MemeBot 3 Sniper Bot Technical Reference Manual

Installation & Environment Setup

Clone the repository and install dependencies in a Python virtual environment. For example:

```
git clone https://github.com/mudanzasalegre/memebot3.git
cd memebot3
python -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt
cp .env.sample .env   # copy the sample .env and edit it
```

Edit the .env file to provide your Solana RPC URL, wallet keys, API keys, and any custom thresholds. At minimum set SOL_RPC_URL and SOL_PRIVATE_KEY (or leave in paper mode). Then run the bot in dry-run mode to test:

```
python -m run_bot --dry-run --log
```

Logs will be written to the logs/ directory (rotating hourly). You can increase verbosity by setting LOG_LEVEL=DEBUG in .env to see every decision.

The bot's Python version, OS, or other prerequisites are standard for Solana development (e.g. an up-to-date Solana SDK installed). Ensure your Python environment has access to the internet for API calls. The code is modular; the main orchestration happens in run_bot.py.

Brief Main Loop

MemeBot uses Python's asyncio extensively. The primary entrypoint (run_bot.py) drives an **async event loop** that schedules tasks for each stage (fetching, filtering, trading, monitoring). For example, price checks for all open positions are batched via await asyncio.gather to Jupiter's API in parallel. Similarly, the bot listens to DexScreener HTTP pulls and Pump.fun WebSocket concurrently. Highlight that the Pump.fun listener runs in an asyncio task with built-in reconnect/backoff, and the DLX fetchers use async HTTP (subject to rate-limit throttling).

At startup the bot launches an async retrain_loop() that sleeps until the configured day/hour (default Thursday 04:00 UTC) and then invokes model training. This task runs in the background without blocking the main trading loop. All async tasks are managed by the main loop, so coroutines can run truly concurrently (subject to I/O waits).

Configuration via .env

All bot behavior is controlled by environment variables in the .env file. These are loaded into a global config (CFG) on startup. Key settings include:

- **Mode & Logging:**
  - DRY_RUN (1 or 0): if 1, no real blockchain transactions are made (paper mode). If 0, the bot uses your Solana private key to sign actual trades.
  - LOG_LEVEL (DEBUG/INFO/etc): logging verbosity.
  - LOCAL_TZ: your local time zone (e.g. Europe/Madrid) used for any time-based trading windows.
- **Paths & Files:**
  - FEATURES_DIR: directory for feature-store Parquet files (default data/features).
  - LOG_PATH: directory for log output (default logs/).
  - MODEL_PATH: path to the ML model file (defaults to ml/model.pkl).
  - SQLITE_DB: path to the SQLite database for positions and token info (default data/memebotdatabase.db).
- **Solana Wallet & RPC:**
  - SOL_PRIVATE_KEY: your Solana wallet private key in base58 (required if DRY_RUN=0).
  - SOL_PUBLIC_KEY: your wallet address (public key) in base58. Used for balance checks.
  - SOL_RPC_URL: Solana RPC endpoint: (default https://api.mainnet-beta.solana.com). Use a reliable node URL.
- **API Endpoints & Keys:**
  - DEX_API_BASE: URL for DexScreener (default https://api.dexscreener.com).
  - RUGCHECK_API_BASE: base URL for the RugCheck API.
  - RUGCHECK_API_KEY: API key for RugCheck. Without it, rug risk always scores 0.
  - HELIUS_API_BASE and HELIUS_RPC_URL: base URL and RPC for Helius (for holder cluster checks). Requires HELIUS_API_KEY to function.
  - GMGN_API_BASE: base URL for the Jupiter/GMGN swap router API (default https://api.gmgn.ai).
  - GECKO_API_URL: GeckoTerminal API base for price/liquidity.

  - BITQUERY_TOKEN, BIRDEYE_API_KEY, etc.: optional API keys for on-chain analytics or price data. Any missing keys simply disable or fall back on free data.

- **Discovery & Timing:**
  - DISCOVERY_INTERVAL (seconds): how often to fetch the latest pairs from DexScreener (e.g. 90s).
  - SLEEP_SECONDS: main loop delay between iterations.
  - VALIDATION_BATCH_SIZE: how many queued tokens to process per loop (e.g. 24).
  - MAX_CANDIDATES: max new addresses to enqueue per DexScreener fetch (e.g. 50).
  - MAX_QUEUE_SIZE: cap on in-memory queue length (e.g. 300).
  - TRADING_HOURS: allowed trading hours window(s) (e.g. "09:00-23:00").
  - BLOCK_HOURS: hours when trading is forbidden (bots will requeue tokens, not terminate).

- **Filtering Thresholds:**
  - MIN_AGE_MIN: minimum token age (in minutes) before buying (default ~0.2–8 min). If a token is younger, it is requeued ("too young").
  - MAX_AGE_DAYS: maximum age of token to consider (default ~1–2 days). Tokens older than this are ignored.
  - MIN_LIQUIDITY_USD: minimum pool liquidity in USD (e.g. ~$3,000–$3,500).
  - MIN_VOL_USD_24H: minimum 24h trading volume (e.g. ~$7,500).
  - MAX_24H_VOLUME: upper bound on volume (tokens with extremely high volume, e.g. > $80M, are skipped).
  - MIN_MARKET_CAP_USD / MAX_MARKET_CAP_USD: minimum/maximum market capitalization (e.g. min ~$3k–$5k, max ~$400k–$8M).
  - MIN_HOLDERS: minimum number of token holders (default 0 or 1). If holders=0 after an initial period, it may defer or reject.
  - BANNED_CREATORS: comma-separated list of known bad deployer addresses to block outright. Tokens created by these addresses are skipped.

- **Scoring & Risk Parameters:**
  - MIN_SCORE_TOTAL: minimum soft score required to buy (default often 0–30 for data collection; raise to 50+ for strict trading).

- **Risk Management:**
  - TRADE_AMOUNT_SOL: amount of SOL to spend per buy (e.g. default 0.15 SOL; set to 0 for pure simulation).
  - TAKE_PROFIT_PCT: profit target percentage (e.g. 25%). When price reaches this above the buy price, a partial profit is taken.
  - STOP_LOSS_PCT: loss threshold (e.g. 20%). If price falls this far below buy price (after the early window), exit.
  - TRAILING_PCT: trailing stop percentage (e.g. 25%). Once profit is made, if price falls by this fraction from its high, exit remaining position.
  - EARLY_WINDOW_S and EARLY_DROP_PCT: early-rug window (e.g. first 600 seconds) and drop percentage (e.g. 12%). A large drop within the first few minutes triggers an immediate sell.
  - MAX_HOLDING_H: maximum holding time in hours (e.g. 3h). Positions older than this are force-closed.
  - MAX_HARD_HOLD_H: optional extension if in profit (e.g. +1h beyond MAX_HOLDING_H) to let winners run.
  - NO_EXPANSION_MAX_PCT: if set (e.g. 0%), exit if after 1h the position has not gained at least this much (default disabled).
- **Liquidity Safeties:**
  - KILL_EARLY_DROP_PCT: same as EARLY_DROP_PCT; set to 0 to disable early-drop kill.
  - KILL_LIQ_FRACTION: drop fraction (e.g. 0.7 for 70%). If current pool liquidity falls below this fraction of the buy-time liquidity, force-sell ("LIQUIDITY_CRUSH"). Set to 0 to disable.
  - LIQ_CRUSH_WINDOW_MIN and LIQ_CRUSH_DROP_PCT: optional drop-over-time criterion (e.g. ≥35% drop within 10 minutes triggers sell).
- **Machine Learning (ML) Parameters:**
  - AI_THRESHOLD: probability cutoff for the ML classifier (default e.g. 0.50). Tokens with ML_predict < AI_THRESHOLD are skipped.
  - AI_THRESHOLD_FILE: path to a JSON file that can store a "recommended_threshold" for dynamic adjustment.
  - RETRAIN_DAY, RETRAIN_HOUR: weekly retraining schedule (defaults: Sunday at 04:00 UTC). Set RETRAIN_DAY negative to disable.
  - TRAINING_WINDOW_DAYS: how many days of data to use when retraining (default ~30 days).
- **Other flags:**
  - REQUIRE_JUPITER_FOR_BUY: if true, the bot will only buy via Jupiter (skip if no Jupiter route).

- FORCE_JUP_IN_MONITOR: if true, always use Jupiter for price quotes during monitoring (else DEX is tried first if the buy was non-Jupiter).
- IMPACT_MAX_PCT: maximum allowed price impact percentage (e.g. 8%). If estimated impact exceeds this, the buy is aborted.

The above list is illustrative and not exhaustive. The full environment reference is included with the code. All these settings can be tuned without code changes to control bot behavior.

Performance & Rate Limits

The bot exposes throttle settings in .env, such as BIRDEYE_RPM and GECKO_RATE_LIMIT, which control how many API calls per minute are made. These defaults are usually safe, but can be lowered if you see rate-limit errors. The multi-source fetch logic caches recent failures briefly – for instance, if the BirdEye or Gecko endpoint fails repeatedly, the code will skip it for a short TTL and try the next source instead. Batch processing is used wherever possible: e.g. price quotes for all open positions are fetched in one batch to reduce latency. The validation queue itself imposes batching: only a fixed VALIDATION_BATCH_SIZE number of new tokens are processed each cycle (to keep CPU/RPC loads bounded).

Because new DexScreener tokens are enqueued in a ring buffer with caps, discovery bursts won't crash the bot." If the queue is full, extra tokens are quietly dropped (with a warning) to prevent runaway memory usage. These controls ensure the bot remains responsive under load.

Internal Architecture & Modules

MemeBot 3 is organized into clear modules, each handling a facet of operation:

- **Configuration (config/):** Reads the .env file and fills a CFG dataclass. All thresholds, API URLs, keys, and global constants are defined here.
- **Data Fetchers (fetcher/):**
  - *PumpFun:* fetcher/pumpfun.py maintains a WebSocket to Pump.fun (via pumpportal.fun), receiving real-time mint events for new tokens on Solana. It buffers recent events and flags new tokens with discovered_via = "pumpfun".
  - *DexScreener:* fetcher/dexscreener.py periodically polls DexScreener's Solana API for the latest trading pairs (every DISCOVERY_INTERVAL seconds). It returns up to MAX_CANDIDATES new token addresses per poll.
  - *RugCheck:* fetcher/rugcheck.py queries the RugCheck API for a safety score (0–100) on demand.
  - *Helius Cluster:* fetcher/helius_cluster.py uses the Helius API to fetch top holders of a token and detect concentration (flagging a "dev cluster" if >20% of supply is in the top 10 addresses).

  - *Socials:* fetcher/socials.py looks at DexScreener's token "profile" (and any metadata) for social links (Twitter, Telegram, Discord, website).
  - *Price Oracle:* jupiter_price.py and jupiter_router.py (optional) obtain up-to-date price quotes via the Jupiter aggregator. utils/price_service.py also integrates with GeckoTerminal and BirdEye to fill missing prices and liquidity data.

- **Data Queue (utils.lista_pares):** All new tokens (from PumpFun or DexScreener) are placed in an in-memory FIFO queue (capped by MAX_QUEUE_SIZE). The queue stores metadata along with each token address. The main loop processes this queue in batches (VALIDATION_BATCH_SIZE) to throttle API usage. A ring buffer and cache ensure PumpFun events aren't processed too many times.

- **Analytics & Filters (analytics/):**

  - *Hard Filters:* analytics/filters.py::basic_filters() implements the core rejection rules (see next section). Each token yields True (pass), False (reject), or None (defer/requeue). There are also helper modules like analytics/insider.py (detect deployer dumping) and analytics/trend.py (optional trend analysis).
  - *Soft Scoring:* analytics/filters.py also has a total_score() function that sums weighted signals (rug risk, holder distribution, social presence, etc.) into a composite soft score. This is compared against MIN_SCORE_TOTAL.
  - *Machine Learning:* analytics/ai_predict.py loads a pretrained model (LightGBM by default, from ml/model.pkl) on demand. It exposes should_buy(features) which outputs a probability. The features/ subpackage contains code that builds the feature vector (e.g. features/builder.py).

- **Trading Engine (trader/):**

  - *Buyer (trader/buyer.py):* Performs the buy operation. In real mode it uses trader/gmgn.py to interact with the GMGN (Jupiter) API. This fetches an optimal swap route and a raw unsigned transaction. The bot signs the transaction locally with the Solana private key (sol_signer.py) and submits it to the network. The response includes the number of tokens bought and the transaction signature. In dry-run mode, trader/papertrading.py simulates this process and returns synthetic buy results.
  - *Seller (trader/seller.py):* Handles sells. It prefers to sell via Jupiter by default (using the Jupiter router for best execution). If Jupiter isn't available or liquidity is low, it can fall back to GMGN swaps (guarded by MIN_LIQUIDITY_USD to avoid illiquid trades). The seller returns the effective sell price and a signature/indicator. In dry-run, the paper trader simulates sells similarly.

  - *Papertrading (trader/papertrading.py):* In dry-run mode, the bot maps both buying and selling to this module. It fetches prices via Jupiter's public price API (or a fallback calculation using SOL/USD rate) but does not sign any transactions. It enforces the same Jupiter-route requirements and impact checks as in real mode, logging skip reasons if a simulated trade is not allowed.

- **Database (db/):** A lightweight SQLite database (using SQLAlchemy) stores persistent data. The main tables are **Position** (one per open or closed trade) and **Token** (metadata if needed). After each buy or sell, the bot records entries here: token mint, symbol, quantity, buy/sell price, timestamps, exit reason, etc. On restart, the bot reloads this DB to resume monitoring open positions and to avoid buying duplicates (it checks for existing open positions on a token before attempting to buy it).

- **Logging:** A centralized logger handles both console output and rotating file logs. At DEBUG level the bot logs every filter decision, token pipeline status ("too young", requeues, rejects, buys, sells, etc.) for troubleshooting. Counters aggregate key metrics (tokens processed, buys/sells, reasons for drop, etc.).

This modular design keeps concerns separate: fetching raw data, filtering/scoring, executing trades, and data logging happen in different parts of the codebase.

Real-Time Token Discovery

MemeBot 3 continuously scans for new memecoin opportunities using two parallel feeds:

- **Pump.fun WebSocket Stream:** On startup, the bot connects to Pump.fun's **PumpPortal** via WebSocket. Whenever a new SPL token is minted on Solana, Pump.fun pushes an event. The bot reads these events into an in-memory buffer (size configurable, with a short time-to-live to avoid backlog). Each event is normalized into a token dictionary (fields like address, symbol, name, created_at, etc.). Pump.fun tokens are tagged discovered_via="pumpfun". These bypass the usual queue and go directly to filtering/evaluation, since they are potentially the very newest tokens. (Because liquidity may not exist yet, the bot relaxes certain checks for these: e.g. allows 60% of normal liquidity and a higher market cap limit on an initial basis.)

- **DexScreener Polling:** Independently, every DISCOVERY_INTERVAL seconds the bot calls the DexScreener API to fetch the latest ~500 Solana trading pairs. It extracts new token addresses from this feed.

  The helper fetcher.dexscreener.fetch_candidate_pairs() returns up to MAX_CANDIDATES fresh addresses per poll. These addresses are sanitized and deduplicated. Any new address is enqueued into the **validation queue** for later filtering.

The utils/price_service.py module tries Jupiter's price API first by default, then falls back to alternate oracles (BirdEye, GeckoTerminal, DexScreener) in a priority chain. The code is prepared to handle Jupiter impact/slippage via fetcher/jupiter_router.py when USE_JUPITER_IMPACT=true. If one source fails or is slow (e.g. BirdEye rate-limited), the bot automatically moves to the next source. In practice, failures are cached briefly to avoid hammering a downed service. Likewise, if an expected data field (like liquidity or price) is missing, the code returns safe defaults. For example, without a RugCheck API key the rug score is simply 0 (no penalty). All web calls are wrapped to catch exceptions quietly and either retry or skip – e.g. the Pump.fun WebSocket uses a reconnect backoff to avoid bans, and API wrappers issue warning logs but do not crash the bot if a service is unreachable.

All discovered tokens (from Pump.fun or DexScreener) first have their data sanitized (utils.data_utils.sanitize_token_data): numeric fields (liquidity, volume, price) are converted to floats, missing values are filled, and a age_min (minutes since creation) is computed. This ensures consistency. New tokens are logged (at DEBUG) with a summary line (symbol, liq, vol24h, mcap, age, source).

The **internal queue** (utils.lista_pares) is a simple FIFO with metadata. It is capped by MAX_QUEUE_SIZE to prevent runaway memory use. The main loop processes this queue in batches of size VALIDATION_BATCH_SIZE, so that if many tokens flood in, the bot throttles its filter pipeline to manage API limits and CPU use. Emphasize that MemeBot's internal queue (managed by utils.lista_pares) has a fixed size to apply backpressure. The queue is capped at MAX_QUEUE_SIZE (default 250); any new tokens beyond this limit are simply dropped until space frees up. A background metric tracks the number of tokens pending, requeued, or in cooldown. The main loop processes tokens in batches (up to VALIDATION_BATCH_SIZE per iteration) to control throughput.

Hard filters may return **None** to indicate "incomplete data" (e.g. a newly minted token has no liquidity yet). In that case, the token is re-enqueued with a delay. The code enforces two limits: **INCOMPLETE_RETRIES** (default 3) controls immediate quick retries for missing data, and **MAX_RETRIES** (default 5) limits total requeue attempts for any token. If a token exhausts these retries without passing the filters, it is dropped. Requeued tokens experience an incremental backoff: for example, delays of ~60s, 3m, 7m... between attempts. These deferrals are logged with a "⏳ symbol (e.g. "⏳XYZ... → too_young, will retry").

Duplicate/Position Check: Before any filtering, the bot queries the SQLite DB to see if it already holds an open position in this token. If so, that token is removed from the queue and ignored (to avoid double-buying the same coin).

Hard Filtering Criteria and Requeue Mechanisms

Each queued token enters a multi-stage **hard filter** pipeline (analytics/filters.basic_filters). A token can pass (True), be rejected (False), or be marked indeterminate (None) meaning "try again later." Key hard-filter rules include:

- **Time Window Gate:** The bot first checks the current time against TRADING_HOURS and BLOCK_HOURS. If the local time is outside allowed trading hours or within a blocked interval, the token is immediately requeued (with reason "off_hours" or "blocked_hour") to be evaluated in the next cycle. This ensures no buys happen when the user doesn't want to trade.
- **Solana Address Check:** The token must be on Solana. If the address is not a valid Solana account (e.g. looks like an Ethereum 0x... address or wrong length) or if a chain ID is present and not Solana, the token is rejected.
- **Age Checks:**
    - *Minimum Age:* If the token's age (in minutes) is below MIN_AGE_MIN (a short grace period, e.g. a few seconds to minutes), the token is considered "too new." In this case the filter returns **None** (defer), and the token is requeued. This delay allows initial liquidity to be added and data (price, volume) to stabilize before risking a trade.
    - *Maximum Age:* If the token is older than MAX_AGE_DAYS (default ~1–2 days), it is rejected outright (these are no longer sniper opportunities).
- **Liquidity and Volume:** After at least a few seconds or minutes have passed, the token must meet minimum liquidity and volume thresholds. The primary pool's USD liquidity must be ≥ MIN_LIQUIDITY_USD (e.g. ~$3k–$3.5k), and 24h volume ≥ MIN_VOL_USD_24H (e.g. ~$7.5k). If either is below threshold, the token is rejected as too illiquid/inactive. Conversely, if the 24h volume exceeds MAX_24H_VOLUME (indicating a big pump), it is rejected as already mature. **Pump.fun Adjustment:** For tokens from Pump.fun (marked is_pf), the bot relaxes these: it accepts liquidity down to 60% of the normal minimum (no lower than a fixed floor, e.g. $1,000), and multiplies the max market cap by 1.5×. This catches very fresh launches that naturally start with near-zero liquidity.
- **Market Capitalization:** If data is available (or can be estimated via price and supply), the token's market cap must fall within a configured range (MIN_MARKET_CAP_USD to MAX_MARKET_CAP_USD). The idea is to target true microcaps: e.g. reject tokens worth only a few hundred dollars (dangerous) or tokens in the millions (already large). Exact defaults vary (e.g. min ~$3k, max ~$400k–$8M in examples).
- **Holders Count:** If the token's holder count is known, it must be ≥ MIN_HOLDERS. By default this may be 0 or a small number, but you can require at least N holders

(to ensure not all tokens are held by the deployer). Special case: if holders=0 **and** there have been no trades in the last few minutes, and the token is still very fresh (e.g. age < 2× MIN_AGE_MIN), the filter returns **None** (delay) rather than False. This allows a newly created pool a chance to get its first swap. Only if, after the early period, holders remain below the minimum does the token get rejected.

- **Early Dump (Anti-Rug) Check:** In the first few minutes of trading (window EARLY_WINDOW_S, e.g. 5–10 minutes), the bot monitors transaction direction. If more than 70% of transactions are sells **and** the price is essentially flat (±5%) during that window, this suggests an insider dump (insiders selling into buyers). If detected, the token is rejected immediately. This protects against obvious rug pulls in progress.
- **Banned Creators:** If the token's mint authority or known creator address matches any in the BANNED_CREATORS list, the token is immediately rejected. This is a user-maintained blacklist of known scams.

If any filter rule returns **False**, the token is dropped from the pipeline. If a filter returns **None**, the token is placed back on the queue to be retried later. A requeue/backoff policy (analytics/requeue_policy.py) governs how many retries occur and how long to wait. For example, the bot may retry "incomplete" tokens (missing liquidity or too-young) a few times with increasing delays (60s, 3m, 7m, etc.) before giving up. Logs annotate requeues with arrows (↩) and reasons like "too_young" or "no_liq". This mechanism ensures tokens that need a moment to fill their pools are not prematurely discarded.

Soft Scoring System (Risk & Quality Signals)

Tokens passing all hard filters advance to the **Soft Scoring** stage. Here the bot gathers additional signals and computes a composite score (typically 0–100) to quantify risk vs. quality. If the total soft score is below MIN_SCORE_TOTAL, the token is not bought. Major soft signals include:

- **Rug Risk Score (RugCheck):** If a RUGCHECK_API_KEY is provided, the bot queries the RugCheck API for the token. RugCheck analyzes the token contract, liquidity lock status, deployer history, etc., and returns a safety score (0–100). The bot adds a bonus (e.g. +15 points) if the rug score is high (typically ≥70). A low or missing score simply means no bonus (the token isn't penalized beyond losing those points).
- **Holder Distribution ("Dev Cluster" Check):** Using the Helius API, the bot examines the top 10 token holders. If they collectively hold more than 20% of the supply, this flags a concentrated distribution (insider-held). If there is *no* such concentration (supply is more distributed), the token earns a +15 point bonus. If a cluster is detected, the bonus is omitted. This rewards tokens with healthier holder spread.

- **Social Presence:** The bot checks if the token has any social links or website listed (from DexScreener profile or metadata). If at least one public link is found (Twitter, Telegram, Discord, etc.), the token gets +10 points. The rationale is that projects with no social presence at all are more suspect than those with at least some trace of community.

- **Insider "Dev Dump" Alert:** If an analytics/insider.py check finds that the deployer (or a single address) is dumping tokens early, it sets a flag. If *no* such insider dump pattern is detected, the bot grants +10 points (i.e. tokens **without** early insider sells get the bonus). Tokens that show suspicious insider selling simply don't earn those points, making it harder to reach the cutoff.

- **Metrics Exceeding Baselines:** Beyond the binary signals above, the bot rewards tokens that significantly outperform the minimum requirements:
  - If **liquidity ≥ 2× MIN_LIQUIDITY_USD**, add +15 points.
  - If **24h volume ≥ 3× MIN_VOL_USD_24H**, add +20 points.
  - If **holders ≥ 2× MIN_HOLDERS**, add +10 points.
    These bonuses favor tokens that are not just barely meeting thresholds but notably exceeding them.

All earned points are summed into total_score. If total_score >= MIN_SCORE_TOTAL, the token qualifies for purchase (subject to the ML gate). If below, it is dropped. By default, MIN_SCORE_TOTAL may be low (even 0) to let data collection for ML be broad; in live trading you'd raise it (e.g. 50+) to be selective.

Optional Machine Learning Gating

If enabled, the bot applies an ML prediction step after scoring. A pretrained **binary classifier** (LightGBM by default) evaluates the token's chance of yielding a profitable trade (within a short horizon, ~30 minutes). Details:

- **Model & Features:** The model file ml/model.pkl is loaded lazily on the first call to analytics/ai_predict.should_buy(). Accompanying it is ml/model.meta.json listing feature names. The feature vector includes all metrics gathered so far (liquidity, volume, age, holders, rug score, cluster flag, social, recent price changes, etc.). Missing features are filled with 0. If MODEL_PATH is unset or the file missing, the bot treats all predictions as 0 (equivalent to skipping ML gating).

- **Threshold Tuning:** The model outputs a probability for "will this be a winning trade?". The environment variable AI_THRESHOLD sets the cutoff. Only if probability ≥ AI_THRESHOLD does the bot proceed to buy. Optionally, the bot can auto-adjust this threshold: if AI_THRESHOLD_FILE contains a "recommended_threshold", the bot will update AI_THRESHOLD at startup (if the change is significant) to calibrate against recent performance.

- **Retraining:** MemeBot 3 includes an asynchronous retraining loop (ml/retrain.py), running periodically (default Sunday 04:00 UTC). It collects recent labeled data from the feature store (last TRAINING_WINDOW_DAYS, default ~30 days) and trains a new model. If the new model shows better metrics (e.g. higher AUC) on a holdout set, it replaces ml/model.pkl and hot-swaps it in memory. A log message "Retrain complete; model reloaded" confirms the update.

- **Feature Logging for ML:** As the bot runs, every token (bought or not) is logged for ML:

- **Filtered Tokens:** If a candidate fails a hard filter or the ML gate (i.e. it is *not bought*), its feature vector is immediately appended to a Parquet file with label 0 (non-profitable). This includes tokens dropped for any reason ("immediate 0" examples).
- **Closed Positions:** When a position is closed (sold or timed-out), the bot appends the feature vector that was recorded at buy time with a label of 1 (if it achieved the configured win profit, e.g. ≥ WIN_PCT fraction of TAKE_PROFIT_PCT) or 0 otherwise. "Shadow positions" (i.e. forced closes after holding max time) are also labeled, generally as losses (0).
- **Feature Store:** These labeled vectors accumulate in monthly Parquet files under data/features/features_YYYYMM.parquet. They can be used offline for analysis or by the bot's retrainer.

Developers can extend the ML pipeline by adding new features (features/builder.py) or replacing the model (any sklearn-compatible classifier that implements predict_proba). Advanced users can tweak retraining frequency or disable it by adjusting RETRAIN_DAY or running the bot without the --retrain flag.

Trade Execution Logic

Buying (Real Mode)

When a token passes all filters and the optional ML gate, the bot attempts to buy it:

1. **Funds & Position Check:** Before buying, the bot checks the SOL balance (get_balance_lamports) to ensure at least TRADE_AMOUNT_SOL + GAS_RESERVE_SOL is available. It also ensures the number of current open positions < MAX_ACTIVE_POSITIONS. If either fails, the buy is aborted.

2. **Route & Transaction (GMGN/Jupiter):** The bot calls trader/gmgn.buy(token_addr, amount_sol). Under the hood, this asks the GMGN (Jupiter) API for the best swap route given the token and SOL amount. The API returns an unsigned Solana transaction. The bot locally signs this transaction with the private key (using sol_signer.py which wraps the Solana SDK) and submits it to the RPC. If successful, the API returns details including:

- qty_lamports: number of token smallest units received,
- buy_price_usd: an estimated USD price per token,
- route: raw route JSON (for debugging),
- price_source: which source provided the price (e.g. "jupiter" or "dexscreener").

3. **Impact Safety Checks:** Before finalizing the buy, the bot assesses price impact to avoid bad trades:

   - If Jupiter routing is available (_JUP_ROUTER_AVAILABLE), the bot can fetch a quote with Jupiter and calculate price_impact_bps. If the impact exceeds IMPACT_MAX_PCT (e.g. 8%), the bot aborts the buy (logging "BUY blocked: route impact > threshold").
   - If Jupiter is not used (or no route), a heuristic uses pool liquidity: estimate impact ≈ (order_usd / liquidity_usd) * IMPACT_EST_K (with factor IMPACT_EST_K, default 2.0). If this estimated impact > IMPACT_MAX_PCT, the bot skips the buy. If pool liquidity is unknown, it compares the price from Dex vs Jupiter: if their difference > PRICE_DIVERGENCE_MAX_PCT (e.g. 15%), it also skips as a safety check.

4. **Post-Buy Recording:** Upon a successful swap, the bot:

   - Deducts the spent SOL from an in-memory balance tracker.
   - Creates a new **Position** record in the SQLite DB. This record stores: token address (and mint ID), symbol, quantity bought (qty in lamports), buy_price_usd, price_source_at_buy (e.g. "Jupiter" or "Dex"), liquidity, market cap, volume at buy time, and opened_at timestamp.
   - Increments a "requeue_success" counter (if the token had been retried).
   - Logs the buy one-line summary (tx signature, price, etc.).

Buying (Dry-Run Mode)

If DRY_RUN=1, the bot routes buys to trader/papertrading.py. This simulates the swap without touching the blockchain. It attempts to fetch a price via Jupiter's public price API. If none is available, it computes an implicit price using the provided SOL amount and expected quantity (factoring SOL/USD). If even that fails, it uses any Dex price hint. The result is a dummy buy_resp with fields qty_lamports, buy_price_usd, and price_source. The paper trader enforces the same Jupiter requirement (REQUIRE_JUPITER_FOR_BUY) and impact checks: it will skip the simulated buy if no Jupiter route exists when required, or if a price impact estimate is too high. This "belt and suspenders" ensures the bot won't simulate an unrealistic or dangerous buy.

Position Monitoring & Exit Strategies

Once a position is open, the bot continuously monitors it in each loop iteration (via _check_positions()):

1. **Price Fetching:** The bot gathers current prices for all open positions. It first tries a batched request to Jupiter's price API for efficiency. Then for each position, it decides whether to prefer a DEX quote or Jupiter: by default it uses the same source as at buy time, unless FORCE_JUP_IN_MONITOR is set (which forces Jupiter).
   - *DEX-first path:* It tries use_gt=True, price_only=True via the GeckoTerminal/DexScreener service to get price (and liquidity). If successful, that price is used with source "dex_full". If that fails, it falls back to the Jupiter batch price, then to an individual Jupiter query.
   - *Jupiter-first path:* It uses the batch price or a direct Jupiter query. If initial attempts yield no price and the position is in a **critical state** (very new or already profitable), it can perform a **critical price fetch** (setting critical=True) which forces Jupiter to give a fresh route quote (bypassing caches). This is rate-limited (controlled by CRIT_PRICE_MAX_PER_CYCLE) to avoid spam.
     The result of this stage is a price_now (USD) and price_src tag (e.g. "jup_batch", "dex_full", "jup_critical"). The bot may also obtain liq_now from Gecko if available.

2. **Exit Conditions:** With current price (or None), the bot checks in order:

3. **Liquidity Rug ("LIQUIDITY_CRUSH"):** If the pool's liquidity has suddenly dropped to ≤ KILL_LIQ_FRACTION of the buy-time liquidity (e.g. a 70% crash), the bot immediately sells to preserve value. This bypasses other checks. It calls seller.sell() for the full remaining quantity and marks the exit reason "LIQUIDITY_CRUSH".
4. **Timeout ("TIMEOUT"):** If no price could be obtained *and* the position has been open > MAX_HOLDING_H hours, the bot will close the position due to timeout. (In practice, lack of price is rare.)
5. **Profit/Loss Checks:** If a price is available, the bot computes the current PnL%. It updates the position's highest PnL% (for trailing logic). Then it evaluates normal exit rules (via _should_exit(pos, price_now, liq_now)):
   - *Early Drop:* If within EARLY_WINDOW_S seconds of buying and price has already dropped ≥ EARLY_DROP_PCT, it triggers an immediate sell ("EARLY_DROP").
   - *Stop-Loss:* If price has fallen ≥ STOP_LOSS_PCT below buy price (after the early window), exit with reason "STOP_LOSS".
   - *Take-Profit & Trailing:* If price is above buy price, the bot may take profit: upon reaching TAKE_PROFIT_PCT, it executes a **partial take-profit** (selling a fraction WIN_PCT of the position, e.g. 33%). The remaining position then follows a trailing-stop strategy: as long as price makes new highs, it stays open. Once price falls by TRAILING_PCT from its peak, the rest is sold ("TRAILING_STOP"). If price never falls that far and keeps rising, the bot allows holding up to an extra MAX_HARD_HOLD_H beyond normal timeout.

  - *No-Expansion:* If enabled (NO_EXPANSION_MAX_PCT > 0), and one hour has passed without PnL% exceeding that threshold, the bot exits ("NO_EXPANSION") to free capital from stagnating trades.

If any of these conditions triggers a sell, the bot proceeds to execute the sale.

1. **Sell Execution:** When a sell is decided, the bot calls trader/seller.sell(token_addr, qty):
    a. It attempts to use Jupiter first (via a sell quote/route). If that fails or Jupiter isn't available and the token's liquidity ≥ MIN_LIQUIDITY_USD, it falls back to gmgn.sell (a GMGN API call similar to buy).
    b. The sell returns details including a transaction signature, a success flag, and price_used_usd.
    c. The bot then determines the final close price: it prefers the last fetched price as a hint, then tries fresh Jupiter or critical fetches, then DEX if needed. If all else fails, it uses the original buy price to calculate PnL (as a last resort).
    d. It records the realized PnL%, sets closed_at timestamp, and labels the exit reason in the Position record (e.g. "TAKE_PROFIT", "STOP_LOSS", etc.).
    e. In real mode, the SOL received from the sale is added back to the bot's internal SOL balance. The position is marked closed in the DB, and a log is emitted (e.g. "SELL executed sig=... price=X.YZ src=Jupiter").

After processing all positions, the bot logs an aggregate summary: how many positions had prices retrieved from each source (batch, single, critical, dex) vs. none, and counts of sells executed.

Feature Storage, Retraining, and Long-Term Data

- **Parquet Feature Store:** Every token evaluated (whether bought or not) has its features appended to monthly Parquet files in data/features/features_YYYYMM.parquet. These rows include all the metrics at buy-time (liquidity, volume, holders, rug score, etc.) plus the eventual outcome label (profit or not). This append-only store serves as the dataset for ML training and analysis. Because it is partitioned by month, it can grow large over time, but the user can archive old files or prune as needed.

- **SQLite Database:** The SQLite file (default data/memebotdatabase.db) persistently tracks all open and closed positions. It ensures that if the bot restarts, it "remembers" what it owns. Safe shutdown/resume is supported: open positions continue to be monitored after restart. The DB also avoids duplicate buys (as mentioned) and can optionally store token metadata in a **Token** table for reference.

- **Retraining & Calibration:** The retrain process uses the Parquet feature data from the last TRAINING_WINDOW_DAYS to build a training set. It holds out recent data for validation. If the newly trained model is better (e.g. higher ROC AUC), it replaces the old model file and updates ml/model.meta.json with feature names and the new recommended threshold. An optional Jupyter notebook (notebooks/calibration.ipynb) is provided for manually exploring threshold vs. precision/recall trade-offs.

Note: Ensure that the data/ directory (and log directories) are persistent between runs (especially if running in a container or VM) so that learning data is not lost. Also periodically back up the SQLite DB or clear it if needed. The SQLite DB and Parquet files form the bot's "memory" for learning and risk tracking.

Utility Modules:

- **Logging (**utils/logger.py**):** The bot uses a JSON logging setup with a custom DedupFilter. This filter suppresses repeated identical log messages for 30s to avoid spam. For example, if many tokens are "too young" at once, you'll only see the "⏳too_young" message once per 30s, then again after the interval. The logger also stamps each message with token symbols (or first-4-chars of address) so one can **grep** logs for a given token's journey. Any sanitization helpers (e.g. sanitize_token_data()) used to clean token info before filtering (the bot warns if critical fields are null).

  Every minute the bot logs a summary of token counts: how many tokens were discovered, how many were deferred (incomplete), filtered out, passed ML, bought, and sold. This helps gauge filter sensitivity at a glance. For example:

  "Stats funnel: discovered=X, incomplete=Y, filtered=Z, ai_pass=U, bought=V"

  Such lines allow a developer to see if, say, most tokens are being filtered early (indicating strict thresholds) or if many reach the buy stage.

  Since every log line includes the token's symbol (or address prefix) and step (e.g. "⛳ New", "⏳ requeue", "✗reject", "[buyer]", "[seller]"), one can grep the log file for that token to reconstruct its path. The structured JSON logs (when file logging is enabled) include these fields explicitly. Highlight that enabling DEBUG level will show *every* filter decision, which combined with the symbols provides a full audit trail.

- **Queue Implementation:** Note that the queue is implemented via utils.lista_pares, a simple in-memory FIFO that tracks each token's address, metadata, and retry count. It enforces the caps (MAX_QUEUE_SIZE, MAX_RETRIES) above. (Code example: queue = ListaPares(max_size=CFG.MAX_QUEUE_SIZE) – where exceeding max_size drops new entries.)

- **Requeue Policy:** The module analytics/requeue_policy.py encapsulates the backoff timing. It reads the token's retry count and schedules the next attempt (e.g. exponential or linear backoff). The documentation can note that by default it waits ~1m/3m/7m before giving up.
- **Sanitization and Error Handling:** If any external call (RPC, API) fails or returns malformed data, helper functions supply defaults (e.g. empty dict, zeros). This prevents exceptions from propagating. For example, fetcher/rugcheck.py logs a warning and returns rug_score=0 on any failure. Social and Helius checks do similar safe-fallbacks (treated as neutral signals).

Summary of Key Files and Workflow

- run_bot.py: main async loop. Discovers tokens, enqueues them, processes filters/scoring, executes trades, monitors positions.
- analytics/filters.py: hard filters and scoring logic.
- analytics/ai_predict.py: loads ML model for should_buy.
- trader/buyer.py / trader/seller.py: execute real trades via GMGN/Jupiter.
- trader/papertrading.py: simulate trades in dry-run.
- fetcher/: modules for external data (Pump.fun, DexScreener, RugCheck, Helius, social checks).
- db/models.py: SQLAlchemy models for persistent tables.
- Parquet files in data/features/: cumulative feature logs for ML.
- .env.sample: annotated template listing all configurable variables.

By combining rule-based filtering with optional ML, and by logging all activity for offline analysis, MemeBot 3 provides a flexible, automated sniper strategy on Solana. All technical behavior – from discovery to exit – is driven by the above configuration options and modules, with no manual intervention needed once set up.

Document Metadata & Signature Page

Title: MemeBot 3 — Sniper Bot Technical Reference Manual
Version: v3.0
Repository: https://github.com/mudanzasalegre/memebot3
Author / Maintainer: José Luis Alegre Llopis (github: @mudanzasalegre)
Contact: mudanzasalegre@hotmail.com
Date: September 28, 2025

Disclaimer
This manual documents the internal operation of MemeBot 3 as of the date above.
Use responsibly. The author is not responsible for financial losses caused by following this guide.
Always test in *DRY_RUN* mode and never store private keys in shared or public locations.

Security & Releases

- Security contact: mudanzasalegre@hotmail.com (use subject: "SECURITY").
- Releases: publish .tar.gz, .sha256 and .asc signature files on GitHub Releases.

Signed by: José Luis Alegre Llopis
Sagunto (Valencia), Spain
Date: September 28, 2025