ChatGPT

Kairos Project Pipeline Owner's Manual

Pipeline Structure Overview (By Phase)

The Kairos modeling pipeline is organized into sequential phases, from raw data ingestion to factor generation, composite signal construction, feature matrix assembly, and ultimately backtesting. Each phase's scripts and modules are described below, along with how they depend on one another.

Phase 1: Data Ingestion and Universe Definition

Purpose: Acquire daily market data and define the stock universe (Option B) for analysis.

- **Daily Data Download:** The script `daily_download.py` (invoked via a daily cron job) fetches the latest daily pricing and fundamental data (e.g. from Sharadar) for the universe. The `daily_update_pipeline.sh` runs this each morning [1]. For initial setup, historical data must be loaded (e.g. via bulk CSV import or Kaggle dataset), establishing base tables like `sep` (Sharadar Equity Prices) and `tickers` (metadata).

- **Merging to DuckDB:** New daily data is merged into the DuckDB database using `merge_daily_download_duck.py`. This updates the incremental price table `sep_base` with fresh records [2] [3]. The `sep_base` table is an "incremental golden copy" of price history that gets appended daily.

- **Option B Universe Construction:** The script `create_option_b_universe.py` defines the investable universe ("Option B"). It filters the `tickers` and `sep_base` tables by criteria like security type, exchange, market cap category, price, and liquidity (ADV) thresholds [4] [5]. Specifically, it selects domestic common stocks on NYSE/NASDAQ/AMEX, in Small to Mega cap buckets, with recent **60-day average dollar volume** $\geq$ `min_adv` and last price $\geq$ `min_price` [5]. The output is: (1) a DuckDB table `sep_base_expanded` containing all historical price data for tickers in the universe [6] [7], and (2) a CSV listing the universe tickers (if `--universe-csv` is provided) [8] [9]. *Dependency:* Requires `sep_base` and `tickers` to be populated [2].

- **Academic Base Table Creation:** The script `create_academic_base.py` builds the optimized `sep_base_academic` table – a clean, sorted price history for just the universe tickers [10]. It reads the Option B universe CSV and filters `sep_base` to those tickers, then sorts by (ticker, date) [11] [12]. The result is a compact ~10 million row table used for all subsequent factor calculations [13] [14]. This step is the successor to the older in-memory "expanded" approach and should be rerun whenever the universe or raw data updates (e.g. new tickers or daily data) to keep `sep_base_academic` current [15] [16].

Phase 2: Technical Feature Generation (Price/Volume Derived)

Purpose: Compute core technical features from price and volume history for each ticker-date, using `sep_base_academic` as the source. These **feature scripts** are typically run *daily* after new data is ingested (they are included in the cron pipeline [17]) to update rolling signals:

- **Price Action & Trend Features:** Scripts like `price_action_features.py` and `trend_features.py` calculate recent price performance indicators (e.g. moving average crossover signals, breakout indicators) and momentum/trend metrics. These likely produce tables such as `feat_price_action` and `feat_trend`. For example, *momentum* features (like 12-month momentum, 1-month reversal) are computed via windowed SQL on `sep` data [18] [19] and stored in `feat_momentum_v2` (or integrated into `feat_trend`). The output includes columns like 12-1 month momentum and 1-month reversal, often standardized (z-scored) with winsorization [20] [21].

- **Statistical & Shape Features:** The `statistical_features.py` and `price_shape_features.py` modules generate features capturing return distributions and patterns (e.g. volatility, skewness, max drawdown periods, etc.). These populate tables `feat_stat` and `feat_price_shape`.

- **Volume & Volatility Features:** `volume_volatility_features.py` computes volume-related signals and volatility measures. One key output is the **average daily volume (ADV) and size factor table** `feat_adv`, which contains `adv_20` (20-day average dollar volume) and `size_z`. Here `adv_20` provides a liquidity metric for each ticker (used later to filter investable names), and `size_z` is likely a z-scored market-cap or size indicator. These are used in both portfolio construction and risk control (e.g. many strategies require ADV $\geq$ 2,000,000 as a liquidity cutoff [22] [23]). If not computed in the above script, ADV and size features may be built by a dedicated script (e.g. an internal `adv_features.py`). In any case, `feat_adv` is listed among required feature tables for matrix assembly [24] and provides `adv_20` and `size_z` columns for later use [25] [26].

- **Volatility Sizing Features:** The script `vol_sizing_features.py` calculates rolling volatility metrics for position sizing. It produces table `feat_vol_sizing` with columns: 21-day vol, 63-day vol, and a blended vol (e.g. 50/50 blend of 1-month and 3-month realized vol) [27] [28]. These are computed from `sep_base_academic` returns and are used to scale portfolio weights (lower weight for higher-volatility stocks). The output `feat_vol_sizing` is built by computing daily returns per ticker and then rolling std dev over 21 and 63 days [29], resulting in `vol_21`, `vol_63`, and `vol_blend` [30].

- **Beta Features:** The `beta_features.py` script computes rolling market beta for each stock. It defines an equal-weighted "market" return (using the Option B universe itself as the market) and then calculates 21-day, 63-day, and 252-day CAPM betas (plus a 63-day residual volatility) for each ticker [31] [32]. The output table `feat_beta` contains columns `beta_21d`, `beta_63d`, `beta_252d`, and `resid_vol_63d` [33]. These beta features are crucial for the long-short strategies (to enable beta-neutral portfolios).

- **Forward Return Targets:** Finally, `generate_targets.py` produces future return **targets** for training and backtesting. This reads `sep_base_academic` (sorted by date) and computes forward

returns, e.g. 1-day and 5-day forward returns (`ret_1d_f`, `ret_5d_f`), as well as a binary label for 5-day positive return [34] [35]. The results are stored in `feat_targets` [36]. These targets serve as the dependent variables for modeling or simply as realized performance to evaluate signals. They should be regenerated whenever new price data is added.

Dependencies: All these feature scripts require `sep_base_academic` to be up-to-date and typically assume the existence of the universe's entire price history. When base data updates daily, these feature computations (especially rolling metrics) are run to append the newest date's features. They are largely independent of each other, each producing a `feat_*` table keyed by (ticker, date).

Phase 3: Fundamental Factor Construction (Academic Factors)

Purpose: Construct "academic" fundamental factors (e.g. value, quality, growth, sentiment) from financial data, often using quarterly fundamentals or specialized datasets. Based on Phase 4 research, the project has v2 versions of several factors with improved quality. Key factor scripts include:

- **Value Factors (v2):** `build_value_factors_v2.py` creates institutional-quality value measures [37] [38]. It uses fundamental ratios from the Sharadar `daily` table (or similar) – e.g. P/E, P/B, P/S, EV/EBITDA – to compute yields (their inverses) with reasonable bounds [39] [40]. It outputs `feat_value_v2` with columns like `earnings_yield`, `book_yield`, `ebitda_yield`, `sales_yield`, and aggregated scores such as `value_composite_z` and a quality-adjusted value (`value_quality_adj`) [41] [42]. The quality adjustment penalizes extremely cheap, distressed stocks (so-called D10 decile value traps) while rewarding "cheap but not too cheap" stocks [43] [44]. This results in a more predictive value factor.

- **Quality Factors (v2):** `build_quality_factors_v2.py` produces a quality composite from fundamentals like profitability and accruals. It likely uses Sharadar SF1 data (quarterly fundamentals) to derive metrics such as ROE, ROA, and accruals, then expands them to daily frequency. Indeed, the code pulls SF1-derived `computed_roe`, `computed_roa`, `computed_accruals` for each quarter, forward-fills them to daily, and then computes cross-sectional z-scores (`quality_composite_z`) [45] [46]. The output `feat_quality_v2` contains the z-scores of ROE/ROA (and negative accruals), representing a combined quality signal. Any NaNs (no data) are filled with neutral 0 after z-scoring [47]. This script should be run whenever new fundamentals are available (e.g. quarterly).

- **Growth Factors (likely v2):** Though not explicitly shown, a similar approach would be used to build growth factors (e.g. revenue or earnings growth rates). These might be integrated into `feat_composite_academic` or another table, or generated by a `build_growth_factors.py`. The *long-horizon composite* scripts (below) reference `growth_z` as part of `feat_composite_long`, indicating that growth z-scores were computed (perhaps as part of an "academic composite" in Phase 3). If separate, `feat_growth_v2` would contain z-scores for growth metrics like EPS growth or sales growth.

- **Momentum Factors (v2):** Long-term price momentum factors are computed in `build_momentum_factors_v2.py`, though these could also be considered technical, they were refined in the academic context. Based on Phase 4 findings, the best momentum signal was 12-month momentum excluding the most recent month (`mom_12_1`), and a complementary factor is 1-

month reversal [18]. This script outputs `feat_momentum_v2` with columns: 1M, 3M, 6M, 12M momentum, the 12_1 variant, 1M reversal, and a combined `momentum_composite_z` [20]. These are calculated via SQL window functions on the `sep` price table [48] [49], and then standardized with winsorized z-scores [21].

- **Sentiment/Other Factors:** The project also incorporates factors like **insider trading activity**. For example, `build_insider_factors.py` (mentioned in search results) processes insider buy/sell data to compute a 30-day net insider buying signal, then cross-sectionally scores it (`insider_composite_z`) [50] [51]. The output `feat_insider` provides an `insider_composite_z` for each ticker-date (with net insider buying considered bullish). Additionally, an **institutional ownership** factor was likely built (perhaps via `feat_institutional_academic`), but not detailed here.

- **ADV & Size Factors:** As noted, `feat_adv` table (from Phase 2) provides **liquidity (ADV)** and **size** metrics. These can be considered part of the factor set because they are used in portfolio construction (e.g. to filter out low-ADV stocks and to include size in long-term composite). `feat_adv` typically would be built by calculating each stock's daily dollar volume (price * volume) and z-scoring the 20-day average (for `adv_z`), and possibly z-scoring the market cap (for `size_z`). The presence of `adv_20` and `size_z` in the feature matrix ensures strategies can enforce liquidity thresholds and account for size effects [52] [26].

Dependencies: These factor scripts depend on **base tables** beyond just price: e.g. Sharadar `daily` fundamentals for value, SF1 fundamentals for quality/growth, insider trade datasets for insider, etc. They also often require the ticker universe to focus on relevant stocks (some scripts might join with `tickers` to ensure the security type is common stocks). After computing these factor tables (`feat_value_v2`, `feat_quality_v2`, `feat_momentum_v2`, `feat_insider`, etc.), subsequent composite-building will use them. Whenever **new fundamental data** arrives (e.g. quarterly updates), rerun the affected factor scripts so that `feat_*_v2` tables stay current.

Phase 4: Composite Signal Construction

Purpose: Combine multiple factors into composite alpha signals that serve as the **"alpha forecasts"** for the strategy. There are two categories of composites developed:

- **Long-Horizon Composite (CL):** The "Composite Long" signal blends fundamental factors that predict longer-term returns. In Phase 3, an initial composite `feat_composite_long` (CL v1) was created from quality, growth, and institutional ownership factors (yielding `quality_z`, `growth_z`, `inst_z`). This was refined in `build_composite_long_v2.py`, which adds value and size components [53] [54]. Specifically, CL v2 combines: quality, growth, institutional (`inst_z` from v1) plus **value_z** (from `feat_value`), and **size_z & adv_z** (from `feat_adv`) [55] [56]. It takes the mean of these z-scores to produce `alpha_CL_v2` [57]. The result is saved as `feat_composite_long_v2` with all component z-scores and the final `alpha_CL_v2` score [58] [59]. CL v2 represents a broad, long-horizon alpha signal that includes quality, growth, value, and size tilts.

- **Cross-Sectional Composite (CS):** Another composite focuses on shorter-term predictive signals (e.g. technical momentum, reversal, insider, etc.). Early on, `feat_composite_v3` was created (referred to as "CS+CL v1 blend" [60]) – possibly combining a cross-sectional score with the initial long composite. This evolved through versions v31, v32, v33, often incorporating improvements and regime considerations:

- **Composite v3 & v31:** These likely combined a cross-sectional factor (like momentum or an earlier alpha composite) with the CL composite. For example, `alpha_composite_v31` is cited as an input to Risk Model v3 [61] [62]. It may have been an improved blend of short-term and long-term factors.

- **Composite v32b and v33_regime:** By Phase 5–6, the project introduced a **regime-aware composite.** `alpha_composite_v33_regime` appears to be a key signal, which likely started as `v33` (perhaps blending the latest CS and CL signals) and then adjusted for market regimes. The `regime_detector_academic.py` script produces a `regime_history_academic` table labeling each date with a regime (e.g. low_vol_bull, high_vol_bear, etc.) based on market volatility and trend. The composite v33 was then modified per regime: for instance, an earlier composite v6 applied multipliers >1 or <1 in certain regimes [63] [64]. The final `alpha_composite_v33_regime` likely scales the base alpha by a factor depending on whether the market regime is favorable or not (e.g. down-weighting alpha in high-volatility or neutral regimes, boosting in bullish regimes). This resulted in a signal that times factor exposure by regime.

- **Next-Generation Blends (v5, v7, v8):** In parallel, the team experimented with weighted combinations of all high-IC factors:

- **Composite v5:** Combined the strongest predictive factors (EBITDA Yield, quality-adjusted value, momentum, reversal, insider, quality composite) with optimized weights [65]. The weights were IC-proportional (e.g. ~35% to EBITDA yield, 24% momentum, etc.) [65]. It produced `feat_composite_v5` containing `alpha_composite_v5` (and possibly a `_regime` variant) [66]. This was an early attempt at a "super-composite" with an expected IC of ~0.02 [67].
- **Composite v7:** Later, composite v7 was introduced as a simple blend: 50% of the regime-adjusted composite and 50% of the quality factor [68] [69]. This was likely motivated by quality's strong standalone performance. It outputs `feat_composite_v7` with `alpha_composite_v7`.
- **Composite v8:** This is the **"best-performing alpha signal"** to date. The formula is **35% v33_regime + 35% quality + 30% value** [70]. In other words, it blends the regime-aware composite with a pure quality and pure value component. For days where `v33_regime` is unavailable (e.g. early periods), it falls back to a 54/46 quality/value mix [71]. The rationale: quality and value factors both have high information coefficients (IC ~0.022 each) and are uncorrelated with each other and with the original composite, so adding value significantly boosts Sharpe while reducing correlation [72] [73]. Backtests showed Sharpe improving from 1.32 (v33) to 1.46 with v8 [74]. `build_alpha_composite_v8.py` creates `feat_composite_v8` with `alpha_composite_v8` and also updates it into the feature matrix if present [75] [76].

After building these composites, the key outputs are **tables like** `feat_composite_long_v2`, `feat_composite_v33_regime`, `feat_composite_v7`, `feat_composite_v8` (as well as older versions for reference). These represent the final **alpha signals** that will be fed into the backtest engines.

Dependencies: Composite scripts require the underlying factor tables to be ready. For example, composite v8 needs `feat_composite_v33_regime`, `feat_quality_v2`, and `feat_value_v2` to exist [77]. If any are missing, the script alerts and may compute with what's available [78] [79]. The composite outputs should be regenerated whenever factor inputs change or new blending ideas are tested. In practice, v33_regime and v8 are likely updated after major re-training or annually, not necessarily daily.

Phase 5: Feature Matrix Assembly

Purpose: Join all feature tables into a single **"feature matrix"** for modeling and backtesting. The feature matrix includes every (ticker, date) in the universe, with all computed features, composites, and targets as columns. This simplifies downstream queries and ensures alignment of data.

- **Feature Matrix Build:** The latest process uses `build_full_feature_matrix.py`, which performs the assembly entirely within DuckDB for efficiency. It first creates a temp base of all (ticker, date) pairs from `sep_base_academic` (filtered to universe tickers) [80] [81]. Then it **iteratively LEFT JOINs** each `feat_*` table onto this base [82] [83]. The script defines an ordered list `ACADEMIC_TABLES` to join in sequence, including all technical features, targets, composites, etc. [84] [85]. Key tables in this list are: `feat_price_action`, `feat_price_shape`, `feat_stat`, `feat_trend`, `feat_volume_volatility`, `feat_targets`, `feat_composite_academic`, `feat_institutional_academic`, `feat_composite_long`, `feat_value`, `feat_adv`, `feat_composite_long_v2`, `feat_composite_v3`, `feat_composite_v31`, `feat_vol_sizing`, `feat_beta`, `feat_composite_v32b`, `feat_composite_v33_regime` [84] [24] (and potentially v7/v8 if added later). As each table is joined, its columns (except the keys) are added to the growing wide table. After all joins, it creates the final `feat_matrix` DuckDB table [86]. This table typically has tens or hundreds of columns (all signals, factors, and targets) and covers the full date range for the universe.

- **Legacy Method (Feature Matrix v2):** Previously, `build_feature_matrix_v2.py` was used to do a similar job in Python/Pandas. It loaded all `feat_` tables and merged them while handling duplicate column names [87] [88]. It also explicitly prioritized newer v2 factors over legacy to avoid confusion in column names [89]. The final output was `feat_matrix_v2` (to distinguish from the older one). In practice, the newer DuckDB-based builder supersedes this, and the primary output is now `feat_matrix` (or `feat_matrix_v2` if named so).

- **Exports:** The `build_full_feature_matrix.py` script also exports the matrix as a partitioned Parquet dataset per ticker (useful for training ML models outside DuckDB) [90] [91]. The directory is named by date (snapshot date) to clearly version the matrix.

Dependencies: All individual `feat_` tables from prior phases must be built before assembling the matrix. The universe CSV is needed to restrict to desired tickers [92]. When base data updates (new dates or if the universe membership changes), you should regenerate the feature matrix. For incremental daily updates, one might choose to just append new day rows to `feat_matrix`. However, the safe approach (as used in code) is to rebuild fully, since it only takes on the order of seconds to a couple of minutes in DuckDB for ~10M rows. Ensure to provide the latest date to include via `--date` and the current universe CSV when running the builder.

Phase 6: Backtesting Suite

Purpose: Evaluate the performance of the alpha signals via historical simulation. The project provides a suite of backtest scripts for different strategy variants, organized as **Phase 8A through 8D** in increasing complexity:

- **Baseline Academic Strategy:** `backtest_academic_strategy.py` is a clean, generic backtester for any given alpha column. It ranks stocks by the chosen `alpha_column` on each rebalance date and builds an **equal-weight** portfolio (long-only or long-short, based on parameters) [93] [94]. It uses the forward return column (default `ret_5d_f`) to compute outcomes [93]. It reports total return, annualized return, volatility, Sharpe, and max drawdown for the portfolio vs a benchmark [94] [95]. This is essentially an "academic baseline" with minimal risk controls: no volatility targeting or sector constraints by default, just top-N equal weighting. You can optionally supply `--regime-filter` to only trade during certain regimes (e.g. only neutral volatility periods) [96]. This baseline is useful to see raw alpha efficacy. It does **not** save results to the DB (outputs stats to console).

- **Risk-Controlled Long-Only Strategies (v1–v4):** These progressively add realistic constraints to the long-only portfolio. Each is a separate script:

- **V1 (Risk model v1):** `backtest_academic_strategy_risk.py` introduces basic risk management: **z-scoring the alpha cross-sectionally** each rebalance, clipping outliers at ±3, picking top N, and applying **volatility targeting** to reach a target annual vol (e.g. 20%) [97] [98]. The portfolio is equal-weight long-only (no short positions) [97]. Essentially, v1 = baseline + vol normalization.
- **V2:** `backtest_academic_strategy_risk2.py` (implied by context) adds further refinements. From code, Risk v2 includes **alpha-proportional weighting** (instead of equal weight) and basic constraints: it ranks by alpha_z and then weights stocks proportional to their alpha (capped at 3% max per stock) and also enforces **sector caps** (no sector's weight > 2× its weight in the universe) [99] [100]. This yields a more balanced long-only portfolio with controlled single-name and sector exposure.
- **V3:** `backtest_academic_strategy_risk3.py` adds **volatility awareness and liquidity filtering**. It requires that each stock's `vol_blend` (volatility) and `adv_20` (ADV) are present [61] [101]. The steps include dropping stocks with missing sector or vol data, then after z-scoring alpha and clipping, it **filters out stocks with ADV below a threshold** (ensuring sufficient liquidity) [102] [101]. It then uses **alpha_z / vol_blend** as the weight basis (so higher volatility stocks get lower weight) [103] [62]. Like v2, it caps at 3% and enforces sector limits [103]. Finally, it computes raw 5-day returns and then **volatility-targets the entire portfolio** to the desired vol [102] [104]. This strategy (Long-Only, Volatility-Aware, Sector-Capped, ADV-filtered) is labeled **Risk Model v3** [105].
- **V4:** `backtest_academic_strategy_risk4.py` is the most advanced long-only model. It includes all of v3's elements (vol targeting, sector and ADV constraints, etc.) and adds **turnover control** [106] [107]. Before each rebalance, it blends the previous weights with the newly proposed weights (`lambda_tc` controls how much to refresh vs. stick with old weights) to smooth turnover [108]. It then caps turnover to a maximum threshold: if the change in weights exceeds the cap, it scales down the changes proportionally [108]. This yields a new weight vector `w_new` that changes more gradually [108]. After this, weights are normalized and the portfolio is formed. Risk4 therefore achieves: long-only, volatility position sizing, sector and per-stock caps, ADV filter, **and turnover minimization** [106]. It's used as the Phase 8A "long-only engine" in later combos. By default, these risk backtests use a top-N (e.g. 75) selection each 5 days and target 20% annual vol [109].

All risk backtest scripts print a performance summary (portfolio vs benchmark returns, vol, Sharpe, drawdown) [110] [111]. The **"risk4"** script has two variants: one just reports stats, and `backtest_academic_strategy_risk4_save.py` which additionally **saves the daily returns to the DB**. The save version writes a table `backtest_results_longonly_r4` containing the time series of portfolio returns (`port_ret`), benchmark returns, and active returns [112]. This is in preparation for regime switching (Phase 8D) which needs access to these results.

- **Long-Short Strategies:** The suite includes two approaches to long-short portfolio construction:
- **Analytic Long-Short (Phase 8B):** `backtest_academic_strategy_ls.py` implements a **dollar-neutral, beta-neutral long-short** strategy without using an optimizer (hence "analytic"). It takes top-N longs and bottom-N shorts (e.g. 75 each) based on the alpha signal. It assigns weights by some rule (likely proportional to alpha ranks or z-scores) and then analytically adjusts those weights to neutralize beta and sector exposures. For example, it uses a regression to ensure the portfolio's net beta ≈ 0 and applies the `apply_sector_caps` routine to limit sector imbalances [113] [114]. It then scales the whole portfolio to a target volatility (often a lower target like 10–12% since it's long-short) [115] [116]. The result is a market-neutral portfolio. This script prints a detailed performance summary including portfolio and benchmark (cross-sectional market) returns, as well as the "active" return (long-short difference) [117] [118]. It also reports the portfolio's average beta to ensure neutrality [119]. **Phase 8B** in the docs corresponds to this analytic LS backtester [120].
- **Optimized Long-Short (Phase 8C):** `backtest_academic_strategy_ls_opt.py` uses a **mean–variance optimizer** to determine long-short weights [121] [122]. It sets up a quadratic program to maximize expected return for a given risk (or equivalently minimize variance for given alpha exposures) subject to constraints: net dollar exposure = 0 (fully dollar-neutral long-short), net beta = 0 (beta-neutral) [123] [124], ADV liquidity filter (exclude low-ADV stocks, e.g. ADV ≥ \$2M) [123], and optional sector and single-name weight caps post-optimization [123] [125]. It also vol-targets the portfolio to a given annual volatility (commonly 10% for LS) [126]. The covariance matrix is estimated on the fly from recent returns (e.g. 60-day window with shrinkage) [127]. This approach can better handle weighting when many signals and correlations matter. By default, it might choose more names (e.g. top 150 longs and 150 shorts) to have a broad portfolio [128]. Like risk4, the optimized LS strategy has a variant that **saves the output**: `backtest_academic_strategy_ls_opt_save.py`. The save script (Phase 8C with saving) writes the results to `backtest_results_ls_opt` in DuckDB [129] [130]. This table stores daily `port_ret`, `bench_ret`, and `active_ret` for the optimized long-short portfolio, to be used in regime combination. When running regime-aware backtests, use the "_save" version so that data is persisted [131] [132]. (The non-save version simply prints stats.)
- **Regime-Aware Switching (Phase 8D):** `backtest_regime_switching.py` combines the above engines based on market regime [133] [134]. It pulls in the time series from **Phase 8A (long-only Risk4)** and **Phase 8C (LS optimized)** – which must have been saved as `backtest_results_longonly_r4` and `backtest_results_ls_opt` – along with the `regime_history_academic` (regime labels per date) [135] [136]. The script's logic: on each date, depending on the regime (volatility and trend state), allocate certain weights to the long-only portfolio vs the long-short portfolio [137] [138]. For example, in bullish regimes (e.g. low_vol_bull) it might use 100% of the long-only strategy and 0% of the LS (since being net long is favorable) [139] [140]. In bearish regimes (high_vol_bear) it might do the opposite (0% long-only, 100% LS) [141], and in neutral regimes a blend (e.g. 50/50). The mapping of regime → weights is defined in `regime_to_weights` [139] [140] but can be tuned. The engine then produces a combined equity

curve. It effectively **switches between long-only and market-neutral strategies** to adapt to market conditions. This strategy helps mitigate drawdowns in hostile regimes by leaning short, while capturing more upside in bull regimes [137] [142]. The output of `backtest_regime_switching.py` is typically printed stats (and possibly it could save a table like `backtest_results_regime` if desired, though not explicitly shown).

All backtest scripts take similar arguments: `--db` (DuckDB path), `--alpha-column` (which alpha signal to test), `--target-column` (forward return, default `ret_5d_f`), `--start-date`, `--end-date`, etc., as seen in usage examples [98] [122]. The backtests use `feat_matrix` (or `feat_matrix_v2`) internally to load the alpha, targets, and any needed ancillary columns (vol_blend, adv_20, beta, sector) [143] [144].

Execution Note: The backtest phase is **on-demand** – you run these when you want to evaluate a strategy. They do not run automatically in the daily pipeline (which focuses on data and feature updates). Typically, after updating features and composites, you would run a backtest with the latest signal (for example, using `alpha_composite_v8`) to see its historical performance. Also note which scripts produce persistent outputs: for regime switching, you must run the "save" versions of long-only and LS backtests first so that the tables `backtest_results_longonly_r4` and `backtest_results_ls_opt` exist [145] [135].

Execution Order and Data Refresh Steps

To perform a **full data refresh** or set up the pipeline from scratch (or for a new day's update), follow this step-by-step execution order. Command-line examples are given assuming you are in the project root and using the default database path `data/kairos.duckdb`. (The **conda environment** `kairos-gpu` should be activated, as configured in `setup_cron.sh` [146] [147].)

1. **Initial Data Load (One-Time):** If starting fresh, load historical data into DuckDB:
2. Import the Sharadar `SEP` dataset and fundamentals (e.g. `SF1` or Sharadar "Daily" fundamentals) into DuckDB. This might be done via provided scripts or manual CSV import. Ensure tables `sep`, `tickers`, and `daily` (if used for fundamentals) are created. *(If you have historical CSVs, you can write SQL COPY statements or use a Python loader.)*

3. Confirm `sep` contains pricing (with columns date, close, volume, etc.) and `tickers` has metadata (including category, exchange, marketcap scale). Also load any other needed raw tables (insider trades, etc., as CSVs) for factor scripts.

4. **Universe Definition:** Run **Option B universe script** to create the investable universe and expanded price history:

```
python scripts/create_option_b_universe.py --db data/kairos.duckdb \
    --min-adv 500000 --min-price 2.0 \
    --universe-csv scripts/sep_dataset/feature_sets/option_b_universe.csv
```

This checks for `sep_base` and merges daily data if not present [2], then filters tickers by the criteria. It will log the universe size and create `sep_base_expanded` [6] [7]. It also saves the universe CSV (path as given). **Note:** If you just updated `sep_base` with new data (e.g. via the daily

merge), rerunning this will update `sep_base_expanded` if any new tickers now meet criteria. In practice, universe membership doesn't change too often (maybe quarterly); daily re-run is optional but recommended periodically.

5. **Academic Base Table:** Build the optimized academic price table:

```
python scripts/create_academic_base.py --db data/kairos.duckdb \
    --universe scripts/sep_dataset/feature_sets/option_b_universe.csv
```

This will drop the old `sep_base_academic` and create a new one filtered to the current universe [148] [149]. It's fairly fast (30-80 seconds) and ensures all downstream features use up-to-date price history. **When to re-run:** do this after any update to `sep_base` (new daily data) or a change in the universe membership. For daily refresh, it's wise to run this each time after merging new data so that `sep_base_academic` extends through the latest date.

6. **Update Technical Features:** Run the daily feature generation scripts to incorporate new data:

```
python features/price_action_features.py --db data/kairos.duckdb
python features/trend_features.py --db data/kairos.duckdb
python features/statistical_features.py --db data/kairos.duckdb
python features/volume_volatility_features.py --db data/kairos.duckdb
python features/price_shape_features.py --db data/kairos.duckdb
python features/generate_targets.py --db data/kairos.duckdb
```

In the daily cron, all these are run in sequence [17]. Each script will typically check for the existence of required base tables (e.g. ensure `sep` or `sep_base_academic` is present) and then create or increment their respective `feat_` tables. Running them in this order covers price-based features first, then targets last. These should be executed **every time new price data is added** (daily) to keep features up to date to the most recent date. The scripts are idempotent (they overwrite or update the tables each run).

7. **Build/Update Fundamental Factors:** Depending on data frequency:

8. **Value factors:** Run when new fundamentals are available or if not done yet:

```
python scripts/features/build_value_factors_v2.py --db data/kairos.duckdb
```

This will check the `daily` table for required columns [150] [151] and then create `feat_value_v2` [152]. Do similarly for quality, momentum, insider, etc.:

9. **Quality factors:**

```
python scripts/features/build_quality_factors_v2.py --db data/kairos.duckdb
```

(Assuming such a script exists – it will produce `feat_quality_v2`.)

10. **Momentum factors:**

```
python scripts/features/build_momentum_factors_v2.py --db data/
kairos.duckdb
```

11. **Insider factors:**

```
python scripts/features/build_insider_factors.py --db data/kairos.duckdb
```

12. **Any other factor scripts** (growth, etc.) as needed.

These factor scripts are usually run after each quarter (or whenever new data is fetched). They typically print progress and summary (e.g. number of rows, coverage percentages). If fundamentals haven't changed since last run, you need not rerun them daily. Ensure they are run at least once before building composites.

1. **Compute Composite Signals:** Now build the composite alpha signals that combine the above factors:
2. **Composite Long v2:**

```
python scripts/build_composite_long_v2.py --db data/kairos.duckdb
```

   This reads `feat_composite_long` (v1) plus `feat_value` and `feat_adv` to produce `feat_composite_long_v2` [153] [154].

3. **Primary Cross-Sectional Composite:** Depending on which version you are using as the main alpha:
   - To build **v33_regime** (if not already built via prior script), run the scripts that produce `feat_composite_v31`, `feat_composite_v32b` (if applicable), and then the regime adjustment:
   - There might be a `build_alpha_composite_v32.py` and a `regime_detector_academic.py` to create `regime_history_academic` and then apply it to make `feat_composite_v33_regime`. For example, a composite v6 script internally merged in regime data [155] [156]. If regime detection is separate, run:

     ```
     python scripts/regime_detector_academic.py --db data/kairos.duckdb
     ```

     followed by a script to apply regime multipliers (e.g. `build_alpha_composite_v33_regime.py`). Ensure `backtest_results_longonly_r4` exists if regime detection requires it for trend (some regime definitions might use market returns).
   - **Composite v7:** If you want the 50/50 blend of v33 and quality (alpha_composite_v7):

     ```
     python scripts/build_alpha_composite_v7.py --db data/kairos.duckdb
     ```

(The script likely automatically picks up `feat_composite_v33_regime` and `feat_quality_v2` and blends them.)

- **Composite v8:** For the latest composite:

```
python scripts/build_alpha_composite_v8.py --db data/kairos.duckdb
```

This will output `feat_composite_v8` and also update the feature matrix with a new column `alpha_composite_v8` if the matrix exists [76] [157]. It prints the expected performance and a sample backtest command at the end [158].

4. If you have a different weighting idea, adjust the script arguments (v8 allows `--v33-weight`, etc.).

When to rerun composites: whenever underlying factors are updated or if weights/formulas change. Typically after updating factor tables quarterly, rebuild the composite so it incorporates the latest data.

1. **Assemble/Update Feature Matrix:** Now that all feature tables (technical, fundamental, composites, targets) are ready, create the unified matrix:

```
python scripts/build_full_feature_matrix.py \
    --db data/kairos.duckdb \
    --date 2025-12-21 \
    --universe scripts/sep_dataset/feature_sets/option_b_universe.csv
```

Use today's date (or last trading date) for the `--date` to timestamp the output directory. This will join all `feat_` tables and produce the `feat_matrix` table [86]. It also writes out `scripts/feature_matrices/<date>_academic_matrix/` as Parquet files [91]. You'll see logs of each table being joined and the final row×column count [159]. If you have an existing matrix and only want to append new dates, you could alternatively load `feat_matrix` and just union the new day's features, but the provided method is a clean rebuild (dropping the old table each time). Re-run this after any new feature or composite is added or when new dates arrive.

2. **Run Backtests:** With the feature matrix and signals ready, you can simulate strategies. The exact sequence depends on which strategy you want to evaluate:

3. **Simple single-strategy backtest:** For example, test the latest alpha composite in a risk-controlled long-only manner:

```
python scripts/backtesting/backtest_academic_strategy_risk4.py \
    --db data/kairos.duckdb \
    --alpha-column alpha_composite_v8 \
    --target-column ret_5d_f \
    --top-n 75 --rebalance-every 5 --target-vol 0.20 \
    --start-date 2015-01-01 --end-date 2025-12-18
```

This will simulate Risk Model v4 on the composite_v8 signal over 2015–2025 and print a summary [158]. Adjust dates and parameters as needed.

4. **Long-short backtest:** To test the alpha in a market-neutral setting, run:

```
python scripts/backtesting/backtest_academic_strategy_ls_opt.py \
    --db data/kairos.duckdb \
    --alpha-column alpha_composite_v8 \
    --beta-column beta_252d \
    --target-column ret_5d_f \
    --top-n-long 150 --top-n-short 150 \
    --rebalance-every 5 --target-vol 0.10 \
    --start-date 2015-01-01 --end-date 2025-12-18
```

This uses the optimized LS engine (dollar & beta neutral, 150 longs, 150 shorts, 10% vol target) [122]. It will log the progress and final performance.

5. **Regime combination:** For the 8D strategy, first ensure you have run the "save" versions of needed scripts:

```
python scripts/backtesting/backtest_academic_strategy_risk4_save.py \
    --db data/kairos.duckdb \
    --alpha-column alpha_composite_v8 --target-column ret_5d_f \
    --top-n 75 --rebalance-every 5 --target-vol 0.20
python scripts/backtesting/backtest_academic_strategy_ls_opt_save.py \
    --db data/kairos.duckdb \
    --alpha-column alpha_composite_v8 --beta-column beta_252d \
    --target-column ret_5d_f \
    --top-n-long 150 --top-n-short 150 --rebalance-every 5 --target-vol 0.
10
```

These will produce `backtest_results_longonly_r4` and `backtest_results_ls_opt` tables [112] [131]. Then run:

```
python scripts/backtesting/backtest_regime_switching.py \
    --db data/kairos.duckdb \
    --start-date 2015-01-01 --end-date 2025-12-18
```

The regime switching script will automatically load `regime_history_academic`, and the two results tables [160] [135], then output the combined performance. It doesn't need the alpha column specified because it uses the pre-run portfolios.

6. **Other variants:** You can also run `backtest_academic_strategy.py` (baseline) for a quick sanity check, or `backtest_academic_strategy_risk.py` (v1) to see the incremental effect of risk controls.

Re-running on Data Update: For a **daily update cycle**, steps 2–4 and 7 are typically automated. In the provided cron, after downloading and merging data (steps 1–2), it goes straight to regenerating features (step 4) [1] [17] – it does **not** rebuild the universe or academic base daily, presumably to save time. In a production daily run: - You would **merge new data**, - (Optionally update `sep_base_academic` if you want the absolutely latest base; if not, the daily features can potentially operate directly on `sep_base` for just the last day's metrics), - Run the feature scripts to append new metrics, - Potentially run a lightweight matrix update (e.g. appending the new day to `feat_matrix`), - Then maybe run a daily scoring using the latest composite (rather than a full backtest).

However, whenever the **universe composition changes or periodically (e.g. weekly)**, it's advisable to re-run steps 2 and 3 to keep `sep_base_academic` in sync and then rebuild everything downstream (factors, composites, matrix). In summary, after any update to **base data** (prices or fundamentals), re-run: (a) academic base, (b) any features/factors derived from that data, (c) composite if needed, (d) matrix assembly. Then use the updated `feat_matrix` for backtests or live predictions.

Backtest Engine Overview and Outputs

The Kairos backtesting framework provides multiple scripts to simulate strategies. Here's an overview of each and how they differ, including their inputs and outputs:

- **Academic Baseline Backtest (`backtest_academic_strategy.py`):** A simple equal-weight strategy for evaluating raw signals. **Inputs:** a chosen `alpha_column` from `feat_matrix` (e.g. a single technical indicator or composite) and a `target_column` (forward return, usually `ret_5d_f`) [93]. **Operation:** On each rebalance date, ranks all stocks by alpha and either: goes long top-N (if long-only) or long top-N and short bottom-N (if `--long-short` mode is indicated; this script might only handle long-only unless modified). It then calculates returns vs an equal-weight benchmark (which is effectively the average return of all stocks each period) [94] [95]. **Outputs:** This script prints the performance statistics to console (total return, annual return, vol, Sharpe, max drawdown) [93]. It does **not** create any database tables. Use this to gauge signal quality without any complex overlays.

- **Risk-Controlled Long-Only Backtests (Risk v1–v4 scripts):** These all require the **feature matrix** as input and pull various columns from it. Common inputs: `alpha_column` (e.g. `alpha_composite_v31` or any composite) and `ret_5d_f` as target, plus internally they use `vol_blend`, `adv_20`, `size_z`, and sector data [23] [161]. They differ as follows:

- **Risk v1 (`backtest_academic_strategy_risk.py`):** Implements cross-sectional z-score ranking and vol targeting [97]. It uses only `feat_matrix` (no external data needed except what's in matrix). **Output:** prints stats. No DB table saved.
- **Risk v2 (`backtest_academic_strategy_risk2.py`):** Builds on v1 by adding per-stock weight caps and sector caps. It joins `tickers` to get sector info [162] [99]. **Output:** prints stats, no table saved.
- **Risk v3 (`backtest_academic_strategy_risk3.py`):** Requires additional features in matrix: `vol_blend` (from `feat_vol_sizing`), and `adv_20` & `size_z` (from `feat_adv`), to filter and weight by volatility [61] [103]. It also needs sector from `tickers`. **Output:** prints stats, no table.

- **Risk v4 (`backtest_academic_strategy_risk4.py`):** Uses all of the above plus keeps track of previous weights for turnover control [106]. It requires the same inputs as v3 (vol, adv, size, sector). **Output:** prints stats. However, for regime combination, you will use `backtest_academic_strategy_risk4_save.py` which outputs a table:
    - `backtest_results_longonly_r4`**:** This DuckDB table is created by the save script [112]. It typically has columns: `date`, `port_ret` (the long-only portfolio's return that period), `bench_ret` (benchmark return, e.g. avg market return), and `active_ret` (the difference) [135] [163]. The save script registers the in-memory DataFrame as `bt_r4` and then creates the table from it (as seen around the drop/create commands) [112]. This table will have one row per rebalance period (e.g. each week) from the start to end date. It's **used as input** for `backtest_regime_switching.py` (which expects `backtest_results_longonly_r4` to exist) [164].
- **Analytic Long-Short Backtest (`backtest_academic_strategy_ls.py`): Inputs:** an `alpha_column`, a `beta_column` (for each stock, e.g. `beta_252d` from `feat_beta`), and `ret_5d_f` target, plus it uses `vol_blend` and `adv_20` from the matrix internally for weighting and filtering [165]. It also likely uses sector data for caps. **Outputs:** prints an extensive summary including active return stats and portfolio beta stats [117] [119]. No DB table output by default (the code does not indicate saving). This is mainly for analysis and is not needed for regime switching (since regime switching uses the optimized LS results instead).
- **Optimized Long-Short Backtest (`backtest_academic_strategy_ls_opt.py`): Inputs:** similar to analytic LS – needs `alpha_column`, `beta_column`, `ret_5d_f`, and uses `adv_20` from matrix as a filter. It also directly queries `sep_base_academic` within the script to compute the covariance (so it needs read access to daily returns) [127] [124]. **Outputs:** the non-save version prints performance stats (likely including portfolio and benchmark Sharpe). The **save version** `*_ls_opt_save.py` creates a DuckDB table:
- `backtest_results_ls_opt`**:** contains the optimized long-short strategy's returns time series [136] [166]. This table has `date, port_ret, bench_ret, active_ret` columns (with naming similar to the long-only results). It's produced by dropping any old table and then creating a new one from the results DataFrame [167] [168]. This is consumed by regime switching.
- **Regime Switching Backtest (`backtest_regime_switching.py`): Inputs:** it **requires** three tables to exist in DuckDB: `regime_history_academic` (with columns date and regime), `backtest_results_longonly_r4`, and `backtest_results_ls_opt` [160] [135]. It loads these into DataFrames [160] [135], then merges by date to align them [169] [145]. The regime table usually has columns `date, vol_regime, trend_regime, regime` indicating categorical regime labels for each day. The script uses the combined daily returns to compute the blended portfolio returns according to the regime weighting scheme (the `regime_to_weights` mapping) [139] [141]. **Outputs:** by default, it prints the final performance metrics of the blended portfolio (Sharpe, drawdowns, etc.). If needed, one could modify it to save a `backtest_results_regime` table, but by design it might be analysis-only. The **key output** to the user is understanding how the combination performed vs the components.

Comparing the Backtest Scripts:

- **Risk4 vs LS Opt vs Regime:** `backtest_academic_strategy_risk4.py` (Phase 8A) produces a long-only strategy which typically has higher beta and will capture market upside with controlled risk (vol targeting, sector caps) [102] [104]. `backtest_academic_strategy_ls_opt.py` (Phase 8C) produces a market-neutral strategy which aims for absolute returns independent of market direction (zero beta, often lower volatility target) [123] [128]. The **regime switching** combines these: in bullish periods, it heavily weights the long-only (to get market exposure plus alpha), in bearish periods it leans on the hedged long-short (to preserve capital or profit from shorts) [139] [141]. Thus, regime switching is like an "adaptive" strategy that picks between the two engines each period.

- `risk4.py` **vs** `risk4_save.py`**:** Both simulate the same strategy. The difference is just in output: the `_save.py` version registers the results into DuckDB as `backtest_results_longonly_r4` [112] for use by other scripts, whereas the plain `risk4` script only prints performance. Similarly for LS opt, `_save.py` writes the table `backtest_results_ls_opt` [131]. Use the save versions when you plan to do multi-stage strategies (like regime switching or any analysis that needs the time series of returns).

- `ls.py` **(analytic) vs** `ls_opt.py`**:** Both target a similar outcome (dollar-neutral, beta-neutral portfolio), but the approach differs:

- The **analytic LS** script uses heuristic weighting (proportional to alpha z-scores, then manually adjusted to neutralize beta and capped by sector) [113] [114]. This is simpler to interpret and faster (no solver), but might not fully optimize the Sharpe.
- The **optimized LS** uses a quadratic optimizer to maximize the risk-adjusted return given the alpha forecasts and estimated covariance [121] [170]. It can consider correlations between stocks to decide weights that maximize the portfolio's predicted Sharpe. This often yields a more efficient portfolio with potentially higher Sharpe, at the cost of requiring a solver and being a bit slower. It also allows more precise constraint enforcement (e.g. exact beta neutrality).

- Both are beta-neutral; both are vol-targeted. The main difference is if you prefer a transparent approach (analytic) or a potentially more optimal but opaque approach (optimized). The project includes both for comparison.

- **Academic Baseline vs Risk Models:** The baseline (no risk controls) will typically have higher volatility and drawdowns and might not be practical, but it establishes a benchmark for pure alpha quality. The risk-controlled models (v1–v4) show how performance improves or trade-offs as you add realistic constraints (vol targeting often reduces drawdown, sector caps reduce concentration risk, turnover control improves consistency at the cost of slight alpha decay). For example, Risk v4 is more constrained than v2, but likely has smoother equity curve. The documentation in the scripts highlights these differences: e.g. risk4 adds turnover smoothing vs risk3 [107].

In summary, the backtest suite allows you to test from a basic idea (did my factor make money assuming equal weighting?) up to a fully-fledged strategy (net of transaction constraints and adaptable to regimes). Each script outputs either console summaries or creates a results table, so you should plan which outputs you need. For instance, if you want to plot the long-short equity curve or do further analysis, you'd run the `_save` version and then query `backtest_results_ls_opt` in DuckDB (or export it).

Outputs for Live Prediction and Production Use

Several outputs of the pipeline are intended to feed into a **live trading or prediction system** once the research transitions to production. Key among these are:

- **Feature Matrix / Design Matrix (`feat_matrix` or `feat_matrix_v2`):** This table is essentially the model-ready dataset containing all predictive features (technical indicators, fundamental z-scores, etc.) and the target. In a live prediction context, you would compute the latest row (or the latest few rows) for each ticker using the same feature generation logic. The **final day's features** can be pulled from DuckDB to make a prediction about next week's return. For production, one could connect the daily update pipeline to produce an updated `feat_matrix` row for each stock by market open. The matrix in DuckDB can also serve as a historical store for retraining models or recalibrating signals.

- **Alpha Composite Signals:** Ultimately, the strategy will rely on one or more composite alpha signals for trading decisions. In the current design, these are columns like `alpha_composite_v33_regime`, `alpha_composite_v7`, `alpha_composite_v8` in the feature matrix. For example, `alpha_composite_v8` (which blends regime, quality, value) is a strong candidate for a production alpha. In live use, after each daily feature update, you would compute the **latest value of** `alpha_composite_v8` **for every stock** and rank accordingly. The pipeline already sets this up: the composite scripts update the `feat_composite_v8` table and even propagate into the matrix [75] [76], so the most recent date's `alpha_composite_v8` is ready in `feat_matrix` for decision-making.

- **Risk Features for Live Allocation:** In production, the portfolio construction would need volatility (`vol_blend`), liquidity (`adv_20`), beta (`beta_252d`), and sector data at hand for each stock to apply position sizing and constraints (like the Risk4 methodology). These are all provided in the feature matrix as well (columns like `vol_blend`, `adv_20`, `size_z`, `beta_*`). The pipeline ensures those are updated (vol and beta features daily from price moves; adv_20 daily from volume).

- **Target Columns (`ret_5d_f` etc.):** These are not used in live trading (since they are forward-looking actual returns), but are crucial for model validation and retraining. However, one could use them to label data for an ML model that might be retrained on the rolling history. In production, you wouldn't have `ret_5d_f` for the latest day of course, but historically it's used to measure performance.

- **Universe CSV:** The Option B universe list (CSV file) is a static output that can be used in production to know which symbols to consider. This could feed into data subscription or downstream systems to limit scope to those tickers. If the universe is updated periodically, the new CSV can be loaded.

- **Backtest Results:** While primarily for research, some results could inform production. For instance, the regime classification (the latest regime from `regime_history_academic`) could be used in a live strategy to decide whether to allocate to a hedged stance. Also, understanding recent performance of long-only vs long-short (from backtest results) might influence tactical decisions. Typically though, **production system** would implement the logic derived from backtests (e.g. "if regime = X, then use 70% long-only and 30% long-short positions").

In practical terms, the output tables **designed for live use** are those in the **features and composite categories**: - `feat_matrix` **(or a real-time slice of it):** Feeds into the model/scoring engine. - `feat_composite_vXX` **tables and the values therein:** Are the actual alpha signals to act on. - `feat_vol_sizing`, `feat_beta`, `feat_adv`**:** Provide risk metrics for position sizing (e.g. use latest `vol_blend` for each stock to scale weights, ensure not to trade stocks with ADV below threshold, use `beta` to maintain neutrality in LS portfolio). - `regime_history_academic`**:** Its latest entry (e.g. regime label for today's date) could be used to decide strategy tilt (the regime detector can likely be run up to T-1 with known data). - The **Parquet feature matrix export** is useful if the live prediction will be done by a separate ML model or if you integrate with a pipeline outside DuckDB (e.g. a Python ML service can load the latest Parquet partition for each ticker and make predictions).

In short, the pipeline readies a daily **"feature snapshot"** which can be used to rank stocks by the chosen composite signal and generate buy/sell lists. The heavy lifting of calculating factors (momentum, value, quality, etc.) is all done ahead of time so that by the time the market opens, you have, for example, each stock's alpha score (`alpha_composite_v8`) and risk characteristics (volatility, beta) available to construct the day's portfolio.

Common Routines, Command Snippets, and Environment Tips

Over the course of using the Kairos pipeline, certain command-line routines and DuckDB usage patterns have proven useful. Below are some tips and snippets, including those gleaned from the user's bash history:

- **Conda Environment Setup:** Ensure you activate the correct environment before running anything. For example:

  ```
  source ~/miniconda3/etc/profile.d/conda.sh
  conda activate kairos-gpu
  ```

  This environment should contain `duckdb`, `pandas`, `numpy`, etc., as required by the scripts. The cron job configuration explicitly activates `kairos-gpu` before execution [146] [147].

- **Running Scripts with Logging:** Many scripts use Python's logging or print outputs. It's often useful to **tee** the output to a log file. For instance, the `daily_update_pipeline.sh` does:

  ```
  python scripts/daily_download.py --date 2025-12-20 2>&1 | tee -a logs/
  daily_update_20251221.log
  ```

  and similarly for each step [1] [171]. Adopting this style helps debug if something failed overnight.

- **DuckDB Command-Line and Inspection:** You can open DuckDB to inspect tables:

  ```
  duckdb data/kairos.duckdb
  ```

Then inside DuckDB, try commands like:

```
SHOW TABLES;
PRAGMA table_info('feat_matrix');
SELECT COUNT(*) FROM feat_matrix;
```

This will list all tables, show columns of `feat_matrix` (so you can see which features are present), and count rows [172] [159]. The history suggests frequent use of such queries to verify data loads (scripts themselves do it too, e.g. value_factors_v2 prints row counts [173]).

- **Viewing Sample Data:** To quickly peek at some feature values, you can do:

```
SELECT * FROM feat_composite_v8 WHERE ticker='AAPL' ORDER BY date DESC
LIMIT 5;
```

Or within Python, use pandas via DuckDB:

```
import duckdb
con = duckdb.connect("data/kairos.duckdb")
df = con.execute("SELECT date, alpha_composite_v8, quality_z, value_z FROM
feat_matrix WHERE ticker='AAPL' ORDER BY date DESC LIMIT 5").fetchdf()
print(df)
```

This can help ensure the latest features are populated and make sense.

- **Frequently Used Composite Versions:** The user often references **v33_regime**, **v7**, and **v8** composites. Knowing their context:

- *v33_regime* is the primary cross-sectional composite with regime timing. Often used as a baseline in backtests (e.g. earlier tests might have used `alpha_composite_v33_regime` as the alpha input).
- *v7* (50/50 blend of v33 and quality) and *v8* (v33+quality+value) are improved signals. These were likely run and tested multiple times. When working interactively, remember to specify the correct column name in backtests or queries: e.g. `alpha_composite_v8` is the column in both `feat_composite_v8` and `feat_matrix` after update. If you forget to update the matrix, you can still join at query time, but the composite script took care of it [75] [76].

- If you maintain multiple versions, it's helpful to keep track in the matrix of all of them for comparison. For example, the feature matrix might have `alpha_composite_v31`, `alpha_composite_v33_regime`, `alpha_composite_v7`, `alpha_composite_v8` all as separate columns. You could run backtests on each to compare performance. The bash history likely includes running backtests on v7 and v8 to validate their Sharpe improvements.

- **DuckDB Performance and Memory:** DuckDB can handle the ~10M×(hundreds of columns) feature matrix in memory if you have sufficient RAM. If you encounter memory issues joining too many

columns, you can drop some irrelevant ones or limit the date range. The `build_full_feature_matrix.py` already partitions work inside the database, which is efficient. If you manually experiment with massive joins, consider the iterative approach used in that script (joining one table at a time) to avoid Cartesian explosion in memory.

- **Partial Pipeline Runs:** You don't always need to run everything. For example, if you adjust just the weighting in composite v8, you can just run `build_alpha_composite_v8.py` and then immediately do a backtest, since all underlying data is already there. It will update the composite table and the matrix column quickly [76] [157]. Similarly, if only price data updated daily and no change in fundamentals or weights, you might skip directly to generating technical features (step 4) and matrix assembly (step 7) on a given day.

- **Troubleshooting Tips:** The scripts include many checks that will throw errors if something is missing. For instance, if you forget to run `merge_daily_download_duck.py` and try to build the universe, it will error "sep_base table not found" [2]. Or if you run a factor script before creating `daily` fundamentals table, it will complain missing columns [152] [151]. Pay attention to these messages – they guide you to which prerequisite step was skipped. The logs and exceptions are meant to be informative (often with or ⚠ symbols in the code).

- **DuckDB SQL snippets:** Some useful snippets seen in code:

- Finding distinct trading dates:

```
SELECT DISTINCT date FROM sep_base_academic ORDER BY date;
```

(Used in expanding SF1 data to daily in quality factors [45] [174].)

- Computing rolling stats in SQL (as in momentum factors):

```
LAG(closeadj, 252) OVER (PARTITION BY ticker ORDER BY date) AS
price_12m_ago
```

This is how lagged features were computed [175].

- Updating or adding new columns to the feature matrix: The composite v8 script demonstrates adding a new column if it doesn't exist, using `ALTER TABLE ... ADD COLUMN` and then an `UPDATE ... FROM ...` join to fill values [176] [76]. This is handy if you want to avoid full rebuild and just enrich the matrix.

- **Favorite Investigative Queries:** The user likely ran quick analyses like:

```
SELECT alpha_composite_v8, ret_5d_f
FROM feat_matrix
WHERE date = '2025-12-18'
ORDER BY alpha_composite_v8 DESC LIMIT 10;
```

to see top 10 picks and maybe their subsequent return, just to eyeball if the signal is working. Or computing IC:

```
SELECT corr(alpha_composite_v8, ret_5d_f)
FROM feat_matrix
WHERE date BETWEEN '2020-01-01' AND '2025-01-01';
```

(This overall IC should match expectations ~0.01–0.02 for a good signal.)

In essence, routine use of the pipeline involves a lot of **small command-line invocations and SQL queries** to verify each step's output. The project is set up so that each script can be run independently, which aligns well with a modular workflow in bash (run one, check output, run next). The bash history likely contains sequences of running a script, then using DuckDB CLI or `head -n` on output CSVs to ensure they look right. Emulating that habit is beneficial.

Modularity and Consolidation Suggestions

While the pipeline is comprehensive, there are opportunities to DRY (Don't Repeat Yourself) and streamline maintenance:

- **Factor Script Unification:** Many factor builders (value, momentum, quality, etc.) share a common pattern: load a base table, apply some filtering, compute some ratios or rolling metrics, then output a `feat_` table with both raw components and a composite z-score. These could be refactored into a single configurable script or function. For example, `build_value_factors_v2.py` and `build_quality_factors_v2.py` both do forward-fill of quarterly data and compute z-scores [47] [177]. A unified "factor_builder" could accept parameters for which base table and which formula to apply, reducing code duplication. Additionally, the z-scoring logic (`zscore_winsorize`) is re-defined in multiple files [21] [178] – this could be placed in a shared utility module (e.g. `factor_utils.py`) that all factor scripts import, ensuring consistency in winsorization and z-scoring across factors.

- **Composite Builder Generalization:** The project currently has separate scripts for each composite version (v5, v7, v8, etc.), which largely perform merges of certain factor tables followed by weighted averaging [78] [179]. Instead, consider a single **config-driven composite script** where you can pass the desired factor weights. For instance, a YAML or JSON config could specify the factors to include and their weights (including regime multipliers), and the script reads that and constructs the composite accordingly. This would prevent the proliferation of scripts for each new idea (v9, v10 could be done by editing config rather than writing new code). It would also ensure any improvements (like better handling of missing data or automatic normalization of weights) apply to all composites uniformly. If separate scripts are still preferred for clarity, at least a shared function could be used to do the merging and weighting (since `build_alpha_composite_v7.py` and `v8.py` are very similar in structure).

- **Backtest Code Refactoring:** There is significant overlap between the risk-controlled long-only scripts (v1–v4) – they differ by a few steps (e.g. whether to filter by ADV, whether to adjust weights by vol, whether to do turnover smoothing). Similarly, the LS analytic vs LS optimized share conceptual

steps (one manually neutralizes, the other solves for neutrality). A single backtest engine class could be created in Python that takes parameters like `long_only=True/False, use_optimizer=True/False, sector_cap=X, turnover_cap=Y` etc., to cover all cases. Then each script could just instantiate with the appropriate settings. This would reduce maintenance (ensuring, for example, that if you fix a bug in how Sharpe is calculated, it fixes across all variants). Currently, one can spot minor differences: e.g. `annualize` functions and max drawdown computations appear in many scripts with slight variations [180] [181]. Centralizing these in a utility (say `performance_metrics.py`) would avoid inconsistency. Furthermore, the **result saving** could be parameterized (instead of having separate save scripts, the main script could accept a `--save-to` flag or similar to decide whether to output a table).

- **Eliminate Hardcoded Specifics:** Some scripts reference specific table names or versions in code. For instance, `feat_matrix_v2` vs `feat_matrix` – eventually it might be better to standardize on one. The `build_full_feature_matrix.py` uses a hardcoded list of table names to join [84], which means whenever a new feature table is added (like `feat_composite_v8`), one must edit that list. This could be made more dynamic: for example, querying DuckDB for all tables starting with `feat_` and filtering out those that shouldn't be included. Or maintain the list in a config file rather than code. A more modular approach would allow the pipeline to automatically include new factors without modifying the join code.

- **Reusable Validation Logic:** Many scripts perform similar validation at start (check if required tables/columns exist, print counts). This can be factored into a shared function. For example, a `ensure_table_has_columns(con, table, columns)` utility could replace repetitive blocks like the one in value_factors_v2 [182] [151]. This would produce consistent error messages and shorten the scripts.

- **Combining Universe and Base Steps:** The division between `create_option_b_universe.py` and `create_academic_base.py` is logical (one for tickers, one for price history), but they could be combined or at least automated sequentially. Perhaps a single script `initialize_universe_and_base.sh` could call both in order, since one must always run the universe builder then the academic base builder. This would reduce the chance of user forgetting one of the two steps.

- **Parameterize Universe Criteria:** Currently the universe criteria (min price $2, ADV $500k by default) are hardcoded defaults in `create_option_b_universe.py` [8]. For flexibility, one might allow adjusting these thresholds (e.g. to test a microcap universe with lower ADV). This is minor, but it improves the modularity of the universe selection.

- **Single Pipeline Entry Point:** To streamline operations, consider creating a **master pipeline script** (could be a Python or bash script) that ties everything together. For instance, `run_full_refresh.sh` that executes all steps in the correct order (with error checking). This would reduce manual steps and ensure nothing is missed. The cron script is a mini version of this for daily update, but a more comprehensive one for a full rebuild would be useful for onboarding.

- **Documentation and Config**: Finally, integrating the pipeline with a configuration file for paths, dates, thresholds can make it more maintainable. E.g., define in one place the start date for

backtests, the target vol levels, etc., so that if strategy parameters change, you don't have to edit multiple scripts.

Adopting these modularity improvements will make the project easier to extend. For example, if a **new factor** comes in (say a sentiment factor), you would just write a new `build_sentiment_factor.py`, add its output to a config, and the matrix build and composites could pick it up without further code changes. Similarly, if trying a **new strategy variant** (maybe a different turnover setting or a sector-neutral long-short), a flexible backtest engine class could support it via parameters rather than copying yet another script.

Overall, Kairos is already structured in logical phases; by refactoring common routines and parameterizing where appropriate, one can avoid redundancy and reduce the risk of inconsistencies. This means less time adjusting boilerplate and more time focusing on research improvements.

---

1 3 17 171 daily_update_pipeline.sh
https://github.com/vjeromel276/kairos-modeling/blob/f63570c06f21b069c65b44716b4e42e5de9a7e06/daily_update_pipeline.sh

2 4 5 6 7 8 9 create_option_b_universe.py
file://file-LhtjNpysreYNmjcp4U2y1W

10 11 12 13 14 15 16 148 149 create_academic_base.py
file://file-XjWWScQjTq12jkyf3XSyXf

18 19 20 21 48 49 175 build_momentum_factors_v2.py
file://file-2bdQG5XAgPUpwLqPcjs1jD

22 23 25 26 52 61 62 101 102 103 104 105 143 144 161 backtest_academic_strategy_risk3.py
file://file-PHzRhnn2MQdz3nzQSRMNjo

24 36 60 80 81 82 83 84 85 86 90 91 92 159 172 build_full_feature_matrix.py
file://file-UjYLjC1EX2Q6m7KxH56AJN

27 28 29 30 vol_sizing_features.py
file://file-MZeTeUaNKAZMmoDnV2NeAK

31 32 33 beta_features.py
file://file-GbQ6C6REJaAyVwtqUbGTyE

34 35 generate_targets.py
file://file-AEHiRZhLSBc7GTqfywkkSQ

37 38 39 40 41 42 43 44 150 151 152 173 182 build_value_factors_v2.py
file://file-3Tp3zJmrituNgDZyiVGbrQ

45 46 47 174 177 build_quality_factors_v2.py
file://file-3rmSX21HUWrdEC7mrTtkoN

50 51 build_insider_factors.py
file://file-TUf85ZGKKVYEuScTUrrWnM

53 54 55 56 57 58 59 153 154 build_composite_long_v2.py
file://file-1Dt1uiFLvepcSKkbQ1TtTM

63 64 155 156 build_alpha_composite_v6.py
file://file-ArBiaxW7bJVjMJedWWpRfR

65 66 67 78 79 178 179 build_alpha_composite_v5.py
file://file-EBzwfMiARRoar9HQ5GLcyC

68 69 build_alpha_composite_v7.py
file://file-63fY3sRJpkpgaxBXVSEGGK

70 71 72 73 74 75 76 77 157 158 176 build_alpha_composite_v8.py
file://file-9P4qSifp3bYq2H6NNfibJw

87 88 89 build_feature_matrix_v2.py
file://file-XFLfZ8UbPGsE56QCm8xoED

93 94 95 96 backtest_academic_strategy.py
file://file-WhjpSa4vvj1NnKqqCr9kXS

97 98 162 180 181 backtest_academic_strategy_risk.py
file://file-J2xP5Ujww7P3W9154a1zZT

99 100 backtest_academic_strategy_risk2.py
file://file-NUDWpKN9E2UMcpSrSB2sV6

106 107 108 109 backtest_academic_strategy_risk4.py
file://file-94SzaZ4S7GnfgoX2iRFa3L

110 111 112 backtest_academic_strategy_risk4_save.py
file://file-EG998qvwtuwLvHCEKGQFFe

113 114 115 116 117 118 119 120 165 backtest_academic_strategy_ls.py
file://file-AR61s6cxrE34ygFdUXNwBA

121 122 123 124 125 126 127 128 170 backtest_academic_strategy_ls_opt.py
file://file-Lzu5E6QsSNgbb1T2zcAUhL

129 130 131 132 167 168 backtest_academic_strategy_ls_opt_save.py
file://file-Hj5NzgMGEYRiZxapFdSa7B

133 134 135 136 137 138 139 140 141 142 145 160 163 164 166 169 backtest_regime_switching.py
file://file-8oL1cgxuT2iSi55845ya68

146 147 setup_cron.sh
https://github.com/vjeromel276/kairos-modeling/blob/f63570c06f21b069c65b44716b4e42e5de9a7e06/setup_cron.sh