ChatGPT

BetterBot Deployment Plan

BetterBot is a Bitvavo-based cryptocurrency trading system that combines traditional ML forecasting with a reinforcement learning (PPO) agent for decision-making. This deployment plan outlines the complete project structure, initialization of a GitHub repository with skeleton code, model training and shadow-testing workflow, regime-switching logic, risk controls, and automation steps. The goal is to create a robust, maintainable trading bot that can start in paper-trading mode and gradually transition to live trading, with safeguards against model degradation and data leakage.

Directory Structure

All project files will reside under `/opt/BetterBot`. The directory hierarchy is organized by component, as shown below:

```
/opt/BetterBot/
├── betterbot/                    # Core Python package for BetterBot
│   ├── __init__.py
│   ├── fetchers/                 # Data fetching modules (Bitvavo API)
│   │   ├── __init__.py
│   │   └── data_fetcher.py       # Stub: fetch latest prices, order book,
etc.
│   ├── features/                 # Feature engineering modules
│   │   ├── __init__.py
│   │   └── feature_engineering.py# Stub: compute indicators (e.g. RSI,
volatility)
│   ├── forecasters/              # Predictive model modules (LightGBM, GRU)
│   │   ├── __init__.py
│   │   ├── lightgbm_model.py     # Stub: LightGBM predictor class
│   │   └── gru_model.py          # Stub: GRU predictor class (PyTorch)
│   ├── ppo/                      # Reinforcement Learning (PPO) components
│   │   ├── __init__.py
│   │   ├── env.py                # Stub: Custom trading environment (Gym
interface)
│   │   └── policy_wrapper.py     # Stub: Wrap PPO policy (load/save, action
masking)
│   ├── orchestrator/             # Training orchestration
│   │   ├── __init__.py
│   │   └── train.py              # Stub: Master training script for models &
PPO
│   ├── executor/                 # Live trading execution
│   │   ├── __init__.py
│   │   └── trader.py             # Stub: Main trading bot runtime (paper/
live)
│   ├── utils/                    # Utility modules (alerts, validation,
registry)
```

```
│   │   ├── __init__.py
│   │   ├── metadata_validator.py # Stub: Check model metadata (age,
integrity)
│   │   ├── registry.py           # Stub: Model registry management (save/
load)
│   │   └── telegram_alerts.py    # Stub: Functions to send Telegram messages
│   ├── configs/                  # Configuration files and templates
│   │   ├── config_template.yaml  # Template for main config (API keys,
settings)
│   │   └── regimes.yaml          # Config for regime thresholds and models
│   └── tests/                    # (Optional) tests for components
│       └── ...                   # e.g., basic unit tests for data fetching,
etc.
├── scripts/                      # Deployment and maintenance scripts
│   ├── sync_to_repo.sh           # Script: git add/commit/push (uses deploy
key)
│   ├── backup_models.sh          # Script: rsync important files to backup
location
│   ├── run_shadow_backtest.sh    # Script: run 7-day shadow backtest
analysis
│   ├── daily_report.sh           # Script: generate & send daily Telegram
report
│   └── cron_setup.txt            # Cron schedule examples (documented
crontab entries)
├── models/                       # Model registry (subfolders per model
release)
│   ├── README.md                 # Info about model registry usage
│   └── {model_hash}/             # e.g., "20250919_abcd1234/" for each
release
│       ├── metadata.json         # Model metadata (timestamp, performance,
config)
│       ├── ppo_policy.zip        # Saved PPO policy (Stable-Baselines3 .zip)
│       ├── gru_model.pt          # Saved GRU model (PyTorch state dict)
│       └── lgb_model.pkl         # Saved LightGBM model (pickled)
├── logs/                         # Log files for trading, training, and
backtests
│   ├── trading.log
│   ├── training.log
│   └── shadow_compare.log
├── README.md                     # Documentation for usage, setup, and
development
└── requirements.txt              # Python dependencies (bitvavo API, sb3,
torch, etc.)
```

Each Python module above will be created with a **stub implementation** (at minimum, a file with a class definition or function and a docstring) rather than leaving empty placeholders. This ensures the repository is importable and sets the stage for further development. For example, `fetchers/data_fetcher.py` might contain a minimal class like:

```
# fetchers/data_fetcher.py
import bitvavo  # (Using Bitvavo API client library)
class DataFetcher:
    """Fetches market data from Bitvavo API (REST/WebSocket)."""
    def __init__(self, api_key: str, api_secret: str):
        # Initialize Bitvavo client (stub)
        self.client = bitvavo.Bitvavo({'APIKEY': api_key, 'APISECRET':
api_secret})
    def get_ticker(self, market: str):
        """Fetch latest ticker price for a given market (e.g., BTC-EUR)."""
        return self.client.getTickerPrice({'market': market})
```

All other components will follow a similar pattern: include relevant imports, a class or function definition, and dummy return values or `pass` statements where actual logic will go. This provides a **GitHub-ready skeleton** where each part of the system is represented.

GitHub Repository Bootstrapping

To initialize the project, create a new Git repository (e.g., on GitHub under a suitable name like `BetterBot`). Clone it into `/opt/BetterBot` and set up the structure as above. Perform the following steps for bootstrapping the repo:

1. **Initial Commit:** Generate the directory structure and stub files as listed. Include an `__init__.py` in each package directory so that `betterbot` can be imported as a Python package. Write a comprehensive `README.md` describing the project, and add a `requirements.txt` specifying dependencies (e.g., `bitvavo` API client, `pandas`, `numpy`, `lightgbm`, `torch`, `stable-baselines3`, `psutil`, etc.). After creating these files, run:

   ```
   cd /opt/BetterBot
   git init .
   git add .
   git commit -m "Initial commit: project structure and stubs"
   ```

   This captures the baseline skeleton in version control.

2. **Repository Settings:** Set up a remote on GitHub and push the initial commit. For secure CI/CD, use a **deploy key** on the server to authenticate pushes. A *GitHub deploy key* is an SSH key granting access to a single repo [1]. On the server, generate an SSH key pair (without passphrase) dedicated to this bot:

   ```
   ssh-keygen -t ed25519 -f ~/.ssh/betterbot_deploy_key -N ""
   ```

   This produces `betterbot_deploy_key` (private) and `betterbot_deploy_key.pub` (public). Add the public key (`cat ~/.ssh/betterbot_deploy_key.pub`) as a **Deploy Key** in the GitHub repository settings (with write access enabled). This allows the `sync_to_repo.sh` script to push updates. The `sync_to_repo.sh` script can be a simple wrapper around git:

```
#!/bin/bash
cd /opt/BetterBot
git add -A
git commit -m "Automated update: $(date '+%Y-%m-%d %H:%M')"
git push origin main
```

Combined with the deploy key loaded (e.g., via `ssh-agent` or using `GIT_SSH_COMMAND` in the script), this enables automated code pushes from the server.

3. **Stub File Initialization:** Ensure each stub Python file has minimal content instead of being empty. For instance, `forecasters/lightgbm_model.py` can include:

```
import lightgbm as lgb
class LightGBMForecaster:
    """Stub forecaster using LightGBM."""
    def __init__(self):
        self.model = None
    def train(self, X, y):
        """Train LightGBM model on features X to predict y (stub
implementation)."""
        self.model = lgb.LGBMRegressor()
        if len(X) and len(y):
            self.model.fit(X, y)  # placeholder training
    def predict(self, X):
        """Predict using the trained LightGBM model (stub: returns 0 if
no model)."""
        return self.model.predict(X) if self.model else [0]*len(X)
```

Similar dummy implementations should be added for the GRU model class, PPO environment, etc., typically with a docstring and `pass` or simple return. This way, all modules are syntactically correct and importable, which will help later when generating full implementations.

4. **Configuration Templates:** In the `configs/` directory, prepare a `config_template.yaml` that users can copy to `config.yaml` and fill in. This template should include placeholders for:

5. API keys (e.g., `api_key`, `api_secret` for Bitvavo).
6. Trading parameters (mode, base asset, trade asset, initial balance for paper, etc.).
7. Risk limits (e.g., `max_position_size`, though initially we limit to €500).
8. Model settings (e.g., `model_max_age_days`, `volatility_threshold`).
9. Telegram bot token and chat ID for alerts.
10. Regime configuration (could also go in a separate `regimes.yaml` as structured in directory).

By keeping secrets in a separate untracked config (the template in git, the actual filled config kept out of version control), we maintain security.

After these steps, the GitHub repo will be bootstrapped with a clear structure and ready for collaborative development or code generation. Each commit should be pushed via the deploy key. (It's

also wise to print the deploy public key to the console or logs upon install, so the user can easily add it on GitHub – the plan is to include that in deployment scripts or documentation.)

Component Overview

BetterBot's architecture is modular, with each component responsible for a specific aspect of the trading system. Below is an overview of each component (with the corresponding stub file in parentheses):

- **Data Fetchers** (`fetchers/`): Handle connectivity to the Bitvavo API for retrieving market data. The fetcher will use Bitvavo's REST endpoints (and WebSocket where real-time updates are needed) [2]. For example, the data fetcher can get current prices, recent OHLC candlesticks, account balances, and open orders. These functions supply the raw inputs for feature calculations. (In our stub, we use the official Python Bitvavo API wrapper for convenience [3].) The fetcher will ensure **spot market data** only, since Bitvavo does not support margin trading or short positions [4].

- **Feature Engineering** (`features/`): Takes raw market data (prices, volumes, etc.) and computes derived features for the models. This could include technical indicators (moving averages, RSI, MACD, volatility measures like ATR, etc.), normalized price changes, or any domain-specific signals. All features are computed using **only past and current data** to prevent any look-ahead bias. A critical part of this component is to enforce no data leakage – the system should **hard fail** if any feature accidentally uses future information (labels) [5]. In practice, this means carefully aligning data in time and possibly implementing a unit test in `metadata_validator.py` or a check in the training pipeline to ensure features at time *t* are not computed with data beyond *t*. The feature module stub will define functions like `compute_features(df)` that returns a feature vector or DataFrame for the latest time step.

- **Forecasters (LightGBM & GRU)** (`forecasters/`): These are supervised learning models that provide predictive signals to the RL agent. The **LightGBM model** could be used for short-term price direction prediction (classification or regression of next return), and the **GRU (Gated Recurrent Unit)** neural network can capture sequential patterns in price data for forecasting. Both models are trained offline (in the orchestrator) using historical data. At runtime, their outputs (e.g., predicted price change or probability of upward move) are included in the RL observation space. This **hybrid approach** leverages powerful predictive modeling for near-term market movement and feeds it into the decision-making agent, combining the strengths of deep learning forecasts with adaptive RL [6]. Each forecaster will have methods to `train()` on historical data and `predict()` on new data. Stub implementations should instantiate a model (for GRU, a simple PyTorch model skeleton; for LightGBM, using `lgb.LGBMRegressor` or `Classifier`) and include serialization (saving to `.pkl` or `.pt`). By storing the trained forecaster outputs, the RL agent can make more informed decisions than using raw features alone.

- **PPO Environment & Policy** (`ppo/`): The PPO environment (`env.py`) defines the reinforcement learning problem for the trading agent. It encapsulates the trading logic in an OpenAI Gym-style interface: state (observation), actions, reward calculation, and episode termination. The **observation space** will include the latest feature vector (from `features/`) **augmented by the forecasters' outputs** (LightGBM prediction, GRU prediction) forming a combined state input to the agent. The **action space** is discrete with three actions: *Buy (go long)*, *Sell (go flat)*, or *Hold* – we restrict to spot long or no position, and explicitly exclude any short or

leveraged actions (aligned with Bitvavo's capabilities [7]). The `policy_wrapper.py` will manage the PPO policy model (likely using Stable-Baselines3 PPO under the hood) – it can load or save the trained policy (`ppo_policy.zip`) and possibly enforce custom behavior (like masking invalid actions if needed, though here all actions are always valid except that "Sell" when no holdings will simply result in staying flat). The **reward function** is critical: we define it as **net profit-and-loss minus a penalty for volatility and trading costs**. This means the agent gets positive reward for increasing net equity, but is penalized if its equity curve is too volatile or if it trades excessively incurring fees. This encourages a high risk-adjusted return (similar in spirit to maximizing Sharpe ratio) [8]. Concretely: `reward = Δ portfolio_value – λ·(portfolio_value_volatility) – trading_fees`. By tuning λ, we control how strongly the agent avoids volatility or large drawdowns. The environment should also factor in a small negative reward for each trade (to account for commission/slippage). The stub `env.py` will outline the class `TradingEnv` with methods `reset()`, `step(action)` and include placeholders for applying an action (simulate buy/sell by updating a position flag and PnL calculation). We will use **episode** definitions that make sense for training (e.g., an episode could be one day or week of trading data).

- **Training Orchestrator** (`orchestrator/train.py`): This is the script that coordinates the end-to-end training pipeline. In a full implementation, running `train.py` will: (1) fetch or load the latest historical data (via fetcher or from a stored dataset), (2) update/generate features, (3) train the LightGBM model on recent data, (4) train the GRU model on recent sequence data, and save their updated weights, (5) construct or update the PPO environment to include these models' predictions in the observation, and (6) train a new PPO policy (using stable-baselines3 PPO algorithm or similar) on this environment. Hyperparameters for training (learning rates, number of timesteps, etc.) will be configured here or in config. The orchestrator essentially **automates the model retraining** process. It will output the new model artifacts and register them. After training, it should evaluate performance on a validation set or recent period. If the new model looks good (e.g., achieves higher reward or Sharpe on backtest), the script can trigger the **model registry** update (creating a new model folder with hash, saving models and metadata). All these steps will be logged to `training.log` for transparency. The orchestrator will also incorporate **data validation checks** – for example, after feature generation, call `utils/metadata_validator.py` to ensure no NaNs, no misaligned labels, and no leakage (e.g., ensure that if we shift labels for training, none of the features includes that label unshifted). If any such issue is detected, it should log an error and stop (fail fast). We plan for training to be done periodically (likely once a day or once a week via cron, depending on strategy needs).

- **Executor/Trader** (`executor/trader.py`): The trader is the live execution script that runs continuously (or at intervals) to make trading decisions and send orders to the exchange. In **paper-trading mode**, it will simulate orders against a virtual portfolio; in **live mode**, it will use real Bitvavo API endpoints to execute trades on the user's account. Initially, we will run in simulation with a fixed capital (e.g., €500) to validate performance without risking real funds [9]. The trader performs the following in a loop:

- Load the current active model from the registry (as specified by config – e.g., a symlink or config field points to the latest approved model folder). It loads the PPO policy, and forecasters if needed (though forecasters could also be re-run live to update predictions).
- Fetch latest market state (through `DataFetcher` – e.g., current price, recent history).
- Compute the feature vector for now and get the forecasters' predictions for the next step.
- Form the observation and query the PPO policy for an action (Buy, Sell, Hold).

- Decide execution: If action is Buy and currently no position, place a buy order (for the allowed amount). If action is Sell and currently in position, place a sell order (to fully exit position). If Hold, do nothing. We restrict to one position at a time and always either fully invested or fully in cash (flat) to keep logic simple and aligned with our discrete actions (this mirrors the approach in a simple Bitvavo bot example where the bot buys with full balance or sells entire holding on a signal [10]). No short orders are sent (if no holdings and signal says Sell, we simply remain flat). All orders are **spot market orders** (or limit orders if desired for safety) – leverage is not used and shorting is not attempted [7].
- Log the action, price, and resulting position/PnL to `trading.log`. Also, handle any errors from the API (network issues, etc.), with retries or graceful abort as needed.

The trader will also implement **regime switching logic** before deciding on the action: using the current volatility or regime indicator, it may load a different model appropriate for the regime (if the models are specialized per regime – see *Regime Switching* below for details). For efficiency, the trader could preload both regime models and just choose which action to take from which model based on the indicator each cycle.

Additionally, the trader script enforces certain **safety checks** each run: - It checks the timestamp of the current model's metadata to ensure the model is not older than the maximum allowed age (from config). If the model is stale (e.g., older than X days), the trader will **halt trading and alert** rather than using an outdated strategy [11]. This prevents the bot from trading on a model that potentially no longer reflects current market conditions. - It ensures that necessary data is available (if data feed has gaps or the fetcher fails to get recent prices, the trader can pause or use fallback logic). Missing data or an inability to compute features should result in no trade (and possibly an alert if prolonged).

Finally, the trader in live mode uses the Bitvavo API keys from config to execute orders. The code must be careful with API usage and keys security (keys will be loaded from a secure config). *Note:* Bitvavo offers both REST and WebSocket APIs – for simplicity, initial implementation might poll REST for latest price every minute. WebSocket can be integrated later for real-time updates if needed.

- **Utility Scripts & Modules**:
- **Model Registry Management** (`utils/registry.py`): Provides functions to save new model versions and load existing ones. For saving, it will create a new subdirectory under `models/` named with a unique identifier (for example, using a hash of the model parameters or a timestamp + short git commit hash). The `metadata.json` in that folder will store important info like training date, data period used, performance metrics (e.g., backtest Sharpe, returns), and a human-readable version or tag. The registry module also might maintain a `latest` symlink or a pointer in the main config to indicate which model is currently active in production. This module ensures that when the trader loads a model, it loads the correct matching trio of PPO, GRU, and LightGBM models from the same release folder.
- **Metadata Validation** (`utils/metadata_validator.py`): Contains routines to verify that data and models meet certain criteria. For example, after training, it can check the new model's performance metrics against thresholds (if the new model's Sharpe or PnL in backtest is worse than the old model, we might decide not to deploy it). It also validates that the model files are present and not corrupted. This module can be invoked by the orchestrator after training, and by the trader at startup (to validate the model it loaded).
- **Telegram Alerts** (`utils/telegram_alerts.py`): Provides a simple interface to send messages to a Telegram chat, using the bot API (with token and chat ID from config). This will be used for both urgent alerts and daily summaries. For example, if the trader encounters a critical error (API failure, model validation failure, etc.), it will call `send_alert("message")` to notify the maintainers immediately. Similarly, at regular intervals (daily report), a summary will be sent.

- **Scripts** (`scripts/` directory): Small executable scripts for maintenance tasks:
  - `daily_report.sh`: This script (or a Python script it calls) gathers system and bot metrics and sends the daily report. Metrics include system resource usage (CPU load, RAM, disk space), process health, current model version and age, number of trades executed in last 24h, and any data gaps or errors detected. The report is sent via Telegram every day at a set time. This ensures we have continuous monitoring of the bot's status.
  - `backup_models.sh`: Uses `rsync` or similar to copy critical data (like `models/` directory, logs, and perhaps any databases) to a remote backup or cloud storage. This can be scheduled to run daily or weekly. It helps in disaster recovery and also allows offline analysis of model versions.
  - `run_shadow_backtest.sh`: Automates the *shadow mode* testing for a new model. Suppose we just trained a new model version; this script can run the model on the last N days of historical data in a backtest mode (or even simulate it forward in parallel with live). It will generate a comparative report between the new model and either the old model or a benchmark. The results (e.g., PnL curves, Sharpe ratio of each, drawdowns) are saved to `shadow_compare.log` or a file, and possibly an alert is sent if the new model underperforms. This script essentially encapsulates the logic of shadow deployment (details in next section).
  - `cron_setup.txt`: A plaintext file (for documentation) that outlines the cron jobs required (the actual crontab is set via `crontab -e`, but we include a reference in the repo for clarity).

All these components work in concert to deliver an automated trading solution that is maintainable. By modularizing, we ensure that each part (data fetch, feature calc, prediction, RL action, execution) can be developed and tested independently or replaced as needed.

Shadow Mode Deployment (7-Day Evaluation)

When a new model is trained and ready, BetterBot employs a **shadow deployment** strategy to validate it before fully switching over. In **shadow mode**, the new model runs in parallel to the current production model for a trial period (configured to 7 days). During this time, the new model's decisions are **logged** but not executed with real money, allowing a safe comparison of performance against the incumbent model or a benchmark [12].

Key aspects of shadow mode implementation:

- The system will **not route trades from the new model to the exchange**. Instead, the new model's actions are recorded (with timestamps and prices) in a separate log (e.g., `shadow_trades.log`). The current active model continues to manage the actual trading (if there is one; if this is the very first model deployment, then "shadow mode" effectively means paper trading alongside a no-trade baseline).

- We collect performance metrics for the new model's strategy over the 7-day window. Since both models operate on the same market data, we can directly compare outcomes. For example, at the end of the period we compute total return, volatility, max drawdown, and number of trades for each. This is automated by either the `run_shadow_backtest.sh` script or by the orchestrator scheduling a comparison routine. (If available, we could also leverage Bitvavo's historical data to simulate the new model on the exact recent 7 days that the old model traded, to ensure fairness in comparison.)

- If the **new model's performance is acceptable or better** (per criteria defined in config, e.g., "new model 7-day Sharpe ratio not less than old model's Sharpe minus 10%"), then we promote the new model to production. Promotion can be as simple as updating a symlink `models/latest` to point to the new model's folder, or updating a field in the main config (e.g., `current_model_hash: abc1234`). The trader will then start using the new model for live decisions after the shadow period (perhaps with a brief downtime or reload signal).

- If the new model **underperforms or shows issues** during shadow testing, the system will not promote it. An alert (via Telegram) will notify that the new model failed validation and the bot will continue using the previous model. The unsuccessful model's artifacts remain in the registry for record-keeping, but might be flagged in its metadata (e.g., `"status": "rejected"` in metadata.json). This triggers the development/training team to either tweak hyperparameters or gather more data and then retrain a better model.

- The shadow mode design follows best practices of testing models in parallel without impact on operations [12]. It provides confidence in the model's real-world behavior beyond offline backtests. In essence, it is a live forward-test. During this period, we also double-check for any unexpected behaviors (e.g., does the model send too many signals, does it oscillate, etc., which we can catch from logs).

- The system supports running in **paper trading shadow** even for the very first deployment. Initially, since no prior model is live, we will effectively run the model in a paper trading capacity for 1–2 weeks to ensure it behaves as expected. Only after this probation will real funds be put at stake. This approach is aligned with industry practices: **always paper-trade a strategy before going live** [9].

Implementationally, shadow mode could be toggled via config (e.g., `mode: shadow` vs `mode: live`). In `trader.py`, if mode is shadow, it will fetch state, get action from model, but instead of executing an order, it will simulate it (update a paper portfolio and log the action). This is similar to a dry-run mode. Meanwhile, if an old model is truly running live concurrently, that would likely be handled by a separate process or instance. However, to keep things simple, we might choose that when a new model is in shadow test, the old model continues normal operation (maybe the old model is just the last model version still in live mode). The architecture should allow either two instances (one live, one shadow) or one instance that can handle both (the latter is more complex). A simpler route: during the initial development phase, shadow = paper trading with no real live trades at all. Once the first model proves itself, we switch to live = true. Thereafter, for updates, one could spin up a second instance on another port or thread for shadow. This detail can be refined in implementation, but the plan ensures the concept is clear.

In summary, **shadow mode ensures safer deployment** by running new models parallel to existing ones and comparing their predictions/outcomes with real-world data before any capital is allocated to them. This way, BetterBot mimics a "blue-green" deployment for models – only switch to the "blue" (new) model after it has proven its merit in shadow.

Model Registry & Versioning

BetterBot includes a **model registry** to organize and manage different versions of the predictive models and policy. Each time a new model is trained and validated, it is assigned a unique identifier and stored with all necessary artifacts for reproducibility. The registry design is as follows:

- Models are stored under the `/opt/BetterBot/models/` directory in sub-folders named by a unique key. The naming could use a timestamp plus a short hash of the model files or git commit. For example, after training on September 19, 2025, we might create `models/20250919_abcd1234/`. This folder encapsulates **all components of that model release**:
- **metadata.json:** A JSON file containing metadata such as:
  - `id` or `hash`: unique identifier.
  - `timestamp`: when the model was trained.
  - `train_data_range`: e.g., "2023-01-01 to 2025-09-18".
  - `performance`: key metrics from backtest (Sharpe, return, max drawdown, win rate, etc.).
  - `parent_model`: if it was based on fine-tuning a previous model.
  - `status`: e.g., "pending", "shadow", "active", "rejected".
  - `regime`: if this model is specific to a regime or "universal".
- **ppo_policy.zip:** The serialized PPO policy (from Stable-Baselines3 or RL library) after training. This includes the neural network weights of the policy and value function.
- **gru_model.pt:** The PyTorch model state for the GRU forecaster.
- **lgb_model.pkl:** The pickled LightGBM model.
- Optionally, any scaler objects or additional data prep objects (e.g., if features were scaled, the scaler parameters could be saved).

- Optionally, training logs or a snapshot of training configuration (learning rates, etc.) could be saved for reproducibility.

- The **registry module** (`registry.py`) provides functions such as `save_new_model(models_dir, metadata)` and `load_model(model_id)` to abstract the filesystem operations. It also might maintain an index (maybe a simple text file or just by scanning subdirs) to know what models exist. The `save_new_model` will:

- Compute a hash (maybe SHA-1 of the model weights or a combination of forecaster hashes).
- Create a folder for the new model.
- Write the metadata.json and save all model files into it.

- Possibly update a symlink `latest` to point to this new folder (if we mark it as the new active candidate).

- **Integration with Orchestrator:** After a successful training session, the orchestrator will call the registry to save the newly trained models. Initially, the new model might be marked as `status: "pending"` or `"shadow"` in metadata. After shadow testing (7 days) passes, a small update can be made to metadata (`status: "active"`) and the config can point to it as active. The old model could be marked `"archived"` or `"previous"`.

- **Regime-specific Models:** If we have separate models for different regimes, the registry will store each as its own entry, and the config (or a mapping in `regimes.yaml`) will reference them. For example, `regimes.yaml` might contain:

```
high_volatility:
  model_id: "20250919_abcd1234"
  threshold: 0.05   # volatility above 5% daily std dev triggers high-
vol regime
low_volatility:
  model_id: "20250801_xyz7890"
  threshold: 0.05   # implicitly below 5%
```

Each of those model IDs corresponds to folders in the registry. The system can load the appropriate model based on current regime (explained in next section). This means multiple models can be "active" in the sense of being used for different conditions, though typically one of them is in use at any given time.

- **Model Loading:** The trader, on startup or on regime change, will use `registry.load_model(id)` to get the needed objects. This function will handle loading the PPO policy (using stable-baselines `PPO.load()` for the zip), the GRU (using `torch.load` on the .pt into a GRU model class), and LightGBM (using `pickle.load` for the .pkl). It will then return these objects (or perhaps wrap them into a single object) to the trader. The trader must ensure it uses the matching forecaster outputs with the PPO that was trained on them (hence loading all from one folder). The metadata can also be loaded to check the age and other properties at runtime.

- **Ensuring Consistency:** The metadata validation includes confirming that all model files in a folder correspond to the same training run. One simple approach: include a `model_hash` field in metadata that was computed from a checksum of all three model files. At load time, we can recompute checksums to ensure no file was accidentally swapped or corrupted. This adds reliability.

- **Model Age Enforcement:** As mentioned, a config parameter (e.g., `model_max_age_days`) will be consulted by the trader. If the current active model's age (current date minus metadata.timestamp) exceeds this threshold, the trader will log a critical message and stop trading (or switch to a safe fallback strategy like all-cash). This "circuit breaker" prevents using an old model indefinitely. The threshold might be, for example, 30 days. The team should aim to retrain or at least revalidate models more frequently than this. This is in line with the expectation that ML models in non-stationary domains degrade over time and need periodic refresh [11]. The Telegram alert for this scenario would state something like " Trading halted: model older than 30 days (needs retraining)".

The model registry approach provides traceability (you can always refer back to an old model if needed), easy rollback (if a new model fails, you still have the old one to revert to), and supports parallel specialized models for different regimes or instruments. By storing all artifacts together, we ensure that the RL policy and its supporting prediction models are always in sync.

Regime Switching (High vs Low Volatility)

Financial markets often cycle through different regimes (e.g., high volatility vs low volatility periods), and a trading strategy might need to adapt its behavior accordingly [13] [14]. BetterBot incorporates a **config-driven regime switching** mechanism to handle two primary regimes defined as **High Volatility**

and **Low Volatility**. The system will adjust which model/policy to use, or adjust parameters, based on the current regime classification.

How regimes are defined: We use a volatility measure over a recent window as the criterion. For instance, we might compute the standard deviation of daily returns over the last 20 days. If this volatility exceeds a threshold (e.g., 5% daily), we label the market as high-volatility; if below, low-volatility. The exact threshold and window can be configured in `regimes.yaml`. We could also incorporate other regime definitions (like trending vs mean-reverting), but the task specifically calls for two regimes by volatility.

Config structure: The `configs/regimes.yaml` (or a section in the main config) will define the regimes. For example:

```
regimes:
  - name: "high_volatility"
    vol_threshold: 0.05    # e.g., std dev of returns >5%
    model_id: "20250919_abcd1234"
    params:
      risk_factor: 0.5     # example: smaller position fraction in high vol
      reward_penalty: 2.0  # example: increase volatility penalty lambda in
reward
  - name: "low_volatility"
    vol_threshold: 0.05    # implies <=5% is low vol
    model_id: "20250801_xyz7890"
    params:
      risk_factor: 1.0     # can invest more since vol is low
      reward_penalty: 1.0  # normal penalty
```

In this hypothetical config, we specify which model to use in each regime and possibly some parameter overrides. The `model_id` corresponds to entries in the model registry (allowing completely different trained policies optimized for that regime). Alternatively, one could use a single model that takes volatility as input, but training separate specialized models often yields better results when regimes diverge.

Switching logic: In the live trader loop, at each decision point (or at least periodically), the bot will: 1. Compute the current volatility metric (this can be done in the feature engineering step – e.g., feature vector includes a `current_volatility` value). 2. Determine the regime by comparing against the threshold. 3. If the regime is different from the last known regime, trigger a regime switch: - Log the regime change (e.g., "Regime switch: Low Volatility -> High Volatility detected, switching model/policy"). - Load the appropriate model for the new regime (via the registry). This means swapping out the PPO policy and possibly forecasters if they differ. - Adjust any runtime parameters according to the config for that regime. For instance, `risk_factor` could scale down position sizes in high volatility. If using a continuous action space for position sizing (not in our current discrete setup, but in future), this could be used. In our simpler discrete case (all-in or all-out), we might enforce that in high volatility, the bot maybe takes no leverage (already none) or even stays out if volatility is extreme. But since we are spot only, the main difference might be using a different trained policy that is more conservative in high vol.

1. Continue trading with the new regime's model.

The separation of models is justified because a policy tuned for low volatility might aggressively assume mean reversion or small range trading, whereas in high volatility that could be dangerous. Conversely, a high-volatility-optimized model might seize big swings but could be too inactive in calm markets. By **classifying market state and switching strategy accordingly**, BetterBot aims to remain profitable across regimes [14].

From an implementation perspective, this regime logic can be encapsulated in the `TradingEnv` or in the `trader.py` loop. A simple approach is to handle it in `trader.py`:

```
# Pseudocode
current_regime = None
while True:
    vol = features.compute_current_volatility()
    regime = "high_volatility" if vol > cfg["vol_threshold"] else
"low_volatility"
    if regime != current_regime:
        # Switch models
        model_id = regimes_cfg[regime]["model_id"]
        active_model = registry.load_model(model_id)
        current_regime = regime
        send_alert(f"Regime switched to {regime}, loaded model {model_id}")
    # ... use active_model to get action and execute ...
```

The above ensures the bot adapts on the fly. We also integrate regime logic into training: we might train separate models on subsets of data (one on historically high-vol periods, one on low-vol periods). The orchestrator could detect regimes in historical data and train or fine-tune models accordingly. For now, config-driven switching suffices as a framework, even if initially one might use a single model for both regimes until enough data is present to differentiate them.

In summary, regime switching is fully configurable: turning it off could be as simple as using the same model for both or setting a threshold that is never triggered. But the scaffolding is there to accommodate market regime adaptation, making the strategy more robust to changing market conditions. This approach aligns with modern adaptive trading systems that adjust policies when market dynamics shift [15] [16].

Alerts and Reporting via Telegram

Monitoring is a crucial part of a deployed trading system. BetterBot integrates **Telegram** alerts and daily reports to keep the operator informed of system status, performance, and any anomalies. Setting up Telegram involves creating a bot via BotFather and obtaining a bot token and a chat ID where the bot will send messages. These credentials are stored in the config (e.g., `telegram_token`, `telegram_chat_id`).

Real-time Alerts: - The system will send immediate Telegram messages for critical events: - **Order Execution Alerts** (optional): e.g., "Bought 0.01 BTC at €25,000". - **Error Alerts**: e.g., "⚠ API error: failed to fetch data, retrying", or " Trade aborted: model validation failed". - **Model/Regime Alerts**: e.g., when a regime switch happens, or when a new model goes live after shadow mode (" New model activated: id=20250919_abcd1234"). - **Stop-condition Alerts**: e.g., " Model age exceeds limit, trading halted!" or "⚠ Data gap detected: missing price data for 10 minutes".

These alerts ensure that if the bot encounters an issue or makes a significant change, the maintainers know immediately. The `utils/telegram_alerts.py` will have a function `send_alert(message)` that formats the message and posts it via an HTTPS request to Telegram's bot API endpoint.

Daily Summary Report: - Once per day (e.g., at 00:00 UTC or a convenient time), the bot will send a summary report. This can be triggered by a cron job calling `daily_report.sh` or directly scheduled in code. The report includes: - **System health**: CPU usage (perhaps average over last day), memory usage, and disk space available. This is to detect if the server is running out of resources. Using Python's `psutil` library, we can gather CPU and RAM stats easily. - **Uptime and Process Check**: Confirm that the trading process is running (if using a supervisor or just by checking last heartbeat in logs). - **Trading stats**: number of trades executed in last 24h, PnL of the last day (realized and unrealized), current position, current balance. - **Model info**: which model is active, how old it is (days since last train), and if shadow mode is in progress for a new model (if so, maybe include preliminary performance of that new model vs current). - **Data continuity**: e.g., "Data OK: no gaps" or if any gaps in price feed were detected (based on timestamp differences in data logs). - If any warning conditions exist (e.g., model getting old, or yesterday's performance was very poor), they can be highlighted.

The daily report message might look like:

```
[Daily Report]
- CPU: 5%, RAM: 1.2 GB (60%)
- Disk free: 20 GB
- Active Model: 20250901_abc123 (age 10 days)
- Trades: 5 trades yesterday, PnL: +2.3%
- Current Position: 0.005 BTC (long)
- Volatility Regime: Low
- No data gaps detected.
```

This concise report helps maintainers ensure everything is running smoothly without having to log into the server.

Technical Implementation: The Telegram functions will use the `requests` library or `http.client` to send a POST request to `https://api.telegram.org/bot<token>/sendMessage` with the `chat_id` and `text`. These calls should handle exceptions (e.g., network issues). We'll likely have a small retry logic or at least error catching so that a failure to send an alert doesn't crash the bot.

We will add a step in the installation to test the Telegram alerts (for example, a script or a flag to send a "Test alert from BetterBot" to ensure the setup is correct).

By integrating Telegram notifications, we add a layer of human oversight even when the bot is unattended. It is especially useful during the initial dry-run period [9] and will continue to be valuable in live trading, to know what the bot is doing or if it needs attention.

Fail-Safes and Risk Management

BetterBot is designed with several **fail-safe mechanisms** to avoid catastrophic errors and enforce good practices:

- **Data Leakage Prevention:** As noted, great care is taken that no future information is used in feature computation or model training (no label leakage). This is enforced by design (features are only from past data relative to decision point). Additionally, we plan a guard in `metadata_validator.py` to run during training: it could, for example, check that the correlation between any feature and the target (future return) at the same timestamp is low on the training data sample (an unexpectedly high correlation might indicate leakage). While not foolproof, it's a heuristic check. If any such test fails, training is aborted. This is aligned with the crucial tip to **always check for data leakage** in time-series ML [5].

- **Model Staleness Check:** As discussed under model registry, the trader will not use models older than X days (configurable). This prevents a scenario where the bot is left running the same model for months while market conditions change. We will set a reasonable default (e.g., 30 days). If triggered, the bot stops trading (goes to cash) and sends an alert. Resuming requires retraining a new model or explicitly overriding the age limit if needed.

- **Capital Exposure Limit:** Initially, the bot will only "play" with a small amount of capital. In paper mode, we simulate €500 as starting balance. When transitioning to real trading, we can enforce that the bot uses at most €500 of the real account. For instance, one could deposit €500 in a separate Bitvavo account or sub-account to limit risk. Alternatively, the bot's logic can be coded to never allocate more than €500 to a position (even if more funds are available). Since we are doing all-in/all-out trades in this simplified strategy, that means effectively the bot shouldn't trade with more than €500. This can be ensured by config (set base capital = 500 in config, and fetcher's balance call will treat anything above €500 as out-of-scope, or simply only trade a fixed amount each time). This way, even if a bug occurred, the maximum loss is limited. Over time, once confidence is gained, this limit can be raised or removed, and more funds can be added.

- **No Leverage or Shorting:** By strategy definition, we disable any leverage. Bitvavo currently doesn't offer margin trading anyway [17], which simplifies things. The bot will never sell more than it holds (which avoids short positions by default). If a short signal arises while holding no asset, the bot will just remain in cash. Keeping the strategy to long/flat not only reduces risk but also avoids compliance with margin requirements. It's explicitly documented and coded that `allow_short = False` and `use_leverage = False` throughout. This is aligned with the limitations noted for similar trading bots like Freqtrade (spot only, no margin) [7].

- **Error Handling and Retries:** All interactions with external systems (Bitvavo API, file I/O, etc.) will be wrapped in try-except blocks. For example, if a price fetch fails due to a network glitch, the bot can wait and retry a few times. If an order submission fails, it should catch the exception and decide: if it's a transient error, retry; if it's a critical error (like insufficient funds or API key invalid), log and alert and possibly stop. Logging of exceptions with stack traces to `trading.log` helps debug later.

- **Cron and Redundancy:** We plan to rely on cron for scheduling some tasks (detailed in next section). For the main trading loop, if we run it as a persistent process, we might use a process manager (like systemd or supervisord) to ensure the bot restarts on crash. Alternatively, running the trading execution via cron every minute (with checks to prevent overlaps) inherently restarts

it fresh each time. In either approach, if the bot stops, the daily report or a separate uptime monitor will catch that no trades have been made or process not running, and alert us.

- **Paper Trading Phase:** As emphasized, the bot will **start in paper-trading only**. This phase can last multiple weeks. During this time, we verify performance, robustness, and that all alerts and logs are working. Only after satisfactory paper results and possibly a code review will we insert the real API keys (enabling live trading). Even then, as mentioned, using limited funds ensures a cautious ramp-up. Essentially, the deployment will follow: Simulation -> Shadow/Paper with small real funds (like €500 in a real account, which one might consider semi-paper if loss is tolerable) -> Full live (bigger capital). This staged deployment is a key risk management practice.

- **Stop-Loss or Emergency Stop:** Although not explicitly requested, it's prudent to mention that we can implement a global stop-loss on the account. For instance, if cumulative losses exceed a certain threshold (say 5% of capital), the bot could pause trading and send an alert. This prevents it from spiraling down if something goes wrong. Similarly, if market conditions become extremely abnormal (e.g., exchange is not responding or giving weird data), the bot should halt rather than continue blindly.

Overall, these safety mechanisms ensure that BetterBot operates within controlled risk parameters and that any deviation or potential problem is quickly communicated and handled. Trading bots must be built with the expectation that anything that can go wrong will go wrong eventually; thus robust error handling and fail-safes are not optional.

Automation with Cron

Cron jobs will be set up to automate various aspects of BetterBot's operation, including data fetching (if separate from trading loop), training, reporting, and code synchronization. Below is a suggested **crontab** configuration (to be installed for the relevant user, e.g., via `crontab -e`):

```
# CRON schedule for BetterBot (minute hour day month weekday)
# 1. Live trading loop - runs every minute
* * * * * /usr/bin/python3 /opt/BetterBot/betterbot/executor/trader.py >> /
opt/BetterBot/logs/trading.log 2>&1

# 2. Daily training orchestration - runs at 03:00 AM every day
0 3 * * * /usr/bin/python3 /opt/BetterBot/betterbot/orchestrator/train.py
>> /opt/BetterBot/logs/training.log 2>&1

# 3. Daily report via Telegram - runs at 09:00 AM every day
0 9 * * * /opt/BetterBot/scripts/daily_report.sh >> /opt/BetterBot/logs/
report.log 2>&1

# 4. Backup important files - runs at 04:00 AM every day
0 4 * * * /opt/BetterBot/scripts/backup_models.sh >> /opt/BetterBot/logs/
backup.log 2>&1

# 5. Git sync (push code changes) - runs at 08:00 PM daily
0 20 * * * /opt/BetterBot/scripts/sync_to_repo.sh >> /opt/BetterBot/logs/
git_sync.log 2>&1
```

Explanation of these entries:

- **Trading loop:** `* * * * *` every minute we run the trading script. This assumes the script executes quickly (within less than a minute). The trader script will handle fetching data, making a decision, and possibly executing an order. If the strategy were faster or needed tick-by-tick, we would not use cron but a while loop internally; however, for simplicity and given likely low frequency (could even be every 5 minutes or 15 minutes depending on strategy), cron is fine. The log output is appended to `trading.log` so we have a continuous record of actions and any errors.

- **Training**: At 3:00 AM daily, we trigger `train.py`. This retrains models using up-to-date data (ensuring the model stays fresh). We schedule it at a time when markets might be quieter (depending on asset; crypto is 24/7, but 3 AM might be a low-volume time). Training daily might be overkill; it could be set to weekly or twice a week depending on how fast the model needs updating. We can adjust in config or via cron schedule. Training output is logged to `training.log`.

- **Daily Report**: At 9:00 AM every day (just an example time, can be whatever the user prefers), call the daily report script which sends the Telegram summary. We log it as well (though it's mainly just sending a message).

- **Backup**: At 4:00 AM, backup the models and logs via `backup_models.sh`. This likely rsyncs `/opt/BetterBot/models` and perhaps `/opt/BetterBot/logs` to a remote server or cloud storage (the backup script would be configured with the destination, possibly using scp or rsync to a NAS or S3 bucket). We schedule it after training so that if a new model was saved at 3:00 AM, by 4:00 AM the backup picks it up.

- **Git Sync**: At 8:00 PM daily, push any local code changes to GitHub. In normal operation, code changes might not happen automatically, but if we have any auto-generated updates (like maybe an updated README or data), or simply to ensure any tweaks we do via hotfix on server get versioned, this keeps the repo up to date. We might also call this script at the end of train.py (so that newly saved model metadata or any code changes from training are committed immediately). In any case, having a daily push as a safety net is useful. The deploy key is used here to authenticate [1].

Additional cron tasks could be added as needed (for instance, if we wanted periodic data fetch jobs separate from trading, or a heartbeat check). But the above covers the essentials. The `cron_setup.txt` file in `scripts/` will basically contain something similar to the above as documentation.

One important note: if using cron for the trading loop, ensure that if one instance takes longer than the interval, we don't start a second instance concurrently. In our case, since it's every minute and the tasks are likely quick, it's fine. If concurrency becomes an issue, we could add a lock file mechanism or simply run the trader in a persistent mode (via `@reboot` in cron or systemd service). However, running in cron has an advantage that it self-resets each minute, which can avoid certain memory buildup issues and ensures each tick is fresh.

We also include an `@reboot` entry (not shown above) if we want the bot to start on server restart:

```
@reboot /usr/bin/python3 /opt/BetterBot/betterbot/executor/trader.py >> /opt/BetterBot/logs/trading.log 2>&1
```

This would kick off the trading as soon as the machine boots (particularly useful if using a continuous while-loop trader approach instead of cron scheduling). In our cron above, `@reboot` isn't strictly needed since cron will handle it every minute anyway after boot.

With cron handling scheduling, we offload timing concerns to the system scheduler, making the implementation simpler and more robust (no need for multi-threading or async in our code for periodic tasks). All cron tasks are logged for auditability.

Git Commit & Deployment Instructions

For deploying BetterBot to a server and maintaining the codebase, follow these version control practices:

- **Git Commits:** Use meaningful commit messages when making changes. For example, if you adjust the reward function or fix a bug in feature engineering, commit with a message like `"Tuned reward function to increase volatility penalty"` [8] or `"Fixed data alignment bug in feature_engineering (no leakage ensured)"`. This will help track the evolution of the strategy. The `sync_to_repo.sh` script automates commits but uses a generic message; for significant changes, it's better to manually commit with context.

- **Deployment Key Setup:** As described earlier, generate a deploy key on the server and add it to GitHub. After adding the deploy key [18] [19], test it by doing `git push` from the server – it should succeed without asking for username/password. If there are multiple servers or staging environments, each can have its own deploy key (GitHub allows multiple deploy keys per repo).

- **Pulling Updates:** If development is also happening on a local machine (or by multiple collaborators on GitHub), when deploying a new version of code to the server, you can simply `git pull origin main` in `/opt/BetterBot` to fetch the latest changes. Our deploy setup is mostly about pushing from server (since the server might create data or model updates). To avoid conflicts, it's best to push any server-side changes (like updated configs or small tweaks done via vim on server) **before** pulling external changes, or use git branches. In a small team scenario, pushing everything from server (which includes model metadata, etc.) to the repo ensures a unified source of truth.

- **Printing Deploy Key:** It's useful to output the deploy public key as part of installation logs or via a script, so that the user can easily copy it. For instance, we can incorporate into `sync_to_repo.sh` (or a separate `print_deploy_key.sh`) a line to `cat ~/.ssh/betterbot_deploy_key.pub` and echo instructions. This is just a quality-of-life improvement during setup.

- **GitHub Repository Structure:** The repository on GitHub will mirror the `/opt/BetterBot` structure (except perhaps the `user_files` or data which are local). It will contain all the stub code, README, etc. After using Codex or manual coding to implement the actual logic in each stub, those will be committed. Sensitive info like actual config with keys should not be committed (use `config_template.yaml` in git and keep the real config out or in a secure store).

- **Commit Hooks (Optional):** We could set up a pre-commit hook to run tests or linters, ensuring code quality before commits. Also, a post-commit hook could potentially push automatically to remote (though our script handles scheduled pushes).

In summary, any changes to the bot should be tracked in Git. Using the deploy key and scripts, even non-development changes (like when the bot trains a new model and updates metadata) can be version-controlled. This is somewhat unusual (you typically wouldn't commit model binaries to Git frequently), but since we want a completely reproducible deployment plan, we mention it. In practice, large model files might be better stored in a storage bucket rather than Git due to size – an alternative is to use Git LFS for model files or have the `models/` directory not under version control. We leave this flexible; a reasonable approach is to exclude `models/` from git (to avoid huge repo) and rely on backups for those. The code and config, however, absolutely should be under Git.

Skeleton Code Structure and Next Steps

At this point, the project is structured and the plan is laid out for each component. The next step is to implement the actual functionality in each of the stub files according to this plan. This can be done manually by writing the code for each module, or by leveraging an AI coding assistant (like OpenAI Codex/GPT-4) to generate boilerplate and even detailed implementations given the specifications.

Below is a **GitHub-ready skeleton** summary (as might be seen in the repository tree), followed by an example prompt that could be used to guide Codex to create the full system from this specification.

```
betterbot/
├── fetchers/
│   └── data_fetcher.py         # Fetches data from Bitvavo API (REST/WS)
├── features/
│   └── feature_engineering.py  # Computes technical indicators, etc.
├── forecasters/
│   ├── lightgbm_model.py       # LightGBM forecaster class
│   └── gru_model.py            # GRU forecaster class (using torch)
├── ppo/
│   ├── env.py                  # Custom trading environment for PPO
│   └── policy_wrapper.py       # Wrapper to manage PPO policy (load/save/
infer)
├── orchestrator/
│   └── train.py                # Training orchestration script
├── executor/
│   └── trader.py               # Live trading execution script
├── utils/
│   ├── metadata_validator.py   # Data/model validation utilities
│   ├── registry.py             # Model registry management
│   └── telegram_alerts.py      # Telegram integration for alerts
└── configs/
    ├── config_template.yaml    # Template for user configuration
    └── regimes.yaml            # Definition of regimes and associated models
scripts/
├── sync_to_repo.sh             # Script to commit & push code to GitHub
├── backup_models.sh            # Backup models/logs via rsync
```

```
├── run_shadow_backtest.sh      # Run shadow model test and comparison
└── daily_report.sh             # Generate and send daily report
```

Each of these files is currently a stub (with basic structure and documentation). The final implementation will involve writing out the logic described in this plan: - Connecting to Bitvavo's API to get data and execute trades. - Computing features like moving averages, RSI, volatility. - Implementing training routines for LightGBM and GRU (likely requiring data preparation, normalization, etc.). - Setting up the PPO environment (this includes defining state, action, reward properly) and training the PPO agent (using Stable-Baselines3's `PPO` class, for example). - Ensuring that the policy wrapper can interface between the environment and the rest of the system (for inference and saving). - Handling the shadow mode operations and comparisons. - Testing regime switching by simulating different volatility conditions. - Integrating Telegram notifications and making sure the cron jobs execute as expected (maybe writing appropriate entries to crontab on installation, or instructing the user to do so).

Given the comprehensive nature of this plan, an AI code assistant could be utilized to expedite development. Below is a suggested **Codex implementation prompt** that can be used to systematically generate the code for BetterBot:

Codex Implementation Prompt

```
You are a coding assistant tasked with implementing a trading bot system
called BetterBot based on the following detailed specification.

**Overview:** BetterBot is a cryptocurrency trading bot for the Bitvavo
exchange that uses machine learning (LightGBM, GRU) and reinforcement
learning (PPO) to decide when to go long or stay flat on a given asset. It
includes modules for data fetching, feature computation, forecasting models,
an RL environment, training orchestration, live trading execution, and
utilities for model management and alerts.

Use the given project structure and implement each component accordingly.
**Write clean, well-documented Python code** for each module. Include
comments and docstrings to explain the logic. Ensure no placeholders remain;
each function or class should have at least a basic working implementation
that matches the specification.

**Project Structure and Requirements:**

1. `betterbot/fetchers/data_fetcher.py`:
   - Implement `DataFetcher` class to interface with Bitvavo API. Use the
Bitvavo Python API client or REST calls.
   - Methods: `get_ticker(market)` returns current price for the given market
(e.g., "BTC-EUR"). Possibly also methods for getting recent OHLC data (e.g.,
`get_ohlc(market, interval)`).
   - Handle API keys and init through constructor.
   - This will be used by both training (to get historical data) and live
trading (to get latest price).

2. `betterbot/features/feature_engineering.py`:
```

```
   - Implement functions (or a class) to compute features from raw data.
E.g., a function `compute_features(price_series)` that returns a dict of
features (moving averages, RSI, volatility, etc.) for the latest timestep.
Use libraries like pandas or ta-lib for indicators if needed.
   - Ensure no forward-looking usage (only past data). Possibly include an
example calculation for volatility (std dev of last N returns).
   - This will be used in both training (to prepare feature matrix for ML
models) and real-time (to compute the state for RL).

3. `betterbot/forecasters/lightgbm_model.py`:
   - Implement `LightGBMForecaster` class.
   - It should handle training (`train(X, y)`) using LightGBM (e.g., using
`lgb.LGBMClassifier` or `Regressor` depending on formulation) and prediction
(`predict(X)`).
   - Save the model to file (`save_model(filepath)`) and load
(`load_model(filepath)`) using LightGBM's built-in save/load or pickle.
   - This model likely predicts something like next price movement or return.

4. `betterbot/forecasters/gru_model.py`:
   - Implement `GRUForecaster` class using PyTorch (or Keras if preferred,
but PyTorch given .pt saving).
   - This should define a GRU neural network for time series prediction. For
simplicity, you can create a small `nn.Module` with a GRU layer and a linear
output.
   - Provide `train(X, y)` which trains the network (one could use PyTorch
training loop) and `predict(X)` which does a forward pass.
   - Include `save_model(path)` and `load_model(path)` to persist the model
(`torch.save` and `torch.load` of `state_dict`).
   - Ensure it's designed to accept sequential data (X could be shaped
[batch, seq_len, features]). In live usage, it might use last N days of
features to predict next day's price change probability, for example.

5. `betterbot/ppo/env.py`:
   - Implement a custom OpenAI Gym environment `TradingEnv` for our trading
scenario.
   - Define `__init__` to accept perhaps the forecasters or their outputs as
part of environment (or they can be globally accessible).
   - The observation space can be Gym Box (continuous) with dimension equal
to number of features plus number of forecaster outputs.
   - The action space is Gym Discrete(3) for {0: Hold, 1: Buy, 2: Sell}.
   - In `reset()`: initialize the environment state (starting with no
position, initial balance, etc.) and maybe prepare initial observation from
data.
   - In `step(action)`: apply the action:
     - If Buy and currently no position, buy as much as possible (or a fixed
unit, but we assume full allocation).
     - If Sell and currently holding, sell all.
     - If Hold, do nothing.
     - Update position status and cash balance accordingly (simulate trade
execution).
     - Advance the environment by one timestep (which could correspond to
```

moving to next index in historical data for training).
     - Calculate reward = (change in portfolio value) - lambda*volatility_penalty - transaction_costs. For training environment, we can compute actual reward from data (since in backtest we know price changes).
     - Return observation (next state), reward, done flag (maybe done at end of dataset or if a certain number of steps).
   - The environment will be used by PPO for training. Provide any additional methods needed (like a method to set the data/episode for training on historical sequences).
   - Also ensure that the environment can incorporate regime logic if needed (e.g., maybe an input feature that indicates current regime, or separate handling, but initial version can ignore regime for training and focus on one regime or full data).

6. `betterbot/ppo/policy_wrapper.py`:
   - This will interface with stable_baselines3 PPO.
   - Possibly define a `PPOPolicyWrapper` class that has methods to `train(env)` (which creates a PPO model, trains it on the given env for a certain number of timesteps), `save(path)` and `load(path)`.
   - Also a method `predict(obs)` that loads the model (if not already loaded) and returns an action (with no exploration, deterministic).
   - This wrapper basically isolates the SB3 dependency and simplifies usage in our code.

7. `betterbot/orchestrator/train.py`:
   - This script glues everything together for training:
     1. Use DataFetcher to get historical data (e.g., last 100 days or more).
     2. Compute features for the training period.
     3. Prepare training dataset for forecasters (e.g., features X and labels y for next-day return or next-period price movement).
     4. Train LightGBMForecaster on this data, save model.
     5. Train GRUForecaster on sequence data (you may need to prepare sequences of features as input and next-step outcome as label), save model.
     6. Initialize TradingEnv with this historical data and the trained forecasters (or their outputs). The environment should simulate the market over the historical period.
     7. Train PPO via policy_wrapper on this environment. Choose number of training timesteps (could be a hyperparameter, e.g., 50k timesteps).
     8. After training, evaluate the policy on a validation set (or the tail of the training set via a separate env run without training). Gather performance stats.
     9. If performance is good (you can define some condition or always accept for now), save the PPO policy.
     10. Use registry to save the new model: create new folder, save ppo, gru, lightgbm, write metadata (with performance metrics).
     11. Possibly initiate shadow mode: e.g., set new model status to "pending" and maybe trigger the `run_shadow_backtest.sh` via subprocess or just log that it's ready for shadow testing.
     - Make sure to handle exceptions and log progress. This script will be run by cron, so any print or exception will go to log.

```
8. `betterbot/executor/trader.py`:
   - This is the live trading loop script:
   - Load config (to know API keys, mode, etc.).
   - Initialize DataFetcher with API keys.
   - In paper mode: initialize a simulated portfolio (cash = 500 EUR,
holdings = 0).
   - Load the latest active model from registry (metadata might tell which
folder is active model for given regime).
   - Possibly load both high-vol and low-vol models if regime switching is
enabled (to avoid delay on the fly).
   - Then either enter an infinite loop with sleep, or if using cron every
minute, just do one iteration per run.
   - Each iteration:
     - Fetch latest price (and any other needed market info).
     - Compute current features.
     - Determine regime (compute vol using recent data window).
     - If regime switching enabled:
       * If current regime != last_regime, load the appropriate model (or if
models already loaded, select appropriate one for action).
     - Get forecaster predictions: run LightGBM and GRU on latest features
(they should output something like a predicted return or probability).
     - Construct observation vector = [features, lgb_pred, gru_pred].
     - Query PPO model for action: e.g., `action = policy.predict(obs)`.
     - Execute action:
       * If mode == "paper": simulate trade: update cash and holdings based
on action and price, and log the action.
       * If mode == "live": use DataFetcher's Bitvavo client to place real
orders (e.g., `createOrder` API call with market = asset, side = buy/sell,
amount or funds = appropriate).
         - Only use at most €500 (or the available balance) for buys. For
sell, sell only the amount we hold.
         - Use immediate-or-cancel or market orders for simplicity, or limit
orders at current price.
         - After an order, update internal portfolio state (for paper and
maybe also check actual fill via API if live).
     - Log the action and relevant info (price, new balance, etc.) to
trading.log.
     - If any exception or API error, catch it, send Telegram alert, and
break or continue safely.
     - (If using cron, the loop is just one iteration; if using while true,
include a `time.sleep(interval)` at end like 60 seconds.)
   - Ensure that if model age > threshold, we do not execute: instead send
alert "model expired, no trading".
   - Ensure to send alerts for critical events (via telegram_alerts
functions).
   - At the end of the run (or periodically), maybe output a heartbeat log
"Still running..." (especially if in a persistent loop).
   - Cleanly handle process termination (e.g., on KeyboardInterrupt if manual
stop, print a message).
```

```
9. `betterbot/utils/metadata_validator.py`:
   - Implement functions to validate data and models. For example:
     - `check_no_leakage(feature_df, target_series)`: simple check to ensure features are not including target (like correlation check or check that no feature column name includes something like 'future' or 'target').
     - `check_data_complete(data_df)`: ensure no large gaps or NaNs in data.
     - `validate_model_files(folder)`: ensure ppo, gru, lgb files exist and perhaps are readable.
   - These can be used in train.py after data preparation and after saving model.
   - If any check fails, raise an Exception or return False so that caller can handle (e.g., abort training or trading).

10. `betterbot/utils/registry.py`:
    - Implement model registry as discussed:
    - Perhaps a function `register_model(new_id, lgb_model, gru_model, ppo_model, metadata)`:
      * Create folder models/new_id
      * Save files there (use forecasters' save_model and policy_wrapper.save)
      * Write metadata.json.
      * Optionally, update a "latest.txt" or symlink.
    - A function `load_model(model_id)`:
      * Read metadata (for any info needed).
      * Load LightGBM, GRU, PPO from that folder (returns a dict or object containing those).
    - Could also have `get_latest_model()` that reads the latest symlink or a config pointer to load current active model.
    - Aim for atomic operations (write temp then rename, etc., for safety if needed).
    - Also handle cleanup of very old models if desired (could keep last N models only, optional).

11. `betterbot/utils/telegram_alerts.py`:
    - Implement `send_alert(message: str)` and possibly `send_report(message: str)` if needing different formatting.
    - Use the `requests` library to POST to Telegram API. (Pre-configure the API URL with the token from config).
    - Also possibly a `format_report(data)` helper to create the daily report text.
    - This module will be used by trader and daily_report.sh.
    - Include basic error handling (if Telegram API call fails, perhaps log it).

12. `scripts/daily_report.sh`:
    - This Bash script can simply call a Python function, or we could implement the report in Python entirely.
    - One approach: have a `betterbot/utils/daily_report.py` that when run, gathers the info and calls telegram_alerts.send_report.
    - But since not specified as a module, we can implement logic directly in the bash script using `python -c "import ...; ..."` calls or just call an
```

````
existing function via a small Python entrypoint.
    - For simplicity, consider creating a `betterbot/utils/daily_report.py`
to generate and send report, and then `daily_report.sh` just calls that:
      ```bash
      #!/bin/bash
      /usr/bin/python3 -c "from betterbot.utils import daily_report;
daily_report.send_daily_report()"
      ```
      where `send_daily_report()` internally gathers CPU (using psutil), etc.
    - (Implement daily_report.py accordingly with psutil usage).

13. `scripts/backup_models.sh`:
    - Use `rsync` (or scp) to a configured backup location. This could be an
environment variable or config (like `backup_target: user@backupserver:/path/
`).
    - Provide flags for rsync to only copy changes.
    - Also perhaps compress logs or rotate them if needed (out of scope, but
worth noting).
    - If no remote given, it might just copy to a `/opt/BetterBot/backups/`
locally.

14. `scripts/sync_to_repo.sh`:
    - Already outlined earlier, just ensure it uses the correct SSH key.
Possibly set `GIT_SSH_COMMAND="ssh -i ~/.ssh/betterbot_deploy_key"` before
`git push` to ensure it uses the right key.
    - Make it executable (`chmod +x` in the repo or instruct to do so after
cloning).

15. `scripts/run_shadow_backtest.sh`:
    - This script can utilize the orchestrator or some backtesting utility:
      * It might call a Python function like `train.py --backtest <model_id>
<days>` or we write a separate `shadow_test.py` in utils that takes the
latest model and runs it on last 7 days of data (perhaps using the TradingEnv
in a purely evaluative mode).
      * Then compare with either previous model or a baseline strategy
(baseline could be "buy and hold" or "do nothing").
      * Compute metrics and output to a log file and maybe Telegram alert
summarizing "Shadow test result: New model Sharpe=1.2 vs Old model
Sharpe=1.0, proceeding to deploy."
      * For simplicity, implement this in Python and just call it here.

16. Configuration handling:
    - There should be a way to load config (YAML) in various parts (train.py,
trader.py, etc.). Possibly create a small util in Python to read YAML and
return a dict (`import yaml`).
    - This config will supply API keys, thresholds, etc. Use it throughout
instead of hardcoding values.

Remember to maintain consistent logging practices (use Python’s logging
module for complex parts, or simple print statements redirected to log files
via cron). Test each module individually if possible.
````

```
Now proceed to implement each file in the project structure as specified.
```

(The above prompt can be fed to an AI code generation system to create the initial codebase. Human review and testing would follow to fine-tune the logic.)

By following this deployment plan and using the skeleton and prompt to guide implementation, we will have a fully functional **BetterBot** trading system. It will be organized, configurable, and prepared for careful testing in shadow mode and paper trading, eventually leading to a robust live trading deployment. The combination of ML forecasters and an RL policy, along with stringent evaluation and monitoring, gives BetterBot a strong foundation to navigate different market conditions while managing risk responsibly. [12] [14]

---

[1] [18] [19] How to Use GitHub Deploy Keys – Dylan Castillo
https://dylancastillo.co/posts/how-to-use-github-deploy-keys.html

[2] [3] [10] [11] How to build an ML based Crypto Trading Bot using Python? | by Oliviervh | Medium
https://medium.com/@oliviervh/how-to-build-an-ml-based-crypto-trading-bot-using-python-e617562e6f4f

[4] Bitvavo trade volume and market listings | CoinMarketCap
https://coinmarketcap.com/exchanges/bitvavo/

[5] [12] Model Evaluation in Machine Learning: Tips and Techniques | EC Innovations
https://www.ecinnovations.com/blog/model-evaluation-in-machine-learning-tips-and-techniques/

[6] Deep neural network approach integrated with reinforcement learning for forecasting exchange rates using time series data and influential factors | Scientific Reports
https://www.nature.com/articles/s41598-025-12516-3?error=cookies_not_supported&code=4e8aed2e-6dfb-43b0-b2c5-a7dc086b109e

[7] [9] Where To Start With Freqtrade | Freqtrade Stuff
https://brookmiles.github.io/freqtrade-stuff/2021/04/20/where-to-start-with-freqtrade/

[8] [13] [14] [15] [16] Building an Adaptive Reinforcement Learning Agent for Regime-Switching Financial Markets | by Sammarieo Brown | Medium
https://medium.com/@sammarieobrown/building-an-adaptive-reinforcement-learning-agent-for-regime-switching-financial-markets-04ecc43ef7dc

[17] Bitvavo vs Coinbase: Features, Fees & Security Compared - Kyrrex
https://kyrrex.com/blog/bitvavo-vs-coinbase