```
import argparse
import math
import os
from dataclasses import dataclass
from datetime import datetime, date, timedelta
from typing import Callable, Dict, List, Optional, Tuple

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
import requests
from scipy.stats import norm


# -----------------------------
# Utilities: dates + expirations
# -----------------------------

def is_third_friday(d: date) -> bool:
    # Third Friday: weekday=4 (Mon=0..Sun=6), day between 15..21
    return d.weekday() == 4 and 15 <= d.day <= 21

def parse_yyyy_mm_dd(s: str) -> date:
    y, m, d = map(int, s.split("-"))
    return date(y, m, d)

def yearfrac_act365(start: date, end: date) -> float:
    return (end - start).days / 365.0

def pick_next_three_monthly_expirations(option_dates: List[str], asof:
date) -> List[str]:
    """
    Given Yahoo option expiration strings, select the next 3 monthly
expirations
    that fall on the 3rd Friday, with expiries >= asof.
    """
    ds = [parse_yyyy_mm_dd(x) for x in option_dates]
    ds = [d for d in ds if d >= asof and is_third_friday(d)]
    ds = sorted(ds)
    # "next three months" in the prompt: we interpret as next 3
monthly expirations available
    picked = ds[:3]
    return [d.isoformat() for d in picked]


# -----------------------------
# Part 2: Black-Scholes + Greeks
# -----------------------------
```

```
def bs_d1_d2(S: float, K: float, r: float, tau: float, sigma: float)
-> Tuple[float, float]:
    if tau <= 0 or sigma <= 0 or S <= 0 or K <= 0:
        return float("nan"), float("nan")
    vol_sqrt = sigma * math.sqrt(tau)
    d1 = (math.log(S / K) + (r + 0.5 * sigma * sigma) * tau) /
vol_sqrt
    d2 = d1 - vol_sqrt
    return d1, d2

def bs_call(S: float, K: float, r: float, tau: float, sigma: float) ->
float:
    d1, d2 = bs_d1_d2(S, K, r, tau, sigma)
    if not np.isfinite(d1):
        return float("nan")
    return S * norm.cdf(d1) - K * math.exp(-r * tau) * norm.cdf(d2)

def bs_put(S: float, K: float, r: float, tau: float, sigma: float) ->
float:
    d1, d2 = bs_d1_d2(S, K, r, tau, sigma)
    if not np.isfinite(d1):
        return float("nan")
    return K * math.exp(-r * tau) * norm.cdf(-d2) - S * norm.cdf(-d1)

def bs_vega(S: float, K: float, r: float, tau: float, sigma: float) ->
float:
    d1, _ = bs_d1_d2(S, K, r, tau, sigma)
    if not np.isfinite(d1):
        return float("nan")
    return S * norm.pdf(d1) * math.sqrt(tau)

def bs_delta_call(S: float, K: float, r: float, tau: float, sigma:
float) -> float:
    d1, _ = bs_d1_d2(S, K, r, tau, sigma)
    return norm.cdf(d1)

def bs_gamma(S: float, K: float, r: float, tau: float, sigma: float)
-> float:
    d1, _ = bs_d1_d2(S, K, r, tau, sigma)
    if not np.isfinite(d1) or S <= 0 or sigma <= 0 or tau <= 0:
        return float("nan")
    return norm.pdf(d1) / (S * sigma * math.sqrt(tau))


# -----------------------------
# Root finders: bisection + Newton
# -----------------------------

def bisection_root(f: Callable[[float], float],
                   lo: float,
                  hi: float,
                  tol: float = 1e-6,
                  max_iter: int = 200) -> Optional[float]:
    flo = f(lo)
    fhi = f(hi)
    if not (np.isfinite(flo) and np.isfinite(fhi)):
        return None
    # Need opposite signs
    if flo == 0:
        return lo
    if fhi == 0:
        return hi
    if flo * fhi > 0:
        return None

    for _ in range(max_iter):
        mid = 0.5 * (lo + hi)
        fmid = f(mid)
        if not np.isfinite(fmid):
            return None
        if abs(fmid) < tol or (hi - lo) / 2 < tol:
            return mid
        if flo * fmid <= 0:
            hi, fhi = mid, fmid
        else:
            lo, flo = mid, fmid
    return 0.5 * (lo + hi)

def newton_root(f: Callable[[float], float],
               fp: Callable[[float], float],
               x0: float,
               tol: float = 1e-6,
               max_iter: int = 100) -> Optional[float]:
    x = x0
    for _ in range(max_iter):
        fx = f(x)
        if not np.isfinite(fx):
            return None
        if abs(fx) < tol:
            return x
        fpx = fp(x)
        if not np.isfinite(fpx) or abs(fpx) < 1e-12:
            return None
        step = fx / fpx
        x_new = x - step
        # keep sigma in sane bounds
        if x_new <= 1e-6:
            x_new = 1e-6
        if x_new > 5.0:
            x_new = 5.0
        if abs(x_new - x) < tol:
            return x_new
        x = x_new
    return x


# ----------------------------
# Part 1: Data download + cleaning
# ----------------------------

@dataclass
class SnapshotInfo:
    tag: str
    timestamp: str
    asof_date: date
    tsla_spot: float
    spy_spot: float
    vix_level: float
    r_annual: float

def get_fed_funds_effective_rate() -> Optional[float]:
    """
    Attempts to fetch Fed funds effective rate from FRED (simple 
endpoint).
    If it fails, returns None and you can fill manually.
    """
    # This is a public CSV endpoint for FRED series DFF (Effective 
Federal Funds Rate)
    # It returns historical data; we take the last non-NaN.
    url = "https://fred.stlouisfed.org/graph/fredgraph.csv?id=DFF"
    try:
        resp = requests.get(url, timeout=15)
        resp.raise_for_status()
        lines = resp.text.strip().splitlines()
        # header: DATE,DFF
        last_val = None
        for line in lines[1:]:
            parts = line.split(",")
            if len(parts) != 2:
                continue
            v = parts[1].strip()
            if v != "." and v != "":
                last_val = float(v)
        if last_val is None:
            return None
        return last_val / 100.0  # convert percent to decimal
    except Exception:
        return None

def get_spot_and_chain(symbol: str, expiries: List[str]) ->
```

```
Tuple[float, pd.DataFrame]:
    """
    Returns spot price and a combined options chain for given
expiries.
    Columns include: symbol, expiry, type, strike, bid, ask, volume,
openInterest, lastPrice, impliedVolatility(yahoo)
    """
    t = yf.Ticker(symbol)

    # Spot: use fast_info if available, else history
    spot = None
    try:
        spot = float(t.fast_info["last_price"])
    except Exception:
        hist = t.history(period="1d", interval="1m")
        if len(hist) > 0:
            spot = float(hist["Close"].iloc[-1])
    if spot is None or not np.isfinite(spot):
        raise RuntimeError(f"Could not fetch spot for {symbol}")

    frames = []
    for exp in expiries:
        oc = t.option_chain(exp)
        calls = oc.calls.copy()
        calls["type"] = "call"
        puts = oc.puts.copy()
        puts["type"] = "put"
        df = pd.concat([calls, puts], ignore_index=True)
        df["expiry"] = exp
        df["symbol"] = symbol
        frames.append(df)

    chain = pd.concat(frames, ignore_index=True)

    # Clean / standardize columns
    keep = ["symbol", "expiry", "type", "strike", "bid", "ask",
"volume", "openInterest", "lastPrice", "impliedVolatility"]
    chain = chain[keep].copy()

    # remove duplicates
    chain = chain.drop_duplicates(subset=["symbol", "expiry", "type",
"strike"], keep="last")

    # ensure numeric
    for c in ["strike", "bid", "ask", "volume", "openInterest",
"lastPrice", "impliedVolatility"]:
        chain[c] = pd.to_numeric(chain[c], errors="coerce")

    return spot, chain
```

```
def option_mid_price(row: pd.Series) -> Optional[float]:
    bid, ask = row["bid"], row["ask"]
    vol = row.get("volume", np.nan)
    if not (np.isfinite(bid) and np.isfinite(ask)):
        return None
    if bid <= 0 and ask <= 0:
        return None
    # HW requires bid/ask exist and corresponding volume is nonzero
    if not np.isfinite(vol) or vol <= 0:
        return None
    return 0.5 * (bid + ask)

def compute_iv_for_chain(chain: pd.DataFrame, S: float, r: float,
asof: date,
                         tol: float = 1e-6) -> pd.DataFrame:
    """
    Computes implied vol using bisection and Newton for each row with
valid mid price.
    Adds columns: tau, mid, iv_bisect, iv_newton, it_money, moneyness
    """
    out = chain.copy()
    out["mid"] = out.apply(option_mid_price, axis=1)
    out["tau"] = out["expiry"].apply(lambda e: yearfrac_act365(asof,
parse_yyyy_mm_dd(e)))
    out["moneyness"] = S / out["strike"]

    def price_fn(opt_type: str, K: float, tau: float, sigma: float) ->
float:
        if opt_type == "call":
            return bs_call(S, K, r, tau, sigma)
        else:
            return bs_put(S, K, r, tau, sigma)

    iv_b = []
    iv_n = []
    for _, row in out.iterrows():
        mid = row["mid"]
        K = float(row["strike"])
        tau = float(row["tau"])
        opt_type = row["type"]

        if mid is None or not np.isfinite(mid) or mid <= 0 or tau <= 0
or K <= 0:
            iv_b.append(np.nan)
            iv_n.append(np.nan)
            continue

        f = lambda sig: price_fn(opt_type, K, tau, sig) - mid
        # Bisection bracket: very low vol to high vol
        lo, hi = 1e-6, 5.0
        root_b = bisection_root(f, lo, hi, tol=tol)

        iv_b.append(root_b if root_b is not None else np.nan)

        # Newton: start guess either bisection result if exists else
0.3
        x0 = float(root_b) if (root_b is not None and
np.isfinite(root_b)) else 0.30
        fp = lambda sig: bs_vega(S, K, r, tau, sig)
        root_n = newton_root(f, fp, x0, tol=tol)

        iv_n.append(root_n if root_n is not None else np.nan)

    out["iv_bisect"] = iv_b
    out["iv_newton"] = iv_n
    return out

def summarize_iv(iv_df: pd.DataFrame, S: float, moneyness_band:
Tuple[float, float]=(0.95, 1.05)) -> pd.DataFrame:
    """
    Builds a summary table by symbol/expiry/type:
    - ATM strike (closest to S)
    - ATM IV
    - average IV in moneyness band
    """
    rows = []
    for (sym, exp, typ), g in iv_df.groupby(["symbol", "expiry",
"type"]):
        gg = g.dropna(subset=["iv_bisect"])
        if len(gg) == 0:
            continue

        # ATM: strike closest to spot
        atm_idx = (gg["strike"] - S).abs().idxmin()
        atm_row = gg.loc[atm_idx]

        band_lo, band_hi = moneyness_band
        band = gg[(gg["moneyness"] >= band_lo) & (gg["moneyness"] <=
band_hi)]
        avg_iv = float(band["iv_bisect"].mean()) if len(band) else
float("nan")

        rows.append({
            "symbol": sym,
            "expiry": exp,
            "type": typ,
            "spot_S": S,
            "atm_strike": float(atm_row["strike"]),
            "atm_iv_bisect": float(atm_row["iv_bisect"]),
            "avg_iv_band_bisect": avg_iv,
            "n_opts_used": int(len(gg)),
            "n_in_band": int(len(band)),
        })
    return pd.DataFrame(rows).sort_values(["symbol", "expiry",
"type"])

def put_call_parity_checks(iv_df: pd.DataFrame, S: float, r: float,
asof: date) -> pd.DataFrame:
    """
    For each expiry/strike, compare parity-implied prices with
observed mids when possible.
    Uses mid price for existing option side.
    """
    df = iv_df.copy()
    df["K"] = df["strike"]
    df["tau"] = df["expiry"].apply(lambda e: yearfrac_act365(asof,
parse_yyyy_mm_dd(e)))

    # Pivot mids
    piv = df.pivot_table(index=["symbol", "expiry", "K"],
columns="type", values="mid", aggfunc="first").reset_index()
    piv["discK"] = piv["K"] * np.exp(-r * piv["expiry"].apply(lambda
e: yearfrac_act365(asof, parse_yyyy_mm_dd(e))))

    # If call exists, parity-implied put = C - S + K e^{-r tau}
    piv["put_from_call"] = piv["call"] - S + piv["discK"] if "call" in
piv.columns else np.nan
    # If put exists, parity-implied call = P + S - K e^{-r tau}
    piv["call_from_put"] = piv["put"] + S - piv["discK"] if "put" in
piv.columns else np.nan

    # Compare to available mids
    piv["call_parity_error"] = np.where(np.isfinite(piv.get("call",
np.nan)) & np.isfinite(piv["call_from_put"]),
                                        piv["call_from_put"] -
piv["call"], np.nan)
    piv["put_parity_error"] = np.where(np.isfinite(piv.get("put",
np.nan)) & np.isfinite(piv["put_from_call"]),
                                       piv["put_from_call"] -
piv["put"], np.nan)
    return piv

def plot_vol_smile(iv_df: pd.DataFrame, symbol: str, outdir: str) ->
None:
    """
    2D plot of implied vol vs strike for the closest maturity, and
overlay 3 maturities.
    Uses iv_bisect.
    """
    d = iv_df[iv_df["symbol"] ==
symbol].dropna(subset=["iv_bisect"]).copy()
    if len(d) == 0:
        return
    expiries = sorted(d["expiry"].unique())
    if len(expiries) == 0:
        return

    closest = expiries[0]
    dd = d[(d["expiry"] == closest) & (d["type"] ==
"call")].sort_values("strike")
    plt.figure()
    plt.plot(dd["strike"], dd["iv_bisect"], marker="o",
linestyle="none")
    plt.xlabel("Strike K")
    plt.ylabel("Implied Vol (bisection)")
    plt.title(f"{symbol} IV Smile (closest expiry {closest}) - Calls")
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(os.path.join(outdir, f"{symbol}
_iv_smile_closest.png"), dpi=160)
    plt.close()

    # Overlay 3 maturities for calls
    plt.figure()
    for exp in expiries[:3]:
        dd = d[(d["expiry"] == exp) & (d["type"] ==
"call")].sort_values("strike")
        plt.plot(dd["strike"], dd["iv_bisect"], marker="o",
linestyle="none", label=exp)
    plt.xlabel("Strike K")
    plt.ylabel("Implied Vol (bisection)")
    plt.title(f"{symbol} IV Smile - Calls (3 maturities)")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(os.path.join(outdir, f"{symbol}_iv_smile_3m.png"),
dpi=160)
    plt.close()


# -----------------------------
# Part 11: Greeks (analytic vs finite diff)
# -----------------------------

def finite_diff_greeks_call(S: float, K: float, r: float, tau: float,
sigma: float,
                            hS: float = 0.01, hsig: float = 1e-4) ->
Dict[str, float]:
    """
    Central differences. hS is absolute dollars; hsig is absolute vol
```

```
units.
    """
    C0 = bs_call(S, K, r, tau, sigma)
    Cp = bs_call(S + hS, K, r, tau, sigma)
    Cm = bs_call(S - hS, K, r, tau, sigma)
    delta = (Cp - Cm) / (2 * hS)
    gamma = (Cp - 2 * C0 + Cm) / (hS * hS)

    Cvp = bs_call(S, K, r, tau, sigma + hsig)
    Cvm = bs_call(S, K, r, tau, sigma - hsig)
    vega = (Cvp - Cvm) / (2 * hsig)
    return {"delta_fd": delta, "gamma_fd": gamma, "vega_fd": vega}

def greeks_table_from_iv(iv_df: pd.DataFrame, S: float, r: float,
asof: date,
                         use: str = "iv_bisect", max_rows: int = 40)
-> pd.DataFrame:
    """
    Computes analytic and FD Greeks for calls, using implied vols.
    """
    d = iv_df[(iv_df["type"] == "call")].dropna(subset=[use]).copy()
    d["tau"] = d["expiry"].apply(lambda e: yearfrac_act365(asof,
parse_yyyy_mm_dd(e)))
    d = d[(d["tau"] > 0) & np.isfinite(d["tau"])].copy()

    # sample rows (avoid enormous outputs)
    d = d.sort_values(["expiry", "strike"]).head(max_rows)

    rows = []
    for _, row in d.iterrows():
        K = float(row["strike"])
        tau = float(row["tau"])
        sig = float(row[use])

        delta = bs_delta_call(S, K, r, tau, sig)
        gamma = bs_gamma(S, K, r, tau, sig)
        vega = bs_vega(S, K, r, tau, sig)

        fd = finite_diff_greeks_call(S, K, r, tau, sig)

        rows.append({
            "symbol": row["symbol"],
            "expiry": row["expiry"],
            "K": K,
            "tau": tau,
            "sigma": sig,
            "delta_analytic": delta,
            "gamma_analytic": gamma,
            "vega_analytic": vega,
            **fd,
            "delta_diff": fd["delta_fd"] - delta,
            "gamma_diff": fd["gamma_fd"] - gamma,
            "vega_diff": fd["vega_fd"] - vega,
        })
    return pd.DataFrame(rows)


# -----------------------------
# Part 3: AMM expected fee revenue (trapezoid)
# -----------------------------

def amm_swap_case1(S1: float, x: float, y: float, gamma: float, k:
float) -> Tuple[float, float, float]:
    """
    Case 1: S1 > P/(1-gamma)
    Derived:
      x1 = sqrt(k / (S1*(1-gamma)))
      y1 = sqrt(k * S1*(1-gamma))
      Δx = x - x1
      Δy = (y1 - y)/(1-gamma)
      R = gamma*Δy
    Returns (dx, dy, R)
    """
    x1 = math.sqrt(k / (S1 * (1 - gamma)))
    y1 = math.sqrt(k * S1 * (1 - gamma))
    dx = x - x1
    dy = (y1 - y) / (1 - gamma)
    R = gamma * dy
    return dx, dy, R

def amm_swap_case2(S1: float, x: float, y: float, gamma: float, k:
float) -> Tuple[float, float, float]:
    """
    Case 2: S1 < P*(1-gamma)
    Derived:
      x1 = sqrt(k*(1-gamma)/S1)
      y1 = sqrt(k*S1/(1-gamma))
      Δx = (x1 - x)/(1-gamma)
      Δy = y - y1
      R = gamma*Δx*S1   (BTC fee converted to USDC)
    Returns (dx, dy, R)
    """
    x1 = math.sqrt(k * (1 - gamma) / S1)
    y1 = math.sqrt(k * S1 / (1 - gamma))
    dx = (x1 - x) / (1 - gamma)
    dy = y - y1
    R = gamma * dx * S1
    return dx, dy, R

def lognormal_pdf(s: np.ndarray, S0: float, sigma: float, dt: float)
```

```
-> np.ndarray:
    # S1 = S0 * exp((-0.5 sigma^2 dt) + sigma sqrt(dt) Z)
    # ln(S1) ~ N(ln(S0) - 0.5 sigma^2 dt, sigma^2 dt)
    mu = math.log(S0) - 0.5 * sigma * sigma * dt
    var = sigma * sigma * dt
    # pdf: 1/(s sqrt(2π var)) exp(-(ln s - mu)^2/(2 var))
    return (1.0 / (s * math.sqrt(2 * math.pi * var))) * np.exp(-
((np.log(s) - mu) ** 2) / (2 * var))

def trapezoid_integral(x: np.ndarray, y: np.ndarray) -> float:
    return float(np.trapz(y, x))

def expected_fee_revenue_trap(sigma: float, gamma: float,
                              x: float = 1000.0, y: float = 1000.0,
                              S0: float = 1.0, dt: float = 1/365) ->
float:
    """
    Computes E[R(S1)] with trapezoidal integration, matching the
assignment’s integrals.
    We integrate piecewise beyond no-arb band:
      upper region: S1 > P/(1-gamma)
      lower region: S1 < P*(1-gamma)
    """
    k = x * y
    P = y / x
    upper = P / (1 - gamma)
    lower = P * (1 - gamma)

    # Build integration grids.
    # Cover almost all lognormal mass by going +/- 8 std dev in log
space.
    sd = sigma * math.sqrt(dt)
    mu = math.log(S0) - 0.5 * sigma * sigma * dt

    s_min = math.exp(mu - 8 * sd)
    s_max = math.exp(mu + 8 * sd)

    # Dense grid for stability
    grid = np.linspace(s_min, s_max, 40000)
    pdf = lognormal_pdf(grid, S0=S0, sigma=sigma, dt=dt)

    # Revenue function on grid
    R = np.zeros_like(grid)

    # Case 1 region
    mask1 = grid > upper
    if np.any(mask1):
        for i in np.where(mask1)[0]:
            _, _, Ri = amm_swap_case1(float(grid[i]), x, y, gamma, k)
            R[i] = Ri
    # Case 2 region
    mask2 = grid < lower
    if np.any(mask2):
        for i in np.where(mask2)[0]:
            _, _, Ri = amm_swap_case2(float(grid[i]), x, y, gamma, k)
            R[i] = Ri

    return trapezoid_integral(grid, R * pdf)

def amm_table_and_plot(outdir: str) -> Tuple[pd.DataFrame,
pd.DataFrame]:
    sigmas = [0.2, 0.6, 1.0]
    gammas = [0.001, 0.003, 0.01]

    rows = []
    for sig in sigmas:
        vals = []
        for g in gammas:
            ER = expected_fee_revenue_trap(sig, g)
            vals.append(ER)
            rows.append({"sigma": sig, "gamma": g, "E[R]": ER})
    df = pd.DataFrame(rows)

    # Best gamma per sigma
    best = df.loc[df.groupby("sigma")
["E[R]"].idxmax()].sort_values("sigma").reset_index(drop=True)

    # Grid sigma -> optimal gamma
    sigma_grid = np.round(np.arange(0.10, 1.001, 0.01), 2)
    opt_rows = []
    for sig in sigma_grid:
        best_g = None
        best_er = -1.0
        for g in gammas:
            er = expected_fee_revenue_trap(float(sig), g)
            if er > best_er:
                best_er = er
                best_g = g
        opt_rows.append({"sigma": float(sig), "gamma_star": best_g,
"best_E[R]": best_er})
    opt = pd.DataFrame(opt_rows)

    # Plot sigma vs gamma*
    plt.figure()
    plt.plot(opt["sigma"], opt["gamma_star"], marker="o",
linestyle="-")
    plt.xlabel("Volatility σ")
    plt.ylabel("Optimal fee γ* (among {0.001,0.003,0.01})")
    plt.title("AMM Optimal Fee vs Volatility")
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(os.path.join(outdir, "amm_sigma_vs_gamma_star.png"),
dpi=160)
    plt.close()

    return df, best, opt


# -----------------------------
# Main runner
# -----------------------------

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--tag", type=str, default="DATA1",
help="DATA1 or DATA2 tag for saved outputs")
    parser.add_argument("--outdir", type=str, default="output",
help="output directory")
    parser.add_argument("--moneyness_lo", type=float, default=0.95)
    parser.add_argument("--moneyness_hi", type=float, default=1.05)
    parser.add_argument("--tol", type=float, default=1e-6)
    args = parser.parse_args()

    os.makedirs(args.outdir, exist_ok=True)

    now = datetime.now()
    asof = now.date()
    ts = now.strftime("%Y-%m-%d_%H-%M-%S")

    # Interest rate
    r = get_fed_funds_effective_rate()
    if r is None:
        print("WARNING: Could not fetch Fed funds effective rate
automatically.")
        print("         Set r manually in the code or rerun with
internet access.")
        r = 0.05  # fallback for practice

    # Pull ^VIX level
    vix_t = yf.Ticker("^VIX")
    try:
        vix_level = float(vix_t.fast_info["last_price"])
    except Exception:
        vix_hist = vix_t.history(period="1d", interval="1m")
        vix_level = float(vix_hist["Close"].iloc[-1]) if len(vix_hist)
else float("nan")

    # Pull TSLA + SPY chains (next 3 monthly expiries)
    # (Yahoo option dates vary; we filter to 3rd Friday monthly)
    tsla_t = yf.Ticker("TSLA")
    spy_t = yf.Ticker("SPY")

    tsla_exp = 
pick_next_three_monthly_expirations(list(tsla_t.options), asof)
    spy_exp = pick_next_three_monthly_expirations(list(spy_t.options), 
asof)

    if len(tsla_exp) < 3 or len(spy_exp) < 3:
        print("WARNING: Could not find 3 monthly (3rd Friday) 
expirations for TSLA/SPY.")
        print("         Using first 3 available expirations instead.")
        tsla_exp = list(tsla_t.options)[:3]
        spy_exp = list(spy_t.options)[:3]

    tsla_spot, tsla_chain = get_spot_and_chain("TSLA", tsla_exp)
    spy_spot, spy_chain = get_spot_and_chain("SPY", spy_exp)

    # Save raw chains
    tsla_chain.to_csv(os.path.join(args.outdir, f"{args.tag}
_TSLA_chain_raw_{ts}.csv"), index=False)
    spy_chain.to_csv(os.path.join(args.outdir, f"{args.tag}
_SPY_chain_raw_{ts}.csv"), index=False)

    # Compute implied vols
    tsla_iv = compute_iv_for_chain(tsla_chain, S=tsla_spot, r=r, 
asof=asof, tol=args.tol)
    spy_iv = compute_iv_for_chain(spy_chain, S=spy_spot, r=r, 
asof=asof, tol=args.tol)
    iv_all = pd.concat([tsla_iv, spy_iv], ignore_index=True)

    iv_all.to_csv(os.path.join(args.outdir, f"{args.tag}
_iv_all_{ts}.csv"), index=False)

    # Summary tables
    sum_tsla = summarize_iv(tsla_iv, S=tsla_spot, 
moneyness_band=(args.moneyness_lo, args.moneyness_hi))
    sum_spy = summarize_iv(spy_iv, S=spy_spot, 
moneyness_band=(args.moneyness_lo, args.moneyness_hi))
    summary = pd.concat([sum_tsla, sum_spy], ignore_index=True)
    summary.to_csv(os.path.join(args.outdir, f"{args.tag}
_iv_summary_{ts}.csv"), index=False)

    # Put-call parity checks (using IV table mids)
    parity_tsla = put_call_parity_checks(tsla_iv, S=tsla_spot, r=r, 
asof=asof)
    parity_spy = put_call_parity_checks(spy_iv, S=spy_spot, r=r, 
asof=asof)
    pd.concat([parity_tsla, parity_spy], ignore_index=True).to_csv(
        os.path.join(args.outdir, f"{args.tag}
_put_call_parity_{ts}.csv"), index=False
    )

    # Vol smile plots
    plot_vol_smile(tsla_iv, "TSLA", args.outdir)
    plot_vol_smile(spy_iv, "SPY", args.outdir)

    # Greeks table (sample calls)
    greeks_tsla = greeks_table_from_iv(tsla_iv, S=tsla_spot, r=r,
asof=asof, use="iv_bisect", max_rows=40)
    greeks_spy = greeks_table_from_iv(spy_iv, S=spy_spot, r=r,
asof=asof, use="iv_bisect", max_rows=40)
    pd.concat([greeks_tsla, greeks_spy], ignore_index=True).to_csv(
        os.path.join(args.outdir, f"{args.tag}
_greeks_compare_{ts}.csv"), index=False
    )

    # Snapshot info
    snap = SnapshotInfo(
        tag=args.tag,
        timestamp=ts,
        asof_date=asof,
        tsla_spot=tsla_spot,
        spy_spot=spy_spot,
        vix_level=vix_level,
        r_annual=r,
    )
    snap_df = pd.DataFrame([snap.__dict__])
    snap_df.to_csv(os.path.join(args.outdir, f"{args.tag}
_snapshot_{ts}.csv"), index=False)

    # AMM section (practice numbers independent of market download)
    amm_df, amm_best, amm_opt = amm_table_and_plot(args.outdir)
    amm_df.to_csv(os.path.join(args.outdir, f"amm_ER_table_{ts}.csv"),
index=False)
    amm_best.to_csv(os.path.join(args.outdir,
f"amm_best_gamma_{ts}.csv"), index=False)
    amm_opt.to_csv(os.path.join(args.outdir,
f"amm_sigma_grid_gamma_star_{ts}.csv"), index=False)

    # Print a quick console summary
    print("\n=== SNAPSHOT ===")
    print(snap_df.to_string(index=False))
    print("\n=== IV SUMMARY (head) ===")
    print(summary.head(12).to_string(index=False))

    print("\n=== AMM E[R] (sigma in {0.2,0.6,1.0}, gamma in
{0.001,0.003,0.01}) ===")
    print(amm_df.pivot_table(index="sigma", columns="gamma",
values="E[R]").to_string())
```

```
    print("\n=== AMM BEST GAMMA ===")
    print(amm_best.to_string(index=False))

    print(f"\nSaved outputs to: {os.path.abspath(args.outdir)}")


if __name__ == "__main__":
    main()
```