Linear Trend Spotter — Technical Specification

- **Project:** Linear Trend Spotter (`linear-trend-spotter`)
- **Repository:** github.com/edwardlthompson/linear-trend-spotter
- **Production Host:** PythonAnywhere (paid tier, always-on)
- **Version:** 1.0.0
- **Contributors:** Edward Thompson (project owner), h8rt3rmin8r (major contributor)
- **Date:** 2026-02-28
- **Status:** DRAFT
- **Audience:** AI-first, Human-second

Table of Contents

1. Document Information

1.1. Purpose and Audience

This document is the authoritative technical specification for Linear Trend Spotter, an automated cryptocurrency trend detection system that continuously scans exchange-listed coins, applies a multi-stage filtering pipeline, and delivers qualified alerts with chart images to a Telegram group.

This specification serves as the single source of truth for the system's behavioral contract, architecture, data flow, and operational requirements. It defines the system as it **should exist** in its ideal state — not as a snapshot of the current implementation. Differences between this specification and the live codebase represent work to be done.

The specification is written for an **AI-first, Human-second** audience. Its primary consumers are AI implementation agents operating within isolated context windows during sprint-based development. Every section provides sufficient detail for an AI agent to produce correct, complete code without requiring interactive clarification. Human developers and maintainers are the secondary audience.

1.2. Scope

In Scope

- The complete scan pipeline: data acquisition, filtering, scoring, entry/exit detection, and notification delivery.
- All external API integrations: CoinMarketCap, CoinGecko, Chart-IMG, and Telegram.
- The database schema for scan history, active coin tracking, exchange listings, symbol mappings, and caching.
- The configuration system covering secrets, tunable parameters, and operational settings.
- The scheduling, process management, and watchdog infrastructure.
- The Telegram bot command interface.
- Logging, metrics, and error handling.

Out of Scope

- Trading logic, portfolio management, or position sizing. This system identifies trends — it does not execute trades.
- Mobile applications or web dashboards. The sole delivery interface is Telegram.
- Historical backtesting or performance analytics beyond basic scan metrics.
- Multi-user authentication or access control. The system serves a single Telegram group.

1.3. Document Maintenance

This specification is maintained as a living document alongside the codebase. When the specification and the implementation disagree, the specification is presumed correct unless a deliberate amendment has been made.

The document header's **Date** field reflects the date of the most recent substantive revision. The **Status** field uses one of the following values:

Status	Meaning
`DRAFT`	Under active development. Sections may be incomplete or subject to change.
`REVIEW`	Believed complete, undergoing review.
`APPROVED`	Reviewed and accepted as the implementation target.
`AMENDED`	Modified after initial approval to reflect post-release changes.

1.4. Conventions Used in This Document

This specification uses RFC 2119 keywords to indicate requirement levels:

Keyword	Meaning
MUST / **MUST NOT**	Absolute requirement or prohibition.
SHOULD / **SHOULD NOT**	Strong recommendation; deviation requires justification.
MAY	Truly optional.

`Monospace text` denotes code identifiers, file paths, configuration keys, and literal values. **Bold text** denotes emphasis or key terms. `§N.N` denotes a cross-reference to another section.

Terminology

Term	Definition
Scan	A single complete execution of the filtering pipeline, from CMC data fetch through notification delivery.
Qualified coin	A coin that has passed all filtering stages and is eligible for entry notification.
Entry	A coin appearing in the qualified set for the first time (or re-entering after an exit).
Exit	A coin that was previously qualified but no longer passes all filters.
Uniformity Score	A value in `[0, 100]` measuring how evenly a coin's gains are distributed across the analysis window. See §6.
Active coin	A coin currently in the qualified set, tracked in the `active_coins` table.

1.5. Reference Documents

Document	Description
`linear-trend-spotter-spec.txt`	Original technical spec. Documents the system as observed. This document supersedes it as the authoritative source.
`3rd-party-map.json`	Maps API key names to the registration URLs for each external service.
`config_json.example`	Example configuration file showing all supported keys and default values.

Document	Description
`.github/copilot-instructions.md`	AI coding agent directives for this project.

2. Project Overview

2.1. Project Identity

Field	Value
Project Name	Linear Trend Spotter
Project Slug	`linear-trend-spotter`
Language	Python 3.10
Repository	github.com/edwardlthompson/linear-trend-spotter
Production host	PythonAnywhere (paid tier, always-on)
Database	SQLite 3 (local files)
Delivery	Telegram Bot API
Telegram Group	Join Link

2.2. What This Tool Does

Linear Trend Spotter is a 24/7 automated scanner that identifies cryptocurrency coins exhibiting **strong, sustained, and uniform** upward price trends across major exchanges. It solves the problem of manual screening — sifting through thousands of coins to find genuine momentum that isn't just a short-lived pump.

The system scans every coin listed on Coinbase, Kraken, and MEXC once per hour through a 10-stage filtering pipeline that progressively narrows ~2,500 coins down to ~15–25 qualified results. The core differentiator is the **Uniformity Score** algorithm (see §6), which measures how evenly a coin's gains are distributed across a 30-day window — filtering out "hockey stick" charts where most gains cluster at one end.

When a coin first qualifies, the system sends a Telegram notification with a TradingView-style chart image, gain percentages, the uniformity score, and per-exchange volume data. When a coin falls out of the qualified set, a single exit notification is sent. No repeated alerts, no spam.

2.3. Design Goals and Non-Goals

Design Goals

G1 — Catch real trends, not pumps. The multi-stage pipeline and uniformity analysis MUST distinguish smooth, sustained uptrends from sudden spikes, dead-cat bounces, and low-volume noise.

G2 — One alert per event. Each coin produces exactly one entry notification when it first qualifies and exactly one exit notification when it drops out. No duplicates, no re-alerts while a coin remains active.

G3 — Minimal API cost. The pipeline is ordered so that the cheapest, most selective filters run first. CoinMarketCap provides bulk gain data in a single API call. CoinGecko's per-coin endpoints are reached only by the ~100 coins that survive the first three filter stages. Aggressive caching reduces redundant calls.

G4 — Resilient unattended operation. The system MUST run continuously without human intervention. Rate limits, API failures, network timeouts, and process crashes are handled automatically through backoff, retry, caching, and watchdog restart.

G5 — Clean separation of concerns. API clients, filter logic, database operations, notification formatting, and orchestration are isolated into distinct modules. The main scanner orchestrator (`main.py`) delegates all domain logic to purpose-built modules.

G6 — AI-agent implementability. This specification provides sufficient detail for an AI agent to implement or modify any component in a single session without interactive clarification.

Non-Goals

NG1 — Trading execution. The system identifies trends. It does not place orders, manage positions, or calculate risk.

NG2 — Web UI or dashboard. Telegram is the sole delivery interface.

NG3 — Multi-tenancy. The system serves one Telegram group with one configuration.

NG4 — Real-time streaming. Scans run on an hourly schedule, not in response to live price feeds.

2.4. Platform and Runtime Requirements

Requirement	Value
Python version	3.10
Operating system	Linux (PythonAnywhere runs Ubuntu)
Network	Outbound HTTPS to all external APIs
Disk	<100 MB total for databases, logs, and lock files
RAM	<256 MB per scan process

3. Repository Structure

3.1. Top-Level Layout

```
linear-trend-spotter/
├── .github/
│   └── copilot-instructions.md
├── .archive/
│   └── (archived files — see §3.3 for naming convention)
├── api/
│   ├── __init__.py
│   ├── coinmarketcap.py
│   ├── coingecko.py
│   ├── coingecko_mapper.py
│   ├── chart_img.py
│   └── tradingview_mapper.py
├── config/
│   ├── __init__.py
│   ├── settings.py
│   └── constants.py
├── database/
│   ├── __init__.py
│   ├── models.py
│   └── cache.py
├── exchange_data/
│   ├── __init__.py
│   ├── exchange_db.py
│   └── exchange_fetcher.py
├── notifications/
│   ├── __init__.py
│   ├── telegram.py
│   └── formatter.py
├── processors/
│   ├── __init__.py
│   ├── gain_filter.py
│   └── uniformity_filter.py
├── utils/
│   ├── __init__.py
│   ├── logger.py
│   ├── metrics.py
│   └── rate_limiter.py
├── main.py
├── scheduler.py
├── telegram_bot.py
├── manage_bot.py
├── bot_watchdog.py
├── update_exchanges.py
├── update_mappings.py
├── .env
├── .env.example
├── .gitignore
├── config.json
├── config_json.example
├── 3rd-party-map.json
├── requirements.txt
├── linear-trend-spotter-spec.md
└── README.md
```

Path	Purpose
`.github/copilot-instructions.md`	AI coding agent directives.

Path	Purpose
`.archive/`	Archived files from prior brainstorming, iteration, and superseded code. Not part of the active codebase. MUST NOT be imported by any active module. See §3.3.
`api/`	External API client modules. Each file encapsulates one external service. See §7.
`config/`	Configuration management. `settings.py` is the centralized settings loader; `constants.py` contains static lookup tables (stablecoin lists, exchange emoji maps, etc.). See §9.
`database/`	Database models and caching layer. `models.py` defines `HistoryDatabase` and `ActiveCoinsDatabase`; `cache.py` provides `PriceCache` for CoinGecko price/uniformity data. See §8.
`exchange_data/`	Exchange listing database and fetcher. `exchange_db.py` manages the SQLite listing store; `exchange_fetcher.py` pulls current listings from Coinbase, Kraken, and MEXC public APIs.
`notifications/`	Telegram notification client and message formatting.
`processors/`	Pure filtering and scoring logic. No API calls, no database access — functions take data in and return results.
`utils/`	Cross-cutting utilities: logging, metrics collection, and rate limiting with exponential backoff.
`main.py`	Scanner orchestrator. Contains `run_scanner()` which executes the full 10-stage pipeline. The only file that imports from every package. See §4.
`scheduler.py`	Cron entry point. Acquires a file lock, runs `main.run_scanner()`, and records stats. See §11.
`telegram_bot.py`	Long-running Telegram bot process. Polls for commands and dispatches responses. See §10.3.
`manage_bot.py`	Bot process lifecycle manager (start/stop/restart/status).
`bot_watchdog.py`	Cron-driven watchdog that restarts the bot if it crashes.
`update_exchanges.py`	Standalone script that refreshes exchange listing data. Run weekly via cron.
`update_mappings.py`	Standalone script that refreshes CoinGecko symbol→ID mappings. Run monthly via cron.
`.env`	Secrets (API keys, Telegram tokens). Gitignored.
`.env.example`	Template showing required environment variables. Committed to repo.
`config.json`	Non-secret tunable parameters. Gitignored (contains per-environment values).
`config_json.example`	Template showing all config keys with defaults. Committed to repo.
`requirements.txt`	Pinned Python dependencies.

Path	Purpose
`linear-trend-spotter-spec.md`	This specification.
`README.md`	Public project overview and Telegram invite link.

3.2. Source Package Layout

The project uses a **flat script layout** (not a `src/` package layout) because it runs as a set of cron-invoked scripts on PythonAnywhere, not as an installable package. The top-level scripts (`main.py`, `scheduler.py`, etc.) add their own directory to `sys.path` at startup to enable imports from the sub-packages.

Each sub-package is responsible for a single domain:

Package	Domain	Key Exports
`api/`	External service clients	`CoinMarketCapClient`, `CoinGeckoClient`, `CoinGeckoMapper`, `ChartIMGClient`, `TradingViewMapper`
`config/`	Settings and constants	`settings` (singleton), `STABLECOINS`, `EXCHANGE_EMOJIS`
`database/`	Persistence and caching	`HistoryDatabase`, `ActiveCoinsDatabase`, `PriceCache`
`exchange_data/`	Exchange listing management	`ExchangeDatabase`, `ExchangeFetcher`
`notifications/`	Telegram delivery	`TelegramClient`, `MessageFormatter`
`processors/`	Pure filter/score logic	`GainFilter`, `UniformityFilter`
`utils/`	Cross-cutting infrastructure	`setup_logger`, `app_logger`, `RateLimiter`, `CircuitBreaker`, `MetricsCollector`

Architectural rule: Sub-packages MUST NOT import from each other laterally. All inter-package coordination flows through `main.py` (the orchestrator) or through the top-level scripts. For example, `processors/` MUST NOT import from `api/`; it receives data as function arguments. `database/` MUST NOT import from `notifications/`. This rule ensures that each package can be understood, tested, and modified in isolation.

3.3. Archived Artifacts

The `.archive/` directory holds files that were part of prior brainstorming, experimentation, superseded approaches, or earlier iterations of active code. These files are retained to provide a historical timeline of the project's evolution. Files in `.archive/` MUST NOT be imported by any active module.

The following categories of files belong in `.archive/`:

- Standalone scripts that have been absorbed into the modular package structure (e.g., a standalone `build_mapping_db.py` superseded by `api/coingecko_mapper.py` and `update_mappings.py`).
- Experimental API clients for services that were evaluated but not adopted.

- Early-iteration pipeline logic that has been rewritten.
- Brainstorming notes, planning documents, and prior drafts of documentation.
- Duplicate copies of the spec or documentation from prior locations (e.g., a `docs/` subdirectory that contained a copy of the spec).

Archive Naming Convention

All files placed in `.archive/` MUST be renamed to follow this scheme:

```
<DateStamp>-<DailyIncrement>-<FileName>.<Extension>
```

Component	Format	Description
`DateStamp`	`YYYYMMDD`	Calendar date the file was archived.
`DailyIncrement`	`NNN`	Zero-padded, three-digit number (001, 002, ...). Resets to 001 at the start of each new calendar day.
`FileName`	Free-form	Original file name (or a descriptive name).
`Extension`	As-is	Original file extension preserved.

Daily increment rules:

- The increment resets to `001` on each new calendar day.
- Files that belong to the same logical group (i.e., files that should be read or processed together) share the same increment value. For example, a sprint plan and its companion implementation prompt template archived on the same day would both use `001`. A technical spec archived in both Markdown and PDF format would also share one increment value.
- Each subsequent file or file group archived on the same day increments by one: `001`, `002`, `003`, etc.

Examples:

```
# Single file archived on 2026-02-28 (first item of the day)
linear-trend-spotter-spec.pdf  →  20260228-001-linear-trend-spotter-spec.pdf

# A file group archived together (sprint plan + prompt template, second batch of
the day)
sprint-04-plan.md              →  20260228-002-sprint-04-plan.md
sprint-04-prompt.md            →  20260228-002-sprint-04-prompt.md

# A file in two formats archived together (third batch of the day)
api-audit.md                   →  20260228-003-api-audit.md
api-audit.pdf                  →  20260228-003-api-audit.pdf

# First file archived on the next day resets the increment
old-scanner.py                 →  20260301-001-old-scanner.py
```

3.4. Runtime Data Files

The following files are created at runtime and MUST be gitignored:

File	Created By	Purpose
`scanner.db`	`database/models.py`, `database/cache.py`	Primary SQLite database: scan history, active coins, price cache.
`exchanges.db`	`exchange_data/exchange_db.py`	Exchange listing data for Coinbase, Kraken, MEXC.
`mappings.db`	`api/coingecko_mapper.py`	Symbol → CoinGecko ID mapping table.
`tv_mappings.db`	`api/tradingview_mapper.py`	Symbol → TradingView symbol mapping with exchange-specific formatting.
`scan.lock`	`scheduler.py`	File lock preventing concurrent scans.
`scan_stats.json`	`scheduler.py`	Last 100 scan durations and timestamps.
`metrics.json`	`utils/metrics.py`	Per-scan performance counters.
`trend_scanner.log`	`utils/logger.py`	Primary application log (rotated, 10 MB × 5 backups).
`bot_output.log`	`manage_bot.py`	Telegram bot process stdout/stderr.
`*.pid`	`manage_bot.py`	Bot process ID file.

Database consolidation. The system uses four SQLite database files. While a single database would be simpler, the separation is deliberate: `exchanges.db` and `mappings.db` are refreshed on independent schedules (weekly and monthly) via destructive rebuild, while `scanner.db` is append-only during scans. `tv_mappings.db` is a lookup cache that can be regenerated at any time. Separating them prevents a weekly exchange refresh from locking the primary scan database.

---

4. Architecture

4.1. High-Level Processing Pipeline

Every scan follows the same linear 10-stage pipeline. The pipeline is orchestrated by `main.run_scanner()` and executes within a file-locked `scheduler.py` invocation. No stage begins until its predecessor completes.

```
[Cron] → scheduler.py (acquire lock)
           → main.run_scanner()
               → Stage 1:  CMC Bulk Fetch          (1 API call)
               → Stage 2:  Exchange Verification    (local DB)
               → Stage 3:  Volume Filter            (in-memory)
               → Stage 4:  Gain Filter              (in-memory)
               → Stage 5:  CoinGecko ID Resolution  (local DB)
               → Stage 6:  Exchange Volume Fetch     (CoinGecko API, cached)
               → Stage 7:  Uniformity Calculation    (CoinGecko API, cached)
```

```
            → Stage 8:  Uniformity Filter         (in-memory)
            → Stage 9:  Entry/Exit Detection      (local DB)
            → Stage 10: Notification Delivery     (Chart-IMG + Telegram API)
        → release lock
```

The pipeline is designed so that **cost scales with selectivity**: the cheapest stages (local DB lookups, in-memory arithmetic) run first and eliminate ~99% of candidates before the expensive per-coin API calls begin.

4.2. Module Decomposition

```
scheduler.py ──────────────────────────────┐
                                           ▼
                                    main.run_scanner()
                                     │  │  │  │  │
                    ┌────────────────┘  │  │  │  └──────────────────┐
                    ▼                   ▼  │  ▼                     ▼
    api/coinmarketcap.py    api/coingecko.py │  api/chart_img.py   notifications/
                    │        api/coingecko_mapper.py                ├──
telegram.py
                    │                      │                        └──
formatter.py
                    ▼                      ▼
    processors/gain_filter.py    processors/uniformity_filter.py
                                           │
                    ┌──────────────────────┘
                    ▼
    database/models.py    database/cache.py    exchange_data/exchange_db.py
```

Key structural rules:

Rule 1 — `main.py` is the sole orchestrator. It is the only module that imports from every package. It wires API clients to processors, processors to databases, and results to notifications. No other module has this breadth of visibility.

Rule 2 — Processors are pure logic. `processors/gain_filter.py` and `processors/uniformity_filter.py` contain no API calls, no database access, and no side effects. They take data in as arguments and return filtered/scored results. This makes them trivially testable.

Rule 3 — API clients are self-contained. Each file in `api/` encapsulates one external service. It manages its own HTTP session, authentication, and response parsing. It does NOT know about the filtering pipeline or the database.

Rule 4 — No lateral imports between sub-packages. See §3.2.

4.3. Data Flow

Data flows linearly through the pipeline as plain Python dictionaries. Each coin starts as a CMC response dict and is progressively enriched:

```
CMC raw response
  → { symbol, name, slug, volume_24h, gains: {7d, 30d, 60d, 90d} }

After exchange verification:
  → + { listed_on: [exchange, ...] }

After CoinGecko ID resolution:
  → + { cg_id: "bitcoin" }

After exchange volume enrichment:
  → + { exchange_volumes: { coinbase: 1234.56, kraken: 789.01, mexc: 456.78 } }

After uniformity calculation:
  → + { uniformity_score: 72.3, total_gain: 45.6 }
```

There is no formal schema class for these intermediate dictionaries. This is a deliberate trade-off — the system is small enough that dictionary-based data flow is sufficient, and introducing dataclasses would add complexity without proportional benefit at this scale. However, each stage's expected input keys are documented in the function signatures and docstrings.

4.4. Process Model

The system runs as three independent processes managed by PythonAnywhere's scheduler:

Process	Lifecycle	Triggered By
Scanner	Short-lived (10–15 min)	Cron, hourly via `scheduler.py`
Telegram bot	Long-running (daemon)	`manage_bot.py start`, restarted by watchdog
Watchdog	Short-lived (<1 sec)	Cron, every 5 minutes via `bot_watchdog.py`

The scanner and the Telegram bot MUST NOT run in the same process. The scanner is a batch job that runs for 10–15 minutes and exits. The bot is a long-polling daemon that must stay alive between scans to respond to user commands. They share data exclusively through the SQLite databases.

---

5. Filtering Pipeline

5.1. Pipeline Overview

Stage	Operation	Data Source	API Calls	Survivors (typical)
1	CMC Bulk Fetch	CoinMarketCap	1	~2,500 (all fetched)
2	Exchange Verification	Local DB	0	~1,500
3	Volume Filter	In-memory	0	~750
4	Gain Filter	In-memory	0	~100
5	CoinGecko ID Resolution	Local DB	0	~90

Stage	Operation	Data Source	API Calls	Survivors (typical)
6	Exchange Volume Enrichment	CoinGecko API	~90	~90 (enrichment only)
7	Uniformity Calculation	CoinGecko API	~20–40	~90 (scoring only)
8	Uniformity Filter	In-memory	0	~20
9	Entry/Exit Detection	Local DB	0	—
10	Notification Delivery	Chart-IMG + Telegram	~5–10	—

Total API calls per scan: ~150–250, dominated by CoinGecko.

5.2. Stage 1 — CoinMarketCap Bulk Fetch

A single API call to `/v1/cryptocurrency/listings/latest` retrieves up to 5,000 coins with their 7-day, 30-day, 60-day, and 90-day percentage changes, 24-hour volume, market cap rank, name, symbol, and slug. The response is parsed into a hash map keyed by uppercase symbol for O(1) lookup in subsequent stages.

Input: None (initiates the pipeline).
Output: `Dict[str, CoinData]` — symbol → market data.
API calls: 1.

5.3. Stage 2 — Exchange Listing Verification

Each coin from the CMC dataset is checked against the local `exchanges.db` to verify it is actually listed on at least one of the target exchanges (Coinbase, Kraken, MEXC). Coins not found on any target exchange are eliminated.

Input: CMC hash map.
Output: Coins confirmed listed on ≥1 target exchange, with `listed_on` field populated.
API calls: 0 (local DB indexed lookup).

5.4. Stage 3 — Volume Filter

Coins MUST have a 24-hour trading volume ≥ `MIN_VOLUME_M` (default: $1,000,000 USD). This eliminates illiquid coins where price movements may be unreliable.

Input: Exchange-verified coins.
Output: Coins meeting volume threshold.
API calls: 0.
Selectivity: ~50% eliminated.

5.5. Stage 4 — Gain Filter

Coins MUST meet **both** of the following gain thresholds, representing >1% average daily growth over each window:

- 7-day gain > 7%
- 30-day gain > 30%

Stablecoins (USDT, USDC, DAI, BUSD, TUSD, USDP, GUSD, etc.) are excluded regardless of gains. The stablecoin list is maintained in `config/constants.py`.

Input: Volume-qualified coins.
Output: Coins meeting both gain thresholds.
API calls: 0.
Selectivity: ~85% of remaining eliminated.

5.6. Stage 5 — CoinGecko ID Resolution

Each surviving coin's symbol is mapped to its CoinGecko API ID (e.g., `"BTC"` → `"bitcoin"`) using the local `mappings.db`. This mapping is required for CoinGecko API calls in subsequent stages.

Coins without a mapping are logged and skipped for this scan. The mapping database is refreshed monthly by `update_mappings.py`.

Input: Gain-qualified coins.
Output: Coins with `cg_id` field populated.
API calls: 0 (local DB).
Pass rate: ~90%.

5.7. Stage 6 — Exchange Volume Enrichment

For each coin with a CoinGecko ID, the `/coins/{id}/tickers` endpoint is called to retrieve per-exchange trading volume. Volumes for Coinbase, Kraken, and MEXC are extracted and attached to the coin data.

Results are cached for 24 hours. Cached values are used without API calls.

Input: Coins with CoinGecko IDs.
Output: Coins enriched with `exchange_volumes` dict.
API calls: 0 (cached) to ~90 (uncached), rate-limited.

5.8. Stage 7 — Price History Fetch and Uniformity Calculation

For each coin, 30 days of daily closing prices are fetched from CoinGecko's `/coins/{id}/market_chart` endpoint. The price array is passed to `UniformityFilter.calculate()` (see §6) to produce a uniformity score and total gain.

Results are cached for 6 hours. Cached values are used without API calls.

Input: Coins with CoinGecko IDs.
Output: Coins with `uniformity_score` and `total_gain` fields.
API calls: 0 (cached) to ~40 (uncached), rate-limited.

5.9. Stage 8 — Uniformity Filter

Coins MUST meet **both** of the following criteria:

- Uniformity score ≥ `UNIFORMITY_MIN_SCORE` (default: 45)
- Total 30-day gain > 0 (positive return)

Input: Scored coins.
Output: Final qualified set.
API calls: 0.
Selectivity: ~80% of remaining eliminated.

5.10. Stage 9 — Entry/Exit Detection

The current qualified set is compared against the `active_coins` table:

- **Entered** = qualified now AND NOT in `active_coins` → insert into `active_coins`, queue entry notification.
- **Exited** = in `active_coins` AND NOT qualified now → remove from `active_coins`, queue exit notification.
- **Unchanged** = in both sets → update `last_seen_date`, no notification.

This is a simple set-difference operation: `entered = current - active`, `exited = active - current`.

Input: Final qualified set + `active_coins` DB state.
Output: Lists of entered and exited coins.
API calls: 0.

5.11. Stage 10 — Notification Delivery

For each **entered** coin:

1. Resolve the TradingView symbol using `TradingViewMapper` (local DB, exchange-priority: MEXC → Kraken → Coinbase).
2. Request a chart image from Chart-IMG using the TradingView symbol (1 API call per coin, rate-limited to 1/second).
3. Format the notification message using `MessageFormatter.format_entry()`.
4. Send via `TelegramClient.send_photo()` with the chart as an inline image and the formatted message as the caption.

For each **exited** coin:

1. Format the exit message using `MessageFormatter.format_exit()`.
2. Send via `TelegramClient.send_message()`.

All coins in the qualified set (entered + unchanged) are saved to `scan_history`.

API calls: 1 Chart-IMG + 1 Telegram per entered coin; 1 Telegram per exited coin.

---

6. Core Algorithm — Uniformity Score

6.1. Purpose

The Uniformity Score measures how evenly a coin's gains are distributed across a 30-day window. A perfect score (100) means the price rose by exactly the same amount each day — a perfectly straight line. A low score means most gains are concentrated in a short burst (e.g., a "hockey stick" pattern where the price was flat for 25 days and then spiked).

This is the system's core differentiator. Volume and gain filters catch noise; the uniformity filter catches *deceptive patterns* — coins that show impressive 30-day returns but did it all in a 3-day pump.

6.2. Algorithm

Input: An array of 30 daily closing prices, ordered oldest to newest.
Output: A tuple of `(uniformity_score, total_gain_pct)`.

```
1. Normalize to cumulative percentage change from day 0:
   base = prices[0]
   cum_pct[i] = ((prices[i] - base) / base) × 100

2. Calculate the ideal uniform growth line:
   total_gain = cum_pct[29]       # final cumulative gain
   daily_gain = total_gain / 29   # uniform daily increment
   ideal[i] = i × daily_gain

3. Calculate total deviation from the ideal:
   total_deviation = Σ |cum_pct[i] - ideal[i]|  for i = 0..29

4. Calculate maximum possible deviation (worst case = all gain on last day):
   max_deviation = Σ total_gain  for i = 0..28

5. Normalize and transform:
   normalized = total_deviation / max_deviation
   raw_score = 100 × (1 - √(min(normalized, 1)))
   score = clamp(raw_score, 0, 100)
```

The square root transformation (`1 - √x`) creates a non-linear scoring curve that is generous to moderately uneven distributions (a coin with some variance still scores well) but punishes severely uneven ones (a hockey stick scores very low). This matches intuition: traders tolerate some daily variance but want to avoid entering after a spike.

If `total_gain ≤ 0`, the score is 0 and the coin is automatically excluded.

6.3. Score Interpretation

Score Range	Category	Meaning
≥ 90	Perfect	Nearly uniform daily gains. Rare.
75–89	Excellent	Very smooth uptrend with minor variance.
60–74	Good	Clear uptrend with some day-to-day noise.
45–59	Fair	Acceptable but uneven. Gains may cluster. Default threshold (45) catches these.
20–44	Poor	Most gains concentrated in a short window. Excluded by default.

Score Range	Category	Meaning
< 20	Bad	Essentially all gains from a single spike.

7. External APIs

7.1. CoinMarketCap

Property	Value
Base URL	`https://pro-api.coinmarketcap.com/v1`
Auth	`X-CMC_PRO_API_KEY` header
Tier	Pro (paid)
Endpoint used	`/cryptocurrency/listings/latest`
Calls per scan	1
Rate limit	30/min, 2s minimum interval enforced client-side
Data retrieved	Symbol, name, slug, 24h volume, 7d/30d/60d/90d % changes, rank
Request limit	5,000 coins per request (configured)

CoinMarketCap is the pipeline's entry point. A single bulk request provides the gain and volume data needed for Stages 2–4 without any per-coin API calls.

7.2. CoinGecko

Property	Value
Base URL	`https://api.coingecko.com/api/v3`
Auth	None (free tier) or `x-cg-demo-api-key` header
Tier	Free / Demo
Endpoints used	`/coins/{id}/tickers`, `/coins/{id}/market_chart`, `/coins/list`
Calls per scan	~100–200 (tickers) + ~20–40 (market chart)
Rate limit	10–30/min (free tier), 12s base interval enforced client-side
Caching	Tickers: 24h, Market chart: 6h, Coin list: 30 days

CoinGecko provides two categories of data that CMC does not offer in its free/pro tiers:

1. **Per-exchange volume breakdown** (`/tickers`) — essential for showing users where a coin actually trades.
2. **Daily price history** (`/market_chart`) — essential for the uniformity score calculation.

The coin list endpoint (`/coins/list`) is used to build the symbol → CoinGecko ID mapping database. It is called by `update_mappings.py` on a monthly schedule, not during normal scans.

7.3. Chart-IMG

Property	Value
Base URL	`https://api.chart-img.com/v2`
Auth	`x-api-key` header
Tier	Paid
Endpoint used	`/tradingview/advanced-chart`
Calls per scan	~5–10 (only for newly entered coins)
Rate limit	1/second enforced client-side

Chart-IMG generates TradingView-style chart images from TradingView symbol strings. The TradingView symbol is resolved by `TradingViewMapper` with exchange-specific formatting:

- Coinbase: `COINBASE:BTC-USD`
- Kraken: `KRAKEN:BTCUSD`
- MEXC: `MEXC:BTCUSDT`

Exchange priority for chart generation: MEXC → Kraken → Coinbase (MEXC has the broadest listing coverage).

7.4. Telegram Bot API

Property	Value
Base URL	`https://api.telegram.org/bot{token}/`
Auth	Bot token in URL path
Methods used	`sendMessage`, `sendPhoto`, `getUpdates`
Calls per scan	~5–15 (only for entries and exits)

Entry notifications use `sendPhoto` with the chart image as inline media and the formatted message as the HTML caption. Exit notifications use `sendMessage` with HTML formatting.

7.5. Rate Limit Strategy

All API clients use a shared `RateLimiter` utility from `utils/rate_limiter.py` that implements:

1. **Minimum interval enforcement.** A configurable floor between consecutive calls (e.g., 12s for CoinGecko free tier).
2. **Exponential backoff on 429s.** On receiving an HTTP 429 (Too Many Requests), the limiter doubles the wait time on each consecutive 429 (60s → 120s → 240s), capped at 300s. The counter resets on a successful response.
3. **Jitter.** A small random component (0–100ms) is added to prevent synchronized retry storms.

4. **Circuit breaker.** After `CIRCUIT_FAILURE_THRESHOLD` (default: 5) consecutive failures to any single service, the circuit opens and skips further calls to that service for `CIRCUIT_RECOVERY_TIMEOUT` (default: 60) seconds. This prevents a downed API from blocking the entire scan for hundreds of seconds of accumulated backoff.

---

8. Database Schema

8.1. Primary Database — `scanner.db`

`active_coins`

Tracks coins currently in the qualified set. Primary source of truth for entry/exit detection.

```
CREATE TABLE active_coins (
    coin_symbol      TEXT NOT NULL,
    coin_name        TEXT NOT NULL,
    gecko_id         TEXT,
    entered_date     TEXT NOT NULL,       -- ISO 8601, when coin first entered
    last_seen_date   TEXT NOT NULL,       -- ISO 8601, last scan where coin
qualified
    last_scan_date   TEXT NOT NULL,       -- ISO 8601, timestamp of the scan
    gain_7d          REAL,
    gain_30d         REAL,
    uniformity_score REAL,
    coinbase_volume  TEXT,
    kraken_volume    TEXT,
    mexc_volume      TEXT,
    slug             TEXT,                -- CMC slug for URL construction
    cmc_url          TEXT,
    PRIMARY KEY (coin_symbol)
);
```

Design note: The primary key is `coin_symbol` alone (not a compound key). Two coins with the same symbol on different exchanges are treated as one logical asset. The `coin_name` field is informational, not part of identity.

`scan_history`

Append-only log of every qualified coin in every scan. Used for historical analysis.

```
CREATE TABLE scan_history (
    id               INTEGER PRIMARY KEY AUTOINCREMENT,
    scan_date        TEXT NOT NULL,       -- ISO 8601
    coin_name        TEXT,
    coin_symbol      TEXT NOT NULL,
    gain_7d          REAL,
    gain_30d         REAL,
    uniformity_score REAL,
    coinbase_volume TEXT,
    kraken_volume   TEXT,
    mexc_volume     TEXT,
    cmc_url         TEXT
);

CREATE INDEX idx_scan_history_date ON scan_history(scan_date);
CREATE INDEX idx_scan_history_symbol ON scan_history(coin_symbol);
```

`price_cache`

Caches CoinGecko price history and uniformity scores to reduce API calls.

```
CREATE TABLE price_cache (
    coin_id             TEXT PRIMARY KEY,   -- CoinGecko ID
    prices              TEXT,               -- JSON array of 30 daily closing
prices
    uniformity_score    REAL,
    gains_30d           REAL,
    cache_date          TEXT NOT NULL       -- ISO 8601
);
```

Cache TTL: 6 hours. Entries older than `CACHE_PRICE_HOURS` are ignored and re-fetched.

8.2. Exchange Listings Database — `exchanges.db`

`exchange_listings`

```
CREATE TABLE exchange_listings (
    exchange        TEXT NOT NULL,
    symbol          TEXT NOT NULL,       -- Uppercase (e.g., "BTC")
    name            TEXT,
    coingecko_id    TEXT,
    first_seen      TEXT,
    last_seen       TEXT,
    source          TEXT,               -- e.g., "coinbase_api", "kraken_api"
    PRIMARY KEY (exchange, symbol)
);

CREATE INDEX idx_listings_symbol ON exchange_listings(symbol);
```

`exchange_metadata`

```
CREATE TABLE exchange_metadata (
    exchange        TEXT PRIMARY KEY,
    last_updated    TEXT,
    total_pairs     INTEGER,
    source          TEXT
);
```

`listing_cache`

```sql
CREATE TABLE listing_cache (
    symbol          TEXT NOT NULL,
    exchange        TEXT NOT NULL,
    is_listed       INTEGER,            -- 0 or 1
    last_checked    TEXT,
    PRIMARY KEY (symbol, exchange)
);
```

8.3. Mapping Database — `mappings.db`

`symbol_mapping`

```sql
CREATE TABLE symbol_mapping (
    symbol          TEXT NOT NULL,       -- Uppercase (e.g., "BTC")
    name            TEXT,               -- Full name (e.g., "Bitcoin")
    coingecko_id    TEXT NOT NULL,       -- CoinGecko API ID (e.g., "bitcoin")
    confidence      INTEGER,            -- 70-100; higher = more reliable
    source          TEXT,               -- e.g., "cryptocurrencies",
"coingecko_direct"
    last_updated    TEXT,
    PRIMARY KEY (symbol, coingecko_id)
);

CREATE INDEX idx_mapping_symbol ON symbol_mapping(symbol);
```

`mapping_metadata`

```sql
CREATE TABLE mapping_metadata (
    key     TEXT PRIMARY KEY,
    value   TEXT
);
```

Stores `last_updated` and `total_mappings` for freshness checks.

---

9. Configuration

9.1. Configuration Architecture

Configuration is split into two layers:

1. **Secrets** — API keys and tokens. Stored in `.env`, loaded via `python-dotenv`, accessed through `os.getenv()`. NEVER committed to the repository.
2. **Tunable parameters** — Thresholds, intervals, feature flags. Stored in `config.json`, loaded at startup by `config/settings.py`. Can be committed (gitignored by default because values are environment-specific).

The `Settings` class in `config/settings.py` is the single point of access for all configuration. It merges hardcoded defaults with `config.json` values, and exposes secrets via `@property` methods that read from environment variables.

9.2. Secrets Management

Secrets are stored in a `.env` file (gitignored) and loaded by `python-dotenv` at startup.

Required environment variables:

Variable	Source	Purpose
`CMC_API_KEY`	CoinMarketCap Pro	Authenticates CMC API requests.
`TELEGRAM_BOT_TOKEN`	BotFather	Authenticates Telegram Bot API requests.
`TELEGRAM_CHAT_ID`	Telegram group settings	Target group for notifications.
`CHART_IMG_API_KEY`	Chart-IMG	Authenticates chart image generation.

Optional environment variables:

Variable	Default	Purpose
`COINGECKO_API_KEY`	*(none)*	CoinGecko demo API key for higher rate limits. Not required for free tier.

`.env.example` template:

```
CMC_API_KEY=your_key_here
TELEGRAM_BOT_TOKEN=your_token_here
TELEGRAM_CHAT_ID=your_chat_id_here
CHART_IMG_API_KEY=your_key_here
# COINGECKO_API_KEY=optional_demo_key
```

9.3. Tunable Parameters

All tunable parameters have hardcoded defaults in `Settings._get_default_config()` and can be overridden via `config.json`.

Key	Type	Default	Description
`MIN_VOLUME_M`	int	`1000000`	Minimum 24h volume in USD.

Key	Type	Default	Description
`TARGET_EXCHANGES`	list	`["coinbase", "kraken", "mexc"]`	Exchanges to scan.
`UNIFORMITY_MIN_SCORE`	int	`45`	Minimum uniformity score to qualify.
`UNIFORMITY_PERIOD`	int	`30`	Number of days for uniformity analysis.
`TOP_COINS_LIMIT`	int	`5000`	Number of coins to fetch from CMC.
`ENTRY_NOTIFICATIONS`	bool	`true`	Send entry alerts.
`EXIT_NOTIFICATIONS`	bool	`true`	Send exit alerts.
`RETRY_MAX_ATTEMPTS`	int	`3`	Max retries on API failure.
`RETRY_DELAY`	int	`2`	Initial retry delay in seconds.
`RETRY_BACKOFF`	int	`2`	Exponential backoff multiplier.
`COINGECKO_CALLS_PER_MINUTE`	int	`30`	CoinGecko rate limit target.
`CMC_CALLS_PER_MINUTE`	int	`333`	CMC rate limit target.
`CACHE_GECKO_ID_DAYS`	int	`30`	Mapping cache TTL in days.
`CACHE_EXCHANGE_HOURS`	int	`24`	Exchange volume cache TTL in hours.
`CACHE_PRICE_HOURS`	int	`6`	Price history cache TTL in hours.
`CIRCUIT_FAILURE_THRESHOLD`	int	`5`	Failures before circuit opens.
`CIRCUIT_RECOVERY_TIMEOUT`	int	`60`	Seconds before circuit retry.

9.4. Configuration File Format

`config.json` is a flat JSON object. Keys match the table above. Unknown keys are silently ignored. Missing keys use defaults.

```
{
    "MIN_VOLUME_M": 1000000,
    "TARGET_EXCHANGES": ["coinbase", "kraken", "mexc"],
    "UNIFORMITY_MIN_SCORE": 45,
    "UNIFORMITY_PERIOD": 30,
    "TOP_COINS_LIMIT": 5000,
    "ENTRY_NOTIFICATIONS": true,
    "EXIT_NOTIFICATIONS": true
}
```

10. Notification System

10.1. Entry Notifications

Entry notifications are sent as Telegram photos with an HTML caption. The chart image is generated by Chart-IMG and attached inline.

Caption format:

```
🟢 <a href='{cmc_url}'>{SYMBOL} ({Name})</a>

📊 Gains:
  7d: +{gain_7d:.1f}%
  30d: +{gain_30d:.1f}%

📈 Uniformity Score: {score}/100

💰 Exchange Volumes:
🟦 Coinbase: ${vol:,.0f}
🐙 Kraken: ${vol:,.0f}
🟩 MEXC: ${vol:,.0f}
```

The `cmc_url` links to the coin's CoinMarketCap page: `https://coinmarketcap.com/currencies/{slug}/`.

Exchange volume lines SHOULD show `No volume` instead of `$0` or `N/A` when a coin is not traded on a particular exchange.

10.2. Exit Notifications

Exit notifications are plain text messages:

```
📅 {timestamp}

🔴 {SYMBOL} ({Name})
🔗 {cmc_url}
has left the qualified list
```

10.3. Telegram Bot Commands

The Telegram bot runs as a long-polling daemon (`telegram_bot.py`) and responds to these commands within the group:

Command	Response
`/status`	Current number of active coins, last scan time, scan duration.
`/list`	List of all currently qualified coins with their uniformity scores.

Command	Response
`/help`	List of available commands.

Bot commands are read-only. They query the database but never modify it or trigger scans.

---

11. Scheduling and Process Management

11.1. Scheduled Tasks

All scheduled tasks are configured in PythonAnywhere's task scheduler (cron equivalent):

Schedule	Script	Purpose
`55 * * * *`	`scheduler.py`	Hourly scan at :55 past the hour.
`0 0 * * 0`	`update_exchanges.py`	Weekly exchange listing refresh (Sunday midnight).
`0 0 1 * *`	`update_mappings.py`	Monthly CoinGecko mapping refresh (1st of month).
`*/5 * * * *`	`bot_watchdog.py`	Bot health check every 5 minutes.

The scanner runs at :55 rather than :00 to avoid overlap with the weekly and monthly maintenance jobs.

11.2. Scan Locking

`scheduler.py` uses an exclusive file lock (`fcntl.flock`) on `scan.lock` to prevent concurrent scan execution. If a scan is already running when cron fires, the new invocation logs a warning and exits immediately without blocking.

The lock file contains the PID of the lock holder for debugging. The lock is released in a `__exit__` handler that also unlinks the file.

11.3. Telegram Bot Process

The bot is managed by `manage_bot.py`, which provides `start`, `stop`, `restart`, and `status` subcommands. It writes the bot's PID to a `.pid` file and redirects stdout/stderr to `bot_output.log`.

11.4. Watchdog

`bot_watchdog.py` runs every 5 minutes via cron. It reads the PID file, checks if the process is alive, and calls `manage_bot.py start` if not. This ensures the bot recovers automatically from crashes within 5 minutes.

---

12. Logging and Diagnostics

12.1. Logging Architecture

The system uses Python's `logging` module with a two-handler setup per logger:

1. **Console handler** — `StreamHandler(sys.stdout)`, simple format (`%(message)s`), `INFO` level. Provides clean cron output.

2. **File handler** — `RotatingFileHandler`, detailed format (`%(asctime)s - %(name)s - %(levelname)s - %(message)s`), `DEBUG` level, 10 MB max with 5 backups.

The `utils/logger.py` module provides `setup_logger(name, log_file)` and a pre-configured `app_logger` instance for the main scanner.

12.2. Log Files

File	Writer	Content
`trend_scanner.log`	`app_logger` (main scanner)	Full scan pipeline progress, filter results, API call outcomes, errors.
`bot_output.log`	`manage_bot.py` (redirected stdout/stderr)	Telegram bot command handling, polling status.

Log files are local to the PythonAnywhere working directory. They are NOT committed to the repository.

12.3. Scan Metrics

`utils/metrics.py` provides a `MetricsCollector` that tracks per-scan counters:

- Total coins fetched from CMC.
- Coins eliminated at each filter stage.
- API calls made to each service.
- Cache hit/miss ratios.
- Wall-clock time per pipeline stage.
- Total scan duration.

Metrics are written to `metrics.json` after each scan and are available via the `/status` bot command.

---

13. Error Handling

13.1. API Failures

All API clients implement a retry-with-backoff strategy:

1. On a transient failure (timeout, 5xx, connection error), retry up to `RETRY_MAX_ATTEMPTS` times with exponential backoff.
2. On a 429 (rate limit), invoke the `RateLimiter`'s backoff escalation (see §7.5).
3. On a non-retryable failure (4xx other than 429), log the error and skip the coin. The pipeline continues with remaining coins.
4. If a service is completely unreachable, the circuit breaker opens and the scan proceeds without data from that service. This means some coins may be skipped for one scan cycle but will be picked up on the next.

No single API failure MUST cause the entire scan to abort. The pipeline is designed to degrade gracefully.

13.2. Database Errors

SQLite `database is locked` errors are retried up to 3 times with 100ms delays. If the lock persists, the operation is logged and skipped.

All database writes use `with conn:` context managers for automatic rollback on exceptions.

13.3. Process Failures

If the scanner process crashes mid-scan, the file lock is released automatically (the OS closes the file descriptor). The next cron invocation starts a clean scan.

If the bot process crashes, the watchdog detects it within 5 minutes and restarts it.

---

14. Performance Characteristics

14.1. Scan Profile

Metric	Typical Value
Input coins	~2,500 (from CMC)
Final qualified	15–25
Total API calls	150–250
Scan duration	10–15 minutes
Bottleneck	CoinGecko rate limiting (12s interval × 100+ calls)

14.2. Caching Strategy

Data	TTL	Storage	Impact
CoinGecko price history	6 hours	`price_cache` table	Eliminates ~60% of `/market_chart` calls
Exchange volumes (tickers)	24 hours	Inline in pipeline	Eliminates repeat `/tickers` calls within a day
Symbol → CoinGecko ID mapping	30 days	`mappings.db`	Eliminates all `/coins/list` calls during scans
Exchange listings	7 days	`exchanges.db`	Eliminates all exchange API calls during scans
TradingView symbol resolution	Indefinite (LRU)	In-memory + `tv_mappings.db`	Eliminates repeated DB lookups

Cache effectiveness after first scan: 60–80% hit rate.

14.3. Filter Selectivity

```
2,500 coins (CMC)
  → 1,500 (exchange-listed)       40% eliminated
  → 750   (volume ≥ $1M)          50% eliminated
  → 100   (gain thresholds)       87% eliminated
  → 90    (CoinGecko ID found)    10% eliminated
  → 20    (uniformity ≥ 45)       78% eliminated
  ─────────────────────────────
  Overall: ~99.2% elimination rate → 0.8% pass rate
```

15. Dependencies

15.1. Python Packages

Package	Purpose
`requests`	HTTP client for all API calls.
`python-dotenv`	Loads `.env` file into environment variables.

Standard library modules used extensively: `sqlite3`, `json`, `logging`, `time`, `os`, `sys`, `math`, `fcntl`, `pathlib`, `datetime`, `io`.

The dependency footprint is intentionally minimal. The system runs on PythonAnywhere where package installation is straightforward but excessive dependencies complicate deployment.

15.2. External Services

Service	Tier	Monthly Cost	Required
PythonAnywhere	Hacker ($5) or higher	~$5–10	Yes
CoinMarketCap Pro API	Basic	Free (10,000 calls/month)	Yes
CoinGecko API	Free / Demo	Free	Yes
Chart-IMG	Paid	~$10	Yes (for chart images)
Telegram Bot API	Free	Free	Yes

16. Future Considerations

The following items are potential enhancements that are explicitly out of scope for the current version but are architecturally anticipated:

- **Additional exchanges.** The `TARGET_EXCHANGES` config and exchange fetcher architecture support adding new exchanges without pipeline changes. Binance and KuCoin are natural candidates.
- **Configurable gain windows.** The 7-day and 30-day gain thresholds are currently hardcoded in `GainFilter`. These could be promoted to `config.json` parameters.
- **14-day gain filter.** A `USE_14D_FILTER` flag exists in the config template but is not implemented. If enabled, it would add a 14-day gain threshold (>14%) as an additional filter stage.

- **Historical trend analysis.** The `scan_history` table accumulates data over time. A future `/history {symbol}` bot command could show when a coin entered and exited the qualified list and how its score evolved.
- **Web dashboard.** A lightweight read-only dashboard pulling from the scan databases. Low priority given the Telegram-first design.
- **Notification preferences.** Allowing users to filter notifications by exchange, minimum score, or minimum gain.