APForecast – Detailed Code Documentation

This document provides a comprehensive overview of the APForecast including all mathematical, theoretical, and logical details. Assumptions and key design decisions are explained, and the statistical methods behind the forecasting are detailed.

File: `src/apforecast/core/constants.py`

- **Purpose:** Defines all the constant values, configuration paths, column names, and fixed parameters used throughout the application.
- **Key Contents:**
    - **Directory paths:** Constants like `DATA_DIR`, `RAW_DIR`, etc., specify where data and configurations are stored. For example, `DATA_DIR = "data"` is the base folder for input/output data, and `MASTER_LEDGER_PATH = f"{PROCESSED_DIR}/master_ledger.parquet"` is where the reconciled master ledger is saved.
    - **Column name mapping:** `COLUMN_MAP` is a dictionary mapping user file headers to the system's internal column names. For example, it might map `"Check #"` or `"Reference"` (user-provided headers) to `"Check_ID"` or `"Vendor_ID"`. This ensures that regardless of input file header variations, the code can unify them.
    - **Internal column names:** Constants like `COL_CHECK_ID`, `COL_VENDOR_ID`, `COL_AMOUNT`, etc., define standardized names (e.g. `"Check_ID"`) used everywhere. This avoids hard-coding strings and prevents typos.
    - **Statuses and cohorts:** It defines status labels (`STATUS_OPEN = "OPEN"`, `STATUS_CLEARED = "CLEARED"`, etc.) and cohort thresholds. For example, `THRESHOLD_SMALL = 10000`, `COHORT_SMALL = "STABLE_SMALL"`, etc. Checks are bucketed into small, medium, or large cohorts based on amount. These cohorts are used for building global models when vendor-specific history is insufficient.
- **Assumptions:** The constants assume certain business logic. For instance, any check older than 45 days is often flagged (as seen in backtesting) – this threshold is implied in

code comments. The cohort thresholds (10k and 50k) are chosen to group vendors by check size, which is an assumption of segmentation strategy.

File: `src/apforecast/ingestion/loader.py`

- **Purpose:** Loads raw data files (CSV or Excel) and standardizes column names.
- **Main Function:** `load_file_smart(filepath)`
  - **Behavior:** Checks if the file exists and then reads it (using `pd.read_csv` or `pd.read_excel` depending on extension). It then **renames columns** using the mapping in `COLUMN_MAP` from `constants.py`. For example, if the raw file has a column `"Vendor Name"`, the loader renames it to `"Vendor_ID"` if so configured. After renaming, it strips whitespace from column names to avoid hidden errors.
  - **Assumptions:** It assumes the user has updated `COLUMN_MAP` to include all variations of their file headers. If critical columns are missing (like no `"Vendor_ID"` after renaming), other parts of the code will handle that (e.g., assigning `"Unknown_Vendor"`).
  - **Note:** This function does not transform data further; it only ensures the data frame has consistent column names for later processing.

File: `src/apforecast/ingestion/reconciler.py`

- **Purpose:** Reconciles daily inputs with historical data to build and maintain a **Master Ledger** of all checks (cleared and open).
- **Main Function:** `ingest_and_reconcile(date_str, run_date)`
  1. **Load Master Ledger:**
     - If a `MASTER_LEDGER_PATH` file exists (a Parquet file), it loads it as the current ledger. If not, it tries to initialize from any file in `data/raw/history/`. The first history file found is loaded via `load_file_smart`.

    - **Initialization:** The initial ledger must contain columns like `"Check_ID"`, `"Vendor_ID"`, `"Post_Date"`, `"Clear_Date"`. The code converts post/clear dates to datetime and computes `Days_to_Settle` (difference between clear and post date) and sets `Status = "CLEARED"` for those entries. This forms the starting historical data.

2. **Load Daily Inputs:**
    - It looks in `data/raw/{date_str}/` for files. It expects an **Outstanding** file (uncleared checks) and optionally a **Cleared** or **bank** file. If found, it loads them.
    - The Outstanding file contains checks that have been issued but not yet cleared by the bank as of the run date. The Cleared file has checks cleared in bank records on that date.

3. **Reconciliation:**
    - **Marking cleared:** If a cleared file with check IDs is provided, the ledger updates any matching `Check_ID` entries to `Status = "CLEARED"`, updates their `Clear_Date`, and recalculates `Days_to_Settle = Clear_Date - Post_Date`. This ensures the master ledger knows which past checks have now cleared.
    - **Adding new opens:** From the outstanding (issued) checks file, it identifies any new `Check_ID` not in the ledger. For those, it adds them with `Status = "OPEN"`, sets their `Post_Date`, and leaves `Days_to_Settle = None` since they haven't cleared yet.

4. **Save Master Ledger:** The updated ledger (with new clears and opens) is saved to `MASTER_LEDGER_PATH` (in Parquet format).

- **Assumptions and Logic:** This function assumes a strict sequence: historical clears are loaded first, then daily outstanding and cleared lists. It also assumes that checks are uniquely identified by `Check_ID`. It removes duplicates by checking existing IDs. It enforces causality by only updating clear dates for existing checks. The concept of a "master ledger" ensures the model always trains on all past cleared checks and knows the current open checks.

File: `src/apforecast/modeling/cohorts.py`

- **Purpose:** Assigns checks to size-based cohorts for modeling purposes.
- **Main Function:** `determine_cohort(amount)`
  - **Behavior:** Given a check amount, it returns a cohort label:
    - If `amount < THRESHOLD_SMALL`, return `"STABLE_SMALL"`.
    - Else if `amount < THRESHOLD_LARGE`, return `"VOLATILE_MED"`.
    - Otherwise, return `"LAZY_GIANT"`.
  - The thresholds come from `constants.py` (10k and 50k by default).
- **Reasoning:** The idea is that very large checks may behave differently than smaller ones. If a vendor has very little data (fewer than 5 cleared checks historically), the code falls back to a “global cohort” model. The cohort groups all checks of similar size (across all vendors) to create enough data for a probability distribution.

File: `src/apforecast/modeling/probability.py`

- **Purpose:** Implements a simple empirical probability model (misleadingly named `BayesianModel`) for days-to-clear distribution and computing probabilities.
- **Class:** `BayesianModel`
  - **Initialization:** Takes an array of historical `days_data` (days it took checks to clear). It sorts this data and records `n = number of data points` and `max_observed_days = maximum days observed`.
  - **Empirical CDF:** `cdf(t)` computes the empirical cumulative distribution function at time `t`, i.e. the fraction of historic checks that cleared in ≤ t days. It does this by counting how many sorted data points are ≤ t and dividing by `n`. If no data (`n=0`), it returns 0.0.
  - **Predict Survival Probability:** (This function is defined but not directly used in current code.) It attempts to compute the probability of clearing *within a future window* given that a check has survived (not cleared) up to its current age. The formula is:

$$P(\text{clear in next } w \text{ days} \mid \text{not cleared by now}) = \frac{F(\text{current age} + w) - F(\text{current age})}{1 - F(\text{current age})},$$

where F(t) is the CDF. If the current age is beyond any observed history or there are no data, it returns `None` or 0.

This is essentially a conditional probability formula.

- **Assumptions and Theory:** The model treats the historical settling days as representative of future behavior (exchangeability). It is “empirical Bayesian” in that it uses the empirical distribution of past events. There is no assumption of any particular theoretical distribution (like normal or exponential); it’s purely data-driven. Because of limited data in many cases, this simple model is used instead of a complex parametric model.

File: `src/apforecast/modeling/engine.py`

- **Purpose:** Trains forecasting models (both vendor-specific and cohort-based) and makes predictions (probabilities) for when checks will clear.
- **Class:** `ForecastEngine`
    - **Initialization:** Takes the full ledger (Pandas DataFrame) as input. It calls `_train_models()`.
    - **Model Training (`_train_models`):**
        - **Vendor-Specific Models:** It selects all cleared checks (`Status == "CLEARED"`) and groups by `Vendor_ID`. If a vendor has at least 5 cleared checks, it creates a `BayesianModel` using that vendor’s `Days_to_Settle` data. These models are stored in `models['SPECIFIC'][vendor]`.
        - **Global Cohort Models:** For each cohort (`STABLE_SMALL`, `VOLATILE_MED`, `LAZY_GIANT`), it selects all cleared checks whose `Amount` falls into that cohort and trains a `BayesianModel` on their `Days_to_Settle`. These models are in `models['GLOBAL'][cohort]`.

        - **Assumption:** Five historical data points is deemed the minimum to create a vendor-specific model. If fewer, the vendor falls back to cohort model.

    - **Predicting a Check (`predict_check`):** Given a single check row, a `forecast_date`, and an optional `current_date_override`, this returns a probability:
        - **Inputs:** The check's vendor, amount, and post date. It computes `forecast_age = days between forecast_date and post_date`. If `current_date_override` is given, it also computes `current_age = days between override date and post_date`.
        - **Model Selection:** It first tries to use the vendor-specific model if available; otherwise it uses the cohort model (based on `determine_cohort(amount)`). If no model exists or no data (`n=0`), it returns 0 probability.
        - **Case A – Conditional Forecast (with current_date_override):** This is used when we want the probability that the check clears on or before `forecast_date`, *given that it is still open at* `current_date_override`. In practice, `current_date_override` is usually the day before the forecast date.
            - If `forecast_age <= current_age`, it returns 0 (cannot clear in the past).
            - If `current_age` is beyond what the model has seen (`current_age > model.max_observed_days`), it tries the cohort model instead; if not possible, returns 0.
            - Otherwise it computes two CDF values: `cdf_current = F(current_age)` and `cdf_forecast = F(forecast_age)`. Then conditional probability:

$$P(\text{clear by } t = \text{forecast\_date} \mid \text{alive at current\_date}) = \frac{F(\text{forecast\_age}) - F(\text{current\_age})}{1 - F(\text{current\_age})}$$

            - This follows from Bayes' rule for continuous distributions (here discrete days). The code then ensures this probability is between 0 and 1.

        - **Case B – Unconditional Forecast (no override):** It simply returns `F(forecast_age)`, the probability of clearing by that day unconditionally.
    - **Usage:** The engine is the core of forecasting. When predicting cash flow for a day, the code will loop over all open checks, call `predict_check` for each check with the forecast date and yesterday as `current_date_override`, then multiply the returned probability by the check's amount to get an **expected cash-outflow**.
- **Assumptions:** The model assumes that past clearing behavior continues into the future. It also assumes checks clear independently (expected values are summed across checks). Using `BayesianModel` means we implicitly assume that every new check is like drawing another sample from the same distribution of days-to-clear.

File: `src/apforecast/main.py`

- **Purpose:** Provides a simple programmatic interface (not UI) to get today's forecast in JSON format. It's essentially a utility that runs ingestion and forecasting and outputs structured results.
- **Main Function:** `forecast_today(run_date_str=None)`

    1. **Date Parsing:** Converts the input string (e.g. `"2026-01-21"`) into a pandas `Timestamp`. If not provided, uses today's date.
    2. **Ingest Data:** Calls `ingest_and_reconcile` to build the ledger up to `run_date`.
    3. **Filter Checks:** It filters the ledger for checks that are still open (`Status == "OPEN"`) and were posted on or before `run_date`. This enforces causality: we don't predict checks not yet issued.
    4. **Forecast Computation:** For each open check, it calls `engine.predict_check` (with `current_date_override = run_date - 1 day`) to get a probability of it clearing on `run_date`. It multiplies that by the check's amount to get its expected contribution for today.
    5. **Aggregate Results:** It sums all expected flows to get `predicted_today`. It also sums all open amounts for `total_outflow_outstanding`. It groups

results by vendor, summing each vendor’s outstanding and predicted. Only vendors with predicted > 0 are kept. The final JSON has:

        - `date`: the run date,
        - `total_outflow_outstanding`: sum of all open check amounts,
        - `predicted_today`: sum of all expected flows for today,
        - `by_vendor`: a map of each vendor’s `{ outstanding: X, predicted: Y }`, sorted by predicted.

- **CLI Interface:** At the bottom, there’s an `if __name__ == "__main__":` section allowing this to run as a standalone script. It accepts an optional `--date` parameter, calls `forecast_today`, and prints the JSON.
- **Assumptions:** This function assumes the ledger ingestion finds the appropriate raw files for that date (`data/raw/YYYY-MM-DD/uncleared_checks.xlsx`, etc.). It doesn’t handle exceptions beyond catching parse errors.

Folder: `experiments/backtesting`

This folder contains tools for validating and analyzing the forecast by comparing predicted vs actuals over historical periods.

File: `experiments/backtesting/core.py`

- **Purpose:** Implements the core walk-forward backtest and plotting of results.
- **Function:** `run_walk_forward_backtest(full_ledger, start_date, end_date, vendor_filter=None)`
    - **Inputs:** A complete historical ledger (Pandas DataFrame with cleared check history), a start date, an end date, and an optional vendor name to filter by.
    - **Preprocessing:** It ensures any row with a non-null `Clear_Date` is marked `Status = "CLEARED"`. Then if `vendor_filter` is set, it filters the ledger to only that vendor. If nothing remains, it returns an empty DataFrame.

- **Main Loop (Rolling):** It iterates from `current_date = start_date` up to `end_date`, one day at a time. For each `current_date`:

    1. **Training Data:** It takes all checks that cleared *before* `current_date` (`Clear_Date < current_date`) as training data.

    2. **Skip If Insufficient Data:** If there are fewer than 50 such checks (or only 5 if doing a single-vendor backtest), it skips forecasting for that date (insufficient history). This threshold ensures models have enough data.

    3. **Train Engine:** It creates a new `ForecastEngine` using the training data. This resets vendor-specific and cohort models based on history up to that point.

    4. **Identify Open Checks:** It defines open checks on that date as those posted on or before `current_date` and not cleared before that (i.e. `post_date <= current_date` and `Clear_Date >= current_date` so they clear at/after or still open).

    5. **Predict for Date:** For each open check, it calculates age = `current_date - post_date`. If `age > 45`, it **flags** the check (adds its amount to `flagged_volume`) and *does not* predict it. The assumption here is that very old checks are not forecasted (maybe considered outliers or handled manually). Otherwise, it calls `engine.predict_check(check, current_date, current_date_override=current_date-1)`. It sums `amount * prob` into `predicted_cash`.

    6. **Actual Cash:** It finds all checks whose `Clear_Date == current_date` and sums their amounts as `actual_cash`.

    7. **Metrics:** It records the residual (`actual_cash - predicted_cash`), cumulative error, and error percentage for that day. It also records `flagged_volume`.

    8. **Store Result:** Appends a row with `Date`, `Predicted`, `Actual`, `Residual ($)`, `Error %`, `Cum. Error`, and `Flagged (>45d)` into a results list.

    - **Return:** At the end, returns a DataFrame of these daily results between the dates.

- **Function:** `plot_backtest_results(df, title_prefix="Global")`
    - **Purpose:** Creates a two-panel Plotly figure for visualizing backtest results.
    - **First Panel:** Plots actual vs predicted cash flow over time. Actual flow is shown as a filled green area, predicted as a blue line with markers.
    - **Second Panel:** Plots the daily residual (error) as a red bar chart, with a dashed horizontal line at 0. This shows how prediction deviated each day. The hover text shows both the dollar error and percent error.
    - **Assumption:** Larger residuals mean worse model performance. Ideally the residuals hover around 0 (balanced over time). A rising cumulative error would indicate consistent bias.

File: `experiments/backtesting/cli.py`

- **Purpose:** Provides a command-line interface for running the backtest.
- **Main Function:** `main()`
    - **Argument Parsing:** It takes `--start`, `--end`, and `--source` (path to historical file).
    - **Load and Normalize:** It reads the source Excel, normalizes columns with its own `normalize_columns` function (similar to `loader` but custom), converts dates to datetime, ensures a `Status` column (assumes cleared if missing), and prints some diagnostics (earliest/latest dates, total records).
    - **History Check:** It checks how much history is available before the start date. If fewer than 50 records exist before `start`, it prints an error instructing to choose a later start date (ensuring enough training data).
    - **Run Backtest:** Calls `run_walk_forward_backtest` from the core. If successful, prints total predicted vs actual for the entire period and net variance (with percentage).
    - **Save Outputs:** Writes the resulting metrics DataFrame to an Excel file and the plot to an HTML file under `experiments/results`, with timestamps in the

filenames.

    - **Feedback:** Prints summary of where results are saved.

- **Usage:** This CLI allows a technical user to validate the forecasting model by comparing it against actual known clears over a period. It encapsulates the logic of `run_walk_forward_backtest` in a user-friendly script.

File: `api/api.py`

- **Purpose:** Implements a REST API (using FastAPI) to expose the forecast functionality over HTTP.
- **Key Endpoint:** `GET /forecast/today?date=YYYY-MM-DD`
    - **Behavior:** When called, it parses the date (or defaults to today), then calls an internal function `forecast_today(run_date)` (note: different from `src/apforecast/main.forecast_today`). This function:
        1. **Loads Ledger:** Calls the same `ingest_and_reconcile` to get the ledger up to that date.
        2. **Trains Engine:** Initializes `ForecastEngine` with the ledger.
        3. **Filters Open Checks:** Keeps checks that are OPEN and posted on or before run_date.
        4. **Prediction Loop:** Like `main.py`, it sums the expected values (`amount * prob`) for all open checks, and accumulates totals per vendor. It uses `current_date_override = run_date - 1 day`.
        5. **Build Response:** Returns a JSON with `run_date`, `total_outstanding`, `predicted_today`, and `by_vendor` mapping each vendor to its predicted amount (rounded). It filters out vendors with zero prediction.
    - **Health Check:** Also provides `GET /health` that returns `{"status": "ok"}`.
- **Assumptions:** Similar to `main.py`, it assumes the same data ingestion works. The API is simply a wrapper around the core forecast logic. The sorting/filtering logic (keep only

vendors with >0 predicted) is done in the API code.

File: `app.py` (Streamlit Frontend)

- **Purpose:** Provides an interactive dashboard (using Streamlit) for users to update the ledger, run forecasts, and backtests, and visualize results. This is the user-facing application “APForecast Commander”.
- **Page Setup:**
  1. Sets title “💸 APForecast Commander” and configures the page layout. Maintains some session state (`ledger`, `forecast_df`, backtest results, etc.) to preserve data between user interactions.
- **Sections (Navigation Tabs):** There are two main modes, selectable in the sidebar:
  1. 🚀 **Forecast & Intelligence:** For running daily forecasts and seeing vendor insights.
  2. 🧪 **Backtest Lab:** For running the backtesting analysis.

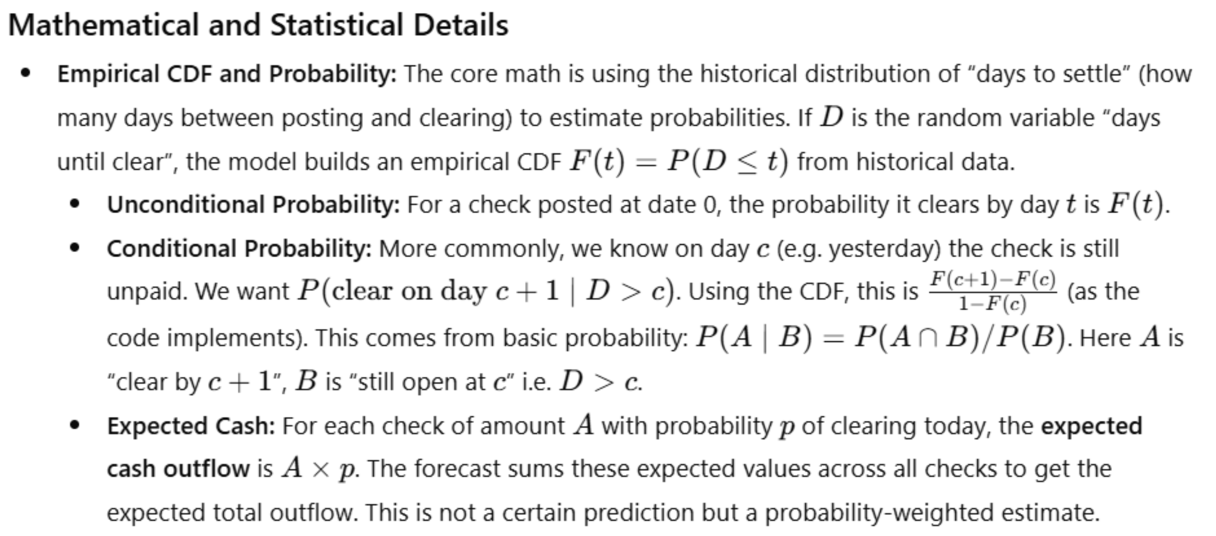
Forecast & Intelligence Mode:

- **Header:** “Daily Cash Forecast”.
- **Update Ledger (Expander):** Allows the user to upload a file of newly cleared checks. The system merges this with the existing master ledger. Specifically:
  - The uploaded file is read, normalized (`smart_normalize_columns`), and compared against `data/master_ledger.xlsx`.
  - New unique checks are appended, and the master ledger is updated.
  - Provides feedback: number of new checks added, or a message if none were new.
  - *Note:* This is essentially a user-driven re-run of part of the reconciler logic to keep the historical data up-to-date.

- **Run Daily Forecast (Expander):** The user selects a **Run Date** (default today) and uploads an **Outstanding Checks** file (uncleared checks).
    - On clicking “RUN FORECAST”:
        - The outstanding checks file is saved to `data/raw/{run_date}/uncleared_checks.xlsx`.
        - Calls `ingest_and_reconcile` to update the master ledger (this will add any new uncleared checks and mark clears if provided in the day folder).
        - Ensures `Vendor_ID` column exists (using `smart_normalize_columns` if needed).
        - Trains `ForecastEngine` on the updated ledger.
        - Finds all open checks on that run date, and for each computes `Probability = engine.predict_check(...)` and `Expected_Cash = Amount * Probability`. Builds a DataFrame of these. Stores it in session state.
        - Shows a success message.
- **Portfolio Overview:** Once forecast data is available:
    - Displays metrics: total expected cash required today (`sum Expected_Cash`) and total exposure (`sum Amount of all open checks`).
    - Provides a button to download the forecast DataFrame as Excel.
    - Shows a **Chart (plot_interactive_landscape)**: a scatter plot of all open checks, where X=age (days since posted), Y=amount, marker size = log(amount), color = age. This visualization helps see how many checks are outstanding and their ages.
- **Vendor Intelligence:**
    - A dropdown to select a vendor (from those in open checks).
    - For the selected vendor, it shows:

        - **Vendor Payment Profile (left):** If enough history exists for that vendor, it shows `plot_vendor_history_legacy`: a bar chart of the empirical probability that the vendor pays on day N (for various N), with a smoothed CDF curve overlay. This tells you the vendor's typical payment speed. If history is insufficient, it shows a message instead.
        - **Delay Scenario (right):** A slider for "Simulate Delay" (0–14 days). It takes the vendor's open checks and plots `plot_snowball_interactive`: expected cash flow over the next 14 days for that vendor under normal conditions (bars) and with an added fixed delay (green line). This visualizes how cash flow would shift if payments are delayed by some days. It also shows a data table of that vendor's open checks (ID, amount, age).
        - Below the charts, an insight message explains that moving the slider shows the effect of delays.

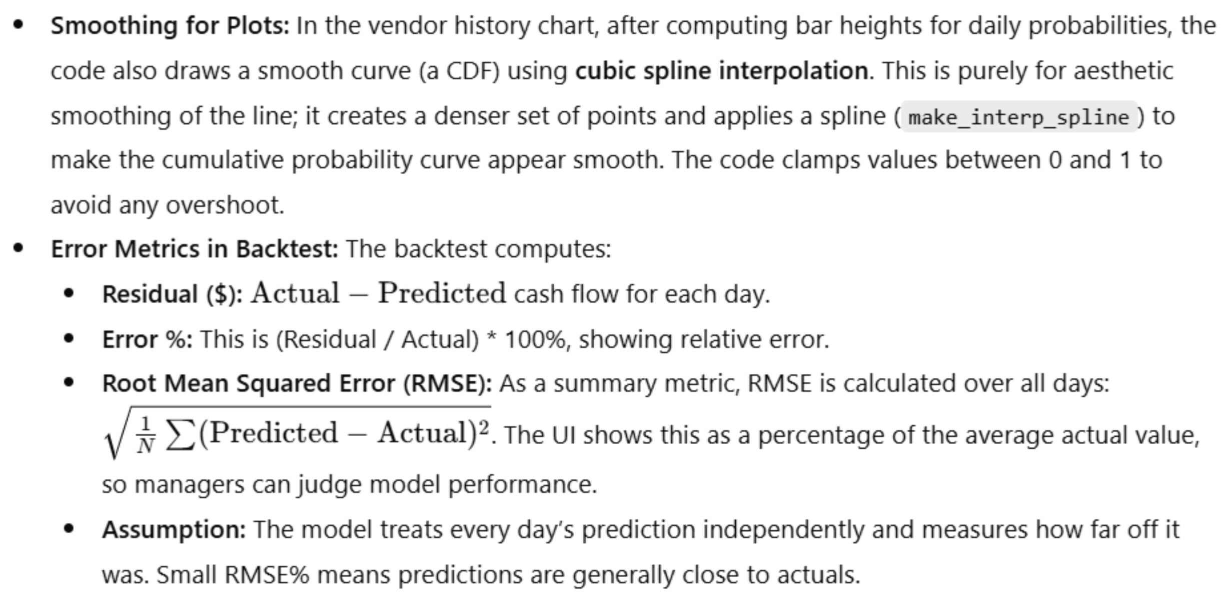
- **Helper Functions Used:**
    - `smart_normalize_columns(df)`: (defined earlier in app.py) Ensures the DataFrame has the right columns (`Vendor_ID`, `Amount`, etc.) by guessing from header keywords. It coerces date columns to datetime and uppercases Vendor_ID. If no vendor column is found, it creates one with `"Unknown_Vendor"` and warns the user.
    - `plot_interactive_landscape(ledger, run_date)`: Creates the scatter age vs amount chart described above.
    - `plot_vendor_history_legacy(model, vendor_name, ledger)`: Builds the vendor history bar/CDF chart (if history exists). It groups past cleared checks for that vendor by days-to-clear, computes probabilities and cumulative probabilities, and plots them with Plotly.
    - `plot_snowball_interactive(engine, open_checks, run_date, custom_delay)`: Computes and plots expected cash over the next 14 days for the given open checks. The normal expected flow is a bar chart; the "delay scenario" shifts the prediction window by `custom_delay` days (effectively simulating if everything were delayed by that amount) and draws a smoothed line. Hovering on bars shows top contributing checks.



Mathematical and Statistical Details

- **Empirical CDF and Probability:** The core math is using the historical distribution of "days to settle" (how many days between posting and clearing) to estimate probabilities. If D is the random variable "days until clear", the model builds an empirical CDF $F(t) = P(D \le t)$ from historical data.
  - **Unconditional Probability:** For a check posted at date 0, the probability it clears by day t is $F(t)$.
  - **Conditional Probability:** More commonly, we know on day c (e.g. yesterday) the check is still unpaid. We want $P(\text{clear on day } c+1 \mid D > c)$. Using the CDF, this is $\frac{F(c+1)-F(c)}{1-F(c)}$ (as the code implements). This comes from basic probability: $P(A \mid B) = P(A \cap B)/P(B)$. Here A is "clear by $c+1$", B is "still open at c" i.e. $D > c$.
  - **Expected Cash:** For each check of amount A with probability p of clearing today, the **expected cash outflow** is $A \times p$. The forecast sums these expected values across all checks to get the expected total outflow. This is not a certain prediction but a probability-weighted estimate.

○



- **Smoothing for Plots:** In the vendor history chart, after computing bar heights for daily probabilities, the code also draws a smooth curve (a CDF) using **cubic spline interpolation**. This is purely for aesthetic smoothing of the line; it creates a denser set of points and applies a spline (`make_interp_spline`) to make the cumulative probability curve appear smooth. The code clamps values between 0 and 1 to avoid any overshoot.
- **Error Metrics in Backtest:** The backtest computes:
  - **Residual ($):** $\text{Actual} - \text{Predicted}$ cash flow for each day.
  - **Error %:** This is (Residual / Actual) * 100%, showing relative error.
  - **Root Mean Squared Error (RMSE):** As a summary metric, RMSE is calculated over all days: $\sqrt{\frac{1}{N}\sum(\text{Predicted} - \text{Actual})^2}$. The UI shows this as a percentage of the average actual value, so managers can judge model performance.
  - **Assumption:** The model treats every day's prediction independently and measures how far off it was. Small RMSE% means predictions are generally close to actuals.

Assumptions and Design Choices

- **Data Quality:** The system assumes input data is reasonably clean. Dates can be parsed (with day-first allowed), and amounts are numeric. The `smart_normalize_columns` and `load_file_smart` functions try to handle variations in column names. Still, completely missing required columns will cause errors.

- **Vendor Independence:** The model treats each check/payment as independent of others (no correlation between vendors or payments). This is common in cash forecasting but may not capture real-world dependencies.

- **Historical Stationarity:** It assumes past behavior predicts future behavior. If a vendor's payment habits change suddenly, the model won't immediately capture that until it gets new cleared checks to retrain on.

- **Thresholds:** Several magic numbers appear (5 checks to build a vendor model, 50 checks to start forecasting in backtest, 45 days cutoff). These are assumptions that come from domain expertise or empirical testing. For example, not predicting checks older than 45 days likely reflects that such outliers skew the model or are handled by exception processes.

- **Cohort Fallback:** If a vendor has too little data, the code uses a global cohort model. This assumes that smaller checks have similar clearing patterns across vendors (and same for larger checks). It's a reasonable fallback but assumes homogeneity within each size cohort.

- **Visualization:** Charts (Plotly) assume the user wants interactive insights. For example, the delay slider shows how expected cash outflows change if payments are delayed. This is not part of the core math but is a useful simulation feature.

Summary of Files and Functionality

- **Constants (`constants.py`)** – Defines all fixed values, column names, and paths.

- **Loader (`loader.py`)** – Reads raw files, applies column header mapping.

- **Reconciler (`reconciler.py`)** – Builds and updates the master ledger by merging historical data with new daily inputs.

- **Modeling (cohorts, probability, engine):**

  - `cohorts.py` – Assigns checks to size cohorts (small/medium/large).

  - `probability.py` – Computes empirical CDF and conditional probabilities from historical days-to-clear.

  - `engine.py` – Trains vendor-specific and cohort models and provides a `predict_check` function returning clearance probabilities.

- **Forecast Interface (`main.py` and `api.py`):**

    - `main.py` – A script that returns a JSON forecast for a given date (for programmatic use).

    - `api.py` – A FastAPI service exposing the forecast function via HTTP endpoints.

- **Streamlit App (`app.py`):** Offers a GUI for uploading data, running forecasts, visualizing results, and running backtests. It includes interactive charts and vendor analysis.

- **Backtesting (`experiments/backtesting`):**

    - `core.py` – Implements rolling backtest of the model over historical periods and plotting functions.

    - `cli.py` – A command-line tool that loads historical data, runs the backtest, and outputs summary metrics and plots.

Conclusion

All components appear logically consistent and functional as designed. Together, they form a full pipeline: ingest historical and daily data, model payment behavior per vendor/cohort, forecast daily cash outflows, and validate via backtesting. The core mathematics is based on empirical probability distributions of payment delays. Care has been taken to handle real-world issues like missing column names, insufficient data, and user-friendly visual output. Assuming the data inputs meet the expected formats and the thresholds align with business experience, this system should provide a transparent and explainable accounts payable cash forecasting solution.