Implementing event-based grid strength and IBR impedance estimation in WaveApps

Prepared by GPA (WaveApps platform team) from a review of the ECCE 2026 paper and the current Python adapter code (`main.py`, `data_proxy.py`, `process_data.py`, `Grid_impedance_funs.py`). The data-stream quality issue is covered separately.

Summary

Both algorithms in the paper are event-based, and that matches how WaveApps applications are meant to work: detect, analyze, report. The data path into the Python adapter is fine. The current code is shaped like a continuous streaming calculation, and in that form it cannot produce valid numbers.

Our recommendation is to keep the Python adapter, but restructure the code around events:

- A light per-second pass runs an excitation detector and keeps the RLS state current.
- When an event is detected, the pre- and post-event window goes through batch least squares (paper eq. 4) and, later, the DMD impedance stage.
- Results go back to the host as one WaveApps event with JSON details, plus Xg, Vg and SCR as low-rate (1 per second) measurements.

Point-on-wave data should never be streamed back to the host. If GE needs a continuous, sample-rate SCR trend, that part belongs in a native C# action adapter, and GPA can port it once the algorithm is validated.

What the two algorithms actually need

The paper's own logic decides the implementation shape. Equation (1), $V_{pcc} = I_{pcc} \cdot Z_g + V_g$, has two unknowns. In steady state I_{pcc} is nearly constant, so Z_g and V_g cannot be separated. The estimate only exists while the current changes.

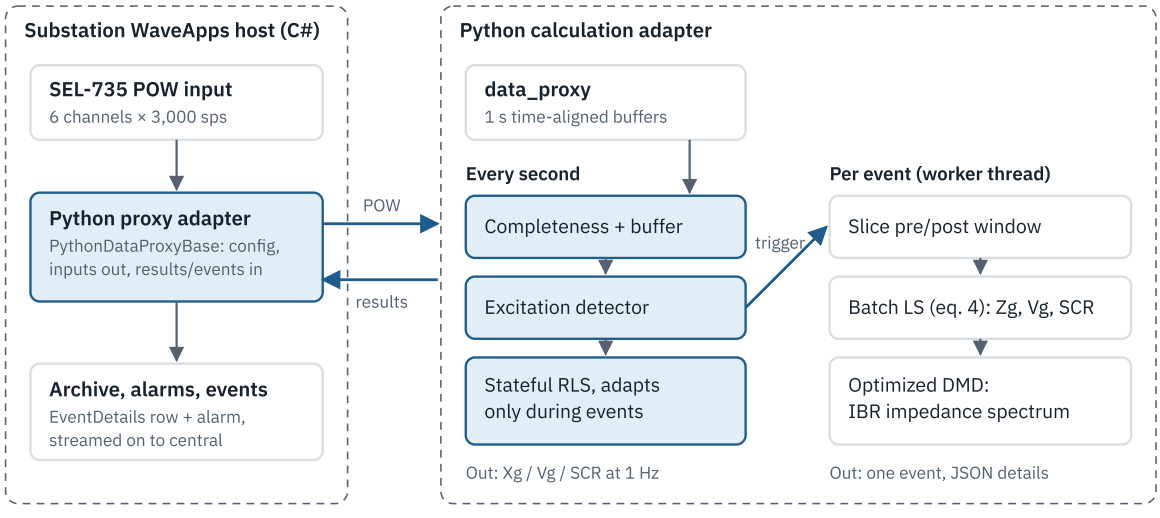
That is why the paper's results converge "immediately after the inverter starts". In the field the inverter is already running, so the estimator has to rely on events: tap changes, PQ steps, POI voltage changes or non-trip faults. The paper presents RLS (eq. 5) only as a way to reduce storage and computation for online use.

ASPECT	GRID STRENGTH (COMPLEX RLS / LS)	IBR IMPEDANCE (OPTIMIZED DMD)
Input	3-phase V and I at the POI, POW at 3 ksps	Same, converted to positive and negative sequence

ASPECT	GRID STRENGTH (COMPLEX RLS / LS)	IBR IMPEDANCE (OPTIMIZED DMD)
Needs excitation?	Yes. Unidentifiable in steady state	Yes. Needs a grid event (paper uses LVRT steps)
Computation	Tiny: 2 complex unknowns. Batch LS on a 2.5 s event window takes milliseconds	Heavy: SVD plus rank optimization over N snapshots. Seconds per event
Data requirement	Contiguous, time-aligned samples	Strictly uniform, gap-free snapshots spanning the event
Natural output	R_g, X_g, V_g, SCR per event; optionally a held 1 Hz trend	An impedance spectrum (|Z|, ∠Z versus frequency): a curve, not a time-series point
WaveApps mapping	Event details JSON plus low-rate measurements	Event details JSON
Best runtime	Python per event; C# only if a sample-rate trend is required	Python (NumPy/SciPy) on a worker thread

Recommended shape inside WaveApps

The platform's design intent applies here. POW is analyzed in the substation, and only low-resolution results and events travel to the central instance, over STTP (IEEE 2664). The Python adapter receives one-second, time-aligned buffers and is called once per second. If a call is still running when the next second is ready, that second is skipped. So per-second work must stay small, and heavy work runs per event, off the per-second path.



Python adapter or native C# adapter?

The worry is that the work turns into a streaming calculation that produces streaming results, with POW going back and forth. Analyzed as a set of events, the algorithm doesn't need that:

REQUIREMENT	FIT	WHERE IT SHOULD RUN
Grid strength per detected event (Zg, Vg, SCR, quality)	Good fit	Python, batch LS on the event window
Held 1 Hz SCR / Xg trend, updated at events	Good fit	Python, stateful RLS. About 30 ms of work per second in our test
IBR impedance spectrum per event	Good fit	Python, DMD on a worker thread
Continuous sample-rate (or sub-cycle) estimate trajectory streamed to the host	Poor fit	Native C# action adapter. Results never leave the host process, and there's no transport hop or Python per-measurement overhead
Many POIs or sites on one substation computer	Watch	Each Python instance ingests 18,000 measurements per second. Consider porting the RLS stage to C# for scale

Validate in Python first, offline and then live. The RLS and dq conditioning are about 100 lines, so they port to C# easily if a continuous trend becomes a requirement. DMD should stay in Python.

Issues in the current code, ranked

Blocks output **An early `return` skips every publish**

The leftover average-frequency logic in `process_data.py` only counts values between 59.5 and 60.5. The inputs are now kV and A point-on-wave samples, so the count is almost always zero and the function returns before any `publish_measurements` call. The log shows "No valid frequency measurements" every second. Remove the frequency and excursion code entirely.

Wrong results **The RLS restarts from zero every second**

`RLS()` re-initializes $\theta = 0$ and $P = \delta I$ on every call, and WaveApps calls it once per one-second buffer. Each second therefore starts over without excitation and returns noise. In our test it ranged from −136 to 586 Ω against a true 397 Ω. Run over one continuous record, the same function converges to 396 Ω. State has to persist between calls.

Wrong results **Non-causal filtering per window**

`filtfilt` is zero-phase and needs the whole record. Applied per window, it creates an edge artifact at every one-second boundary. Use `sosfilt` with saved initial conditions for streaming, and `sosfiltfilt` only on a complete event window, trimming the edges.

Blocks output **Output signal IDs are random each run**

`Xg/Vg/SCR_estimate_signalid` default to `uuid.uuid4()`. They must be set by matching C# adapter `MeasurementKey` properties with `[AmbientValue("…")]`, and they must exist in the Python publisher's metadata (`Metadata.xml`, or the definitions in the current sample's `DataProxy_Publisher`). Otherwise the STTP publisher silently drops them.

Scale / config **Units and frequency are hard-coded**

`RLS()` multiplies inputs by 1000, assuming PSCAD kV/kA, but the WaveApps stream is already in volts and amps. Your plots show about ±28,000 V and ±21 A, so V_g comes out 1000× too large. `f = 60` is hard-coded while the excursion limits assume 50 Hz. The paper's cases are 50 Hz and the SRP demo is 60 Hz. Make both adapter parameters.

Design **The estimator adapts through steady state**

With continuous adaptation, noise walks the estimate along the unidentifiable Z_g/V_g direction: X_g drifted from 397 to 381 Ω over 5 s of steady state in our test. Adapting only in windows that contain a detected event held 397.0 Ω through two events. The open design choice is how to re-open the estimator so it can track a real change in grid impedance (see the comments in `grid_strength.py`).

Check **Complex RLS transpose**

The code uses `B.T` everywhere, but the textbook complex least-squares RLS gain uses conj(φ). It still converged in our test, so treat this as a correctness check against your MATLAB reference. In MATLAB, `B'` is the conjugate transpose and `B.'` is the plain transpose.

Throughput **Per-sample work in the hot path**

Writing about 3,000 CSV rows every second, debug `print` calls for every measurement, and a per-sample Python loop around `abc_to_pn`, which is already vectorized NumPy, all add up. A call that runs over one second causes WaveApps to skip the next second.

Evidence: synthetic test with a known grid

34.5 kV, 60 Hz, 3,000 sps, SCR 3, X/R 2 (true R_g 198.4 Ω, X_g 396.8 Ω, V_g 28,169 V peak). The inverter is already running, with current steps at 4 s and 7 s. Data was fed one second at a time, the way WaveApps calls `process_data`. This is reproducible with `test_grid_strength_synthetic.py`.

METHOD	X_G ESTIMATE (Ω)	SCR	VERDICT
Original `RLS()`, restarted every 1 s	`−136 … 586`	`−9.5 … 7.6`	Unusable
Stateful RLS, adapting continuously	`397 → 381`	`3.00 → 3.12`	Drifts
Stateful RLS, adapting only on events	`397.1, 397.0`	`3.00`	Holds
Batch LS, eq. 4, 2 s window spanning each event	`396.7, 396.6`	`3.00`	Within 0.1%
Batch LS on a steady-state window	`rejected`	—	Correctly refused

The guided `process_data.py` was also exercised end to end with a mock proxy. It took about 30 ms per second of data, published X_g/V_g/SCR at 1 Hz once the first event arrived, and published two WaveApps events with X_g = 396.3 Ω and 396.4 Ω. A second with 14% of its samples missing was flagged and excluded from adaptation and event detection. The test used synthetic data only; it has not run against GE's recordings or a live WaveApps host.

Guided code provided

These are new files in a `guidance/` folder; your original files are unchanged. They are heavily commented starting points, not finished software.

`grid_strength.py`	Stateful complex RLS with causal filtering and event-gated adaptation, batch LS (eq. 4) with an identifiability gate, a placeholder excitation detector, a rolling POW buffer with gap reporting, and positive/negative sequence helpers for DMD. It has no STTP dependency, so it can be tested offline.
`process_data.py`	A drop-in WaveApps callback. It logs completeness each second, runs the detector and RLS, publishes 1 Hz results, runs event-window analysis on a worker thread and publishes one event with JSON details. It lists every adapter parameter and signal ID to register in Python and expose in the C# adapter, and includes the `run_ibr_impedance_dmd()` hook with the expected inputs and output schema.

`test_grid_strength_synthetic.py`	The offline harness behind the table above. Swap in a loader for your PSCAD or CSV exports. If the offline result is right and the streamed result is wrong, the problem is the data path, not the algorithm.

> We also recommend rebasing on the current WaveAppsSample. Your copy predates the latest changes: code-defined output metadata (`DataProxy_Publisher`), phase-aware POW metadata, and the `publish_event` signature. Event publication requires `sttpapi` 0.7.6 or later.

Questions for GE that shape the implementation

1. Is the deliverable a grid-strength result per event, or a continuous SCR trend? If a trend, at what rate: 1 Hz, or sub-second?
2. Which event types should trigger estimation (tap change, PQ step, POI voltage step, fault), and what thresholds define them?
3. The paper says POW was "recorded at 3000 samples per cycle". Is that 3,000 samples per second, which is the SEL-735 rate?
4. What system frequency and units does the replayed CSV use (50 or 60 Hz; V/A or kV/kA)?
5. Does the MATLAB reference RLS use the conjugate transpose (`B'`)?
6. What is the status of the DMD implementation, and which pre- and post-event window lengths does it need?
7. When the grid impedance genuinely changes (a topology change), how quickly must the estimate follow? This sets the forgetting and reset strategy.