Light DQM — Refactor & Streamlining Plan

Below is a practical, step-by-step breakdown of your script into a clean, testable package with reusable components, less duplication, and clearer data flow. It includes a proposed folder layout, core APIs, and targeted refactors with concrete examples.

---

1) Goals

- **Separation of concerns:** I/O, computation, plotting, reporting, CLI.
- **DRY plotting:** one generic plotting path parameterized for different metrics (noise, baseline, clipped, neg).
- **Testability:** pure functions with small signatures and typed inputs/outputs.
- **Performance:** avoid repeated conversions; precompute masks/ROIs; reduce memory churn.
- **Robustness:** structured logging, consistent error/exception boundaries; replace ad-hoc JSONL with schema'd artifacts.

---

2) Proposed Package Layout

```
light_dqm/
├─ __init__.py
├─ cli.py                   # Typer/argparse definitions only
├─ config.py                # Dataclasses & defaults loaded from CLI or YAML
├─ io.py                    # HDF5 reads, channel status, file discovery
├─ compute.py               # Core math: baselines, noise, masks, spectra, CP
intervals
├─ grouping.py              # Channel group logic (channels, EPCB/TPC groupers)
├─ plot.py                  # Generic plotting utilities + specific figure
builders
├─ grafana.py               # PNG mask renderers & legend helpers
├─ report.py                # PDF merge / run-args page
├─ persist.py               # Parquet/NPZ write+read (prev runs), versioning
├─ pipeline.py              # Orchestrates a single file run (pure-ish)
└─ main.py                  # Program entry (calls cli -> pipeline)
```

(Optional) `tests/` with pytest; `bin/light-dqm` as console entrypoint.

---

3) Config & Types (remove globals at import time)

```python
# config.py
from dataclasses import dataclass
from pathlib import Path
from enum import Enum
```

```python
class Units(str, Enum):
    ADC16 = "ADC16"
    ADC14 = "ADC14"
    V = "V"

@dataclass(frozen=True)
class Paths:
    input_path: Path
    output_dir: Path
    temp_dir: Path
    channel_status_csv: Path

@dataclass(frozen=True)
class RunConfig:
    file_syntax: str
    units: Units = Units.ADC16
    ptps16bit: int = 500
    start_run: int = 0
    nfiles: int = 1
    ncomp: int = -1
    powspec_nevts: int = 500
    max_evts: int = 500
    write_json_blobs: bool = False
    merge_grafana_plots: bool = False
    plot_all_clipped: bool = False
    plot_all_negatives: bool = False
```

Load from CLI (Typer) or YAML → create `Paths` + `RunConfig`. Never parse args at import.

---

4) I/O & Discovery

```python
# io.py
from pathlib import Path
import h5py

def discover_files(root: Path, prefix: str) -> list[Path]:
    return sorted(p for p in root.iterdir() if p.name.startswith(prefix) and
p.suffixes[-2:] == [".FLOW", ".hdf5"])  # .FLOW.hdf5

class LightFile:
    def __init__(self, path: Path):
        self.path = path
        self._h5 = h5py.File(path, "r")
    def events(self):
        return self._h5["light/events/data"]
    def waveforms(self):
        return self._h5["light/wvfm/data"]["samples"]
```

```
    def close(self):
        self._h5.close()
    # Context manager support
    def __enter__(self): return self
    def __exit__(self, *_): self.close()
```

---

5) Grouping (single source of truth)

```
# grouping.py
import numpy as np

N_ADCS = 8
N_CH = 64
SAMPLES = 1000
SAMPLE_DT_US = 0.016

# Channels 4–15 in each block of 16 (as in your code)
CHANNELS_ACTIVE = np.concatenate([np.arange(s+4, min(s+16, 64)) for s in
range(0, 64, 16)])

# EPCB groups: 8 groups × 6 channels
EPCB_GROUPS = [np.arange(i*6, (i+1)*6) for i in range(len(CHANNELS_ACTIVE)//
6)]

# TPC pairing rule (encode once here)
def tpc_pairs_for_adc(i_adc: int) -> tuple[int, int]:
    d = 1 if i_adc % 2 == 0 else -1
    return (i_adc, i_adc + d)
```

All EPCB/TPC logic lives here—plotting/computation just asks for indices.

---

6) Core Computation API (unified + typed)

```
# compute.py
from __future__ import annotations
import numpy as np
from scipy.fft import rfft, rfftfreq
from scipy.stats import beta
from .config import Units
from .grouping import SAMPLES, SAMPLE_DT_US

ADC14_MAX = 8191
ADC14_TO_16 = 4
ADC_V_RANGE = 2.0
ADC_V_OFFSET = -1.0
```

```
@dataclass(frozen=True)
class Stats:
    centre: np.ndarray  # (8,64)
    err_low: np.ndarray # (8,64)
    err_up: np.ndarray  # (8,64)

# Units & thresholds

def ptp_thresholds(units: Units, ptps16bit: int) -> np.ndarray:  # (8,)
    base = np.repeat(ptps16bit, 8).astype(float)
    if units == Units.ADC16: return base
    if units == Units.ADC14: return base / ADC14_TO_16
    if units == Units.V:     return base * (ADC_V_RANGE) / (ADC14_MAX *
ADC14_TO_16)
    raise ValueError(units)

# Conversions are pure & vectorized

def to_units(adc16: np.ndarray, units: Units) -> np.ndarray:
    if units == Units.ADC16: return adc16
    if units == Units.ADC14: return adc16 / ADC14_TO_16
    if units == Units.V:     return adc16 * (ADC_V_RANGE + ADC_V_OFFSET) /
(ADC14_MAX * ADC14_TO_16)
    raise ValueError(units)

# Waveform features (mask events via indices, never mutate input)

def features(wv: np.ndarray, units: Units, event_mask: np.ndarray | None,
ths: np.ndarray) -> tuple:
    sel = wv[event_mask] if event_mask is not None else wv
    wf = to_units(sel, units)
    noise = np.std(wf[..., :50], axis=-1)
    baseline = np.mean(wf[..., :50], axis=-1)
    max_val = np.max(wf - baseline[..., None], axis=-1)
    clipped = np.any(wf >= (ADC14_MAX-1)*ADC14_TO_16, axis=-1)
    negs = np.any(wf - baseline[..., None] < -ths[None, :, None, None],
axis=-1)
    return wf, noise, baseline, max_val, clipped, negs

# Clopper-Pearson (safe at edges)

def clopper_pearson(passed: np.ndarray, total: np.ndarray, interval: float =
0.68) -> Stats:
    alpha = 1 - interval
    lower = beta.ppf(alpha/2, passed, total - passed + 1)
    upper = beta.ppf(1 - alpha/2, passed + 1, total - passed)
    frac = np.divide(passed, total, out=np.zeros_like(passed, dtype=float),
where=total>0)
    lower = np.nan_to_num(lower, nan=0.0)
    upper = np.nan_to_num(upper, nan=1.0)
```

```
    pct = 100*frac
    err_lo = np.where((total==0)|(pct==0), 0.0, 100*(frac - lower))
    err_up = 100*(upper - frac)
    return Stats(pct, err_lo, err_up)

# Spectra (return freq + central/quantiles)

def spectra(wf_adc16: np.ndarray, nevts: int) -> tuple[np.ndarray,
np.ndarray]:
    wf = wf_adc16[:nevts]
    fft_N = rfft(wf, axis=-1) / (SAMPLES/2 + 1)
    power = 2*np.abs(fft_N)**2
    freq = rfftfreq(SAMPLES, d=SAMPLE_DT_US*1e-6)
    median = np.nanmedian(power, axis=0)
    return freq, median
```

This consolidates duplicated math and makes everything return consistent shapes.

---

7) One Generic “Bar-Error” Plot Builder

```
# plot.py
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages
from dataclasses import dataclass
import numpy as np

@dataclass
class Series:
    y: np.ndarray        # (64,)
    err_lo: np.ndarray   # (64,)
    err_up: np.ndarray   # (64,)
    alpha: np.ndarray    # (64,)
    label: str

def points_with_error(ax, x, series: Series):
    for idx in x:
        ax.errorbar(idx, series.y[idx], yerr=[[max(series.err_lo[idx],0)],
[max(series.err_up[idx],0)]],
                    fmt='.', markersize=5, linewidth=0.5, capsize=4,
color='C0', alpha=series.alpha[idx])

def step_band(ax, x_span, y, lo, up, alpha=0.25):
    ax.step([x_span[0], x_span[1]], [y, y], where='post', color='C3',
alpha=alpha, linewidth=1)
    ax.fill_between([x_span[0], x_span[1]], [y-lo, y-lo], [y+up, y+up],
step='post', alpha=alpha/2, color='C3')

def save_pdf(fig, out_path):
```

```
    with PdfPages(out_path) as pdf:
        pdf.savefig(fig)
    plt.close(fig)
```

All four “fraction” plots (total/TPC/EPCB/channel) become thin wrappers that compute their denominators and call this generic layer.

---

8) Collapse 4× “clipped/neg fraction” into One Function

```
# pipeline helpers (pseudo)
from .compute import clopper_pearson
from .grouping import CHANNELS_ACTIVE

def fraction_by(granularity: str, event_flags: np.ndarray, max_vals:
np.ndarray, ths: np.ndarray) -> Stats:
    """granularity ∈ {"channel","epcb","tpc","total"}.
    Returns Stats over (8,64) so plotting pipeline doesn’t change."""
    # compute passed per-channel
    passed = event_flags.sum(axis=0)   # (8,64)
    # compute totals per-channel depending on granularity
    totals = np.zeros_like(passed)
    if granularity == "channel":
        totals = (max_vals > ths[None, :, None]).sum(axis=0)
    elif granularity == "epcb":
        # fill totals for each epcb’s 6 channels based on "any over PTP in
that EPCB"
        # (use grouping.EPCB_GROUPS)
        ...
    elif granularity == "tpc":
        # fill totals based on paired ADC x 32 channels (use pairing helper)
        ...
    elif granularity == "total":
        totals[...] = event_flags.shape[0]
    else:
        raise ValueError(granularity)
    return clopper_pearson(passed, totals)
```

This removes three near-duplicate functions for clipped and another two for negatives.

---

9) Grafana Masks → One Grid Renderer

```
# grafana.py
from matplotlib.lines import Line2D

def grid_mask(fig, ax, mask: np.ndarray, ch_status: np.ndarray | None, title:
str, emoji: bool):
```

```
    # Render once; choose symbols via flags instead of branching inside nested loops.
    ...
```

`plot_flatline_mask` and `plot_baseline_mask` become a thin wrapper that selects legend + title + "emoji mode".

---

10) Persistence: from JSONL → Parquet/NPZ

- JSONL was fine for prototyping, but for numeric arrays (8×64), use **Parquet** (via pandas) or **NumPy NPZ** for compact, typed, and fast I/O.
- Provide `persist.write_stats(name, idx, stats)` and `persist.read_rollup(name, indices)` that return exactly the shapes the plotting code needs. This deletes repeated add-in-quadrature logic scattered in the script.

```
# persist.py
import numpy as np, pandas as pd

def write_stats_parquet(file_index: int, centre: np.ndarray, lo: np.ndarray, up: np.ndarray, path: Path):
    df = pd.DataFrame({
        "file_index": file_index,
        "adc": np.repeat(np.arange(8), 64),
        "ch": np.tile(np.arange(64), 8),
        "centre": centre.ravel(),
        "err_lo": lo.ravel(),
        "err_up": up.ravel(),
    })
    df.to_parquet(path, engine="pyarrow", compression="zstd", append=True)
```

---

11) Reporting & Merging

- Keep **PDF creation** outside plotting (single responsibility): figures return `matplotlib.figure.Figure`, and `report.py` saves/merges.
- The "args page" is a small helper that renders a monospaced list; keep it in `report.py`.

---

12) Orchestration Flow (per file)

```
# pipeline.py (sketch)

def process_one(paths: Paths, cfg: RunConfig, idx: int, prev_indices: list[int]):
    with LightFile(paths.input_path / files[idx]) as lf:
        ev = lf.events(); wf = lf.waveforms()
```

```python
        # choose event indices once
        sel = pick_indices(total=len(wf), max_evts=cfg.max_evts)
        # split by trigger
        beam_mask = (ev['trig_type'][sel] == 1)
        self_mask = (ev['trig_type'][sel] == 0)
        # thresholds
        ths = ptp_thresholds(cfg.units, cfg.ptps16bit)
        # features
        all_feats = features(wf, cfg.units, sel, ths)
        beam_feats = features(wf, cfg.units, sel[beam_mask], ths)
        self_feats = features(wf, cfg.units, sel[self_mask], ths)
        # spectra
        freq, median_pow = spectra(wf[:cfg.powspec_nevts], cfg.powspec_nevts)
        # compute stats for each view, persist if requested, then build figs
        ...
        # save figs via report helpers and return a summary object
        return Summary(...)
```

---

13) Logging & Progress

- Add `logging` with module-level loggers; avoid `print`.
- Use `tqdm` for long loops (if any remain). Most loops disappear after vectorization.

---

14) Testing Strategy (high-value targets)

- **compute.clopper_pearson**: edge cases total=0, passed=0, passed=total.
- **grouping**: EPCB/TPC mappings; `CHANNELS_ACTIVE` shape.
- **fraction_by**: unit tests per granularity with synthetic masks.
- **spectra**: known impulses → peaks at expected bins.

---

15) Quick Wins in Your Current Script

- Parse args **only** in `main()`; don't compute globals (`args = parse_args()`) at import time.
- Replace multiple `plot_*clipped*` / `plot_*neg*` with **one** function + a `granularity` enum.
- Factor **mask/legend rendering** (Grafana vs. PDF) into a single `grid_mask()` helper.
- Centralize constants (sample rate, sizes, ADC ranges) under `grouping.py` / `compute.py`.
- Replace repeated JSON read/try/except blocks with a tiny `persist.read_rollup()`.
- Have plotting functions **return the figure** and not call `plt.show()` inside the library; `main` / `report` decides to show/save.

---

16) Example: One “Clipped Fraction by EPCB” in New API

```
# in pipeline step
stats_prev = persist.read_rollup("clipped_epcb", prev_indices)
stats_now  = fraction_by("epcb", clipped_flags, max_vals, ths)
fig = build_fraction_figure(stats_prev, stats_now, title="Beam trigger (%
clipped per EPCB)")
report.save_pdf(fig, temp_dir/"clipped_epcb.pdf")
```

---

17) Example: Unified Baseline/Flatline Mask

```
# grafana.py usage
fig, ax = plt.subplots(figsize=(16,4))
grid_mask(fig, ax, flatline_mask, ch_status, title="Alive and dead channels",
emoji=True)
report.save_png(fig, outdir/f"{stem}_light_dqm_flatline.png")
```

---

18) Performance Notes

- Avoid building large intermediate arrays repeatedly (e.g., do units conversion once per slice).
- Use `np.unique(np.linspace(...))` as you do, but compute masks once and pass around.
- FFT: limit to `powspec_nevts`, and delete large arrays (`del`) after use; or operate on memory-mapped arrays if needed.

---

19) Migration Plan (low risk)

1. Create package skeleton + move pure helpers (`clopper_pearson`, conversions, masks).
2. Port plotting to return-fig style; keep old signatures, adapt calls in `main`.
3. Replace JSONL with NPZ/Parquet under `persist` while keeping a compatibility reader for old runs.
4. Collapse duplicated fraction functions behind `fraction_by()`.
5. Introduce `pipeline.process_one()` to replace the giant `for` body.
6. Cut over `main.py` to `cli -> pipeline -> report`.

---

20) Optional Enhancements

- YAML config file (`light_dqm.yaml`) to avoid long CLI invocations.
- Rich console output (`rich`) for tables and status spinners.
- Small HTML summary (Plotly or static PNGs) in addition to PDFs.

TL;DR

This plan turns your single, long script into a tidy package with: - clean computation kernels, one generic plotting pathway, - predictable data models, - replaceable persistence layer, - and a slim main pipeline that's easy to test and maintain.

When you're ready, I can generate stub files for the modules above so you can start committing immediately.