# Dual-Signal 'Smart Money' Tracker: Implementation Guide

This document provides the canonical technical specification and implementation plan for the Dual-Signal 'Smart Money' Tracker project. It is structured into a series of epics and tasks, designed to guide the development process from initial setup through the creation of a Minimum Viable Product (MVP) and into future enhancement phases. The intended audience is the development team responsible for building, testing, and deploying the system.

## Project Configuration & Foundational Data

This initial section establishes the core, static data and configuration parameters required for the project. It serves as the single source of truth for the system's primary strategic input: the curated list of institutional investment managers, or "whales," whose holdings will form the basis of the initial filter.

The entire strategy is predicated on tracking a specific, qualitatively selected list of "smart money" institutional investors.1 The research document "An Analysis of 'Smart Money': Profiling Elite Institutional Investment Managers" provides a vetted list of 18 such managers, their investment philosophies, and their U.S. Securities and Exchange Commission (SEC) Central Index Keys (CIKs).2 This list is the direct and most critical input for the system's "whale" filter.

The system's architecture must therefore treat this list as a manageable configuration rather than a hard-coded constant. This design choice is fundamental, as the quality of the system's output is directly dependent on the quality of this input list. Future strategic reviews may necessitate adding, removing, or modifying the managers being tracked. By storing this information in a dedicated Funds table within the database, the system is built with the necessary flexibility to accommodate such changes without requiring code modifications. This elevates the task of populating the Funds table from a simple data-entry job to a strategic task of creating a flexible and updateable "source of truth" for the system's core filter.

### Curated 'Smart Money' Manager List

The following table contains the foundational dataset for the application. It codifies the qualitative research into a machine-readable format and will serve as the primary input for populating the Funds table in the database.

| Firm / Manager Name | Primary Strategy Archetype(s) | SEC Central Index Key (CIK) |
| --- | --- | --- |
| Elliott Investment Management L.P. | Activist Investing, Distressed Debt | 0000904495 |
| Starboard Value LP | Activist Investing, Concentrated | 0001517137 |
| Trian Fund Management, L.P. | Activist Investing, Concentrated | 0001345471 |
| Pershing Square Capital Management, L.P. | Activist Investing, High-Conviction & Concentration | 0001336528 |
| ValueAct Capital | Activist Investing, Long-Term Engaged | 0001351069 |
| Berkshire Hathaway Inc. | High-Conviction & Concentration, Deep Fundamental | 0001067983 |
| Akre Capital Management, LLC | High-Conviction & Concentration, Deep Fundamental | 0001112520 |
| Pabrai Investment Funds | High-Conviction & Concentration, Deep Fundamental | 0001571785 |
| Himalaya Capital Management LLC | High-Conviction & Concentration, Deep Fundamental | 0001709323 |
| The Baupost Group, L.L.C. | Deep Fundamental, Multi-Strategy Value | 0001061768 |
| Southeastern Asset Management, Inc. | Deep Fundamental, Concentrated, Engaged | 0000807985 |
| Ruane, Cunniff & Goldfarb Inc. | Deep Fundamental, High-Conviction & Concentration | 0001720792 |
| Dodge & Cox | Deep Fundamental, Long-Term Value | 0000029440 |
| First Eagle Investment Management, LLC | Deep Fundamental, Global Value | 0001325447 |
| Greenlight Capital, Inc. | Deep Fundamental, Concentrated, Long/Short Value | 0001079114 |
| Sachem Head Capital Management LP | Activist Investing, Concentrated, Value-Oriented | 0001582090 |
| JANA Partners LLC | Activist Investing, Event-Driven | 0001159159 |
| Engaged Capital, LLC | Activist Investing, Concentrated (Small/Mid-Cap) | 0001559771 |

## EPIC-01: MVP - Foundational Setup & Core Engine

This epic covers all non-interactive backend components required for the MVP. The objective is to build a robust, testable, and scalable foundation before any user-facing elements are developed. This includes setting up the project environment, designing the database, and implementing the core data parsing and signal analysis logic.

### Task 1.1: Project Scaffolding & Environment Setup

* **Description:** Establish the project's directory structure, initialize version control, and configure the Python environment with all necessary dependencies.
* **Implementation Details:**
  + Create the main project directory.
  + Initialize a Git repository using git init.
  + Set up a Python virtual environment (e.g., using python -m venv venv).
  + Create a requirements.txt file listing all core dependencies. Per the recommended technical stack, this file must include: psycopg2-binary, sqlalchemy, pandas, numpy, requests, and python-dotenv.1
* **Acceptance Criteria:** A clean development environment can be fully replicated on another machine by cloning the repository, creating a virtual environment, and running pip install -r requirements.txt.

### Task 1.2: API Credential and Configuration Management

* **Description:** Implement a secure method for managing sensitive information such as API keys and database credentials, ensuring they are not hard-coded into the source.
* **Implementation Details:**
  + Create a .env.template file in the project root. This file will serve as a template for developers, specifying all required environment variables (e.g., API\_KEY, DB\_HOST, DB\_PORT, DB\_NAME, DB\_USER, DB\_PASSWORD).
  + Create a .gitignore file and add entries to exclude .env, \_\_pycache\_\_/, and other sensitive or temporary files from version control.
  + Implement a central configuration module (e.g., config.py) that uses the python-dotenv library to load these variables from the .env file into the application's environment at runtime.1
* **Acceptance Criteria:** All sensitive data is loaded from environment variables. No credentials or API keys are present in any version-controlled file.

### Task 1.3: PostgreSQL Database and Schema Implementation

* **Description:** Design and implement the PostgreSQL database schema that will store all data related to funds, institutional holdings, and insider transactions. The choice of PostgreSQL is a foundational decision that dictates the project's viability beyond the MVP.1
* **Implementation Details:**
  + Write a SQL script (schema.sql) that defines the complete database schema.
  + The schema must include three primary tables as defined in the project plan: Funds, Quarterly\_Holdings, and Insider\_Transactions.1
  + Define appropriate data types (VARCHAR(10) for CIKs, BIGINT for share counts, TIMESTAMP WITH TIME ZONE for filing dates), primary keys, foreign key relationships, and indexes to ensure data integrity and optimize query performance.
  + A critical feature of the design is the inclusion of a raw\_json column of type JSONB in both the Quarterly\_Holdings and Insider\_Transactions tables. This leverages a key advantage of PostgreSQL over simpler databases like SQLite.1 Storing the original, complete JSON response from the API provider serves as an invaluable archive for debugging, reprocessing data without re-fetching, and future analysis on fields not initially parsed.
  + The selection of PostgreSQL is deliberate and forward-looking. Its sophisticated Multi-Version Concurrency Control (MVCC) system is essential for the live operational phase, where a daily script will be writing new Form 4 data into the database while other processes or users may be concurrently reading from it. Unlike SQLite, which typically uses file-level locking, PostgreSQL allows these simultaneous read and write operations without blocking, preventing a critical performance bottleneck that would otherwise require a major architectural refactor.1
* **Acceptance Criteria:** The schema.sql script can be executed against a clean PostgreSQL database to create the entire required schema, including all tables, columns, constraints, and indexes, without any errors.

### Task 1.4: Data Parser Development

* **Description:** Develop a set of pure, testable functions for parsing the raw JSON data returned by the SEC filings API provider. This explicit separation of parsing logic from data acquisition and analysis is a critical design choice for ensuring testability and maintainability.
* **Implementation Details:**
  + Create a dedicated Python module, parsers.py.
  + Implement a function parse\_13f\_filing that accepts a raw JSON object representing a 13F filing and returns a structured list of dictionaries, where each dictionary corresponds to a row in the Quarterly\_Holdings table.
  + Implement a function parse\_form4\_filing that accepts a raw JSON object for a Form 4 filing and returns a structured dictionary corresponding to a row in the Insider\_Transactions table.
  + These functions are responsible for all data extraction, cleaning, and type conversion.
  + This approach allows for the development and validation of the complex data transformation logic using static sample JSON files, without needing to make a single live API call. This de-risks development, speeds up debugging, and makes the entire system more modular and robust.
* **Acceptance Criteria:** Each parser function can be tested in isolation. When provided with a sample static JSON file representing a filing, each function correctly processes the file and returns structured, validated data that matches the database schema.

### Task 1.5: Core Signal Analysis Engine

* **Description:** Implement the central analysis engine that contains the project's core strategic logic. This engine will be responsible for identifying the dual-signal alerts based on the data stored in the database.
* **Implementation Details:**
  + Create a new module, signal\_generator.py.
  + Implement build\_whale\_watchlist(quarter, year): This function queries the Quarterly\_Holdings table to construct and return a set of stock tickers that meet the "strong ownership" criteria for a given historical period.1
  + Implement find\_insider\_triggers(start\_date, end\_date): This function queries the Insider\_Transactions table for significant insider buys within a specified date range. The filtering criteria must be strict, focusing on Transaction Code 'P' (open-market purchase) and prioritizing purchases by C-suite executives (CEO, CFO, COO).1
  + Implement generate\_signals(watchlist, triggers): This function takes the output from the previous two functions, compares the set of insider buy tickers against the active watchlist, and returns a list of any tickers present in both, representing the final dual-signal alerts.
* **Acceptance Criteria:** The engine's logic can be validated with a pre-populated test database. Given a known historical scenario, the engine correctly identifies the expected signal.

### Task 1.6: Database Seeding and Test Data

* **Description:** Create a utility script to populate the database with the initial list of curated funds and a small, static set of historical data for testing the MVP.
* **Implementation Details:**
  + Create a script named seed\_db.py.
  + This script will first populate the Funds table with the 18 managers and their CIKs from the configuration table defined at the beginning of this document.2
  + The script will then read from a local directory containing a small, curated set of historical 13F and Form 4 JSON files.
  + It will use the functions from parsers.py (Task 1.4) to process these static files and load the resulting structured data into the Quarterly\_Holdings and Insider\_Transactions tables.
* **Acceptance Criteria:** After running the seed\_db.py script, the database is fully populated with the 18 funds and contains a small but complete set of historical data sufficient for testing the functionality of the historical simulator in the next epic.

## EPIC-02: MVP - Interactive Historical Simulator

This epic focuses on building the user-facing component of the MVP. The goal is to create an interactive command-line tool that allows a user to explore the strategy's performance on historical data. This tool is not merely a prototype; it is a critical de-risking mechanism that serves as a "manual backtest," allowing for the qualitative validation of the strategy before investing in full automation or live deployment.

### Task 2.1: Command-Line Interface (CLI) Scaffolding

* **Description:** Create the main executable script for the simulator and implement the command-line argument parsing.
* **Implementation Details:**
  + Create the main script file, simulator.py.
  + Utilize Python's built-in argparse library to handle command-line arguments.
  + The primary required argument will be the historical quarter to analyze, which should be accepted in a clear and consistent format (e.g., --quarter Q4-2020).
  + Implement input validation to ensure the user provides a valid quarter format and that the corresponding data exists in the database.
* **Acceptance Criteria:** The script can be executed from the command line (e.g., python simulator.py --quarter Q4-2020) and correctly parses the specified historical quarter without errors.

### Task 2.2: Simulation Orchestration Logic

* **Description:** Implement the main control loop of the simulator, which orchestrates the calls to the core analysis engine and manages the flow of the historical simulation.
* **Implementation Details:**
  + Upon launch, the script will call the build\_whale\_watchlist function from the core engine (Task 1.5) for the user-specified quarter. It is crucial that this step correctly models the 45-day filing delay inherent in 13F data; for a quarter ending December 31, the watchlist is only considered "known" on February 15 of the following year.1
  + The script will then enter a loop, iterating day-by-day through the subsequent quarter (e.g., from February 16 to May 15).
  + On each simulated day, it will call the find\_insider\_triggers and generate\_signals functions to check for any dual-signal alerts that occurred on that historical date.
* **Acceptance Criteria:** The simulator correctly constructs the point-in-time watchlist for the specified period and iterates sequentially through the subsequent 90 days of historical data from the database.

### Task 2.3: Alert Generation and Console Output

* **Description:** Develop a well-formatted and detailed console output for displaying signal alerts. The quality of this output is central to the MVP's primary function as a validation and research tool.
* **Implementation Details:**
  + When the generate\_signals function returns one or more alerts for a given day, the simulation loop must pause.
  + A dedicated function will be developed to print a detailed, human-readable report to the console for each alert.
  + The report must include all relevant information as specified in the project plan 1:
    - **Stock Identifier:** Ticker and Company Name.
    - **Trigger Event:** Date of the insider purchase.
    - **Insider Details:** The insider's name and their role (e.g., CEO, CFO).
    - **Transaction Details:** Number of shares purchased and the total transaction value.
    - **Conviction Context:** A list of the curated "Whales" from the active watchlist that held the stock at the start of the quarter.
  + This detailed context allows the user to examine each signal individually, building intuition and helping to identify potential patterns or flaws in the logic (e.g., are signals clustering in a specific sector? Are they often followed by price drops?). This qualitative feedback loop is invaluable for refining the signal definition.
* **Acceptance Criteria:** The console output is clear, readable, contains all required information, and accurately reflects the underlying database data that triggered the signal.

### Task 2.4: User Interactivity

* **Description:** Implement a simple mechanism for the user to control the pace of the simulation.
* **Implementation Details:**
  + After displaying a signal alert (or a block of signals for a given day), the script will pause execution.
  + It will use a simple input() prompt to wait for user confirmation before proceeding. The prompt should give the user options, such as continuing to the next day with a signal, or exiting the simulation.
* **Acceptance Criteria:** The user can effectively control the flow of the simulation, allowing for careful review of each signal before proceeding.

## EPIC-03: Post-MVP - Live Data Integration

This epic outlines the generic phase for transitioning the validated MVP from a historical analysis tool into a live, automated signal generator. It focuses on building the data acquisition pipeline that will keep the system's database current with the latest SEC filings. The system's operational rhythm is defined by two distinct, asynchronous data streams, mirroring a strategic (quarterly) and tactical (daily) decision-making process.

### Task 3.1: Quarterly 13F Harvester Script

* **Description:** Create a standalone, automated script to fetch and store the latest quarterly Form 13F-HR filings for the curated list of "whales." This script performs the "Quarterly Strategic Review," updating the entire "hunting ground" of potential investments.
* **Implementation Details:**
  + Develop a new script, quarterly\_13f\_harvester.py.
  + The script will first query the Funds table to retrieve the current list of CIKs to track.
  + It will then iterate through this list, making API calls to the data provider's 13F endpoint (e.g., sec-api.io) to fetch the latest quarterly holdings for each manager.1
  + For each API response, it will utilize the parse\_13f\_filing function (from Task 1.4) to process the data.
  + The structured holdings data will then be inserted into the Quarterly\_Holdings table.
  + The script must include robust error handling (e.g., for API failures or network issues) and detailed logging to allow for unattended execution. A failure in this quarterly script is a high-severity event that compromises the entire strategy for the next three months and requires immediate manual intervention.
* **Acceptance Criteria:** When executed on its schedule, the script successfully fetches and stores the latest available 13F data for all curated funds in the database without manual intervention.

### Task 3.2: Daily Form 4 Monitor Script

* **Description:** Create a standalone, automated script to fetch and store all new Form 4 filings from the SEC. This script performs the "Daily Tactical Scan" for actionable triggers within the established universe.
* **Implementation Details:**
  + Develop a new script, daily\_form4\_monitor.py.
  + The script will make an API call to the data provider's Form 4 endpoint, requesting all filings submitted within the last 24 hours.1 For providers like  
    sec-api.io that offer a real-time streaming service, a more advanced implementation could maintain a persistent WebSocket connection for lower latency.1
  + It will iterate through the filings in the response, using the parse\_form4\_filing function (from Task 1.4) to process each one.
  + The structured transaction data will be inserted into the Insider\_Transactions table.
  + The script must include logic to prevent the insertion of duplicate records, for instance, by checking the unique accession number of each filing before insertion.
  + Robust error handling and logging are required. An error in this daily monitor is less severe than a quarterly failure (a single day's triggers might be missed) and could potentially be retried automatically.
* **Acceptance Criteria:** The script, when run, successfully fetches and stores all new Form 4 filings from the previous 24-hour period into the database.

### Task 3.3: Scheduling and Automation

* **Description:** Document the process for scheduling the two harvester scripts for automated execution on a server.
* **Implementation Details:**
  + Provide clear, copy-pasteable examples of cron entries for a standard Linux/macOS environment.
  + **Quarterly 13F Harvester:** The schedule must be set to run after the 45-day filing deadline. For example: 0 2 16 2,5,8,11 \* /path/to/venv/bin/python /path/to/project/quarterly\_13f\_harvester.py (This runs at 2:00 AM on the 16th of February, May, August, and November).1
  + **Daily Form 4 Monitor:** The schedule should be set to run at least once per day, ideally after market close to capture the full day's filings. For example: 0 22 \* \* 1-5 /path/to/venv/bin/python /path/to/project/daily\_form4\_monitor.py (This runs at 10:00 PM every weekday).1
* **Acceptance Criteria:** The documentation provides clear and correct instructions for automating the entire data acquisition pipeline using a standard scheduling utility like cron.

## EPIC-04: Future Enhancements (Product Backlog)

This section serves as a high-level product backlog. It captures the generic next steps for evolving the system beyond its core signal-generation function, as requested by the user. These items represent a logical progression from a data-processing tool to a comprehensive, interactive trading system. They are not fully specified but provide a clear direction for future development sprints.

### Backlog Item 4.1: Signal Scoring System

* **User Story:** As a user, I want signals to be scored based on their conviction level so that I can prioritize the most promising opportunities.
* **Implementation Notes:** This enhancement involves refactoring the generate\_signals function to return a numerical score for each alert instead of a simple boolean. The score would be a weighted sum of several factors, such as the number and quality of "whales" holding the stock, the seniority of the purchasing insider (e.g., CEO > Director), the absolute and relative size of the insider purchase, and the presence of a "cluster buy" where multiple insiders purchase shares in a narrow time frame.1

### Backlog Item 4.2: Automated Alerting System

* **User Story:** As a user, I want to receive real-time notifications when a new high-conviction signal is generated so that I can act on it in a timely manner.
* **Implementation Notes:** This task involves integrating the signal generation logic with a notification service. A robust scoring system (Backlog Item 4.1) is likely a prerequisite to avoid sending low-quality alerts. The implementation could use Python's built-in smtplib for email notifications or a dedicated library like python-telegram-bot to push alerts to a private Telegram channel for lower latency.1

### Backlog Item 4.3: Portfolio Construction & Risk Management Module

* **User Story:** As a user, I want the system to suggest position sizes and manage portfolio-level risk so that I can translate signals into a disciplined trading strategy.
* **Implementation Notes:** This represents a major evolution from a signal generator to a full trading system. It would require the development of new modules to handle position sizing (e.g., fixed fractional sizing based on account equity), the definition of clear rules for setting stop-loss and take-profit levels, and the ability to track and manage overall portfolio-level risk, such as maximum exposure to a single stock or sector.1

### Backlog Item 4.4: Web-Based Dashboard

* **User Story:** As a user, I want a graphical interface to visualize the current watchlist, review historical signals, and analyze performance so that I can interact with the system's data more intuitively.
* **Implementation Notes:** The backend logic and database developed for the MVP can serve as the foundation for a web application. A framework like Streamlit or Dash, which are specifically designed for creating interactive data applications in Python, would be ideal. The dashboard would provide a user-friendly front-end for the functionality currently available in the command-line simulator, greatly improving usability.1

### Backlog Item 4.5: Automated Backtesting Engine

* **User Story:** As a user, I want to run a full-scale, automated backtest of the strategy over many years of data to quantitatively assess its historical performance and risk characteristics.
* **Implementation Notes:** This task involves building a new, sophisticated simulation engine that can rigorously test the strategy on historical data in a point-in-time correct manner, avoiding any look-ahead bias. This would require acquiring historical market price data (daily open, high, low, close) from a provider like Polygon.io. The engine would simulate trades based on the strategy's signals and produce a comprehensive report with standard performance metrics (e.g., Compound Annual Growth Rate, Sharpe Ratio, Maximum Drawdown) using a quantitative finance library like quantstats or pyfolio.1

#### Works cited

1. Project Plan: Dual-Signal 'Smart Money' Tracker
2. Smart Money Fund Identification