Got it. I read through your InvestmentStrategy.txt (it's a Jupyter notebook serialized as JSON) and distilled what it actually does, how it decides allocations, and how to turn it into a clean, fast, and testable Python + Gradio app. I'll keep things crisp and table-first where helpful.

Executive summary

The provided strategy computes monthly momentum features (1–12M) for a curated set of "aggressive" (equity) and "passive/defensive" (bonds, bills, TIPS, IG/HY credit, EM bonds, KR gov bond) ETFs via Yahoo Finance, then makes an allocation with two rules:

1. **Algorithm 1 (Rule-based):**
   If any **aggressive** asset has **positive 12-month** return → allocate **100% to the best one**.
   Otherwise → pick **top-3 passive** by **6-month** return; replace any negative-return picks with **Cash**.

2. **Algorithm 2 (Score-based):**
   For a few designated assets (e.g., SPY, VEA, TLT, 148070.KS), compute the **fraction of positive returns across 1–12M** (12 features). Add a **fixed Cash score** (e.g., 0.15). Allocate **proportionally to scores**.

3. **Blend:** Weighted blend of (1) and (2). Save the final allocation to CSV (Google Drive path in the notebook).

What the current notebook contains

Component	What it does	Where it is (in the txt)
Data import	yfinance download over a 1-year window (data_period="1y") for a curated ETF list	Import cells & tickers = [...]
Feature engineering	Builds monthly return columns **1M … 12M**, and summary tables: aggressive_table, passive_table, full_table	Mid cells building "Change (%)" columns

Algorithm 1	Rule: best 12M aggressive → 100% else top-3 passive by 6M with negative→Cash	def run_algorithm_1(...)
Algorithm 2	Score = share of {1M..12M} that are positive; add Cash score; normalize	def run_algorithm_2(...)
Blending	Weighted combine two allocations	def blend_allocations(...)
Output	Console print; append CSV to Google Drive path	Final cells (path = "...Investment_Decisions.csv")

Strategy logic (concise)

Step	Inputs	Output	Notes
Price load	Ticker list (US & KR ETFs)	Panel of close prices	Daily → monthly return aggregation
Monthly returns	Close prices	1M..12M returns	Can compute with resample('M').last().pct_change(k)
Aggressive rule	aggressive_table["12M"]	1-ticker 100% OR fallback	Positive 12M → all-in; else passive fallback
Passive fallback	passive_table["6M"]	Top-3 equally weighted; negatives→Cash	Replaces each negative constituent with Cash slice
Score rule	full_table[1M..12M]	Proportional weights across set + Cash	Score = positive-count / 12
Blend	alloc1, alloc2 + weights	Final allocation	alloc = w1*alloc1 + w2*alloc2

Key design choice: This is a "momentum tilt with defensive fallback." It behaves like a trend-following switch between risk-on (single equity) and diversified bonds/cash.

Risks & practical notes

Concern	Impact	Mitigation
Data gaps / delistings	NaNs lead to dropped assets or wrong ranks	Strict data validation; require min history; fallback logic
Monthly boundary effects	End-month timing can change ranks	Fix a **rebal day** (e.g., last trading day) and use PX as of T-1
All-in equity	Large drawdowns in whipsaws	Impose **max weight** caps; add **cooldown** after big losses
Passive negative returns	Cash substitution may overweight Cash	Add **floor** & **cap** for Cash; consider 4–6 assets in fallback

---

Clean repository design (Python + Gradio)

```
investment-strategy/
├─ README.md
├─ pyproject.toml            # or requirements.txt + setup.cfg
├─ requirements.txt          # yfinance, pandas, numpy, matplotlib, gradio, pydantic, polars (optional)
├─ .env.example              # API keys/paths if needed
├─ config/
│  ├─ base.yaml              # tickers, groups, weights, lookbacks, paths
│  └─ presets/
│     ├─ conservative.yaml
│     └─ aggressive.yaml
├─ data/
│  ├─ cache/                 # downloaded prices cache (parquet/csv)
│  └─ outputs/
│     └─ decisions.csv
├─ src/
│  ├─ strategy/
│  │  ├─ __init__.py
│  │  ├─ config.py           # load/validate YAML → pydantic models
│  │  ├─ data.py             # download, cache, validate prices
│  │  ├─ features.py         # monthly returns, tables
│  │  ├─ algo_rules.py       # Algorithm 1
│  │  ├─ algo_scores.py      # Algorithm 2
```

```
│   │   ├── blend.py            # blending & normalization
│   │   ├── allocate.py          # orchestration: run_all(config) → allocation df
│   │   └── io.py               # save/load decisions, append-safe
│   └── app/
│       ├── __init__.py
│       └── gradio_app.py        # Gradio UI
├── tests/
│   ├── test_features.py
│   ├── test_algo_rules.py
│   ├── test_algo_scores.py
│   └── test_pipeline.py
├── scripts/
│   ├── run_once.py             # CLI one-shot run
│   └── backtest.py             # optional: quick backtest
└── Makefile                    # tidy commands (run, test, lint)
```

Why this shape?

- Separation of concerns (data, features, algorithms, blend, IO).
- Config-driven tickers/weights/paths.
- Cache to avoid repeated downloads.
- Tests target each layer.

Minimal config/base.yaml (example)

```
data:
  period: "5y"
  interval: "1d"
  cache_dir: "data/cache"

universe:
  aggressive: ["SPY", "294400.KS", "VEA"]
  passive: ["BIL", "SHY", "IEF", "TLT", "TIP", "LQD", "HYG", "BWX", "EMB", "148070.KS"]
  score_set: ["SPY", "VEA", "TLT", "148070.KS"]
```

```
  cash_ticker: "CASH"   # synthetic

features:
  months: 12          # build 1..12M returns
  rebalance_day: "EOM"  # or specific day-of-month

allocation:
  algo1:
    passive_top_n: 3
    passive_rank_col: "6M"
    aggressive_pos_col: "12M"
  algo2:
    cash_score: 0.15
  blend:
    w1: 0.6
    w2: 0.4
  caps:
    max_weight: 0.8
    min_weight: 0.0
    cash_cap: 0.7

io:
  decisions_csv: "data/outputs/decisions.csv"
```

Core modules (concise code)

src/strategy/data.py

```
import os
import pandas as pd
import yfinance as yf
from pathlib import Path

def download_prices(tickers, period="5y", interval="1d", cache_dir=None):
    cache_dir and Path(cache_dir).mkdir(parents=True, exist_ok=True)
    key = f"{'_'.join(tickers)}_{period}_{interval}.parquet"
    cache_path = Path(cache_dir)/key if cache_dir else None
```

```
    if cache_path and cache_path.exists():
        return pd.read_parquet(cache_path)

    df = yf.download(tickers, period=period, interval=interval, auto_adjust=True)["Close"]
    if isinstance(df, pd.Series): df = df.to_frame()
    df = df.dropna(how="all")
    if cache_path: df.to_parquet(cache_path)
    return df
```

src/strategy/features.py

```
import pandas as pd

def monthly_returns(prices: pd.DataFrame, months=12) -> pd.DataFrame:
    mpx = prices.resample("M").last().dropna(how="all")
    feats = {}
    for k in range(1, months+1):
        feats[f"{k}M"] = mpx.pct_change(k)
    out = pd.concat(feats, axis=1)
    out.columns = out.columns.get_level_values(0)  # flatten
    return out

def latest_snapshot(monthlies: pd.DataFrame) -> pd.DataFrame:
    # take last row as current features
    return monthlies.iloc[[-1]].T.reset_index().rename(columns={"index":"Ticker",
monthlies.index[-1]:"Value"}).pivot_table(index="Ticker", columns=None, values="Value",
aggfunc="first")
```

src/strategy/algo_rules.py

```
import pandas as pd

def run_algorithm_1(full_table: pd.DataFrame, aggressive: list, passive: list,
                    aggressive_pos_col="12M", passive_rank_col="6M", passive_top_n=3, cash_ticker="CASH"):
    agg = full_table.loc[full_table.index.intersection(aggressive)]
    pas = full_table.loc[full_table.index.intersection(passive)]

    alloc = {}
    pos_aggs = agg[agg[aggressive_pos_col] > 0]
    if not pos_aggs.empty:
```

```
        best = pos_aggs[aggressive_pos_col].idxmax()
        alloc[best] = 1.0
        return alloc

    # fallback: top-N passive by 6M
    top = pas.sort_values(passive_rank_col, ascending=False).head(passive_top_n)
    w = 1.0 / passive_top_n
    cash_add = 0.0
    for t, row in top.iterrows():
        if row[passive_rank_col] > 0:
            alloc[t] = alloc.get(t, 0) + w
        else:
            cash_add += w
    if cash_add > 0:
        alloc[cash_ticker] = cash_add
    return alloc
```

src/strategy/algo_scores.py

```
def run_algorithm_2(full_table, score_set, cash_score=0.15):
    scores = {}
    for t in score_set:
        if t not in full_table.index: continue
        row = full_table.loc[t]
        pos = sum(1 for m in range(1,13) if row.get(f"{m}M", float("nan")) > 0)
        scores[t] = pos / 12.0
    total = sum(scores.values()) + cash_score
    alloc = {t: s/total for t,s in scores.items()}
    alloc["CASH"] = cash_score / total
    return alloc
```

src/strategy/blend.py

```
def blend(alloc1: dict, alloc2: dict, w1=0.6, w2=0.4, caps=None):
    keys = set(alloc1) | set(alloc2)
    out = {k: round(alloc1.get(k,0)*w1 + alloc2.get(k,0)*w2, 6) for k in keys}
    # normalize
    s = sum(out.values()) or 1.0
    out = {k: v/s for k,v in out.items()}
    # caps
```

```
    if caps:
        maxw = caps.get("max_weight")
        if maxw is not None:
            for k in out: out[k] = min(out[k], maxw)
        s = sum(out.values()) or 1.0
        out = {k: v/s for k,v in out.items()}
    return out
```

src/strategy/allocate.py

```
import pandas as pd
from .data import download_prices
from .features import monthly_returns
from .algo_rules import run_algorithm_1
from .algo_scores import run_algorithm_2
from .blend import blend

def run_all(cfg) -> pd.DataFrame:
    uni = cfg["universe"]
    prices = download_prices(uni["aggressive"] + uni["passive"], **cfg["data"])
    feats = monthly_returns(prices, months=cfg["features"]["months"]).iloc[-1]  # last row
    full = feats.unstack().to_frame(name="ret").reset_index()
    # Rebuild a table Ticker x {1M..12M}
    table = prices.resample("M").last().pct_change(range(1, cfg["features"]["months"]+1)).iloc[-1]
    table.index = table.index.rename("Ticker")
    full_table = table.to_frame().T
    full_table = full_table.T  # rows=tickers, cols=1M..12M
    full_table.index.name = "Ticker"

    a1 = run_algorithm_1(full_table, uni["aggressive"], uni["passive"],
                         cfg["allocation"]["algo1"]["aggressive_pos_col"],
                         cfg["allocation"]["algo1"]["passive_rank_col"],
                         cfg["allocation"]["algo1"]["passive_top_n"],
                         uni.get("cash_ticker","CASH"))
    a2 = run_algorithm_2(full_table, uni["score_set"], cfg["allocation"]["algo2"]["cash_score"])
    final = blend(a1, a2, cfg["allocation"]["blend"]["w1"], cfg["allocation"]["blend"]["w2"],
                  caps=cfg["allocation"].get("caps"))

    df = (pd.Series(final, name="weight").to_frame()
          .assign(timestamp=pd.Timestamp.utcnow().normalize()))
    return df[["timestamp","weight"]].sort_values("weight", ascending=False)
```

src/strategy/io.py

```
import pandas as pd
from pathlib import Path

def append_decision(df: pd.DataFrame, csv_path: str):
    p = Path(csv_path); p.parent.mkdir(parents=True, exist_ok=True)
    if p.exists():
        old = pd.read_csv(p)
        out = pd.concat([old, df.reset_index().rename(columns={"index":"ticker"})], ignore_index=True)
    else:
        out = df.reset_index().rename(columns={"index":"ticker"})
    out.to_csv(p, index=False)
```

Gradio app (compact, single file)

src/app/gradio_app.py

```
import gradio as gr
import yaml
from pathlib import Path
from strategy.allocate import run_all
from strategy.io import append_decision

CFG_PATH = Path("config/base.yaml")

def run_strategy(config_yaml: str):
    cfg = yaml.safe_load(config_yaml)
    df = run_all(cfg)
    append_decision(df, cfg["io"]["decisions_csv"])
    return df.reset_index()

def default_config():
    return CFG_PATH.read_text()

with gr.Blocks(title="Momentum Blend Strategy") as demo:
    gr.Markdown("## Momentum Strategy — Aggressive/Passive with Blend")
```

```
    with gr.Row():
        config = gr.Textbox(lines=22, value=default_config(), label="YAML Config")
    run_btn = gr.Button("Run & Save Decision")
    out = gr.Dataframe(label="Allocation", interactive=False)
    run_btn.click(run_strategy, inputs=[config], outputs=[out])

if __name__ == "__main__":
    demo.launch()
```

Run locally

```
pip install -r requirements.txt
python -m src.app.gradio_app
# then open the local URL Gradio prints
```

Tests (examples)

tests/test_algo_rules.py

```
import pandas as pd
from src.strategy.algo_rules import run_algorithm_1

def test_algo1_aggressive_wins():
    ft = pd.DataFrame({"12M":[0.1, -0.05], "6M":[0.05,0.02]}, index=["SPY","TLT"])
    alloc = run_algorithm_1(ft, aggressive=["SPY"], passive=["TLT"])
    assert alloc.get("SPY",0) == 1.0

def test_algo1_passive_fallback_cash():
    ft = pd.DataFrame({"12M":[-0.1], "6M":[-0.02]}, index=["TLT"])
    alloc = run_algorithm_1(ft, aggressive=["SPY"], passive=["TLT"], cash_ticker="CASH")
    assert abs(alloc["CASH"] - 1.0) < 1e-9
```

---

requirements.txt

```
pandas>=2.2
```

numpy>=1.26
yfinance>=0.2
gradio>=4.0
pydantic>=2.7
pyyaml>=6.0

(Optional: polars, duckdb for speed; matplotlib if you want plots.)

Makefile (quality of life)

.PHONY: run test lint

run:
\tpython -m src.app.gradio_app

test:
\tpytest -q

lint:
\trufflehog . || true
\truff -q format src tests

Performance & reliability tips

- **Cache** downloads (Parquet under data/cache/) to avoid slow repeated calls.
- **Validate** that each ticker has ≥ 12 months of data; skip or backfill otherwise.
- **Deterministic rebal day**: e.g., last business day of month at 23:59 UTC.
- **Caps**: set max_weight (e.g., 80%) to reduce all-in risk.
- **CI**: run unit tests on PR; smoke test the Gradio launch.

What you'll get with this repo

- Reproducible one-click allocation from a YAML config.
- Clear logic parity with your notebook (Alg 1 + Alg 2 + blend).
- GUI for non-technical users via Gradio.
- CSV append (same as notebook) but in a safe, atomic way.

If you want, I can generate this repository as a zip (with the files pre-filled), or wire in extras like **backtests**, **charts**, or **Docker** for deployment.