# RetirePlan Engine and Architecture Review

## 1. Project Purpose and Structure

\*\*RetirePlan\*\* is a single-window GUI application for detailed retirement scenario modeling. It enables users to input personal, financial, and policy data, then produces a year-by-year table of projected finances, including balances, income, taxes, spending, RMDs, Social Security, and more.

- \*\*Inputs:\*\* Provided via structured YAML, GUI forms, or both.

- \*\*Outputs:\*\* Yearly projections presented in a table, exportable as CSV.

- \*\*Architecture:\*\* Python, modular GUI, core calculations in `engine.py`, with supporting modules for taxes, precision, policy, and accounts.

## 2. Strengths

- \*\*Feature-rich:\*\* Covers a wide range of retirement planning needs.

- \*\*Separation of Concerns:\*\* Inputs, calculations, and UI are clearly separated.

- \*\*Extensibility:\*\* Existing modules for taxes, policy, and accounts lay a foundation for further modularization.

- \*\*Scenario Support:\*\* Batch scenario logic is present (see `scenarios.py`).

## 3. Current Weaknesses

### A. Engine Monolithism and Flow

- \*\*Monolithic Logic:\*\* `engine.py` contains nearly all yearly calculation logic, state handling, and output aggregation, making the code hard to maintain, test, or extend.

- \*\*Year 1 Special Case:\*\* Year 1 is handled with custom code, not reusing the generic annual logic, leading to potential confusion and bugs.

- \*\*State Management:\*\* State passed year-to-year is not explicit or encapsulated in a single structure, risking subtle bugs.

### B. Data and Calculation Clarity

- \*\*Ambiguous Definitions:\*\* Not all input/output columns are well-documented. For example, the meaning and sign convention for "Cash Events" can be unclear.

- \*\*Column Calculation Consistency:\*\* Some calculations (e.g., Target Spend, Shortfall) do not always align with their intended definitions, leading to output inconsistencies.

### C. Modularity and Testability

- \*\*Insufficient Modularization:\*\* Many calculations that could be in modules (accounts, income, RMDs, spending) are inlined in `engine.py`.

- \*\*Minimal Unit Testing:\*\* There is little evidence of unit tests for individual components or for year-by-year calculations, reducing confidence in correctness.

### D. UI/UX and Output

- \*\*Redundant Features:\*\* Some UI/button features or output files may be unnecessary or duplicative, adding clutter.

- \*\*Column/Field Drift:\*\* GUI, YAML config, and engine field names/meanings sometimes diverge, risking confusion for users and maintainers.

## 4. Recommendations

### A. Definitions and Documentation

- \*\*Finalize Definitions:\*\* For every input and output column, write a plain-English definition and calculation method. Make these available as docstrings, a Markdown file, or in the GUI.

- \*\*Clarify Sign Conventions:\*\* Explicitly document the sign for values like "Cash Events" or "Withdrawals".

### B. Engine Refactor & Modularity

- \*\*Break Up `engine.py`:\*\* Move annual calculation logic into a `year.py` module, with helper modules for accounts, income, spending, and RMDs.

- \*\*Single Responsibility:\*\* Each module should do one thing (e.g., `accounts.py` for withdrawals/growth, `taxes.py` for taxes).

- \*\*Consistent Yearly Flow:\*\* Treat Year 1 as just the first year of calculation, using the same logic as other years, with parameter overrides as needed.

- \*\*State Passing:\*\* Use a clearly defined state object to pass balances, ages, and other variables year-to-year.

### C. Calculation Flow and Output

- \*\*Explicit Calculation Order:\*\* Clearly order calculations: (1) update ages/state, (2) apply growth, (3) calculate income, (4) RMDs, (5) spending/withdrawals, (6) taxes, (7) update balances.

- \*\*Ensure Data Consistency:\*\* All outputs should match their definitions and add up as expected (e.g., Total Assets = sum of balances).

### D. UI/UX and Config Consistency

- \*\*Align GUI and Engine:\*\* Ensure all field names and meanings are the same in GUI, config, and engine code.

- \*\*Reduce Redundancy:\*\* Remove or merge unnecessary buttons, exports, or fields, focusing on user clarity.

### E. Testing and Debuggability

- \*\*Unit Tests:\*\* Write unit tests for each module. Test edge cases (e.g., withdrawal order, RMDs, survivor transitions).

- \*\*Verbose/Debug Mode:\*\* Add an optional mode that logs intermediate calculations each year for troubleshooting.

### F. Maintainability and Extensibility

- \*\*Scenario Support:\*\* Modular code will make it easier to add scenario analysis, batch runs, or Monte Carlo features in the future.

- \*\*Documentation:\*\* Keep code, user documentation, and output definitions in sync.

## 5. Suggested Directory Structure

```

retireplan/

engine/

\_\_init\_\_.py

engine.py # Orchestrator/loop

year.py # One year's calculation

income.py # All income source logic

taxes.py # Tax calculations (move/merge as needed)

spending.py # All spending logic

accounts.py # Withdrawals, contributions, balances

rmd.py # RMDs logic

policy.py # Constants/rules

precision.py # Rounding, fixed-point

utils.py # Utility functions

```

## 6. Questions/Clarifications Before Refactoring

1. \*\*Column Definitions:\*\* Should we finalize a dictionary of all output columns and formulas before breaking up the engine?

2. \*\*Year 1 Logic:\*\* Should Year 1 be handled as just the first year with overrides, or does it need fundamentally different logic?

3. \*\*GUI/Config Alignment:\*\* Are you open to renaming or moving fields for clarity, or is backward compatibility essential?

4. \*\*Output Flexibility:\*\* Should the engine always output the same columns, or do you want scenario-specific outputs?

5. \*\*Testing Requirements:\*\* Should unit tests be added for each module as we refactor?

---

\*\*In summary:\*\*

RetirePlan is a strong foundation for personal retirement projection, but the engine needs modularization, clearer definitions, calculation consistency, and improved testability. These steps will make the codebase more maintainable, reliable, and user-friendly for scenario analysis and long-term evolution.