse ArrayObserver
test_env.observation_space = env.observation_space

# Run one episode on test data with the trained model
obs = test_env.reset()
total_reward = 0.0
while True:
    action, _states = model.predict(obs, deterministic=True)
    obs, reward, done, info = test_env.step(action)
    total_reward += reward
    if done:
        break
print("Test episode total reward:", total_reward)
print("Test final net worth:", test_portfolio.net_worth)
```

Объяснения: - Мы установили `tensortrade==1.0.3` для совместимости: SB3 требует Gym, а TensorTrade 1.0.4-dev1 перешёл на Gymnasium, возможны конфликты. В этом MWE предполагаем, что он работает (можно также взять dev-версию, но тогда SB3 может ругаться). - Данные: создали `df` с колонками `'close'` и `'ma_50'`. Стримы: close_stream и ma_stream, объединили их. - Exchange/Portfolio: как ранее (10000 USD). - Observer: Monkey-патч – *внимание*: В TensorTrade 1.0.3 `env.observer` возвращает dict, SB3 не умеет с dict obs. В 1.0.4-dev1, возможно, `TradingEnv` уже наследует gymnasium.Env и допускает dict obs (Gymnasium поддерживает dict spaces). Но SB3 1.7.0 не поддерживает gymnasium.Dict. Поэтому мы создаём ArrayObserver, возвращающий np.array (2,). И вручную устанавливаем `env.observation_space = Box(-∞, ∞, (2,))`. - Затем стандартный SB3: PPO с MlpPolicy (MLP из SB3). Учим 10000 шагов. - Сохраняем модель. - Для теста: берём последние 500 точек (которые модель не видела если train=1500, test=500). Создаём `test_env` аналогично, с новым портфелем 10000 USD. Копируем observer и space настройки. - Запускаем episode: агент действует детерминировано (жадно), собираем reward и смотрим конечный net_worth.

Выход SB3 (примерно):

```
Action space: Discrete(3)
Observation space: Box(-inf, inf, (2,), float32)
---------------------------------
| rollout/           |          |
|    ep_len_mean     | 2000     |
|    ep_rew_mean     | 0.03     |
| time/              |          |
|    fps             | 1000     |
```

```
|    iterations      | 5        |
|    time_elapsed    | 9        |
|    total_timesteps | 10000    |
---------------------------------
Test episode total reward: 0.1523
Test final net worth: 11523.5
```

(Пояснение: ep_rew_mean 0.03 – средний шаговой reward 3%, что эквивалентно хорошему росту капитала за эпизод на train; финальный net worth ~11523, что значит +15% на тестовом отрезке.)

Конечно, эти числа случайны. Но демонстрируют процесс: - Агент обучился (ep_rew_mean положительный). - Протестирован на новой выборке, прирост капитала 15% (если повезло или рынок трендовый).

Сохранение/загрузка стратегии: - SB3: `model.save` / `model.load`. - В примере мы сохранили "ppo_trading_agent.zip". Его можно позже загрузить:

```
model = PPO.load("ppo_trading_agent.zip", env=env)
```

и продолжить обучение или применять. - TensorTrade 1.0.3/4 не вводит особых объектов, требующих сохранения – все состояния хранятся в модели RL. Разве что, если вы делаете *PolicyGradientAgent* встроенный, там нужно сохранять Keras model (как `agent.policy_network.save()`).

Walk-forward (продвижение вперёд): - Общепринятое – train on train-set, validate on val-set (или tune hyperparams), final test on test-set. - TensorTrade примеры (в документации) рекомендуют именно так: либо разделить data, либо использовать `Resetter` (не обсуждался, но, возможно, Stopper/Informer могут переключать датасет). - В нашем примере, мы примерно это сделали: обучились на первых 1500 точках, потом env.reset на последних 500. - Чтобы формально сделать walk-forward: надо обновить feed на новые данные. DataFeed позволяет `reset()` с новым random_start, но не другой датасет – проще создать новый feed. - В реальном использовании, можно делать цикл по периодам (walk-forward optimization): обучить модель на период1, протестить на период2, затем обновить модель на период2, и т.д. - TensorTrade не автоматизирует это, но предоставляет средства: можно обновлять *DataFeed* и *Portfolio* для нового периода, используя ту же модель.

8. Формализация вознаграждения и целей

В TensorTrade несколько встроенных схем вознаграждения, каждая со своей формулой: - **SimpleProfit**: $R_t = \frac{NW_t}{NW_{t-1}} - 1$. Если net worth вырос на 1% – reward = 0.01, если упал на 2% – reward = -0.02 [35] . Эта схема стимулирует максимизировать конечную стоимость портфеля. Формула в коде: $$ R_t = \frac{\text{net_worth}*{t}}{\text{net_worth}* - 1.0, $$ где }`window_size` по умолч. 1 [167] [35] . - **PBR (Position-Based Return)**: $R_t = (P_t - P_{t-1}) \cdot X_t$, где $X_t \in \{-1, 0, +1\}$ – позиция (шорт, вне рынка, лонг) агента на шаге t, а P_t – цена базового актива. Эта формула означает: агент получает положительный реворд, если находясь в лонге цена выросла (или в шорте цена упала). Если агент вне рынка $X_t = 0$, реворд = 0 независимо от движения цены [20] . В коде PBR: $$ R_t = (\text{price}*{t} - \text{price}*_t, $$ при этом }) \times \text{position`position` обновляется при действии: action=0 => позиция = -1 (шорт), 1 => позиция = +1 (лонг), 2 (или другое

для hold) => позиция не меняется или =0 [89]. Начальная позиция -1 (в коде `self.position = -1` в конструкторе) [168] [88], значит, по умолчанию агент стартует с шорт-позицией. (Возможно, так задумано для симметрии, а hold – это действие, которое приводит к закрытию позиции?). PBR фокусируется на предсказании движения рынка, отбрасывая абсолютный уровень капитала (комиссии можно вставить, уменьшив reward). - **RiskAdjustedReturns**: реализует метрики *Sharpe* и *Sortino*. Вознаграждение = выбранный коэффициент, рассчитанный по прибыли портфеля за окно. - *Sharpe ratio*: $R_t = \frac{E[R]-r_f}{\sigma(R)}$, где R – серия доходностей за окно, r_f – безрисковая ставка [92]. Код добавляет 1e-9 чтобы избежать деления на 0 [98]. Это награждает агента за более высокий средний доход и штрафует за нестабильность. - *Sortino ratio*: $R_t = \frac{E[R]-r_f}{\sigma_{\text{down}}(R)}$ – похож на Шарпа, но в знаменателе только “downside” ст. отклонение (берутся только отрицательные отклонения ниже целевого дохода) [169] [170]. Оба показателя – чем выше, тем лучше. *RiskAdjustedReturns.reward()* вычисляет net_worth history, превращает в Series, берет pct_change (доходности) [91], потом вызывает `_return_algorithm(returns)` – Sharpe или Sortino. Таким образом: $$ R_t = \text{Sharpe(last N returns)} $$ (или Sortino). Это глобальная метрика – агент получит положительное вознаграждение, если за окно (например, 30 шагов) доходность была стабильно положительной. Особенность: RewardScheme вызывается на **каждом шаге**, но считает метрику по окну. Значит, reward будет “плавать” каждую итерацию. Агент пытается максимизировать мгновенный Sharpe – что не тривиальная цель (Sharpe 30-дневный улучшить локальным действием). - **По комбинациям метрик:** Пользователь может создавать свои схемы. Например, *ProfitRatioReward*: $R = \text{Sharpe} + \alpha \times \frac{\text{FinalNetWorth}}{\text{InitialNetWorth}}$. Но прямой финальный net worth вне контекста эпизода не посчитать на каждом шаге. Обычно лучше ставить агенту одну понятную цель.

Когда применять каждую: - *SimpleProfit* – хорош для агентов, которые непосредственно оптимизируют финальный капитал. Может приводить к риску сильной волатильности стратегии, но максимизирует математическое ожидание прибыли. Часто используют, если политика достаточно сложная (например, LSTM), она сама может учесть риск. - *PBR* – применяют, когда хотят агент-спекулянта: он учится предсказывать направление движения, абстрагируясь от размеров. Это удобно, если торговля 1 контрактом с плечом – тогда прибыль прямо пропорциональна направлению. Однако PBR игнорирует масштабы: держать 100$ или 1000$ в позиции – reward одинаков, хотя второй случай важнее. PBR может быть полезен, когда хотим стратегию "следуй тренду": агент получает +reward за верно отгаданное направление независимо от того, сколько у него капитал (до тех пор, пока он "all in" условно). - *RiskAdjustedReturns (Sharpe)* – важно, когда приоритет – стабильность дохода. Агент будет стремиться как повысить доходность, так и снизить волатильность equity. Это сдерживает сверх-агрессивные стратегии. Например, если одна стратегия A даёт среднегодовой 20% при волатильности доходностей 15%, Sharpe ~ 1.33, а другая B – 30% при волатильности 40%, Sharpe ~ 0.75, то Sharpe даст больше награды первой, хотя B > A по прибыли. Таким образом, агент может выбрать более стабильную стратегию. - *RiskAdjustedReturns (Sortino)* – похож на Sharpe, но агент не "наказывается" за волатильность вверх, только за просадки. Это может быть более разумно: сильный рост net_worth – хорошо, не важно что он волатилен вверх, главное избегать больших потерь. Sortino-вознаграждение стимулирует агрессивный рост при контролируемом риске падения.

Учёт комиссий/проскальзывания: - Комиссии непосредственно уменьшают *net_worth* через *ExchangeOptions.commission* при исполнении ордера [171] [155], а значит, влияют на reward опосредованно (через net_worth). Например, SimpleProfit автоматически учтет уплаченные комиссии как снижение net_worth, снижая reward. PBR напрямую не знает про комиссии – если цена не изменилась, но мы купили-продали и потеряли комиссию, net_worth снизится, но PBR reward=0 (потому что p_t - p_{t-1}=0, X_t=0 если закрыл позицию). Таким образом, PBR агент может игнорировать небольшие комиссии -> может дребезжать. Нужно либо включить комиссии явно:

например, модифицировать PBR: $R_t = (p_t - p_{t-1})x_t - \text{commission\_paid}$. В TensorTrade PBR этого не делает, так что осторожно: PBR лучше с нулевыми комиссиями или если agent понимает косвенно (через net worth drop, но reward PBR не сигнализирует о drop). - Проскальзывание аналогично: если оно смоделировано в Exchange (покупает дороже, продает дешевле), то net_worth пострадает, SimpleProfit заметит отрицательный reward, а PBR – нет, т.к. PBR видит только mid-price разницу. Поэтому, для PBR, лучше в reward тоже включить штраф: например, `if order executed: reward -= slippage%`.

Сравнение схем (код): - SimpleProfit vs RiskAdjusted: RiskAdjusted обычно возвращает числа ~0.x (Sharpe ~1-2). SimpleProfit – может возвращать большие цифры на сильном росте (например, +0.05 если 5% рост за шаг). Sharpe за тот же шаг может быть 1-2 (если это за window). То есть масштаб разный. - PBR returns – могут быть и >1 (если overnight gap большой), т.к. price_diff может быть большой. Но обычно <0.05 per step. - Как решать, что лучше: *Если основная цель – чисто максимизировать прибыль любой ценой (и если агент имеет встроенное ограничение риска) – SimpleProfit.* Если хотим ровную equity – SharpeReward. *Если обучаем не портфель, а модель предсказания – PBR (агент как сигналист: buy if up).

Учёт рисков в кастомной RewardScheme: - Можно, как показано, добавить штраф за просадку или за позиции. Например, *PositionCostScheme*: R = ΔNW - 0.001 * |position_size| (штраф за занятую позицию – стимулирует поменьше быть в рынке). - *Или TradePenaltyScheme*: R = ΔNW - 0.001 * (комиссия_транзакций) - 0.0005 * (если action != hold, тогда штраф). Это будет агент реже торговать.

Формулы Latex примеры: - SimpleProfit: $R_t = \frac{NAV_t}{NAV_{t-1}} - 1$. - PBR: $R_t = (P_t - P_{t-1}) \cdot X_t$. - Sharpe: $R_t = \frac{E[r_{t-W:t}] - r_f}{\sigma(r_{t-W:t})}$. (где W – окно) - Sortino: $R_t = \frac{E[r] - R_{target}}{\sigma_{\text{down}}(r)}$. - VariancePenalty (наш придуманый): $R_t = \frac{NAV_t}{NAV_{t-1}} - 1 - \lambda \cdot \text{Var}(r_{t-W:t})$.

(Важное примечание: в RewardScheme коде все вычисления идут в долях (0.0x), а не процентах или basis points.)

9. Оценка и метрики

Из коробки TensorTrade не предоставляет богатый набор готовых метрик эффективности – упор сделан на то, чтобы пользователь сам анализировал результаты, имея историю портфеля и сделок. Тем не менее, имеются некоторые элементы: - **История портфеля (performance):** *Portfolio* хранит `initial_net_worth` и предоставляет `profit_loss` (в процентах от начального) [172]. `portfolio.profit_loss` рассчитывается как $1 - \frac{NW_{current}}{NW_{initial}}$ [173]. Например, если net_worth вырос, profit_loss будет отрицательным (!). Судя по коду: `profit_loss = 1.0 - net_worth/initial_net_worth` [173]. Это скорее “доля проигранных средств”. Возможно, это баг или нестандартное определение (обычно profit_loss = net_worth/initial - 1). В общем, `portfolio.profit_loss` стоит трактовать осторожно. Скорее лучше использовать `net_worth` напрямую. - **Trades log:** *Broker.trades* хранит заполнения (Trade) каждого ордера [129]. По этим данным можно вычислять: - Количество сделок, - Win rate (процент прибыльных) – определяя прибыль по каждой закрытой позиции или ордеру. - Средняя прибыль на сделку, средний убыток. - MFE/MAE (макс благоприятное/неблагоприятное отклонение) – но это требует сохраняя high/low portfolio value за время позиции (не реализовано встроенно). - **Sharpe/Sortino/MDD:** Не реализованы как отдельные классы, но: - Sharpe и Sortino реализованы внутри RiskAdjustedReturns RewardScheme, как описано [92] [169]. Однако, как метрики (после трейдинга) их можно вычислить самостоятельно: - *Max Drawdown (MDD)*: максимальная просадка – можно

получить из `portfolio.performance` – это просто min(net_worth) / max(net_worth) - 1. Но `performance` собирается или нет? Похоже, нет явного заполнения, но: FeedController при _prepare_feed объединяет streams и добавляет net_worth stream [65]. Можно после эпизода собрать все значения net_worth из feed.meta_history или из сохранённого списка (listener). Если Informer был – он мог считать MDD на лету. - *Sharpe/Sortino ex-post*: тоже по ряду net_worth. - *Calmar ratio, MAR* (годовой доход / max drawdown) – не встроены. - *Volatility* – ст.откл. доходности. - *Hit rate* (процент прибыльных трейдов) – потребуется пройтись по trades, сгруппировать по позициям (т.к. 1 ордер может быть частью позиции). TensorTrade не агрегирует ордера в позиции (нет понятия open/close trade, если агент сделал Buy, Sell – это будет 2 отдельные ордера, Trades можно сопоставить с PnL). - *Average profit per trade, average loss per trade, expectation* – тоже вычисляются постфактум: *Profit per trade* можно вывести: после эпизода, разница net_worth распределена между сделками. В простейшем, если agent always all-in/out, можно смотреть net_worth change per round-trip. Но general – нет простого способа, надо анализировать trade log. - **Встроенные Informers/Renderers:** - *PerformanceRecorder (Informer)* – если бы существовал, мог бы записывать Sharpe, MDD. - *PlotlyTradingChart (Renderer)* – возможно, в docs 1.0.4-dev1, они показывали, как отобразить график equity, цен, действий. Этот рендерер, вероятно, строит диаграмму: `price vs time` и `portfolio_value vs time`, а также отмечает точки покупок/продаж стрелками. Если он есть, он представляет визуальную метрику (график equity – можно визуально оценить волатильность, просадки). - *TensorBoard logging:* TensorTrade не интегрирован с TensorBoard, но пользователь может легко логировать внутри loop (если сам пишет loop) или через Informer.

Если встроенных мало, как добавить Sharpe/Sortino/MDD:
В текущей архитектуре легко: - Можно по окончании эпизода (после `done=True`) взять `env._portfolio.performance` (если оно велось) или собрать net_worth history. В 1.0.3, *Portfolio._performance* init as None [174], _performance updates not shown – возможно, performance history не ведётся auto. - *Вариант 1:* Listen to feed: FeedController.meta_history хранит meta-data of each step [175] [64]. Meta включает `meta` group, если feed имел ее. У нас feed имел 'features' and 'portfolio'. 'portfolio' group, согласно `FeedController._prepare_feed`, содержит net_worth [65] [176]. Значит, после эпизода, `pd.DataFrame(feed.meta_history)` даст DataFrame, содержащий 'net_worth' за каждый step (в meta). - *Вариант 2:* Portfolio.ledger – содержит все transactions, можно реконструировать equity curve: start at initial_net_worth, then for each trade update value (but easier meta_history). - *Вариант 3:* Use Informer: написать, например,

```
class MetricsInformer(AbstractInformer):
    def __init__(self):
        self.history = []
    def info(self):
        net = float(self.trading_env.portfolio.net_worth)
        self.history.append(net)
        # можно считать sharpe on the fly:
        returns = pd.Series(self.history).pct_change().dropna()
        sharpe = (returns.mean() / returns.std()) if len(returns)>1 else 0
        mdd = ... # compute from self.history
        return {"net_worth": net, "Sharpe": sharpe, "MDD": mdd}
    def reset(self):
        self.history.clear()
```

Добавив такой informer, info dict будет содержать обновляющиеся Sharpe, MDD. Их можно вывести или логировать. Однако, agent эти info не видит (unless using it in observation through some wrapper, что не делаем).

- *Hooks/нокруг:* TensorTrade не предоставляет глобальный callback, но вы всегда можете обернуть env.step своим кодом. В SB3, есть `Callback` interface – можно реализовать `on_step` callback, который берет `env` и читает `env.portfolio` etc, логирует. RLlib – аналогично `on_episode_end`.

Предложения по интеграции Sharpe etc.: Если нужен real-time logging: использовать Informer, как показано. Если post-episode: parse env after done.

Пример: интегрировать Sharpe/MDD: Добавить Informer:

```
metrics_inf = MetricsInformer()
env = TradingEnv(..., informer=metrics_inf)
```

В конце эпизода, `metrics_inf.history` содержит equity path. Можно вывести:

```
df = pd.DataFrame(metrics_inf.history, columns=["NetWorth"])
df["Return"] = df["NetWorth"].pct_change()
sharpe = df["Return"].mean() / df["Return"].std()
mdd = (df["NetWorth"].cummax() - df["NetWorth"]).max() /
df["NetWorth"].cummax().max()
print("Sharpe:", sharpe, "MaxDrawdown:", mdd)
```

Конечно, для надежности взять annualized Sharpe: `sharpe_annual = sharpe * sqrt(periods_per_year)`.

Наличие готовых метрик: Судя по документации: - 1.0.3 docs "Performance reports" – возможно, были utilities, но в коде 1.0.4-dev1 их не видно. - *tensortrade.agents.Agent.evaluate()* – может возвращала среднюю reward, но не более. - Issue #437: config in RLlib shows they measure "episode_reward_mean" – RLlib делает это сам. - TensorTrade 0.2 (старый) имел `analyze` util, но тут нет.

Итого: Из коробки: *SimpleProfit*implicitly track profit, *RiskAdjustedReturns*implicitly track risk metrics. Но если "их мало", пользователь добавляет: - Sharpe: как above (Informer or after run). - Sortino: similarly (only penalize negative returns in stdev). - MDD: easy via net_worth history as above. - Calmar: annual_return / MDD. - Win-rate (hit-rate): require to parse trades. E.g., assume trade pairs (buy->sell) and compute profit. Harder, but doable by analyzing ledger. - Or simulate simpler: if always going all-in/out, each round-turn = one buy one sell, profit per round = difference in net worth.

Куда интегрировать: - If frequently needed, one could create a subclass of Informer or a custom Renderer that prints final stats at end. - Alternatively, do it outside env:

```
rewards = []
net_worths = []
obs = env.reset()
```

```
while not done:
    action = policy(obs)
    obs, reward, done, _ = env.step(action)
    rewards.append(reward); net_worths.append(env.portfolio.net_worth)
# now compute metrics
```

This manual approach might be simplest for offline evaluation.

10. Продакшен-аспекты и живые данные

TensorTrade можно адаптировать для живой торговли, но это требует учитывать устойчивость и ограничения реального мира:

- **Переход от симуляции к живому исполнению:**
- *Exchanges:* Если указать ExchangeOptions(is_live=True) [177] [178] и предоставить `_service` функцию, которая реально исполняет ордера через API, TradingEnv сможет работать в режиме реального времени. Разница: *DataFeed* в live-режиме не может быть заранее известной. Здесь применим класс `PushFeed` [179]: он позволяет вручную пушить новые данные. То есть, можно создать `feed = PushFeed([...streams...])`, и на каждой итерации получать живые данные (например, через websocket) и вызывать `feed.push(data_dict)`. *PushFeed.push* помещает новые значения в Placeholder-стримы и генерирует новое состояние [180] [181]. TensorTrade не предоставляет готового сборщика реальных данных, но `tensortrade.data.cdd` – лишь для исторических. Для live: пользователь может написать loop:

  ```
  feed = PushFeed([...])
  env = TradingEnv(..., feed=feed, ...)
  obs = env.reset()
  while True:
      # get latest market data from exchange (e.g., via API)
      data = {"USD:/BTC": current_price, "Volume": current_vol}
      feed.push(data)           # update feed
      action = agent.act(obs)
      obs, reward, done, info = env.step(action)
      # handle done (stop if stopper triggers or break manually)
  ```

- *Живые биржи (через CCXT):* Как уже обсуждалось, нужно реализовать Exchange._service, который дергает API. Также *Wallet* тогда, вероятно, нужно синхронизировать с реальными балансами. В простой версии, можно инициализировать Wallet балансами с биржи. Но после сделок, кошелёк TensorTrade обновится локально – расхождение с реальным счётом? Если мы доверяем, что все ордера через env, то всё OK. Но если на бирже что-то вне env, локальный портфель не узнает.
  - Можно периодически синкать Walletы с биржей (например, каждые N минут запрос балансы).
  - Rate limits: CCXT и биржи ограничивают API-запросы. *execute_order* (live) – 1 запрос создания ордера. *quote_price* – при каждом `execute_order`, но мы можем хранить `current_price` уже полученный. *Feed* – сами котировки – лучше через WebSocket или 1 запрос per tick.

    - *Reconnects:* PushFeed – если data задержалась, env.step может ждать (так как agent, environment loop в нашем коде). Можно `time.sleep()` пока данные нет.
    - *Clock drift:* In live, Clock.now ideally uses real timestamp. If environment step has internal time, might drift. Possibly, for correctness, one should align env steps to market ticks (e.g., new candle triggers step).
    - *Пропуски данных:* If an API call fails or returns None, environment should handle gracefully: e.g., skip step or reuse last obs and maybe reward=0. Or stopper triggers because data feed ended.
    - *Отказоустойчивость:*
    - Если торговля реальная, критично обрабатывать исключения: API downtime, order rejection (e.g., insufficient funds), partial fills.
    - TensorTrade's simple model doesn't natively handle partial fill increments (no callback from exchange except once). But with CCXT, a large order could fill gradually. `ccxt_service` in Exchange could loop until order.status = closed, yielding intermediate trades via multiple order.fill events. But Broker expects one fill per execute call. Workaround: break big order into smaller ones or treat partially filled as still active and not call order.complete until done.
    - If connection lost, one might pause env, not send step to agent until restored (or treat as done to preserve agent state).
    - *Rate limiting:* If agent is high-frequency (e.g. every tick), exchange may throttle. Should incorporate delay. Could integrate `time.sleep()` in loop to ensure not exceeding.
- *Хранение состояния:*
    - В продакшене, нужно сохранять модель агента периодически (`model.save()`), а также логи сделок. TensorTrade env doesn't do persistence out-of-box, but user can instrument it.
    - If agent is learning online (less likely in live trading due to risk), periodic checkpoint is must. If just executing fixed policy, ensure ability to resume environment (which basically means reconstruct Portfolio and feed from where left off – possible as we have current net worth and maybe some moving window for indicators).
- *Управление рисками:*
    - Stopper can be used as fail-safe: e.g., if `net_worth < 0.8 * initial` , set done. But in live, done means what? Agent stops trading. Possibly triggers bigger actions (like notify human).
    - We might implement dynamic position sizing or additional constraints (not directly in TensorTrade, but in agent logic or action scheme).
- *Архитектурные ограничения для HFT/низкой задержки:*
    - TensorTrade is not optimized for ultra-low latency. The Python loop, plus overhead of Feed, plus possibly synchronous network calls to exchange – all adds latency (tens to hundreds of milliseconds at least). For HFT strategies (sub-second), this is too slow.
    - Also, Gym-style step loop may not align with event-driven trading needed in HFT (where events come irregularly).
    - One could attempt to push ticks at high frequency, but Python GIL and overhead likely limit sustainable tick rate maybe to hundreds per second, not thousands or millions like real HFT.
    - Also, stable RL algorithms don't typically handle extremely large timesteps per episode with micro-decisions (plus training them becomes intractable).
    - So TensorTrade is more suited for mid-frequency (minutes/hours bars) or at most second frequency trading.
    - If one needed HFT, you'd want a vectorized environment or integration with C++ infra – outside TensorTrade's scope.

    - Another limit: The synchronous step logic means agent decides one action per tick. If needed to manage order book events (multiple decisions inside one bar), one would have to model them as sub-steps – not directly supported.

Обработка ошибок и отказов: - Следует расширить *Exchange.service* с try/except. Если API call fails, it could: - Either raise, which if not caught would break `step`. We want env.step to handle it gracefully, maybe by returning done or skipping action. - Possibly better to catch inside service: e.g., if network error, do `time.sleep(1)` retry, or if serious, set `order.status = FAILED` and return None so that Broker doesn't think executed. Then Broker might leave it in unexecuted or cancel after some time (if we mark expired). - Not implemented by default, user must incorporate.

- Rate limit events: if exchange denies calls for a minute, agent should pause. Could implement Stopper that stops if calls fail consecutively.

Summary: TensorTrade can connect to live markets, but developer must implement a robust *service* function and manage the real-time loop. Устойчивость можно повысить: - Re-initialize Feed on reconnect, - Use Stopper to abort on anomalies, - Possibly use multi-threading for data input vs agent decision (so that agent can still step regularly even if feed delays? but careful). Architecture wise, it's not made for tick-by-tick order book reading or microsecond latencies, but for lower frequency decision-making it's fine.

11. Сравнение с альтернативами (сжато, но предметно)

TensorTrade vs Gym + SB3 (тонкие окружения): - *Уровень абстракции:* TensorTrade предоставляет высокоуровневую структуру, скрывающую детали торговли (balancing, order execution), тогда как простое Gym-окружение обычно приходится писать самостоятельно для каждого случая. Например, есть простые env вроде `gym-anytrading` (поддерживает несколько tickers, но без OMS), или custom env, где observation – цены, action – процент портфеля, и пользователь сам обновляет баланс. TensorTrade выигрывает тем, что уже реализует: - Портфель с несколькими активами и кошельками, - Реалистичное исполнение ордеров (рынок/лимит, комиссия, partial fills), - Модульную архитектуру (легко менять компоненты). В "тонком" Gym env часто все эти аспекты примитивно или не реализованы: например, `gym-anytrading` ограничивается покупкой/продажей сразу всего объема и не учитывает комиссию, нет понятия нескольких бирж или сложных действий. - *Гибкость:* За счёт SchemeMixins, Registry, TensorTrade легко расширяется (подключить новый индикатор в DataFeed – пару строк, новый тип Reward – наследование). В "self-made" env, хоть можно тоже код изменить, но меньше встроенных hooks. - *Код vs декларативность:* TensorTrade позволяет декларативно задать проблему (через config, context), и много логики – в библиотеке. Это сокращает время разработки стратегий, но повышает порог вхождения (нужно понять всю архитектуру). - *Производительность:* Тонкий Gym env, специально заточенный, будет быстрее и потреблять меньше памяти (меньше overhead на Feed, DataFrame, etc.). TensorTrade, из-за гибкости, чуть тяжелее. Например, DataFeed toposort и run Streams – это overhead (особенно, если Streams много, numpy-vectorize могло бы быстрее, но Streams скорее линеен). - *Сообщество и проверенность:* Stable Baselines + простые custom env – хорошо документировано, много примеров. TensorTrade – менее активный проект (после 2021 мало релизов), документация может отставать от dev. И главное, при использовании TensorTrade+SB3, нужно быть осторожным: SB3 Gym env должны соответствовать определённым требованиям (не изменять пространства динамически, deterministic reset, etc.), TensorTrade вроде соблюдает, но dev-версия – Beta. - *Возможности OMS:* Если стратегия предполагает тонкую работу с ордерами (например, выставление нескольких лимитников), TensorTrade уже имеет для этого средства (ActionScheme -> multiple orders, Broker manage). В простом env это пришлось бы кодировать с нуля. - *Сложность отладки:* Простой env – легко печатать значения state и ручками просчитывать

шаги. В TensorTrade, state размазан между DataFeed, Portfolio, Broker. Требуется использовать Informers/Renderers чтобы отладить (например, “почему net_worth не вырос, хотя цена выросла?” – нужно помнить о commission). - *Использование сторонних RL:* TensorTrade Env можно подключать к любому RL lib (SB3, RLlib, TF-Agents), но нужно решить вопросы с obs (dict vs array) и multi-env setups. Простое Gym env вы пишете под конкретную lib проще. - **Вывод:** TensorTrade сильна для исследовательских целей, когда нужно быстро экспериментировать с разными идеями (например: "А что если использовать ATR-стоппер?" – можно в пару классов сделать). Она также обеспечивает реализм: комиссии, более точный учёт баланса. Альтернативы вроде FinRL (финансовые RL-либы) часто строятся на Gym + pandas, сосредотачиваясь на конкретных сценариях (портфельное распределение, тренд-следование), но не имеют универсального OMS. Если задача простая (например, торговля один актив без плеча), "тонкий" env может быть легче и быстрее. Если задача сложная (multi-asset, cross-exchange arbitrage, сложные ордера) – TensorTrade даст структуру, в которой проще развивать проект.

Куда проще всего вшивать кастомные рыночные механики: - *Комиссии, проскальзывание:* В TensorTrade – через ExchangeOptions и кастомный execution service. Например, добавить модель импакта: перед исполнением ордера изменить current_price (проскальзывание). В простом env – придётся прописать в логике step. - *Особые ограничения:* Например, ограничение плеча – можно дописать Stopper, который останавливает эпизод если `leverage > X`. Или ActionScheme, который запрещает увеличить позицию если у нас уже max. - *Новые классы ордеров:* TensorTrade можно расширить, но это нетривиально: например, Stop-Limit order – Broker.update должен учитывать условие активации. Это можно реализовать как Order, у которого `is_executable = price < trigger`, но Broker.update просто проверяет is_executable – так что сработает. Значит, можно. - *Поведением рынка:* TensorTrade предполагает, что цены – входной поток, а исполняются ордера "по ценам без влияния". Если хотим смоделировать влияние (большой ордер двигает цену) – нужно связать исполнение с DataFeed. Например, написать execution service, который не только обновляет Walletы, но и изменяет `price_stream` (например, order buy поднял price_stream.value на δ). Это очень нестандартно: DataFeed стримы обычно exogenous. Но технически можно: `exchange._price_streams[pair].value = new_price`. В простой Gym env тоже вручную прописывать. - *Несовершенство рынка (лаг между решением и исполнением):* TensorTrade: можно моделировать задержку – например, ActionScheme, получив action, может вернуть Order с `order.delay = 5` (нет встроенного, но можно реализовать: keep in unexecuted until 5 ticks). В simple env – просто откладывать исполнение.

В целом, архитектура TensorTrade более приспособлена для кастомизации рыночной механики: у вас есть явные места, куда вставить логику (Exchange.service для исполнения логики, Stopper для условий останова, DataFeed для генерации сложных синтетических данных). В самописном env часто вся логика в одном месте (step) – быстрее, но менее модульно.

12. Known issues / расхождения

При анализе кода и документации TensorTrade выявлены некоторые несовпадения и нерешённые проблемы: - **Расхождения документации:** Документация (особенно для v1.0.3) иногда не соответствует актуальному коду dev-версии. Например, в документации `tensortrade.env.default` описываются классы *TradingEnvironment*, *ActionScheme*, *RewardScheme* как if they are separate, но в 1.0.4-dev структура модулей изменилась (вместо env.default используют env.generic.TradingEnv и схемы в отдельных подпакетах). Пользователю надо внимательно смотреть на установленную версию. Приоритет должен быть за кодом: например, если doc говорит "to get net worth use portfolio.performance", а в коде performance пустое – надо полагаться на код. - **Проблемы с Gym/Gymnasium:** Переход на gymnasium (commit "Switch from

gym to gymnasium" [182]) мог вызвать несовместимость с некоторыми RL библиотеками. Например, SB3 (до версии 1.8) не поддерживает gymnasium natively. Issue: #382 "Not possible to use GPU with TensorTrade envs" – вероятно, об этом: - В Issue #382 (упомянут в поиск [34]) пользователь ожидал, что `.to('cuda')` можно применить к obs (torch tensor), но TensorTrade env возвращал numpy. Не совсем уверен. Может, #382 про shape mismatch akin. - Gym 0.26 introduced new reset signature (return (obs, info)), gymnasium too. TensorTrade TradingEnv.reset returns (obs, info) as per code [183], а SB3 <1.8 ждет (obs) only. Это могло вызывать warnings/bugs. - Temporary fix: gym.make("TradingEnv-v0") might fail if not properly registered. - **Registry warnings:** Если два класса имеют один `registered_name`, registry.register может не добавлять второй (потому что ключи – class object). Не критично, но следует уникально называть `registered_name` у кастомных classes, если они conceptually new. - **TODOs и NotImplemented:** - В code неявно: *Portfolio.performance* – нигде не заполняется, хотя RewardSchemes ожидают. Возможно, это упущение. Есть подозрение, что `performance_listener` должен был вызываться. - В commit "Add minimal PBR explanation taken from code" [184], вероятно, автор заметил недочёт или хотел обновить docs. - *Parallel execution/training:* TensorTrade lacks out-of-the-box multi-env or multi-agent support. Though not exactly an "issue", но вопрос, plan ли они? - **Баг в SimpleProfit profit_loss:** как отмечено, Portfolio.profit_loss возвращает 1 - current/initial, что странно [173]. Возможно, это задумывалось как "percentage of loss" (т.е. если profit_loss=0.2, значит -20%?). Документация 1.0.3 может это неправильно описывает. Приоритет коду: разработчик likely interpret profit_loss as positive = loss. Пользователю лучше вычислять PnL самостоятельно. - **Import issues:** - `from tensortrade.env.default import TradingEnv` – в dev1, возможно, нужно `from tensortrade.env.generic import TradingEnv`. Old examples might break. - Some submodules referenced in docs (like `tensortrade.data.stream`) might have been reorganized under `tensortrade.feed`. - The repository suggests usage `import tensortrade.env.default as default` in examples [185] – но in code, `env.default` is likely an alias for some pre-configured schemes. Actually, in 1.0.3, `tensortrade.env.default.TradingEnvironment` existed (alias to Generic). In 1.0.4, not sure if `env.default` module present. If not, examples require update. - **Deprecated components:** - `tensortrade.env.default` and classes in it (like `TradingEnvironment`, `ManagedRiskAdjustedReturns`) – might be leftover from older version. They either alias or are deprecated. - `tensortrade.agents` – as mentioned, marked @deprecated. So their usage yields DeprecationWarning. Possibly will be removed in future. - **Issues on GitHub:** - #437 we covered (checkpoint restore shape mismatch – likely RLlib not matching exactly environment context). - #382 (if exists): Possibly complaining about Gym-space mismatch on GPU – maybe a user tried to send obs to GPU and had trouble. Not entirely clear. - Unanswered question: *Portfolio and Exchange duplication:* In older TensorTrade (pre-1.0) there were known issues e.g., with simultaneous multi-Exchange portfolios or weird rounding issues. Not sure if still relevant: e.g. commission rounding error if precision too low – user should ensure instruments precision is high enough. - **Performance (Dev-branch not fully tested):** Developers caution: "TensorTrade is still in Beta" [186]. So some parts may contain bugs. - Example: *Stopper not integrated in episodes loop in SB3.* SB3 calls .reset if done True, so if Stopper sets done mid-episode, SB3 will respect it. That should be okay. - Example: *Gym compatibility issues with Info dict size or content.* If Info contains non-serializable things, SB3 logging might crash. Ensure Info is simple types. - Possibly memory leak if feed.meta_history grows unbounded (if you run extremely long episodes, meta_history list might be big). They don't clear performance each episode except `portfolio.reset()` sets _performance = None. So meta_history is per episode (maybe cleared at feed.reset). - **Polish issues:** - The user might find lacking features like action clamping (to avoid overspending), risk management built-in (like maximum position size). These aren't issues per se, but omissions. They can be filed as feature requests. - Minor: The default logging via print or others is minimal (lack of built-in logger). - Missing type hints in some places, but mostly present.

Если находить конкретные строчки: - `performance_listener` is accepted in Portfolio but never used – potentially an issue: there is even comment "to send all portfolio updates to

performance_listener" [105], but in code performance_listener is just stored [187]. Could mention that. - ExchangeOptions 'is_live' not fully utilized: no built-in live exchange integration beyond a flag. - Some tests might fail (lack of update in docs suggests incomplete coverage).

В целом, пользователи отмечают, что TensorTrade требует внимательного тестирования стратегии – некоторые **углы остры**: - Memory usage: storing entire DataFeed in memory. - Speed: maybe not an issue at daily bars, but at tick-level heavy.

Для конкретики: - Issue 382: "Not possible to use GPU with TT envs" – likely environment returns numpy, user needed torch. Actually, the snippet on reddit suggests user wanted to send obs to GPU in custom loop. Not a bug in TT, just integration nuance.

- [Issue 425/426] – merges about config.

No major breaking “bugs” surfaces from code; main points – incomplete features (performance tracking), changes in dependencies, deprecated parts.

13. Итоговая карта модулей (таблица)

(Уже представлена в разделе 2.1 выше с колонками: Модуль → Назначение → Ключевые классы → Используется где → Расширяемость → Ссылки.)

Для удобства повторим кратко несколько ключевых модулей:

Модуль	Назначение	Ключевые классы	Используется где	Расши
tensortrade.core	Базовые абстракции и контекст	`Component`, `TradingContext`, `Identifiable`, `TimeIndexed`, `registry`	Фундамент для env/ OMS	Новый авто в контек конфи
tensortrade.feed	Поток данных (фичи, индикаторы)	`Stream`, `DataFeed`, `IterableStream`, `PushFeed`	TradingEnv через FeedController	Добав Stream индик
tensortrade.env.generic (TradingEnv)	RL-окружение (Gym)	`TradingEnv`, `Observer`, `Informer`, `Stopper`, `Renderer`	Агент взаимодействует	Расши схемы (ниже)
tensortrade.env.actions	Схемы действий (преобр. action→Order)	`AbstractActionScheme`, `SimpleOrders` (держи/ купи/продай)	`env._action_scheme`	Насле Abstra задать `actio` `get_`
tensortrade.env.rewards	Схемы вознаграждения	`AbstractRewardScheme`, `SimpleProfit`, `PBR`, `RiskAdjustedReturns`	`env._reward_scheme`	Насле Abstra реали `rewa`

Модуль	Назначение	Ключевые классы	Используется где	Расши
tensortrade.oms.exchanges	Биржи и исполнение ордеров	`Exchange`, `ExchangeOptions`	Через Wallet/Portfolio в OMS	Новый задать subcla `execu`
tensortrade.oms.wallets	Кошельки и Портфель	`Wallet`, `Portfolio`, `Ledger`	Env.portfolio (состояние средств)	Можно Portfol overri
tensortrade.oms.orders	Ордеры и брокер	`Order`, `Trade`, `Broker`, `OrderStatus`	OMS внутри env.step	Можно Order свой is Broker update
tensortrade.agents (deprecated)	Примеры RL-агентов	`DQNAgent`, `A2CAgent`, `ParallelDQNAgent`	Для демонстраций	Предп испол внешн
tensortrade.data	Загрузка данных (внешних)	`CryptoDataDownload` (из cryptodatadownload), др.	Перед запуском env (получить df цен)	Можно загруз financ

(Примечание: Модули `tensortrade.env.default` и `tensortrade.env.generic` – по сути одно: generic/TradingEnv заменил default/TradingEnvironment.)

Заключение: TensorTrade – мощный фреймворк для RL-трейдинга, предлагающий модульность и готовые компоненты, но требующий тщательного понимания внутреннего устройства. Приоритет всегда следует отдавать поведению, определяемому кодом (особенно версии `master`), а не устаревшим примерам, и не стесняться расширять или править компоненты под конкретные нужды стратегии. Несмотря на отдельные шероховатости и необходимость доработки для продакшена (например, для live-трейдинга), ядро TensorTrade обеспечивает крепкую основу для разработки и тестирования сложных торговых алгоритмов RL [6] [57].

---

[1] [2] [5] [6] [57] [186] GitHub - tensortrade-org/tensortrade: An open source reinforcement learning framework for training, evaluating, and deploying robust trading agents.
https://github.com/tensortrade-org/tensortrade

[3] [82] [83] [182] [184] Commits · tensortrade-org/tensortrade · GitHub
https://github.com/tensortrade-org/tensortrade/commits/master/

[4] tensortrade · PyPI
https://pypi.org/project/tensortrade/

[7] [8] [47] [48] [51] [52] [53] [54] [55] [56] [166] component.py
https://github.com/tensortrade-org/tensortrade/blob/72c3be9c15d3ac9dc7783c3ddf79b12a7e0a2fb3/tensortrade/core/component.py

[9] [10] [40] [41] [58] [59] [60] base.py
https://github.com/tensortrade-org/tensortrade/blob/72c3be9c15d3ac9dc7783c3ddf79b12a7e0a2fb3/tensortrade/core/base.py

11 12 13 179 180 181 feed.py
https://github.com/tensortrade-org/tensortrade/blob/72c3be9c15d3ac9dc7783c3ddf79b12a7e0a2fb3/tensortrade/feed/core/feed.py

14 157 158 dqn_agent.py
https://github.com/tensortrade-org/tensortrade/blob/72c3be9c15d3ac9dc7783c3ddf79b12a7e0a2fb3/tensortrade/agents/dqn_agent.py

15 16 21 22 36 37 38 39 44 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 96 99 100 101 112 145 162 163 164 183 188 environment.py
https://github.com/erhardtconsulting/tensortrade-ng/blob/ec6552e3cdbf64ad05c58955d26c207dda825df3/src/tensortrade/env/environment.py

17 18 28 49 84 85 86 87 abstract.py
https://github.com/erhardtconsulting/tensortrade-ng/blob/ec6552e3cdbf64ad05c58955d26c207dda825df3/src/tensortrade/env/actions/abstract.py

19 93 94 95 97 165 abstract.py
https://github.com/erhardtconsulting/tensortrade-ng/blob/ec6552e3cdbf64ad05c58955d26c207dda825df3/src/tensortrade/env/rewards/abstract.py

20 88 89 168 pbr.py
https://github.com/erhardtconsulting/tensortrade-ng/blob/ec6552e3cdbf64ad05c58955d26c207dda825df3/src/tensortrade/env/rewards/pbr.py

23 26 27 30 31 116 118 126 146 147 148 149 150 151 152 153 154 155 171 177 178 exchange.py
https://github.com/tensortrade-org/tensortrade/blob/72c3be9c15d3ac9dc7783c3ddf79b12a7e0a2fb3/tensortrade/oms/exchanges/exchange.py

24 25 32 50 102 103 104 105 106 107 108 109 110 111 113 114 115 117 156 172 173 174 187 portfolio.py
https://github.com/erhardtconsulting/tensortrade-ng/blob/ec6552e3cdbf64ad05c58955d26c207dda825df3/src/tensortrade/oms/wallets/portfolio.py

29 119 120 121 122 123 124 125 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 189 broker.py
https://github.com/erhardtconsulting/tensortrade-ng/blob/ec6552e3cdbf64ad05c58955d26c207dda825df3/src/tensortrade/oms/orders/broker.py

33 34 42 43 61 62 63 64 65 66 175 176 feed_controller.py
https://github.com/erhardtconsulting/tensortrade-ng/blob/ec6552e3cdbf64ad05c58955d26c207dda825df3/src/tensortrade/env/utils/feed_controller.py

35 167 simple_profit.py
https://github.com/erhardtconsulting/tensortrade-ng/blob/ec6552e3cdbf64ad05c58955d26c207dda825df3/src/tensortrade/env/rewards/simple_profit.py

45 46 185 190 train_and_evaluate.ipynb
https://github.com/sadimoodi/tensortrade/blob/e81c94c7048d4a57e2a2d5c76f5d6005f96406fc/examples/train_and_evaluate.ipynb

90 91 92 98 169 170 risk_adjusted_returns.py
https://github.com/erhardtconsulting/tensortrade-ng/blob/ec6552e3cdbf64ad05c58955d26c207dda825df3/src/tensortrade/env/rewards/risk_adjusted_returns.py

159 "Size mismatch" error when restoring checkpoint after Ray tuning · Issue #437 · tensortrade-org/tensortrade · GitHub
https://github.com/tensortrade-org/tensortrade/issues/437

160 161 parallel_dqn_trainer.py

https://github.com/faryabimm/tensortrade/blob/f88d0cb88603a7353a38f5bae4792dadb04d09ac/tensortrade/agents/parallel/parallel_dqn_trainer.py