Cross-Layer Diagnostics for Operator-Side Platform Porting of GPU-Accelerated 5G RAN: An Experience Report on Cable-on Validation of NVIDIA Aerial cuBB on DGX Spark

Hsiu-Chi Tsai
Arete Honors Program
National Yang Ming Chiao Tung University (NYCU)
Hsinchu, Taiwan
Email: hctsai1006@cs.nctu.edu.tw
ORCID: 0000-0001-7421-8027 GitHub: @thc1006

***Abstract*—GPU-accelerated 5G/6G Radio Access Network (RAN) stacks such as NVIDIA Aerial cuBB introduce *cross-layer interaction surfaces* that are absent from traditional CPU-only RAN implementations and are particularly visible during operator-side platform porting. We report on the early bring-up of NVIDIA Aerial CUDA-Accelerated RAN (ACAR) Release 26.1.0 on the DGX Spark (GB10 ARM64) entry-level reference platform in the cable-on physical-function-to-physical-function self-loopback topology, and on the cross-layer defects encountered while extending it with an end-to-end dApp inference pipeline. Because NVIDIA's public 26.1.0 release does not ship a DGX Spark cable-on cuphycontroller or L2 adapter YAML for the F08 launch-pattern family, the operator must derive these configurations locally from server-class sibling references; we document three defects that surface in this practice and the diagnostic techniques that localize them. With explicit grading of evidence: (i) an *operator-derivation oversight* (Finding 1) in which the locally derived L2 adapter YAML omitted a five-line static-slot override that all public sibling configurations contain, causing 3GPP TS 38.211 DMRS scrambling to use the runtime FAPI slot instead of the test-vector-baked slot 0 and producing 100% PUSCH CRC failure; (ii) a *vendor-side validation-mode policy gap* (Finding 2) in which eight unconditional `EXIT_L1(EXIT_FAILURE)` call sites in the publicly released `task_function_ul_aggr.cpp` escalate recurrent cond 3 timeout events in cable-on validation to process death without an opt-in soft-recovery knob; and (iii) an *operator-derivation CPU-map error* (Finding 3) in which a real-time priority-95 L2 adapter thread shared a CPU with a default-priority Data Lake worker in the derived YAML pair, silently starving the dApp telemetry pipeline. Three diagnostic techniques—*sibling-config diff, latent-code inspection, and cross-YAML CPU-map audit*—each localized one defect in minutes rather than weeks. A cumulative patch (YAML edits and a local source modification for Finding 2, approximately fifteen lines in total) restored the expected peak-cell throughput on our F08_1C_59 validation profile, with 1 h 6 min continuous soak stability and dApp end-to-end inference at the configured periodicity, in a single-run feasibility validation. We hypothesize the three techniques generalize to other configuration-heavy GPU-accelerated RAN SDK ports; broader validation is left to future work.**

***Index Terms*—5G NR, O-RAN, GPU-accelerated RAN, NVIDIA Aerial, cuBB, DGX Spark, GB10, PUSCH, DMRS, real-time scheduling, dApp, E3 interface, experience report.**

I. Introduction

Open Radio Access Networks (O-RAN) increasingly disaggregate the protocol stack across multiple compute components and external interfaces. NVIDIA Aerial CUDA-Accelerated RAN (ACAR), announced as open-source in October 2025 with GitHub availability starting December 2025 (and the Aerial Omniverse Digital Twin following in March 2026) [1], accelerates 5G/6G layer-1 (PHY) processing on GPUs through the cuBB SDK family (cuPHY, cuMAC, cuphydriver, cuphycontroller), and exposes a real-time AI inference path (dApps [2], [3]) through the new E3 interface in Release 26.1.0 [4].

In parallel, NVIDIA introduced the DGX Spark workstation, the first entry-level reference platform for Aerial built on the GB10 Grace Blackwell Superchip—a 20-core ARM64 (Cortex-X925 + A725) integrated SoC with 128 GB unified memory and a single ConnectX-7 Smart NIC providing two QSFP network ports. ATB1.0 (Aerial Total Builder Release 1.0) and Aerial Sample Apps 1.0 [5] were released contemporaneously, designating DGX Spark as a supported platform for cable-on PF↔PF self-loopback validation.

Cross-layer interaction surfaces. Traditional CPU-only RAN bring-up is dominated by single-component bugs (a MAC scheduling defect, an RLC misconfiguration, an LDPC threshold). GPU-accelerated RAN introduces three new *interaction surfaces* that traditional component-level review misses: (i) *specification-to-implementation alignment*, where 3GPP-prescribed Gold-sequence scrambling depends on slot/symbol/identity inputs sourced from YAML that may diverge from the test-vector context; (ii) *GPU runtime ↔ host OS*, where deterministic GPU-kernel timing budgets (e.g., the cuphy uplink "order kernel") interact with host-side scheduling, NIC PTP precision, and Linux real-time priority; and (iii) *cross-component configuration consistency*, where cuphycontroller and L2 adapter consume separate YAMLs whose CPU affinity, priority, and resource declarations are not automatically cross-checked.

Our contribution. This paper makes four contributions:

1) We present, to our knowledge, one of the first publicly documented experience reports on the bring-up of NVIDIA Aerial cuBB 26-1 on DGX Spark in the cable-on PF↔PF topology[1], reaching the published one-peak-cell reference range for our F08_1C_59 validation profile (DL 1544.14 Mbps; UL up to 213.84 Mbps, with 211.99 ± 1.20 Mbps mean over a 300 s observation as detailed in Table II; CRC=0 post warm-up) with 1 h 6 min continuous soak and end-to-end dApp inference at the configured periodicity, in a single-run feasibility validation.
2) We identify and root-cause three integration defects encountered while porting Aerial cuBB to DGX Spark in cable-on validation, each illustrating a distinct cross-layer anti-pattern (§IV–§VII). Two are *operator-side derivation errors* that surfaced while we were deriving DGX Spark-specific YAML files (Findings 1 and 3) from public sibling references, and one is a *vendor-side validation-mode policy gap* in NVIDIA's publicly released uplink driver source (Finding 2). Total cumulative patch: approximately fifteen lines of YAML and source comments.
3) We describe a diagnostic methodology—*sibling-config diff, latent-code inspection (i.e., inspecting code paths following non-returning fail-fast calls), and cross-yaml CPU-map audit*—that distilled each finding (§IX). We hypothesize these techniques may transfer to similar configuration-heavy GPU-accelerated RAN SDK porting tasks; broader validation is left as future work.
4) We report all three findings as GitHub Issues #43, #44, #45 [9]–[11] and an NVIDIA Aerial Developer Forum thread [12], contributing the diagnostic record to the open-source Aerial community. Following community feedback from NVIDIA's Aerial team, Issues #43 and #44 were closed with retractions clarifying that the underlying YAML files (`cuphycontroller_F08_GB10_CABLE_v8.yaml` and `l2_adapter_F08_GB10.yaml`) are operator-derived rather than NVIDIA-shipped; Issue #45 (the vendor-side fail-fast policy gap of Finding 2) remains open at submission time.

The remainder of this paper is organized as follows. §II provides background on 5G PHY, O-RAN 7.2x fronthaul, NVIDIA Aerial, DGX Spark, and the operator-side derivation context. §III describes our bring-up and instrumentation methodology. §IV introduces the cross-layer anti-pattern taxonomy. §V–§VII detail each of the three findings. §VIII reports our KPI validation results. §IX discusses generalizable lessons. §X states threats to validity and limitations, including the operator-derivation scope. §XI positions our work against concurrent literature. §XII concludes.

[1]We use "one of the first" conservatively. We do not claim priority over unpublished vendor bring-up or independent community notes that may exist; among the publicly searchable comparable reports we identified are the Red Hat OpenShift GH200 deployment [6], the cuSense GH200 dApp framework [7], and the DGX Spark LDPC-kernel characterization [8], none of which document cable-on full-chain bring-up on DGX Spark with end-to-end dApp inference.

II. Background

Note on terminology. We use the following acronyms consistently throughout: *ACAR* = Aerial CUDA-Accelerated RAN (NVIDIA's open-source GPU-accelerated 5G/6G SDK family); *cuBB* = the CUDA Baseband software stack within ACAR (cuPHY + cuMAC + cuphydriver + cuphycontroller); *ATB1.0* = Aerial Testbed 1.0 (NVIDIA's reference hardware-plus-software deployment, successor to the earlier ARC-OTA testbed); and *Aerial Sample Apps 1.0* = the open-source dApp reference repository released alongside ACAR 26.1.0 / ATB1.0.

A. 5G PHY: PUSCH, DMRS, and CRC

The Physical Uplink Shared Channel (PUSCH) carries 5G NR uplink user data. Each transport block (TB) is encoded with LDPC channel coding, modulated up to 256-QAM, and transmitted across a configurable set of physical resource blocks (PRBs). The Demodulation Reference Signal (DMRS) is a known, slot-and-symbol-specific pseudo-random sequence embedded in the same PRBs, used by the receiver to estimate the channel response and demodulate the data.

The DMRS sequence is initialized from a value c_{init} specified by 3GPP TS 38.211 §6.4.1.1.1.1 [13]:

$$c_{\text{init}} = \Big[\, 2^{17} \cdot \big(N_{\text{symb}}^{\text{slot}} \cdot n_{s,f}^{\mu} + l + 1\big) \cdot \big(2N_{\text{ID}} + 1\big) + 2N_{\text{ID}} + n_{\text{SCID}} \,\Big] \bmod 2^{31} \tag{1}$$

(here, 2^{17} denotes two raised to the seventeenth power, 2^{31} denotes two raised to the thirty-first power, and $\bmod$ denotes the modulo operation; we spell these out explicitly because automated PDF-to-text extraction may flatten superscripts.) The remaining parameters are: $N_{\text{symb}}^{\text{slot}} = 14$ for normal cyclic prefix, $n_{s,f}^{\mu}$ is the slot number within a frame at numerology μ, l is the OFDM symbol index within the slot, N_{ID} is the configured scrambling identity, and n_{SCID} is a 1-bit scrambling switch. The dependence on $n_{s,f}^{\mu}$ is critical: a misalignment of even one slot produces a completely uncorrelated Gold sequence (by design), so a receiver computing DMRS with the wrong slot yields a near-zero correlation with the actual transmitted reference signal. Only the slot number $n_{s,f}^{\mu}$ changes across our before/after comparison; all other parameters remain unchanged.

A 24-bit CRC is appended to each TB. The receive pipeline—channel estimation, equalization, demodulation, descrambling, LDPC decode, CRC check—fails closed when DMRS misalignment yields effectively noise at the equalizer output.

B. O-RAN 7.2x Fronthaul

The O-RAN ALLIANCE Working Group 4 standardizes the lower-layer functional split known as 7.2x [14], [15]. The PHY layer is split between the distributed unit (DU) and radio unit (RU): the DU handles high-PHY (scrambling, modulation, precoding, resource mapping), while the RU handles low-PHY (iFFT/FFT, CP add/remove, digital beamforming) and RF. The

fronthaul wire carries four logical planes over eCPRI [16]: U-plane (IQ samples), C-plane (per-slot scheduling and beamforming weights), M-plane (NETCONF/YANG management), and S-plane (IEEE-1588 PTP synchronization).

C. NVIDIA Aerial cuBB Stack

NVIDIA Aerial's cuBB SDK [17], [18] runs the entire DU-side 5G baseband on a GPU. Key components:

- **cuPHY**: GPU-accelerated L1 library implementing channel estimation, equalization, LDPC encode/decode, modulation, and scrambling as fused CUDA kernels for maximum memory bandwidth utilization.
- **cuMAC**: L2 layer providing MAC scheduling and HARQ management.
- **cuphydriver**: layer between cuPHY and the NIC, using DPDK poll-mode drivers and per-slot "order kernel" to gate frame ingress.
- **cuphycontroller_scf**: the integration daemon that hosts the cuPHY pipeline, exposes the SCF FAPI [19] interface to upstream L2, instantiates the Data Lake, and serves the E3 Agent.
- **L2 adapter**: a thin shim that bridges cuphycontroller to test simulators (`test_mac`) or production L2 stacks (e.g., OpenAirInterface).
- **ru_emulator**: a TV-replay simulator that emulates an O-RAN 7.2x compliant RU, useful for validation before real-RU procurement.

The new Data Lake architecture (introduced in 26-1) collects per-slot PHY telemetry (28 channels, including PRB power, SNR, HARQ counts) into shared memory, publishes via ZeroMQ to subscribed dApps. The E3 Application Protocol [2] between cuphycontroller's E3 Agent and dApps uses three sockets: REQ/REP for setup and subscribe (port 5555), PUB for indication messages (port 5556), and PUB for control messages (port 5557).

D. DGX Spark / GB10 Platform

The DGX Spark workstation [20] integrates the GB10 Grace Blackwell Superchip—10 Cortex-X925 performance cores at 4 GHz and 10 Cortex-A725 efficiency cores at 2.8 GHz, an integrated Blackwell GPU (6144 CUDA, 192 5th-gen Tensor cores, compute capability `sm_121`), 128 GB unified-memory LPDDR5x at 8533 MT/s, 4 TB NVMe, and a single ConnectX-7 Smart NIC exposing two QSFP rear-panel ports. (On our unit, the SR-IOV/multi-PF configuration enumerates as four netdev interfaces `aerial00`–`aerial03` via four mlx5 PCIe functions in domains `0000:01` and `0002:01`, per `lspci` and `ibdev2netdev`.) It operates at a 240 W power budget, positioning it as an entry-level reference platform relative to server-class Aerial deployments on GH200 [1]; precise relative-performance comparisons against GH200 are beyond the scope of this paper.

The unified memory architecture, while ideal for AI inference, eliminates separate GPU performance counters: `nvidia-smi` reports the `Memory-Usage` field as `Not Supported`, and `nsys` provides limited per-kernel insight. This drives our reliance on application-level instrumentation (NVLOGC counters) for diagnostic methodology (§III).

E. Cable-on PF↔PF Self-Loopback Topology

NVIDIA's recommended early bring-up topology connects two of DGX Spark's NIC physical functions (aerial00 PF and aerial02 PF) directly via a QSFP56 DAC cable. The cuphycontroller binds to aerial00 and `ru_emulator` to aerial02, completing a full fronthaul chain on a single host without requiring a physical RU, UE, SIM card, or external PTP grandmaster. NVIDIA's published spec for this topology [21] is approximately 1.5 Gbps downlink and 210 Mbps uplink for a single 4T4R 100 MHz cell with 30 kHz subcarrier spacing.

This topology is the focus of our paper. The absence of an external PTP grandmaster necessitates a software TAI/PHC synchronization recipe [22] as a prerequisite, with implications discussed in §VI.

F. Operator-Side Derivation of DGX Spark YAMLs

A practical consequence of running cable-on F08-pattern validation on DGX Spark is that the operator must derive the cuphycontroller and L2 adapter YAML files locally. NVIDIA's public 26.1.0 release tree [18] ships reference cuphycontroller configurations for a range of server-class platforms (`cuphycontroller_F08_R750.yaml`, `F08_CG1.yaml`, `F08_GL4.yaml`, `F08_BF3-ARM.yaml`) and two WNC-RU production paths (`P5G_WNC_DGX.yaml` for DGX Spark, `P5G_WNC_GH.yaml` for GH200), but does not ship a DGX Spark F08-pattern cable-on cuphycontroller or L2 adapter (verified via `git ls-files` against tag 26.1.0; no `cuphycontroller_F08_GB10*.yaml` or `l2_adapter_F08_GB10.yaml` entries exist). The Phase 4 performance-testbench framework [4] further restricts `VALID_HOST_CONFIGS` to three server-class combinations (`CG1_R750`, `CG1_CG1`, `GL4_R750`), none of which match the GB10 host. To run F08-pattern cable-on validation prior to procuring a WNC R1220 RU, we derived a local cuphycontroller YAML (`cuphycontroller_F08_GB10_CABLE_v8.yaml`) and an L2 adapter YAML (`l2_adapter_F08_GB10.yaml`) by combining (i) the DGX-aligned CPU/SM pinnings from `cuphycontroller_P5G_WNC_DGX.yaml` and (ii) the F08 launch-pattern parameters from `cuphycontroller_F08_R750.yaml` and `l2_adapter_config_F08_R750.yaml`. This operator-side derivation is the source of Findings 1 and 3; we make this provenance explicit at the start of each finding and in the Reproducibility Appendix (§A).

III. Bring-up and Diagnostic Methodology

A. Bring-up Sequence

We followed NVIDIA ATB1.0's published install path [4]: the `cubb_scripts/install/Makefile` sequentially provisions kernel, drivers, network stack (OFED, DOCA), NIC firmware, and supporting services, with sentinel

`.stamps/` files marking each stage. We verified all five stages green via `make check`. We then applied the published software TAI/PHC recipe [22] as a prerequisite to chain startup. The chain itself was driven by `test_mac` (the included L2 simulator) sending FAPI commands per the `F08_1C_59` launch pattern (one cell, 4T4R, MCS 27, 100 MHz, 30 kHz subcarrier spacing).

B. Expected Baselines

NVIDIA Forum announcement [21] publishes the following targets for this topology:

- DL throughput: ∼1.5 Gbps
- UL throughput: ∼210 Mbps
- CRC error rate: 0/s after warm-up
- UCI ONTIME: high-90 % sustained
- dApp inference cadence: per configured periodicity

Any deviation from these baselines signals an integration issue.

C. Diagnostic Toolchain

Owing to DGX Spark's unified memory architecture, conventional GPU profilers (`nvidia-smi`, `nsys`) provide limited per-component insight. We rely on three lightweight techniques:

NVLOGC counter instrumentation. NVIDIA's `NVLOGC` macro emits low-overhead counter log lines and is widely scattered through the cuPHY source. We added counters at three Data Lake publish-path entry points to identify which layer was missed:

- `data_lake::notify()` — per-slot work-ready signal,
- `data_lake::collectSlot()` — telemetry-payload assembly,
- `e3_agent::notifyDataReady()` — ZMQ publication.

Linux scheduling probes. For each thread of interest we issued two queries: `chrt -p <tid>` to confirm scheduling policy (`SCHED_FIFO` vs. `SCHED_OTHER`) and effective priority, and `taskset -p <tid>` to confirm the CPU affinity mask. Cross-referencing runtime values against the configured YAML produced an authoritative CPU map (§VII).

Sibling configuration diff. Aerial ships multiple reference `l2_adapter_*.yaml` files for different reference platforms (F08 base, Dell R750, SE-DU, GH200, P5G_WNC_DGX). When an operator-derived configuration targets a platform with no shipped reference (as with our DGX Spark F08-pattern cable-on case, see §II-F), diffing the derivation against its public siblings is a high-yield first step in any platform-port debugging session: derivations frequently omit entries present in the source siblings. This technique localized Finding 1 (§V) in approximately ninety seconds, versus three weeks of unstructured debugging that preceded it.

IV. Cross-Layer Anti-Pattern Taxonomy

We propose a tentative three-category taxonomy of cross-layer integration anti-patterns observed during cable-on bring-up of NVIDIA Aerial cuBB on DGX Spark, with each finding instantiating one category. *Type 1: spec-config drift* arises when a 3GPP-prescribed behavior (e.g., DMRS scrambling) parameterized by YAML inputs falls back to a runtime-derived value diverging from the test-vector context, producing systematic failures that look like noise; kernel-level unit tests pass while live chain runs fail (Finding 1, §V). *Type 2: GPU-to-OS impedance mismatch* arises when GPU kernels with deterministic budget timing (e.g., the cuphy uplink "order kernel" enforcing a ∼1.5 ms first-packet completion window) interact with host-side timing primitives whose precision varies (hardware PTP versus software phc2sys), and the SDK's fail-fast policy escalates intermittent budget violations to process death (Finding 2, §VI). *Type 3: cross-component resource conflict* arises when independently configured components (cuphycontroller, L2 adapter, dApp) over-allocate shared OS resources—particularly CPU cores under Linux real-time scheduling—with the resulting starvation manifesting silently (Finding 3, §VII). Two of the three patterns (Types 1 and 3) are best understood as *operator-side derivation anti-patterns* arising during the platform porting described in §II-F; the third (Type 2) is a *vendor-side validation-mode policy gap* in NVIDIA's publicly released uplink driver. On a CPU-only platform these patterns either do not exist (Type 1 requires the standard-to-config-to-kernel chain), are less acute (Type 2 with software PHY runs in user space), or are diagnosed by routine profiling (Type 3). On a GPU-accelerated entry-level platform like DGX Spark, all three are simultaneously visible—which we suggest is the substantive lesson for any platform port of a GPU-accelerated RAN SDK. Epistemic grading and vendor-confirmation status for each finding are stated in the Provenance note at the top of §V–§VII and consolidated in §X.

V. Finding 1: Operator-Derivation Drift Causes 100% PUSCH CRC Failure

Provenance note. The L2 adapter YAML in which the missing entry was observed (`l2_adapter_F08_GB10.yaml`) is operator-derived in our environment, as established in §II-F; it is not present in the public 26.1.0 release tag tree. We retain this finding in the paper because the diagnostic technique that surfaced it (sibling-config diff against public reference yamls) is the methodological contribution, not the specific YAML omission. GitHub Issue #44 was closed with retraction [10].

A. Symptom

Upon initial chain start, `test_mac` reported:

```
[SCF.PHY] Cell 0 | DL 1544.14 Mbps 1600 Slots
                 | UL    0.00 Mbps  400 Slots CRC 400
```

Every PUSCH slot received reported a CRC failure (400 fails per second, 100% rate). PUCCH HARQ, CSI, and PRACH all operated normally. Only PUSCH UL data decode failed, deterministically, in steady state.

B. Diagnosis Path

Standard suspects—wire format (BFP9 compression), MCS configuration, RNTI assignment, scrambling identity—were

eliminated by inspection. Channel-estimation and equalizer-coefficient algorithm variants were swept; none corrected the symptom.

Critically, the GPU kernel was eliminated as a fault source: `cubb_gpu_test_bench`, fed the same test vector directly without the rest of the chain, produced 100% CRC pass. The defect must therefore lie at the chain integration layer between the test-vector slot context and the cuPHY input parameters.

Reading the cuphy channel-estimation kernel implementation (`cuPHY/src/cuphy/ch_est/ch_est.cu:776`):

```
uint32_t cInit = TWO_POW_17 *
    (slotNum * OFDM_SYMBOLS_PER_SLOT +
        symIdx + 1) *
    (2 * dmrsScramId + 1) +
    (2 * dmrsScramId) + scid;
```

This implements (1) faithfully, with `slotNum` the slot number within the frame. The Gold-sequence generator initialized by this c_{init} produces the locally expected DMRS reference signal against which the receiver correlates the incoming IQ samples.

Tracing back through the SCF FAPI L2 adapter's slot-command handler (`scf_5g_slot_commands.cpp:158`; full path in App. A) revealed:

```
cell_info.slotNum =
    ((staticPuschSlotNum > -1)
       ? staticPuschSlotNum
       : cell_grp_cmd->slot.slot_3gpp.slot_);
```

The YAML parameter `staticPuschSlotNum` overrides the runtime FAPI slot when explicitly set; otherwise the runtime slot is used.

The test vector `TVnr_ULMIX_3040_gNB_FAPI_s0.h5` has `gnb_pars.slotNumber=0` baked into its IQ samples (DMRS sequence corresponds to slot 0). The `F08_1C_59` launch pattern places PUSCH at runtime slots 4, 5, 14, 15 within each 10 ms frame. With `staticPuschSlotNum` unset, cuphy computes its expected DMRS using slot 4 while the wire-side IQ contains slot 0 DMRS. The correlation is statistically zero, channel estimation produces noise, the LDPC decoder converges to random codewords, and the CRC check fails with probability $1 - 2^{-24}$ per TB—empirically 100%.

C. Sibling Configuration Diff

Comparing `l2_adapter_F08_GB10.yaml` against sibling configurations:

Config file	`staticPuschSlotNum`
`l2_adapter_config_SE_DU.yaml`	0 (present)
`l2_adapter_config_F08_R750.yaml`	0 (present)
`l2_adapter_F08_GB10.yaml`	*absent*

Our operator-derived DGX Spark configuration omits the entry present in the two server-class siblings (Dell R750 and SE-DU) from which it was adapted. This is the classic spec-config drift pattern in operator-side derivation: a YAML entry assumed by the implementation is silently dropped during the porting copy.

D. Fix

Adding five lines to `l2_adapter_F08_GB10.yaml`:

```
staticPucchSlotNum: 0
staticPuschSlotNum: 0
staticPdcchSlotNum: 0
staticPdschSlotNum: 0
staticCsiRsSlotNum: 0
```

E. Result

```
Before: UL   0.00 Mbps CRC fail/sec 400
After:  UL 213.84 Mbps CRC fail/sec   0
```

UL throughput jumped instantly from zero to the published one-peak-cell reference range, with CRC error rate dropping to zero post warm-up. This finding was reported upstream as GitHub Issue #44 [10] and NVIDIA Aerial Developer Forum thread 369748 [12]; following community feedback that clarified the YAML provenance (see Provenance note at the top of this section), Issue #44 was closed with an operator-side retraction on 2026-05-16 and a corresponding clarification was posted to the Forum thread.

VI. Finding 2: Fail-Fast Policy Prevents Long-Running Cable-on Validation under Recurrent Cond 3 Timeout Events

Framing and safety disclaimer. This finding is best described not as a bug in NVIDIA Aerial but as a missing operator-facing knob for the cable-on validation topology: the production fail-fast policy is appropriate for deployments with hardware PTP grandmasters, but absent for the high-frequency-yet-benign jitter events characteristic of cable-on testing. We propose a YAML knob and demonstrate a local source patch *only as a viability prototype*—we explicitly do *not* recommend deploying it in production. Required pre-conditions for production adoption (per-slot HARQ/UCI/PRACH state correctness audits, 24-h+ soak under concurrent dApp workload, leak audits under repeated `abortTasks()`, scope-gating to the specific cond 3 condition, and side-by-side hardware-PTP comparison) are deferred and enumerated in §X.

A. Symptom

After Finding 1 was resolved, chain throughput reached NVIDIA-validated peak-cell rates, but cuphycontroller exited with a fatal error after approximately ten minutes of wall-clock runtime:

```
ERR ... SFN 294.5  Order kernel timeout error
                    (exit condition 3) ...
ERR ... SFN 491.15 Order kernel timeout error
                    (exit condition 3) ...
FATAL ... task_function_ul_aggr.cpp line 514:
            pipeline failed, exit
```

(Line 514 is the `NVLOGF_FMT` print preceding the `EXIT_L1` call hit on this invocation in our locally instrumented copy; the full list of eight call sites in the public 26.1.0 release is below.)
The exit was deterministic and reproducible: every chain restart died within ∼10 minutes.

B. Cond 3 Events Correlated with Cable-on Topology Timing

The cuphydriver GPU "order kernel" polls per-slot for arrival of all uplink IQ packets, exiting on one of four conditions

defined in `order_entity.hpp:43` (full path in App. A). Condition 3, `ORDER_KERNEL_EXIT_TIMEOUT_RX_PKT`, fires when:

```
if (first_packet &&
    (current_time - first_packet_start
     > timeout_first_pkt_ns)) {
  *exit_cond_d = ORDER_KERNEL_EXIT_TIMEOUT_RX_PKT;
}
```

The default `ul_order_timeout_gpu_ns = 3 000 000` (3 ms) implies a first-packet timeout of 1.5 ms.

In production deployments with a hardware PTP grandmaster (e.g., VIAVI Qg 2), wire-side jitter is expected to remain well below 1.5 ms and cond 3 essentially not to fire. In our cable-on PF↔PF topology, only software `phc2sys` synchronizes the two ConnectX-7 NIC PHCs without an external grandmaster. We measured the cond 3 event rate at ∼0.33 events/s averaged over a 1-hour run (with bursts peaking at ∼20 events/s), accumulating to 1302 events in 1 h 6 min wall clock. We hypothesize that software-only PHC synchronization is a major contributor to these events in our topology, but we have not isolated this factor from interrupt affinity, DPDK polling priority, kernel real-time tuning, container scheduling, or driver/firmware versions; cond 3 root-cause attribution therefore remains uncertain in our environment.

We swept `ul_order_timeout_gpu_ns` to test whether tuning could eliminate cond 3:

Timeout	cond 3 rate	Note
3 ms (default)	∼20 events/s	—
4 ms	∼15 events/s	−21% vs default
6 ms	0 events	FATAL at `scf_5g_fapi_phy.cpp:272`

Beyond 6 ms, another internal timer dependency triggers at startup. Thus the YAML timeout alone is bounded above by an unrelated constraint and cannot fully eliminate the issue.

C. Fail-Fast Policy: Eight `EXIT_L1` Sites

In the uplink pipeline driver source file `task_function_ul_aggr.cpp` (full path in App. A), every error path terminates the process via `EXIT_L1(EXIT_FAILURE)`, a macro evaluating to a non-returning `_exit()` call. In the publicly released 26.1.0 source, eight distinct call sites exist at lines 515, 777, 958, 1137, 1302, 1526, 1944, and 2189, each with the identical structure (line numbers cited here correspond to the public release; our locally instrumented copy, which carries an additional approximately eleven lines of NVLOGC counter code, shifts these by the same offset and was the source of the seven-site count of 788 / 969 / 1148 / 1313 / 1537 / 1955 / 2200 reported in earlier versions of this work and in GitHub Issue #45 [11]):

```
error_next:
  // Currently we do not support pipeline
  // recovery from CUDA/FH errors
  NVLOGF_FMT(TAG, ..., "pipeline failed, exit");
  EXIT_L1(EXIT_FAILURE);  // process dies here
  // The next three lines are unreached:
  slot_map->abortTasks();
  NVLOGE_FMT(TAG, ..., "aborted ...");
  return -1;
```

The code following `EXIT_L1` provides a latent path at the source level. We do not claim this reflects any intended recovery semantics; the in-source comment explicitly states recovery is not supported. We observe only that the path's mere existence at this location provides a natural implementation site if a future configurable recovery knob were to be added.

D. Proposed Configuration Knob (Narrow, Scope-Gated)

We propose a narrowly scoped, rate-limited, condition-specific YAML knob set rather than a broad `fatal_on_pipeline_error` flag. A representative skeleton:

```
validation_mode:
  # Soft-recover cond 3 only (opt-in):
  recover_ul_order_timeout_rx_pkt: false
  # Rate cap (0 = unlimited):
  max_soft_recoveries_per_min: 0
  # Always abort on non-cond-3 errors:
  abort_on_non_cond3_pipeline_error: true
```

The intent: (i) soft-recovery is gated to exactly the `ORDER_KERNEL_EXIT_TIMEOUT_RX_PKT` (cond 3) condition, not all eight `EXIT_L1` error paths; (ii) recovery is rate-capped to prevent indefinite masking of a developing fault; and (iii) any pipeline error of a different class still aborts. Defaults preserve the current production behavior. We emphasise that this is a knob *request* to NVIDIA; we have not implemented the gating logic ourselves, only the broader proof-of-viability described next.

This proposal has been submitted as GitHub Issue #45 [11] as a configuration-knob request, structured as a follow-up to a closely related issue #41 [23] that addressed the same fail-fast design pattern at `nic.cpp:609` for cable-less topologies.

E. Local Prototype Patch (Not for Production)

To verify that the soft-recovery path is technically viable for cond 3 events that appeared non-fatal in this short validation run after soft recovery (but whose underlying state-safety remains unproven, see §X), we locally commented out the seven `EXIT_L1` call sites we had identified at the time of validation (lines 788, 969, 1148, 1313, 1537, 1955, 2200 in our instrumented copy) and supplemented with a defensive YAML threshold (`aggr_obj_non_avail_th: 5 → 1000`). The eighth site, at line 515 in the public release (which we identified only during post-submission audit, see §VI, paragraph 5), was not commented out in our validation; it did not fire during the 1 h 6 min run. For a complete soft-recovery patch matching the public release, all eight sites should be commented out:

```
error_next:
  NVLOGE_FMT(TAG, ..., "pipeline failed");
  // EXIT_L1 disabled for validation:
  // EXIT_L1(EXIT_FAILURE);
  slot_map->abortTasks();
  NVLOGE_FMT(TAG, ..., "aborted ...");
  return -1;
```

We rebuild cuphydriver and cuphycontroller_scf. With this prototype patch, chain uptime extends to 1 h 6 min wall clock with zero FATAL exits and 1302 cond 3 events soft-recovered. *This patch is a viability prototype, not a production recommendation*; the pre-conditions for production adoption (state-corruption verification, extended soak, leak audits, condition-specific gating) are enumerated in §X.

F. Result

Within the scope of validation testing on cable-on PF↔PF self-loopback, the prototype patch enabled continuous chain uptime of 1 h 6 min wall clock with zero FATAL exits, 1302 cond 3 events soft-recovered, throughput unchanged (§VIII), and UCI ONTIME stable in the 98–100 % range during the run. We caution that the UCI ONTIME observation alone does not establish absence of subtle data-plane state inconsistencies post-recovery; per-slot HARQ and UCI histogram analysis is enumerated in §X as a required next step.

VII. Finding 3: Operator-Derived Cross-YAML CPU Conflict Silently Starves dApp Pipeline

Provenance note. Both YAML files involved in this finding (`cuphycontroller_F08_GB10_CABLE_v8.yaml` and `l2_adapter_F08_GB10.yaml`) are operator-derived in our environment, as established in §II-F. The colliding CPU pinnings (`data_core: 11` and `cpu_affinity: 11, sched_priority: 95`) were both operator-side choices made when porting CPU assignments from server-class sibling references without re-mapping for the 20-core GB10 layout. The public NVIDIA DGX Spark dApp baseline (`cuphycontroller_P5G_WNC_DGX.yaml`) uses `data_core: 12` and avoids the collision, as confirmed by NVIDIA Aerial engineering in the GitHub Issue #43 thread. We retain this finding because the diagnostic technique (cross-YAML CPU-map audit) is the methodological contribution. GitHub Issue #43 was closed with retraction [9].

A. Symptom

After Findings 1 and 2 were addressed, chain throughput and stability matched NVIDIA spec. The prb-power-python sample dApp (NVIDIA Aerial Sample Apps 1.0 [5]) was attached, configured to subscribe to eight Data Lake telemetry channels via the E3 interface. E3 setup succeeded, subscription returned `confirmed`, and ZMQ ports 5555/5556/5557 reported `ESTABLISHED`. However, a raw ZMQ `SUB` on `tcp://127.0.0.1:5556` received *zero* indication messages over a 10-second observation window. The dApp reported no error logs.

B. Locating the Silence

The Data Lake publish path comprises:

1) `data_lake.cpp::notify()` called per PUSCH slot by the cuphy main thread; sets a work-ready flag.
2) A separate polling thread (`datalake_thread`) waits on the flag and dispatches to a per-slot collector function when signalled.
3) `collectSlot()` assembles the telemetry payload and calls `e3_agent::notifyDataReady()`.
4) `notifyDataReady()` publishes the indication on ZMQ port 5556.

Adding NVLOGC counter instrumentation at each step:

Counter	Observed rate
`notify()` entry	∼410 /s
`collectSlot()` entry	**0 /s**
`notifyDataReady()` entry	0 /s

The `notify()` flag was being set, but `collectSlot()` was never entered—meaning the `datalake_thread` was not progressing. We hypothesized CPU starvation.

C. CPU Affinity Audit

In `cuphycontroller_F08_GB10_CABLE_v8.yaml`, the `datalake_thread` is pinned to the CPU named in `data_core`:

```
data_config:
  data_core: 11   # avoid CPU 12 collision
                  # with debug_worker
```

The inline comment in our derived YAML (carried over from the sibling we adapted) flags the cuphy-internal `debug_worker` thread on CPU 12, but our derivation did not extend the collision check across components. Cross-referencing with our derived `l2_adapter_F08_GB10.yaml`:

```
message_thread_config:
  {name: msg_processing,
   cpu_affinity: 11,
   sched_priority: 95}
```

The L2 adapter's `msg_processing` thread is also pinned to CPU 11, but with `SCHED_FIFO` real-time priority 95.

We confirmed at runtime with `chrt -p` and `taskset -p`: `datalake_thread` runs `SCHED_OTHER` priority 0 pinned to CPU 11; `msg_processing` runs `SCHED_FIFO` priority 95 also pinned to CPU 11.

D. Linux Real-Time Scheduling Behavior

Under Linux scheduling semantics [24], [25], a runnable `SCHED_FIFO` priority-95 thread can indefinitely delay a `SCHED_OTHER` thread pinned to the same CPU unless the RT thread blocks, yields, or is throttled. In our run, the observed counters (`notify()` at ∼410 /s while `collectSlot()` remains at 0 /s) together with the one-line relocation of `data_core` from CPU 11 to CPU 13 restoring full operation strongly support a CPU-collision mechanism.

E. Scheduler-Level Evidence (post-fix state)

To corroborate the scheduling explanation at the host-OS level, we captured `/proc/<tid>/sched` snapshots and a 30-second `perf sched record` on the long-running production chain (post-fix state, `data_core: 13`). Table I summarises the per-thread scheduling deltas.

The 100% voluntary-switch fraction for both threads indicates clean blocking between work items rather than involuntary preemption; `datalake_thread` accruing ∼1.18 s of cumulative CPU time over the 30 s window on CPU 13 demonstrates that a default-class thread can be scheduled at this rate when not co-located with an RT-95 thread, which by contrapositive supports the original starvation explanation when both were on CPU 11. The failure-state trace (prefix, `data_core: 11`) would require reverting the YAML and restarting the 4-day-uptime production chain and is

TABLE I: 30-second `/proc/<tid>/sched` delta on the live chain (post Finding 3 fix). `datalake_thread` accrues stable nonzero CPU runtime on CPU 13 with 100% voluntary switches, demonstrating that the same default-class thread *can* be scheduled when given a CPU not shared with the RT-95 `msg_processing`; this is the scheduler-level corollary of the original starvation observation.

Counter (30 s window)	`datalake_thread`	`msg_processing`
Scheduling policy / priority	SCHED_OTHER / 0	SCHED_FIFO / 95
CPU affinity (hex / decimal)	0x2000 / CPU 13	0x0800 / CPU 11
`sum_exec_runtime` delta	1178.97 ms	17700.40 ms
Effective CPU utilization	3.93% of CPU 13	59.00% of CPU 11
`nr_switches` delta	562,672	4,085,362
Voluntary-switch fraction	100.0%	100.0%
Sched events per second	18,756/s	136,179/s

deferred to a controlled follow-up; raw `perf.data` and `sched` snapshots are archived in the reproducibility pack. Alternative explanations (ZMQ HWM drops, subscription-routing mismatch, Data Lake periodicity gating, thread-creation failures, IPC namespace mis-binding) were each excluded by negative evidence: subscription confirmed, ZMQ endpoints in `ESTABLISHED`, thread present in `ps -eLf`, IPC namespace shared via `--ipc=container:c_aerial_$USER`.

F. Fix

In `cuphycontroller_F08_GB10_CABLE_v8.yaml`, one line changes:

```
data_config:
  data_core: 13   # was 11; collided with
                  # l2_adapter msg_processing
                  # (cpu_affinity:11 prio:95)
```

CPU 13 was unused in our 20-core deployment map: CPUs 0–4 are kernel/systemd, 5–10 are cuphy uplink/downlink workers, 11 is the L2 adapter `msg_processing` (RT 95), 12 is the cuphy `debug_worker`, 14–17 run `test_mac`, and 18–19 are reserved.

G. Result

```
After fix:
  collectSlot()      ~410/s (was 0/s)
  notifyDataReady()  ~10/s  (was 0/s; 100ms period)
  dApp inferences:  249 in 25s; avg 2528us
  Min/Max latency:  1875/3620us (prb_power_numpy)
```

The dApp end-to-end pipeline—cuphy GPU L1 → Data Lake → E3 ZMQ → Python inference → visualization—becomes operational. This finding was reported upstream as GitHub Issue #43 [9]. Following NVIDIA Aerial engineering feedback that the official DGX Spark dApp baseline (`cuphycontroller_P5G_WNC_DGX.yaml`) uses `data_core: 12` and does not exhibit the collision (i.e., the collision arises specifically from operator-side derivation), Issue #43 was closed with retraction on 2026-05-16. Negative evidence excluding common alternative explanations: after the fix, a raw `zmq.SUB` listener on the same container path receives ∼100 messages per 10 s window; the E3 `Subscription` response was returned with `positive` status by the cuphy E3 Agent and a corresponding subscription record was created server-side; and `data_lake.cpp::notify()` continued firing at ∼410 /s throughout, demonstrating the upstream source path was active.

H. Discussion

The defect is one digit: `11` → `13`. The diagnostic effort was approximately four hours. The disparity is characteristic of cross-component configuration conflicts on a constrained 20-core platform: any individual yaml is internally self-consistent, but the union over yaml files exceeds the available CPU map. Detection requires reading all relevant yaml files and computing the union, then comparing against the host CPU layout—an audit step we propose generalizing to a CI lint check (§IX).

VIII. Validation

Following all three fixes, we ran a single-run feasibility validation against the NVIDIA-published one-peak-cell reference range for our F08_1C_59 profile. The cumulative patch totals fewer than twenty lines: five YAML additions for Finding 1, one YAML change plus eight source-line modifications for Finding 2, and one YAML change for Finding 3.

A. Throughput and Decode Quality

Table II summarizes our measured KPIs against the NVIDIA spec. All headline values are at or above spec.

TABLE II: Multi-window KPI statistics computed from a single 300 s steady-state observation on the post-patch chain, split into five non-overlapping ∼60 s windows ($n = 299$ one-second samples total). This characterizes **intra-run variance within one steady-state observation**, not cold-start variance across independent chain restarts. Reference column from NVIDIA forum 369091.

KPI	Mean ± std (min/max)	Ref.	Comparison
DL (Mbps)	1544.14 ± 0.000	∼1500*	matches
UL (Mbps)	211.99 ± 1.20 (202.39/213.84)	∼210*	+0.95%
CRC err/s	2.57 ± 1.74 (0/many)	0*	see §VIII-D
UCI ONTIME (%)	99.77 ± 0.03 (99.66/99.82)	n/a	high
HARQ ACK/NACK /s	∼11,900 (peak 12,000)	12,000†	matches
CSI Part 1/2 /s	∼2,380 (peak 2,400)	2,400†	matches
PRACH detect /s	∼695 (peak 700)	700†	matches

*Reference value cited from NVIDIA forum thread 369091 short description for the "1 peak cell" baseline. †F08 launch-pattern ceiling. Specific Mbps values vary across release versions and profile masks; see comparison with the Red Hat demo measurement later in this section.

B. One-Hour Soak Validation

A 1 h 6 min continuous wall-clock soak (longer-soak validation extending to 24 h+ with concurrent dApp workload remains a required next step, §X). Statistics:

- Wall-clock uptime: 3960 s
- FATAL exits: 0
- Cond 3 events: 1302 total (∼0.33/s average, with bursts)
- Cond 3 outcomes: all 1302 soft-recovered, chain continued
- DL throughput drift: none in first hour
- CRC pass rate: 100% post warm-up

The 1302 soft-recovered cond 3 events are direct evidence that the Finding 2 patch (commenting out the eight `EXIT_L1` sites) functions as intended—without it, the first cond 3 would have terminated the process within minutes.

C. End-to-End dApp Inference

The prb-power-python sample dApp, configured with 100 ms publish periodicity, subscribed to channels `[1, 4, 5, 6, 16, 17, 18, 19]` (PRB power, SNR per-layer, HARQ ACK/NACK counts, PRB power averages, UE throughput). Results over a 25-second observation:

- Inferences fired: 249 ($\sim$9.96/s, matching 100 ms periodicity)
- Average end-to-end dApp latency: 2528 μs (model: `prb_power_numpy`)
- Min/Max latency: 1875 / 3620 μs (single-run, $n = 249$ samples)
- Visualizer (Flask + Plotly on port 5001): HTTP 200, live PRB power heatmap update

The latency figure is measured end-to-end within the `prb-power-python` container and covers four stages: ZeroMQ `recv` on the 5556 SUB socket, Python dispatch through `e3_manager.py`, `prb_power_numpy` model inference, and the result-serialization plus `publish` step on port 5559. It is not a kernel-only latency; the GPU-accelerated Triton variant included with Aerial Sample Apps 1.0 would be expected to reduce the inference component substantially but was not the focus of this validation. We report mean and min/max only; CDF / $p_{50}/p_{95}/p_{99}$ characterization is enumerated as required next work in §X.

D. Long-Run Drift and Cross-Validation

The CRC-error rate of 2.57 ± 1.74/s in Table II reflects a 4-day uptime chain rather than a fresh one—the chain trends from CRC$= 0$/s immediately post warm-up toward a small nonzero rate over multi-hour operation, likely from accumulated software PHC drift in the cable-on topology; fresh-chain re-validation with cold-start characterization is left to future work (§X). Independent cross-validation comes from a publicly available Red Hat OpenShift demo using the closely related `F08_1C_59c` profile [6], which reports DL $= 1544.1432$ Mbps over a 30-second PASS interval; our DL (1544.14 Mbps) agrees to four significant figures. UL differences (their 196.6992 Mbps versus our 213.84 Mbps) likely reflect profile-mask differences between `F08_1C_59` and `F08_1C_59c` variants which we have not diffed; we therefore caution against reading 213.84 Mbps as canonical for all F08_1C_59 variants.

IX. Lessons and Methodology

We distill three diagnostic techniques and one engineering practice recommendation; we expect each to be useful for similar configuration-heavy GPU-accelerated RAN SDK porting tasks.

Sibling-config diff. When porting an SDK to a new reference platform, the platform-specific YAML is derived from one or more existing siblings; cross-diffing the derivation against its closest sibling is a high-yield first step. In our case, diffing `l2_adapter_F08_GB10.yaml` against `l2_adapter_config_F08_R750.yaml` immediately surfaced the missing `staticPuschSlotNum` entry that would otherwise have required three additional weeks of unstructured debugging to locate. This is not novel as a software engineering technique but is under-applied in RAN bring-up where configurations are dense and per-platform; we propose that any SDK ship with a CI-integrated sibling-diff lint that warns when a new platform's configuration omits keys present in established siblings.

Inspecting code paths following a non-returning call. When debugging unexpected fail-fast behavior, inspecting the code paths that follow a non-returning call (such as `EXIT_L1`) can still be informative. In our Finding 2, the statements after `EXIT_L1` (`slot_map->abortTasks()` and `return -1`) do not demonstrate any intended recovery semantics—the in-source comment above the call explicitly states that pipeline recovery is not supported—but their *location* identifies a natural implementation site at which a future, explicitly gated validation-mode recovery knob could be inserted without major refactoring, which is exactly what Issue #45 proposes.

Cross-YAML CPU-map audit. When multiple components of a GPU-accelerated RAN are independently configured (each loading its own YAML), shared OS resources can be over-allocated. We propose treating CPU affinity declarations across all component YAMLs as constituting a single logical schedule, against which the host CPU layout is cross-checked: walk every component YAML, collect each thread's CPU pin, scheduling policy, and priority, and warn whenever a CPU is shared by a `SCHED_FIFO` and a default-class thread. We have included this recommendation in Issue #43, alongside a suggestion that cuphycontroller emit a startup WARN-level log when cross-YAML CPU-affinity collision is detected.

Reference linter implementations and practice recommendation. To make the methodology concrete we implemented two Python utilities ($\sim$100 lines each, no external dependencies): `cross_yaml_lint.py` parses CPU affinity / scheduling priority across component YAMLs and reports SCHED_FIFO/default-class collisions (correctly surfacing the Finding 3 collision on CPU 11 when run on the derived YAML pair); `sibling_config_diff.py` reports keys present in the sibling intersection but missing from a target (correctly surfacing all five missing `static*SlotNum` entries underlying Finding 1 when run on the derived L2 adapter against its R750 and SE-DU siblings). Both scripts are released with the reproducibility pack [26]. The collective engineering practice recommendation is small and applies to vendor and operator alike: any platform-specific configuration—shipped or derived—should be programmatically validated against (i) the host CPU map under cross-component affinity declarations and (ii) the union of YAML keys present in sibling references. Cost: one engineering afternoon to integrate into CI/pre-commit; prevention: approximately three weeks of bring-up debugging per platform port.

X. Threats to Validity and Limitations

Operator-derivation scope. Findings 1 and 3 are observations on *operator-derived* YAML files, not on NVIDIA-shipped configurations; the diagnostic techniques we propose demonstrate utility in catching exactly the class of derivation errors we made, and we do not claim that these specific defects exist in NVIDIA's public configurations. The present work establishes only that at least one operator (us) made these errors during platform porting and the proposed techniques surfaced them in under two minutes versus weeks of unstructured debugging; prevalence across the broader Aerial operator community is a separate study.

Single operator, unit, version, run. All experiments were performed by a single operator on one DGX Spark unit running ACAR 26.1.0 with the locally derived F08_GB10 YAML pair (§II-F) and the `F08_1C_59` launch pattern; KPI values in Table II reflect a single 300 s observation split into five $\sim$60 s windows (intra-run variance only). Multi-unit, multi-version (e.g., 25-x or hypothetical 26-2), multi-launch-pattern (e.g., `ULMIX_3060`), and multi-operator cross-validation, together with multi-run mean/variance/percentile statistics, are required next steps; the results should be read as a feasibility data point rather than statistically grounded performance characterization.

Finding 2 soak-limit and cond 3 attribution. The 1 h 6 min uptime establishes that the soft-recovery path is not immediately unsafe but does not prove production-grade correctness: per-slot HARQ/UCI/PRACH state histograms following each soft-recovered cond 3, extended soak ($\geq$24 h) with concurrent dApp workload, memory/file-descriptor/shared-memory leak audits under repeated `abortTasks()`, comparison against an external hardware PTP grandmaster, and MCS/bandwidth/cell-count sensitivity analysis are all unperformed. Our local source patch is therefore a viability prototype only. Separately, we attribute cond 3 events primarily to software-only PHC synchronization but cannot exclude contributions from interrupt affinity, DPDK polling priority, CPU isolation, kernel real-time tuning, container scheduling, or driver/firmware versions; isolating these confounders requires a controlled comparison left as future work.

Finding 3 correlation versus causation. Our evidence for CPU starvation is the union of static configuration evidence (`chrt -p`, `taskset -p`), Linux scheduling semantics, observed counter rates, and the one-line fix restoring operation; post-fix scheduler trace appears in Table I, and a symmetric pre-fix trace is within scope of a controlled follow-up.

Alternative explanations not strictly ruled out. For Finding 1 these include TV metadata loader gaps, an F08-specific FAPI adapter defect, or an undocumented launch-pattern contract; for Finding 2, NIC e-switch latency or per-component RT tuning; for Finding 3, ZMQ HWM drops, subscription routing, Data Lake gating, thread-creation failures, or IPC namespace mis-binding. Each is consistent with our observations but not strictly excluded.

Generalization. The diagnostic techniques in §IX are derived from three findings on one platform. Generalization across GH200, real-RU, OAI, multi-cell topologies, or non-RAN SDKs is hypothesized but unsubstantiated by this study.

External validation status. Following community feedback from the NVIDIA Aerial engineering team in the GitHub Issue #43 thread, GitHub Issues #43 and #44 were closed with operator-side retractions on 2026-05-16 acknowledging the YAML files were operator-derived rather than NVIDIA-shipped; the HackMD whitepaper and NVIDIA Forum thread 369748 received corresponding corrections on the same date. GitHub Issue #45 (Finding 2, independent of YAML provenance) remains open at submission time, with the EXIT_L1 line numbers corrected to the publicly released values (515, 777, 958, 1137, 1302, 1526, 1944, 2189). Any subsequent vendor confirmation on Issue #45 will be reflected in camera-ready.

XI. Related Work

Concurrent NVIDIA Aerial work. The cuSense framework [7] demonstrates a programmable GPU-accelerated dApp on the Aerial Testbed (GH200 + BlueField-3 + ConnectX-7, hardware PTP, 77 cm mean indoor-localization error, $\sim$150 μs framework overhead). Our work is complementary: we operate on the entry-level DGX Spark (GB10) platform whose cross-layer surfaces (no BlueField-3, software-only PTP, 20-core CPU) differ from the server-class GH200 environment, and our contribution is methodological—characterizing the cross-layer integration risks rather than the dApp applications themselves. The *Six Times to Spare* paper [8] characterizes GPU-accelerated 5G NR LDPC decoding throughput on DGX Spark using Sionna LDPC5G, finding substantial GB10 compute headroom; our results extend this to the *full chain* integration, where kernel headroom is necessary but not sufficient when configuration defects break the integration layer.

dApp lineage and 5G testbeds. The dApp concept was introduced by Lacava et al. [2] and elaborated by Polese et al. [3], motivated by the inability of E2-based xApps to operate at sub-10 ms latency due to SCTP round-trips and Near-RT RIC overheads. Earlier work by Polese et al. [27] reports a multi-vendor O-RAN testbed using NVIDIA Aerial ARC and OpenAirInterface, the testbed predecessor of the ATB1.0 release we validate.

Linux real-time scheduling and configuration drift. The interaction between `SCHED_FIFO` and `SCHED_OTHER` threads on shared CPUs is well documented in the Linux kernel literature [25], [28], [29]; Finding 3 is a known anti-pattern surfacing in a new domain (GPU-accelerated 5G). Two of our findings (Findings 1 and 3) reduce to operator-side configuration drift during platform porting, a well-studied class of failures at scale (Tang et al. [30] on Facebook holistic configuration; Xu et al. [31] on configuration sprawl); our diagnostic methodology (§IX) is a RAN-specific instantiation of that broader validation discipline. Our framing aligns with the blameless-postmortem tradition (Allspaw [32], Google SRE [33]): rather than attribute the derivation defects to individual oversight, we identify the structural conditions (absent shipped reference, server-class CPU-map carry-over) under which such defects arise.

Position of this work. Our contribution is an *experience-report-with-methodology* on operator-side platform porting: a documented record of two operator-side derivation defects plus one vendor-side validation-mode policy gap, alongside three diagnostic techniques that we hypothesize generalize. Such experience reports are common in the OSDI/USENIX ATC tradition but rare in 5G/6G literature; we hope to encourage more of this style of contribution to the GPU-accelerated RAN community.

XII. Conclusion

We have presented an experience report on the operator-side bring-up of NVIDIA Aerial cuBB 26.1.0 on the DGX Spark ARM64 entry-level reference platform in the cable-on PF↔PF self-loopback topology. Because the public 26.1.0 release does not ship a DGX Spark cable-on F08-pattern cuphycontroller or L2 adapter, we derived these configurations locally from server-class siblings; this derivation is the immediate source of two of the three defects we report. We *firmly establish* Finding 1 (an operator-side static-slot-override omission causing 100% PUSCH CRC failure) and Finding 3 (an operator-side CPU-map collision silently starving the dApp telemetry pipeline), and *identify a plausible* vendor-side validation-mode fail-fast policy gap in Finding 2 (eight unconditional `EXIT_L1` call sites in the publicly released uplink driver). A cumulative patch of approximately fifteen lines restored the expected peak-cell throughput on our F08_1C_59 profile, with 1 h 6 min continuous soak and end-to-end dApp inference at the configured periodicity. The methodological contribution is three diagnostic techniques (§IX)—sibling-config diff, latent-code inspection, and cross-YAML CPU-map audit—supported by two ∼100-line Python linter prototypes that correctly surface the underlying defects on minimal synthetic configurations. Following community feedback from NVIDIA's Aerial team, GitHub Issues #43 and #44 were closed with retractions clarifying the operator-side derivation; Issue #45 (Finding 2) remains open at submission time. Future work includes WNC R1220-079L real-RU validation, OAI gNB integration, multi-run variance characterization, and multi-cell scalability under SR-IOV.

Acknowledgments

We thank the NVIDIA Aerial engineering team for the open release of the cuBB SDK and ATB1.0, and the NVIDIA Aerial Developer Forum community for the TAI/PHC synchronization recipe in thread 368569 that was a prerequisite for our work. Research conducted at NYCU.

AI Tool Disclosure

Per IEEE policy on AI-generated content,[2] we disclose use of large language model tools (Anthropic Claude) for (i) sentence-level editing and grammar refinement, (ii) literature-search assistance (all candidate references verified by the author against primary sources before inclusion), and (iii) LaTeX template scaffolding. All technical content (problem definition, root-cause analysis, experimental design, measurement, scheduler-trace interpretation, KPI computation, configuration patches, source modifications, and decisions about claims and their epistemic grading) was authored, executed, and verified by the named human author, who takes full responsibility for the manuscript.

[2]IEEE, "Using AI-Generated Content in an IEEE Article," IEEE author guidelines, 2024.

IP and Reproducibility Disclosure

All source-code references and analyses draw from the publicly released NVIDIA Aerial CUDA-Accelerated RAN repository under Apache License 2.0 [1], [18], the public Aerial Developer documentation [4], [17], [20], and the cited public NVIDIA Aerial Developer Forum threads [12], [21], [22]. We did not access, reproduce, or rely on NVIDIA Confidential Information. The YAML files modified for Findings 1 and 3 are operator-derived in our environment (§II-F); no NVIDIA-shipped YAML is redistributed by this paper or its artifact pack. All artifacts—YAML derivation deltas against public sibling references, source-line changes, validation logs, dApp inference traces, NVLOGC instrumentation, linter scripts, reproducibility checklist—are archived at the companion whitepaper [26] with SHA-256 manifest; the artifact pack will be archived to Zenodo for a permanent DOI on acceptance. Exact derivation deltas and SHA-256 hashes of the locally derived files appear in Appendix A.

Appendix A
Reproducibility

Platform. NVIDIA DGX Spark / GB10 (10×Cortex-X925 @ 4.0 GHz + 10×Cortex-A725 @ 2.8 GHz aarch64, integrated Blackwell `sm_121`, 128 GB LPDDR5x unified, ConnectX-7 with two QSFP ports), Ubuntu 24.04 (NVIDIA DGX OS), kernel 6.17.0-1014-nvidia, driver 590.48.01, CUDA 13.1.1, DOCA 3.2.1025-1, OFED 25.10-1.7.1, ConnectX-7 firmware 28.47.1088. Container `nvcr.io/nvidia/aerial/aerial-cuda-accelerated-ran:26-1-cubb` (source tag 26.1.0), run via NVIDIA-supplied `run_aerial.sh` (`--ipc=host`).

YAML provenance. `cuphycontroller_F08_GB10_CABLE_v8.yaml` and `l2_adapter_F08_GB10.yaml` are operator-derived and not in the public 26.1.0 release tag (verified by `git ls-files --error-unmatch`). The cuphycontroller was adapted from `cuphycontroller_P5G_WNC_DGX.yaml` taking F08-pattern parameters from `cuphycontroller_F08_R750.yaml`; the L2 adapter was adapted from `l2_adapter_config_F08_R750.yaml` with CPU pinnings re-mapped (incorrectly, as Finding 3 documents) for the 20-core GB10 layout. All three public sibling L2 adapters we referenced (`F08`, `F08_R750`, `SE_DU`) contain the `static*SlotNum:0` entries omitted from our derived L2 adapter (Finding 1).

Patch summary. *Finding 1*: add five `static*SlotNum:0` entries to the derived L2 adapter, matching public siblings. *Finding 2*: `aggr_obj_non_avail_th: 5` → `1000` in the derived cuphycontroller, plus commenting out the eight

`EXIT_L1(EXIT_FAILURE);` call sites at public-release lines 515, 777, 958, 1137, 1302, 1526, 1944, 2189 in `task_function_ul_aggr.cpp`; rebuild via the standard cmake chain. *Finding 3*: `data_config.data_core: 11 → 13`.

Measurement sequence. (i) apply the TAI/PHC synchronization recipe [22] from cold boot; (ii) start `cuphycontroller_scf` with profile `F08_GB10_CABLE_v8` (wait 13 s); (iii) start `ru_emulator F08 1C 59 --config config_cable_v6.yaml` (wait 7 s); (iv) start `test_mac F08 1C 59`, collect 60 s of steady-state stats after a 30 s warm-up; (v) for dApp runs bring up `prb-power-python` in the ACAR IPC namespace via `docker compose up -d`, observe inference rate over 25 s. All reported numbers are single-run; multi-run / multi-unit / multi-operator characterization is future work (§X).

References

[1] NVIDIA Corporation, "NVIDIA Open Sources Aerial Software to Accelerate AI-Native 6G," NVIDIA Blog, Oct. 2025, posted October 28, 2025. https://blogs.nvidia.com/blog/open-source-aerial-ai-native-6g/.

[2] A. Lacava *et al.*, "dApps: Enabling Real-Time AI-Based Open RAN Control," *arXiv preprint arXiv:2501.16502*, Jan. 2025, https://arxiv.org/abs/2501.16502.

[3] M. Polese *et al.*, "dApps: Distributed Applications for Real-time Inference and Control in O-RAN," *arXiv preprint arXiv:2203.02370*, 2022, https://arxiv.org/abs/2203.02370.

[4] NVIDIA Corporation, "Aerial CUDA-Accelerated RAN Release 26.1.0 Release Notes," Apr. 2026, https://docs.nvidia.com/aerial/cuda-accelerated-ran/latest/aerial-cuda-accelerated-ran.pdf.

[5] NVIDIA Corporation, "Aerial Sample Apps 1.0," May 2026, https://github.com/NVIDIA/aerial-sample-apps.

[6] B. Wensley and B. Rowsell, "Using NVIDIA Aerial CUDA-Accelerated RAN on Red Hat OpenShift to Accelerate Development of AI-Native 5G and 6G RAN Solutions," Red Hat Blog, Apr. 2026, red Hat Blog, 29 Apr 2026; redhat.com/en/blog/using-nvidia-aerial-cuda-accelerated-ran-red-hat-openshift-...

[7] D. Villa, M. Belgiovine, N. Hedberg, M. Polese, C. Dick, and T. Melodia, "Programmable and GPU-Accelerated Edge Inference for Real-Time ISAC on NVIDIA Aerial Testbed," *arXiv preprint arXiv:2512.06493*, Dec. 2025, https://arxiv.org/abs/2512.06493.

[8] R. Barker, J. Boone, T. Seyfi, A. E. Dorcheh, F. Afghah, and J. Boccuzzi, "Six Times to Spare: Characterizing GPU-Accelerated 5G LDPC Decoding for Edge-RSU Communications," *arXiv preprint arXiv:2602.04652*, Feb. 2026, https://arxiv.org/abs/2602.04652.

[9] H.-C. Tsai, "[ACAR 26-1][DGX Spark] PRB Power dApp receives no E3 indications when data_core collides with RT msg_processing thread," GitHub Issue, NVIDIA/aerial-cuda-accelerated-ran #43; closed with operator-side retraction 2026-05-16, May 2026, https://github.com/NVIDIA/aerial-cuda-accelerated-ran/issues/43.

[10] H.-C. Tsai, "[ACAR 26-1][DGX Spark] Cable-on loopback: 100% PUSCH UL CRC fail unless staticPuschSlotNum: 0 is added to the L2 adapter YAML," GitHub Issue, NVIDIA/aerial-cuda-accelerated-ran #44; closed with operator-side retraction 2026-05-16, May 2026, https://github.com/NVIDIA/aerial-cuda-accelerated-ran/issues/44.

[11] H.-C. Tsai, "[ACAR 26-1][cuphydriver] Request config knob for recoverable UL pipeline timeout in cable-on self-loop (follow-up to #41)," GitHub Issue, NVIDIA/aerial-cuda-accelerated-ran #45; open as of 2026-05-12, May 2026, https://github.com/NVIDIA/aerial-cuda-accelerated-ran/issues/45.

[12] H.-C. Tsai, "Aerial 26-1 DGX Spark Cable-on Loopback: 100% PUSCH CRC Fail; staticPuschSlotNum: 0 Appears Required for TV Replay," NVIDIA Aerial Developer Forum, thread 369748, May 2026, https://forums.developer.nvidia.com/t/369748.

[13] 3GPP, "NR; Physical channels and modulation," 3GPP TS 38.211, Release 18, 2025, https://www.etsi.org/deliver/etsi_ts/138200_138299/138211/.

[14] O-RAN ALLIANCE, "O-RAN.WG4.CUS-Plane: Control, User and Synchronization Plane Specification," O-RAN ALLIANCE Technical Specification, 2024.

[15] O-RAN ALLIANCE, "O-RAN.WG4.M-Plane: Management Plane Specification," O-RAN ALLIANCE Technical Specification, 2024.

[16] CPRI Cooperation, "Common Public Radio Interface: eCPRI Specification V2.0," 2019, http://www.cpri.info/.

[17] NVIDIA Corporation, "NVIDIA Aerial CUDA-Accelerated RAN Documentation," 2026, https://docs.nvidia.com/aerial/cuda-accelerated-ran/latest/index.html.

[18] NVIDIA Corporation, "NVIDIA/aerial-cuda-accelerated-ran GitHub repository," 2026, https://github.com/NVIDIA/aerial-cuda-accelerated-ran.

[19] Small Cell Forum, "5G FAPI: PHY API Specification, SCF222 v5.0," 2024, https://www.smallcellforum.org/work-items/fapi/.

[20] NVIDIA Corporation, "DGX Spark User Guide," May 2026, https://docs.nvidia.com/dgx/dgx-spark/dgx-spark.pdf.

[21] NVIDIA Aerial Team, "Aerial CUDA-Accelerated RAN 26-1, Aerial Testbed 1.0 and Aerial Sample Apps 1.0 Release Notification," NVIDIA Aerial Developer Forum, thread 369091, Apr. 2026, https://forums.developer.nvidia.com/t/369091.

[22] NVIDIA Aerial Team, "TAI/PHC Synchronization Recipe for Cable-on Self-Loopback," NVIDIA Aerial Developer Forum, thread 368569, 2026.

[23] H.-C. Tsai, "[26.1] aerial-fh-driver: NIC link-down should not be FATAL — nic.cpp:609 blocks cable-less ru_emulator/cuphycontroller bring-up," GitHub Issue, NVIDIA/aerial-cuda-accelerated-ran #41, May 2026, https://github.com/NVIDIA/aerial-cuda-accelerated-ran/issues/41.

[24] Linux man-pages project, "sched(7) — overview of CPU scheduling," Linux manual page, 2024, https://man7.org/linux/man-pages/man7/sched.7.html.

[25] J. Corbet, "SCHED_FIFO and realtime throttling," LWN.net, 2008, https://lwn.net/Articles/296419/.

[26] H.-C. Tsai, "DGX Spark Aerial cuBB Cable-on Bring-up White Paper," HackMD whitepaper, 13 chapters, May 2026, https://hackmd.io/@thc1006/HJKy85kJzl.

[27] M. Polese *et al.*, "An Open, Programmable, Multi-vendor 5G O-RAN Testbed with NVIDIA ARC and OpenAirInterface," *arXiv preprint arXiv:2310.17062*, 2023, https://arxiv.org/abs/2310.17062.

[28] Red Hat, "Tuning scheduling policy — RHEL real-time documentation," 2024, https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/10/html/monitoring_and_managing_system_status_and_performance/tuning-scheduling-policy.

[29] Linux kernel community, "Real-Time Linux Wiki (PREEMPT_RT)," wiki.linuxfoundation.org/realtime, 2024, https://wiki.linuxfoundation.org/realtime/start.

[30] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic configuration management at Facebook," in *Proc. 25th Symposium on Operating Systems Principles (SOSP)*, 2015, pp. 328–343.

[31] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs! understanding and dealing with over-designed configuration in system software," in *Proc. 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015, pp. 307–319.

[32] J. Allspaw, "Blameless PostMortems and a just culture," Etsy Code as Craft engineering blog, May 2012, https://www.etsy.com/codeascraft/blameless-postmortems.

[33] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Eds., *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.