5G OAI Neural Receiver Testbed with USRP X410

Contents

AN-829

Bharat Agarwal and Neel Pandeya

This Application Note presents a practical, system-level benchmarking platform leveraging NI USRP software-defined radios (SDRs) and the OpenAirInterface (OAI) 5G/NR stack for evaluating AI-enhanced wireless receivers in real-time. It addresses one of the key challenges in deploying AI/ML at the physical layer-ensuring reliable system performance under real-time constraints.

AI and ML techniques hold promise for improving both wireless and non-wireless KPIs across the stack, from core-level optimization (e.g., load balancing, power savings), to tightly-timed PHY/MAC innovations such as:

- ML-based digital predistortion to improve power efficiency.
- Neural receivers for channel estimation and symbol detection with improved SNR tolerance.
- Intelligent beam and positioning prediction, even under fast channel dynamics.

Consortia such as 3GPP (Release-18/19) and O-RAN are actively defining how AI/ML can be incorporated into future cellular network standards.

We demonstrate a real-time implementation of a neural receiver that is based on a published model architecture called DeepRX, which replaces the traditional OFDM receiver blocks (channel estimation, interpolation, equalization, detection) with a single neural network that treats the time-frequency grid data as image-like input. Model training and validation are performed using the NVIDIA Sionna link-level simulator, and training data is stored using the open SigMF format for reproducibility.

More information about the SigMF file format can be found on the project website, on the Wikipedia page, and on the GitHub page.

The original paper, "DeepRx: Fully Convolutional Deep Learning Receiver", by Mikko Honkala, Dani Korpi, Janne M.J. Huttunen, can be found here and here.

To validate the performance of the neural receiver in real hardware, the prototype integrates:

- The OAI real-time 5G protocol stack (complete core, RAN, and UE) running on commodity CPUs.
- NI USRP SDR hardware as the RF front-end.
- An optional O-RAN Near-RT RIC (via FlexRIC) integration.
- Neural receiver inference performed on a GPU (e.g., Nvidia A100, RTX 4070, RTX 4090), accessed via TensorFlow RT C-API for seamless integration within OAI.

This setup enables a direct comparison between the traditional receiver baseline against the neural receiver in an end-to-end real-time system.

Initial testing focuses on uplink performance using various MCS levels (MCS-11, MCS-15, MCS-20 are specifically highlighted in this document) and SNR ranges (5 dB to 18 dB) under a realistic fading channel profile (urban micro, 2 m/s, 45ns delay spread). Each measurement is averaged over 300 transport blocks.

Some of the key findings are listed below.

- The neural receiver shows a clear Bit Error Rate (BER) advantage at lower MCS and lower SNR.
- At higher MCS levels, the performance gap narrows (a trade-off that merits further analysis).
- A reduced uplink bandwidth was used to meet strict real-time latency requirements (500 ?s slot duration with 30 KHz SCS).
- The neural receiver model complexity was reduced by 15 times (from 700K to 47K parameters) to achieve real-time GPU inference.

These results underscore the crucial balance between complexity, latency, and performance in AI-enhanced wireless physical-layer deployments.

The testbed demonstrates a realistic path from simulation to real-time deployment of neural receiver models. This workflow supports rapid prototyping, robust AI model validation, and exploration of architecture-performance trade-offs.

Some key takeaways are listed below.

- AI/ML models can be efficiently integrated into real-time wireless stacks using SDR hardware and GPU inference.
- Low-complexity models offer promising performance improvements while satisfying real-time constraints.
- Synchronized dataset generation and automated test workflows enable scalable ML benchmarking across scenarios.
- The framework allows researchers to investigate unexpected behaviors and robustness in AI-native wireless systems.

Ultimately, the methodology bridges AI/ML conceptual research and realistic deployment, advancing trust and utility in AI-powered future wireless systems.

The Universal Software Radio Peripheral (USRP) devices from NI (an Emerson company) are software-defined radios which are widely used for wireless research, prototyping, and education. The hardware specifications for the various USRP devices are listed elsewhere on this Knowledge Base (KB). For this Neural Receiver implementation described in this document, we use the USRP X410. The USRP X440 may also be used, with some further adjustments to the system configuration.

The resources for the USRP X410 are listed below.

The Hardware Resource page for the USRP X410 can be found here.

The product page for the USRP X410 can be found here.

The User Manual for the USRP X410 can be found here.

The resources for the USRP X440 are listed below.

The Hardware Resource page for the USRP X410 can be found here.

The product page for the USRP X410 can be found here.

The User Manual for the USRP X410 can be found here.

The USRP X410 is connected to the host computer using a single QSFP28 100 Gbps Ethernet link, or using a QSFP28-to-SFP28 breakout cable, which provides four 25 Gbps SFP28 Ethernet links. On the host computer, a 100 Gbps or 25 Gbps Ethernet network card is used to connect to the USRP.

The USRP X410 devices are synchronized with the use of a 10 MHz reference signal and a 1 PPS signal, distributed from a common source. This can be provided by the OctoClock-G (see here and here for more information).

For control and management of the USRP X410, a 1 Gbps Ethernet connection to the host computer is needed, as well as a USB serial console connection.

Further details of the hardware configuration will be discussed later in this document.

The software stack running on the computers used in this implementation are as listed below.

- Ubuntu 22.04.5, running on-the-metal, and not in any Virtual Machine (VM)
- UHD version 4.8.0.0
- Nvidia drivers version 535
- Nvidia CUDA version 12.2
- TensorFlow 2.14

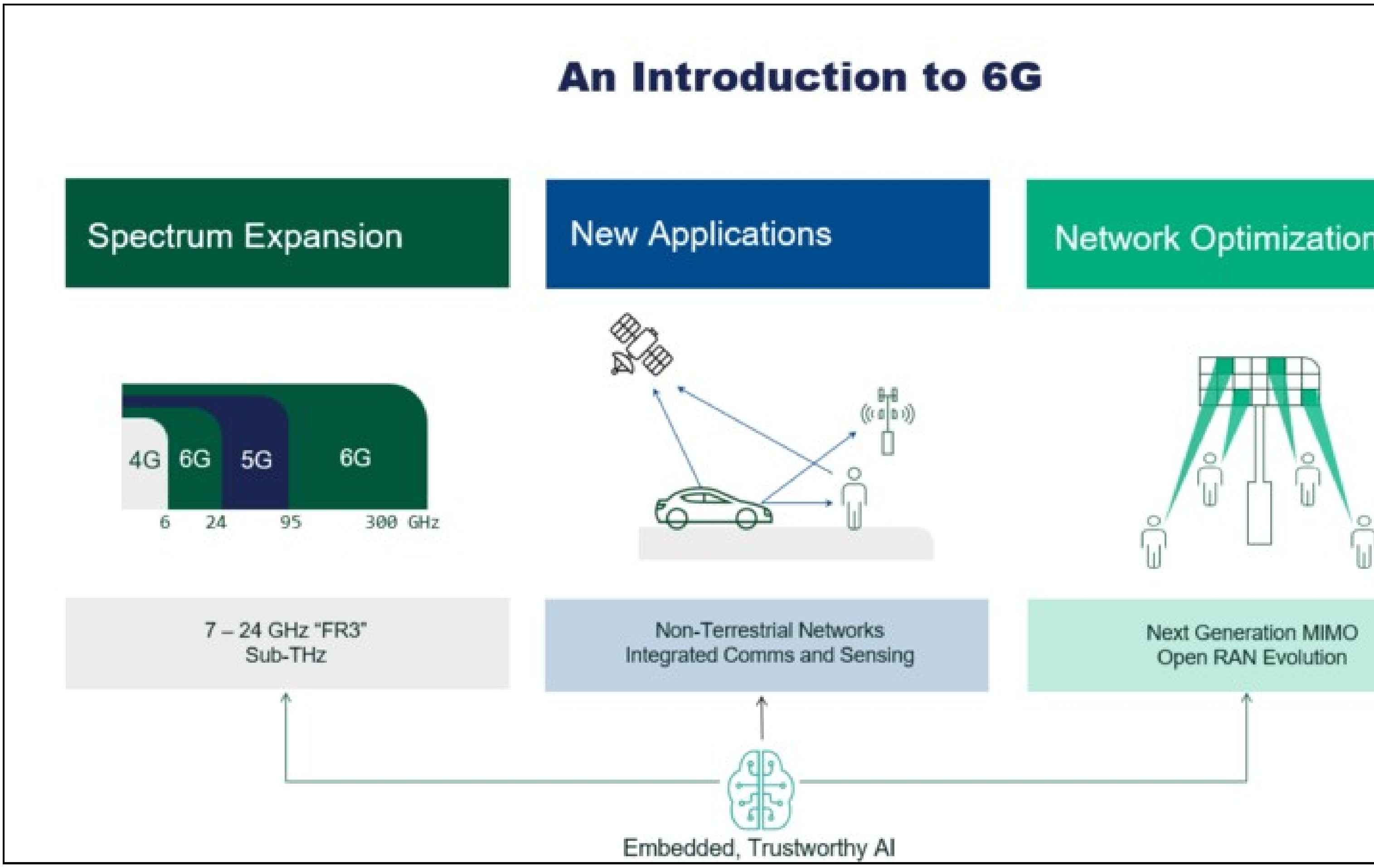
For the OAI gNB, the OAI UE, and the FlexRIC, there will be NI-specific versions used, and these will be obtained from an NI repository on GitHub.

Note that the Data Plane Development Kit (DPDK) is not used in this implementation. However, it may may be helpful when using higher sampling rates.

Further details of the software configuration will be discussed later in this document.

The figure listed below highlights the vision for sixth-generation (6G) wireless systems. Beyond incremental improvements, 6G introduces three major advances, as listed below.

- **Spectrum Expansion**: Extending from traditional sub-6 GHz and mmWave bands into FR3 (7 to 24 GHz) and sub-THz (up to 300 GHz), enabling ultra-wide bandwidths and unprecedented data rates.
- **New Applications**: Integration of non-terrestrial networks (NTN) with terrestrial infrastructure and joint communication-and-sensing (JCAS) functionalities, supporting use cases such as connected vehicles, satellite-augmented IoT, and immersive XR.
- **Network Optimization**: Advancements in massive MIMO, multi-user beamforming, and Open RAN disaggregation, improving spectral efficiency, flexibility, and energy sustainability.

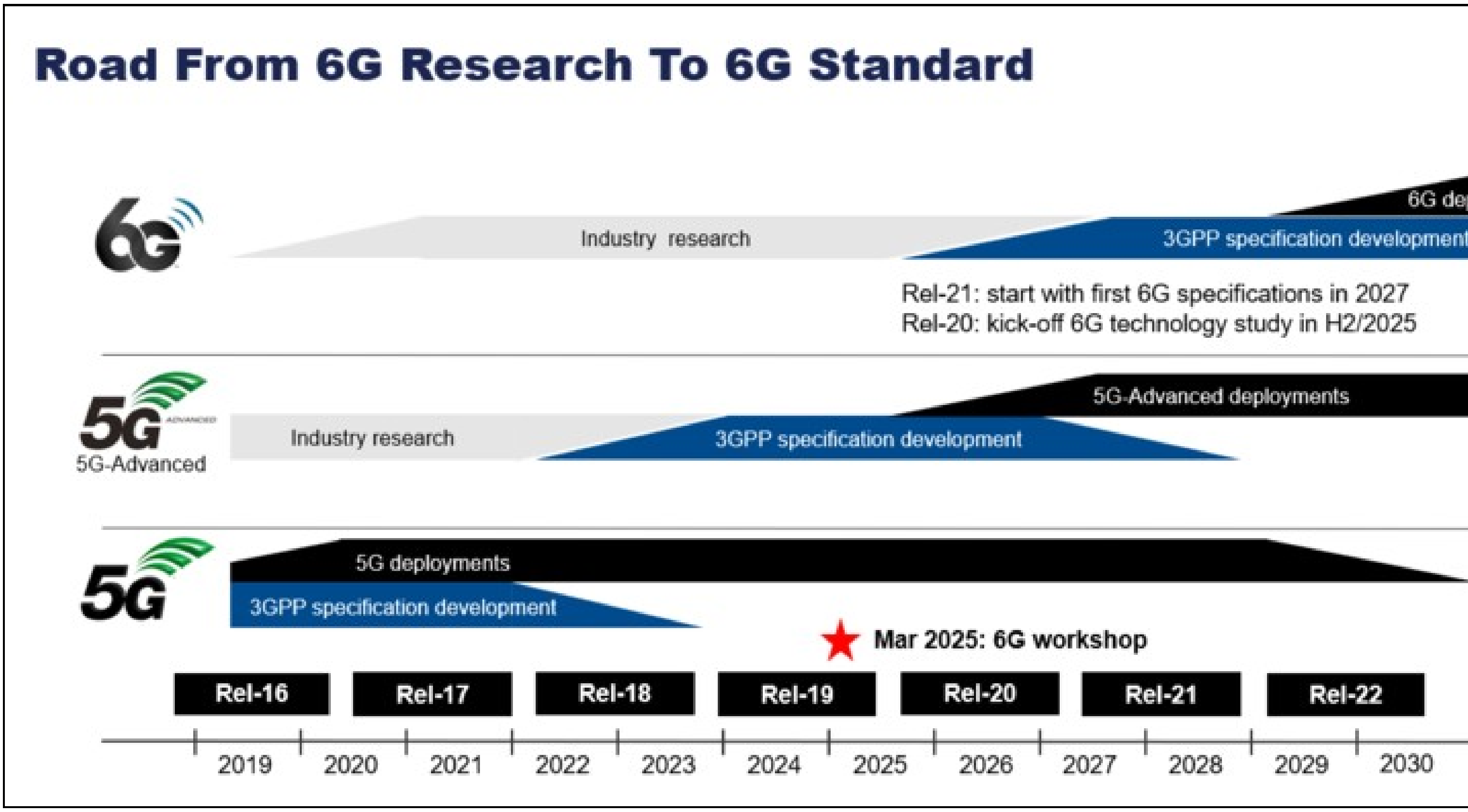


Key pillars of 6G development. (Left) Spectrum expansion into FR3 (7?24 GHz) and sub-THz (up to 300 GHz) to support wider bandwidths. (Center) New applications such as non-terrestrial networks (satellite and UAV integration) and integrated communications and sensing (ICAS). (Right) Network optimization through next-generation MIMO and Open RAN evolution. Across all pillars, embedded and trustworthy AI acts as a central enabler Across these pillars, embedded and trustworthy AI is the key enabler, providing intelligence for spectrum management, adaptive receivers, and end-to-end optimization.

These trends highlight that 6G will operate in highly challenging environments with wideband sub-THz channels, dynamic non-terrestrial links, and complex multi-user MIMO topologies. Traditional linear detection techniques such as ZF or MMSE struggle to cope with hardware non-idealities, nonlinear channel distortions, and the stringent latency and reliability targets of 6G. To address these limitations, the concept of a Neural Receiver has emerged. By embedding deep learning models directly into the receiver chain, neural receivers can learn from real measured impairments, jointly optimize channel estimation and detection, and deliver significant performance gains over classical approaches. This makes neural receivers a key building block for realizing the vision of embedded, trustworthy AI in 6G physical layer design.

The figure listed below illustrates the expected timeline from ongoing 5G research through to the first 6G deployments.

- **5G (Release-16 to Release-18)**: 3GPP initiated 5G specification development in Release-15 and Release-16, followed by commercial deployments from 2019 onward. Work on Release-17 and Release-18 (2021 to 2024) extends 5G capabilities in areas such as URLLC, industrial IoT, and positioning.
- **5G-Advanced (Release-18 to Release-20)**: Industry research and specification development converge to define 5G-Advanced features. Deployments are expected around 2025 to 2027, focusing on improved energy efficiency, AI/ML-native functions, and expanded NTN integration.
- **6G (Release-20 onward)**: Formal 6G technology studies will begin with Release-20 in H2 2025, marked by the first official 6G workshop in March 2025. Standardization of 6G specifications is planned for Release-21 in 2027, with early 6G deployments projected for the end of the decade (around 2030).


Road From 6G Research To 6G Standard
6G
Industry research
6G de
3GPP specification development
Rel-21: start with first 6G specifications in 2027
Rel-20: kick-off 6G technology study in H2/2025
5G ADVANCED
5G-Advanced
Industry research
3GPP specification development
5G-Advanced deployments
5G
5G deployments
3GPP specification development
Mar 2025: 6G workshop
Rel-16
Rel-17
Rel-18
Rel-19
Rel-20
Rel-21
Rel-22
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030

Roadmap from 6G research to standardization. The timeline shows the progression from 5G and 5G-Advanced to 6G, aligned with 3GPP releases and industry milestones.
The figure above highlights the transition from 5G deployments to the research and standardization cycles of 5G-Advanced and 6G. This staged process ensures backward compatibility, while paving the way for disruptive innovations in spectrum use, AI-native networks, and new application domains.

As shown in the figure above, the transition from 5G to 6G is not only a matter of spectrum expansion and new use cases, but also of embedding AI-native functionalities into the air interface itself. Release-20 (2025) will mark the start of 6G technology studies, providing an opportunity to evaluate disruptive physical layer techniques such as Neural Receivers. These receivers directly integrate deep learning models into the detection chain, enabling them to cope with nonlinearities, hardware impairments, and the extreme bandwidths expected in FR3 and sub-THz bands. By Release 21 (2027), as 6G specifications are defined, neural receivers and other AI-based PHY innovations will play a crucial role in realizing the vision of AI-native 6G, where intelligence is embedded from the physical layer up to the application layer.

This section highlights the three main challenges hindering the seamless integration of AI into wireless communication systems. The challenges are listed with increasing levels of AI readiness.

- **Data Scarcity**
    - Meaning:
        - Wireless networks often lack sufficient labeled and diverse datasets.
    - Why it's a problem:
        - Collecting and labeling large wireless datasets is expensive and time-consuming.
        - Real-time data is sparse or kept proprietary by operators/vendors.
        - Rare but critical scenarios (handover failures, deep fades, interference spikes) are underrepresented.
    - Impact:
        - Models trained on limited data risk poor generalization and biased decision-making.

- **Data Quality**
    - Meaning:
        - Available data may not be clean, representative, or consistently labeled.
    - Why it's a problem:
        - Measurements are noisy due to sensors or network logging errors.
        - Labeling mistakes propagate errors into AI models.
        - Data is biased toward specific environments (e.g., urban, indoor) and not generalizable to others.
    - Impact:
        - Low-quality data reduces model reliability, leading to unstable or inaccurate predictions.

- **Data Relevance**
    - Meaning:
        - Even when data exists, it may not directly match the target AI task or deployment scenario.
    - Why it's a problem:
        - LTE datasets may not transfer well to 5G/6G systems.
        - Lab-collected data ignores mobility, blockage, or coexistence effects.
        - Training distributions drift away from real-time operational data.
    - Impact:
        - AI performs well in simulation but degrades in live networks (gap between simulation and real-world).

The takeaway is that the three challenges of Scarcity, Quality, and Relevance form the key bottleneck for wireless AI, and that addressing them requires:

- Synthetic data generation (digital twins, simulators, ray-tracing),
- Federated learning (distributed training without data centralization),

- Data curation pipelines (cleaning, validation, domain adaptation).

A neural receiver is a machine learning-based physical layer receiver that replaces or augments traditional signal processing blocks?such as channel estimation, equalization, and detection?with a unified, data-driven model. In contrast to conventional receivers that rely on handcrafted algorithms and strict mathematical models of the wireless channel, neural receivers learn to perform these operations jointly by training on large datasets of labeled I/Q samples or OFDM resource grids.

The table below explains some of the differences between components in traditional receiver architectures and components in neural receiver architectures.

Comparison of Traditional vs Neural Receiver Architectures

Component	Traditional Receiver	Neural Receiver
Channel Estimation	Least Squares (LS), MMSE estimators	Learned directly from pilot and data patterns
Equalization	Zero-Forcing, MMSE equalizers	Implicitly learned during training
Symbol Detection	QAM demodulation, hard/soft decision	Jointly learned with other tasks
Architecture	Modular, deterministic	End-to-end differentiable neural network
Input	OFDM resource grid or raw IQ samples	IQ tensors or pilot+data grid
Output	Estimated bits or LLRs	Bits or probabilities

A commonly used architecture like DeepRx treats the resource grid as a 2D input, similar to an image, where time and frequency correspond to axes. This allows the use of convolutional neural networks (CNNs), recurrent neural networks (RNNs), or transformer-based models.

- **Input**: Complex-valued OFDM resource grid, with pilot and data symbols.
- **Layers**: Convolutional or attention-based layers extract spatial and temporal features.
- **Output**: Recovered symbols or bits with associated confidence scores.

The items below describe the training process that was used in this implementation.

The use of the SigMF data format allows for the storage of raw IQ data with comprehensive metadata, which provides context.

- Dataset: Generated from link-level simulation tools, such as Sionna, under various channel models (AWGN, LOS, NLOS, 3GPP, etc.).

- Format: Datasets are stored in the SigMF format, containing raw IQ samples with metadata.

- Loss Function: Cross-entropy or binary cross-entropy; optionally soft LLR loss for reliability-aware decoding.

- Optimizer: Adam, SGD, or custom schedulers suitable for low-SNR scenarios.

Trained models are deployed in real-time using TensorRT run-times on edge hardware such as:

- GPUs: NVIDIA A100, RTX 4090, RTX 4070, RTX 4060.

- Integration: Plugged into OAI physical-layer receiver chains.

- Latency: Achieves under 500 ?s processing delay for 30 KHz SCS, meeting real-time subframe timing requirements.

The table below shows a comparison of various performance metrics between traditional receiver architectures and neural receiver architectures.

Performance Comparison of Traditional vs Neural Receiver Architectures

Metric	Traditional Receiver	Neural Receiver
Block Error Rate (BLER) under low SNR	Higher	Lower (up to 3 dB gain)
Complexity	Fixed, low	Tunable, moderate
Latency	Very low	Real-time, under 500 ?s
Generalization	Poor to unseen channels	Better with diverse training data
Interpretability	High (white box)	Lower (black box)

The list below highlights some common use-cases for Neural Receivers.

- **Uplink Neural PHY Receiver**: Real-time decoding at gNB.
- **Channel Tracking**: Adaptive to fast-fading and mobility scenarios.
- **Joint Equalization and Detection**: Reduces end-to-end BER and BLER.
- **Massive MIMO**: Scalable to high-dimensional antennas with deep models.

There are several challenges to the practical realization of Neural Receivers, as listed below.

- Requires extensive datasets for generalization.
- Less interpretable compared to traditional receiver implementations.
- Hardware deployment must meet strict real-time constraints.
- Careful calibration and interface with existing stacks (such as OAI) are needed.

The neural receiver presents a promising paradigm for future 6G systems by enabling adaptive, intelligent, and performance-enhancing physical-layer decoding using machine learning. When paired with platforms like the NI USRP radios and AI accelerators, it opens the path for real-time AI-native physical-layer design.

To ensure a consistent and reproducible set-up for our AI-enabled wireless testbed, we employ a systematic validation checklist. The table below summarizes the key checks, commands, and expected outputs that must be verified before conducting experiments. This process guarantees that both the hardware (e.g., USRPs, GPUs) and the software (e.g., operating systems, drivers, TensorFlow, UHD) are correctly installed and aligned with the requirements.

The checklist covers three broad areas:

- **Operating system and hardware readiness**: Includes verification of the installed OS, BIOS version, kernel, and GPU drivers.
- **USRP connectivity and configuration**: Ensures that USRPs are discovered, their file systems are compatible with UHD, and network parameters (e.g., MTU size, socket memory) are tuned for high-throughput streaming.
- **Software stack and runtime optimization**: Covers validation of TensorFlow, NVIDIA TensorRT, and TF C-API installation, as well as disabling unnecessary system services (e.g., updates, GNOME display manager) that may negatively impact performance.

This structured approach minimizes setup errors and improves reproducibility across different machines and deployments.

Listing of checks for system bring-up and set-up

Check Item	Command	Desired Output
Operating System	`hostnamectl`	Ubuntu 22.04.5
BIOS version	`sudo dmidecode -s bios-version`	Check system vendor for latest version
Verify GPU	`lspci | grep -i nvidia`	Example: RTX 4090 may appear as `17:00.0 VGA compatible controller: NVIDIA Corporation Device 2684 (rev a1)`
Nvidia driver version and CUDA version	`nvidia-smi`	Nvidia driver version 535.183.01 and CUDA version 12.2
GPU Load	`nvidia-smi`	Load in percentage
Kernel version	`uname -r`	`6.5.0-44-generic`
Cores operation mode	`cat /sys/devices/system/cpu/cpu*/cpufreq/scaling governor`	All cores should show `performance`
Cores clock rate	`watch -n1 "grep ^[c]pu MHz" /proc/cpuinfo`	Should be larger than base clock rate and less than turbo clock rate
UHD version	`uhd config info --print-all`	`4.8.0.0`
IP address of USRP	`uhd find devices`	List of all connected USRPs with IP addresses
USRP specifications	`uhd usrp probe`	All USRP specifications printed without any errors
USRP claimed status	`uhd find devices`	`Claimed: false`
MTU of Ethernet ports	`ifconfig`	`9000`

Socket buffer sizes	`cat /proc/sys/net/core/rmem max`
	`cat /proc/sys/net/core/wmem max`
		`62500000`

Disable System Update	Disable updates in OS settings	Ensures stable test environment
Disable GNOME Display Manager	`sudo apt remove gdm gdm3`	Prevents unnecessary excess load on system

The table below summarizes the wireless scenario and associated gNB and UE configuration used in the demo setup. The experiment is simplified to a single link between one base station and one user equipment, without interference from neighboring cells or other users.

The system operates in 3GPP NR band n78 with a 40 MHz bandwidth and a subcarrier spacing of 30 KHz using CP-OFDM modulation. A total of six PRBs are scheduled per UE, corresponding to an effective transmission bandwidth of approximately 2.16 MHz.

The TDD frame structure follows a 5 ms periodicity, with 7 downlink slots and 6 additional downlink symbols, complemented by 2 uplink slots and 4 uplink symbols. Uplink transmissions employ PUSCH mapping type B, spanning 13 OFDM symbols. DMRS pilot signals are configured as Type 1 with an additional position pos2, leading to DMRS placement at OFDM symbols 0, 5, and 10.

This baseline configuration provides a controlled environment for evaluating the fundamental link performance of the testbed.

gNB and UE configuration for the wireless scenario in the demo setup

Parameter	Configuration
Wireless Scenario	Single link between one BS (gNB) and one UE No interference from neighboring cells or other UEs
Config File	`gnb.band78.sa.fr1.106PRB.1x1.usrpx410_3300MHz.conf`
Operating Band	n78
Waveform	CP-OFDM
Subcarrier Spacing (SCS)	30 KHz
System Bandwidth	40 MHz
Scheduled PRBs per UE	6
Scheduled Transmission Bandwidth	2.16 MHz
TDD Pattern	DL/UL periodicity of 5 ms
Downlink Slots	7
Downlink Symbols	6
Uplink Slots	2
Uplink Symbols	4
PUSCH Mapping	Type B, duration = 13 OFDM symbols
PUSCH DMRS Configuration	Type 1, dmrs-AdditionalPosition = pos2 DMRS in OFDM symbols 0, 5, and 10

The table listed below details the bill of materials used in the experimental 5G testbed setup. The infrastructure is divided into separate blocks for data recording, gNB, UE, and RF hardware.

The host computers for the gNB and UE are each implemented on a Dell Precision 5860 server, which has an Intel Xeon W7-2495X CPU (24 physical cores with turbo frequency of 4.8 GHz), and 25 Gbps Ethernet network cards. The gNB system also includes an Nvidia RTX 4090 GPU to accelerate AI-based signal processing workloads. The UE system uses a much less powerful GPU, as no AI acceleration is needed.

Both gNB and UE connect to USRP X410 radios via a Mellanox Nvidia ConnectX-5 MCX512A-ACAT network card (the Intel X710-DA2 may also be used), and QSFP28-to-SFP28 breakout cables.

The OctoClock-G distributes a common 10 MHz reference signal and 1 PPS signal across all USRP devices.

This hardware configuration ensures support for wideband operation, supporting channel bandwidths from 40 MHz to 100 MHz, and multiple MIMO layers, enabling advanced AI-driven signal processing and performance evaluation in a realistic testbed.

The testbed is designed to operate on a range of high-performance workstation and server platforms that provide sufficient compute, I/O, and GPU acceleration for 5G/6G physical-layer and application-layer experimentation. The hardware listed below represents configurations that have been tested and validated for different use cases such as baseband processing, AI-driven link adaptation, and machine learning training. Note that other hardware configurations are possible.

Listed below are the supported, validated, and tested server platforms:

- Dell Precision 5860 Tower: This system is the current baseline workstation for gNB and UE host computer deployments. It supports PCIe Gen-4, 25 Gbps and 100 Gbps Ethernet network cards, and large DDR5 memory configurations.
- Lenovo ThinkStation P8: Verified for both inference and control-plane workloads; features comparable expansion and thermal capacity to the 5860.
- Gigabyte Server Platform: Used for high-throughput data recording and multi-GPU configurations, supporting rackmount deployment scenarios.
- Dell Precision 5820 Tower: This is a legacy system that has been succeeded by the Dell 5860. Previously used in early iterations. It is still functional, but limited to PCIe Gen-3 and DDR4 memory.

Listed below are the tested and recommended GPUs:

- Nvidia RTX 4060, 4070, 4090: Validated for real-time baseband acceleration, AI inference, and model training tasks. The RTX 4090 provides the best trade-off between power, cost, and tensor performance.
- Nvidia A100: This is a legacy GPU that was previously used for large-scale neural receiver training. It has been superseded by RTX-class GPUs for compact testbed setups.

These GPU and server combinations provide the flexibility to deploy either component (either the gNB or the UE) on the same hardware architecture, simplifying replication and scaling across the testbed. Systems with PCIe Gen-4 or higher are recommended to ensure sufficient bandwidth for 25 Gbps and 100 Gbps Ethernet cards and for real-time data streaming from USRP X410 devices.

Additional compatible systems and updated configuration guidance can be found on the USRP X410 page in the Knowledge Base (KB), which includes validated NICs, timing sources, and synchronization accessories.

The Dell Precision 5860 Tower workstation is used as the main server platform for the gNB and UE in the 5G/6G testbed. It provides the compute, memory, and I/O performance required for advanced AI-driven wireless communication experiments.

Some of the key features of this system are:

- High-performance workstation designed for AI-driven wireless testbeds.
- Supports multiple NICs and high-throughput data interfaces.
- Optimized for GPU acceleration (e.g., NVIDIA RTX 4090).
- Expandable PCIe slots for USRP connectivity and RF front-ends.
- Large DDR5 memory capacity for parallel workloads.
- Suitable for both gNB and UE roles in 5G/6G experiments

The Dell Precision 5860 Tower was selected as the system for the gNB because it supports: ? Multiple 25 Gbps Ethernet cards for high-speed connectivity with USRP X410 radios and data recording servers. ? Nvidia RTX-class GPUs to accelerate physical-layer baseband processing and AI workloads. ? Large DDR5 memory (up to 1 TB) for handling real-time scheduling and parallel computation. ? The Intel Xeon W7-2495X CPU, with 24 physical cores and 4.50 GHz turbo clock, to meet the demands of high sampling rates and multi-layer MIMO.

This flexibility allows the same hardware platform to serve as both gNB and UE, simplifying testbed deployment and ensuring scalability for future AI-driven enhancements.

Note that as the sample rate increases, for example, to 122.88 Msps for a 100 MHz channel bandwidth, and also as the number number of MIMO layers increases, for example, to 2x2 and 4x4, then the computational demands increase significantly, and more and more physical cores would be needed to support this load.

Dell Precision 5860 Tower Workstation

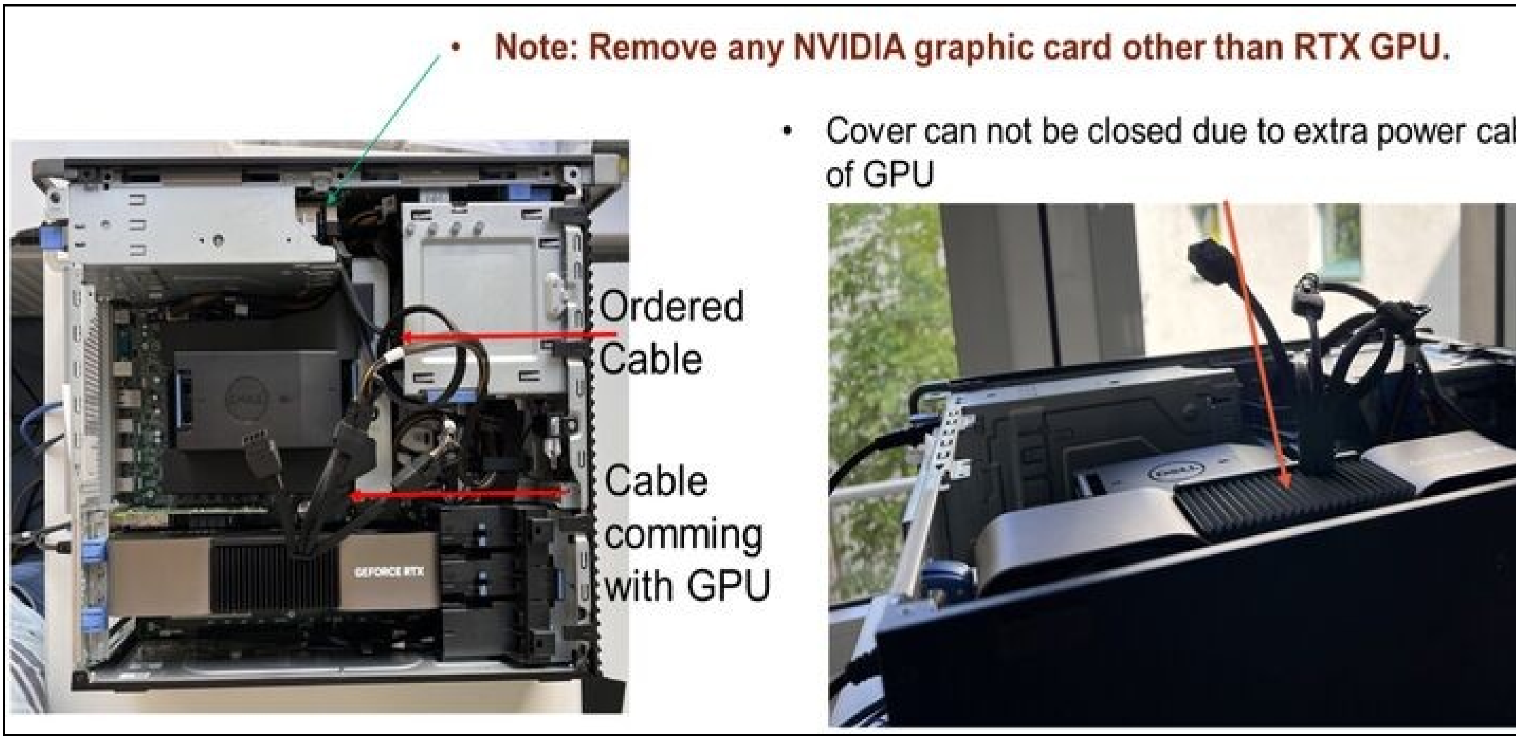
Listed below is a consolidated Bill of Materials (BoM) for all the components used in the implementation of this system. Note that the specific computers listed are not necessarily or specifically required, and that computers from other vendors may be used as long as they have a similar level of performance and capability.

- Core Network (CN) host system:
    - Dell 3680 Precision Tower, with Intel i9-10980XE CPU, with 18 physical cores, with 3.00 GHz base clock frequency.
    - Mellanox Nvidia ConnectX-5 MCX512A-ACAT network card, with two SFP28 ports.
- gNB host system:
    - Dell 5860 Precision Tower, with Intel Xeon W7-2495X CPU, with 24 physical cores, and with turbo frequency of 4.8 GHz.
    - Mellanox Nvidia ConnectX-5 MCX512A-ACAT network card, with two SFP28 ports.
    - Nvidia GeForce RTX 4090 GPU.
    - GPU power cable, 30cm, 12+4-pin male to 2x8-pin female sleeved 12v extension cord for GeForce RTX 4090.
    - QSFP28-to-SFP28 25 Gbps Ethernet breakout cable to connect with USRP X410.
        - Nvidia MCP2M00-A002E30N 100 GbE to 4x25GbE (QSFP28 to 4xSFP28) Direct Attach Copper (DAC) splitter cable (here).
        - NI 100 GbE to 4x25GbE (QSFP28 to 4xSFP28) splitter cable (NI Part Number 788214-01).
- USRP X410 radio (Quantity 2).
    - One connected to the gNB system.
    - One connected to the UE system.
- OctoClock-G CDA-2990 for synchronization between the gNB USRP X410 and the UE USRP X410.
- RF cables (1 meter) (Quantity 8).
- 20 dB RF attenuators (Quantity 2).
- Two-port RF splitter, 6 GHz, SMA connectors (Quantity 1).
- Ethernet switch, 5 ports (Quantity 1).
- Cat-6A RJ-45 Ethernet cables (Quantity 4).

On some systems, the chassis may not be able to be fully closed after installing the GPU. This is true for the Dell 5860, where after installing the Nvidia RTX 4090 GPU, the chassis can no longer be fully closed because of limited clearance between the power connectors for the GPU and the lid of the chassis. Reference the photo shown below.

Be sure to remove any other GPUs installed in the system, other than the primary Nvidia RTX 4090 GPU. This ensures optimal power delivery and avoids resource conflicts.

For compact systems, it is recommended to use custom low-profile or angled power cables with the GPU to reduce space usage. If frequent access is required, maintain the system in an open or semi-open configuration. Consider future integration with 2U or 4U rackmount servers that have dedicated GPU clearance and improved airflow management.



- Note: Remove any NVIDIA graphic card other than RTX GPU.
- Cover can not be closed due to extra power cal
of GPU

Left: Installed RTX 4090 GPU with ordered power cable. Right: Cover cannot be closed due to cable protrusion.

The desire is to run the CPU at the highest clock speed supported to ensure maximum performance for real-time and large-data processing. All power-saving features of the CPU should be turned off in the BIOS, and the CPU performance governors should be set to `performance` . Be sure that the system has adequate airflow and sufficient cooling.

Turn off power-saving features that make the CPU go to sleep or change the CPU speed automatically or dynamically. These features save energy but make the system less stable for real-time processing.

Disable Hyper-Threading (SMT) in the BIOS. This capability enables one CPU core to act like two cores, but it can cause timing problems when both threads share the same resources, and can add latencies when context-switching.

Disable C-states in the BIOS and in Linux, and keep the CPU in the C0 state only. The C0 state means that the CPU is always awake and ready to run tasks without delay.

The P-states control CPU frequency. P0 is the fastest. Higher numbers like P1, P2 mean lower speed. Select the highest P-state (P0).

Disable or blacklist the `intel powerclamp` driver. This Linux driver forces the CPU to rest for cooling, but that can reduce real-time performance.

Enable Intel SpeedStep, and set the P-state policy. If disabled, then CPU runs at its base clock frequency, and is limited to that speed. If enabled with the `performance` policy setting, then the CPU can use its higher turbo boost frequency, depending on the CPU temperature and the thermal policy limits.

The following settings are recommended.

- BIOS
  - Hyper-Threading: Disable.
  - C-States: Disable.
  - P-States: Enable.
- Linux
  - CPU governor: Set to `performance` for all cores (must be set for each individual core).
  - C-States: Block deep C-states using kernel parameters, and set C0 state.
  - Turbo speed policy: Allow turbo speed whenever the CPU is within its thermal limits.
  - `intel powerclamp` driver: Disable or blacklist this kernel module.

These settings can be made using the Linux commands listed below.

To set the performance governors, first install the `cpufrequtils` package.

```
sudo apt-get install cpufrequtils
```

Next, edit the following file (if it does not exist, then create it).

```
sudo nano /etc/default/cpufrequtils
```

Add the following line to the end of the file.

```
GOVERNOR = "performance"
```

Save the file, and exit the text editor.

Next, disable deep C-states by adding some options to the GRUB bootloader.

Edit the GRUB configuration file.

```
sudo nano /etc/default/grub
```

Modify the line with the `GRUB_CMDLINE_LINUX_DEFAULT` parameter to include various options, as show below.

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash intel_pstate=disable processor.max_cstate=1 intel_idle.max_cstate=0 idle=poll iommu=pt intel_iomm
```

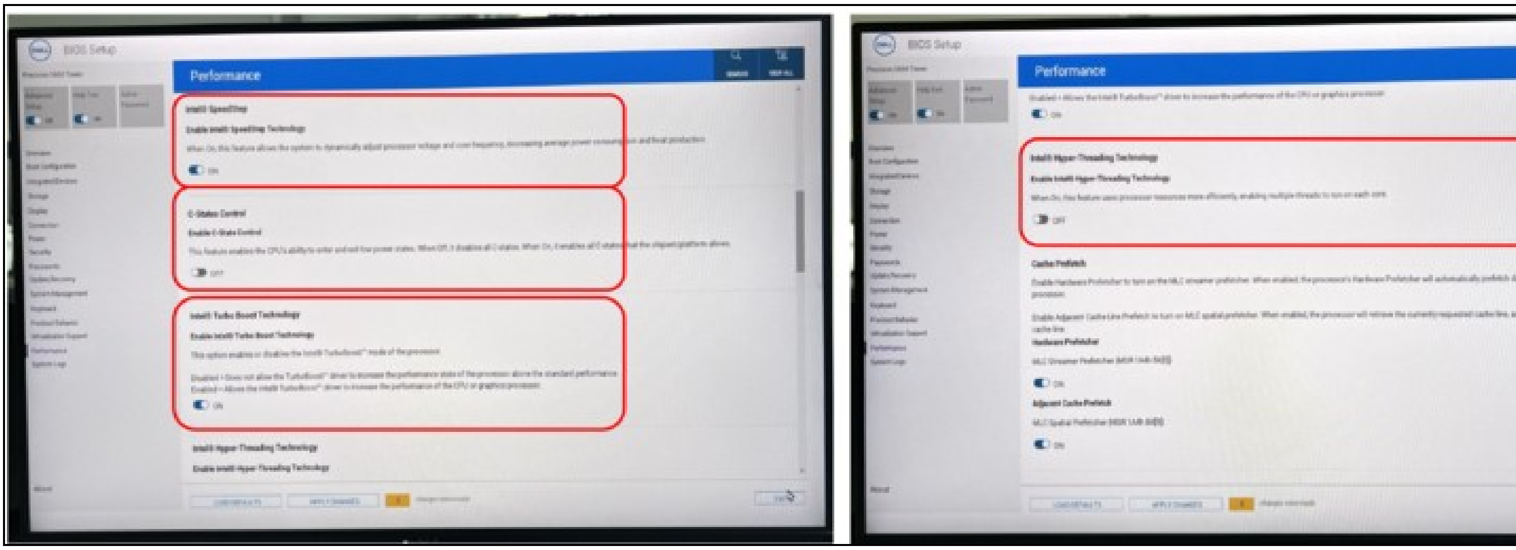
Update the GRUB bootloader with the newly edited configuration file.

```
sudo update-grub
```

Append `blacklist intel powerclamp` to the end of the `/etc/modprobe.d/blacklist.conf` file. This will blacklist this kernel module from being loaded. If the file does not exist, then create it, and add the line into it. Then save the file, and exit the text editor.

```
sudo nano /etc/modprobe.d/blacklist.conf
```

The figure listed below shows the BIOS settings from the Dell Precision 5860 Tower server that are recommended for achieving stable and consistent performance when running the real-time physical-layer processing. The BIOS settings for the Dell 3680 are similar. These optimizations ensure that the CPU runs at full performance and avoids unpredictable latency caused by power-saving modes.



Intel® SpeedStep
Enable Intel® SpeedStep Technology
ON

C-States Control
Enable C-State Control
OFF

Intel® Turbo Boost Technology
Enable Intel® Turbo Boost Technology
ON

Intel® Hyper-Threading Technology
Enable Intel® Hyper-Threading Technology
OFF

Cache Prefetch
Hardware Prefetcher
ON
Adjacent Cache Prefetch
ON

Recommended BIOS configuration on the Dell Precision 5860 for high-performance Linux operation
Each of the areas highlighted in the red boxes is explained below.

- Intel SpeedStep (Enable): This feature allows the CPU to automatically adjust voltage and frequency as needed. Keeping it enabled with a performance governor in Linux lets the processor use higher

frequencies when possible (Turbo mode).

- C-State Control (Disable): Disabling C-states prevents the CPU from entering low-power or idle modes (C1, C2, etc.). This keeps all cores active and ready to execute instructions without delay, which is

essential for low-latency wireless tasks.

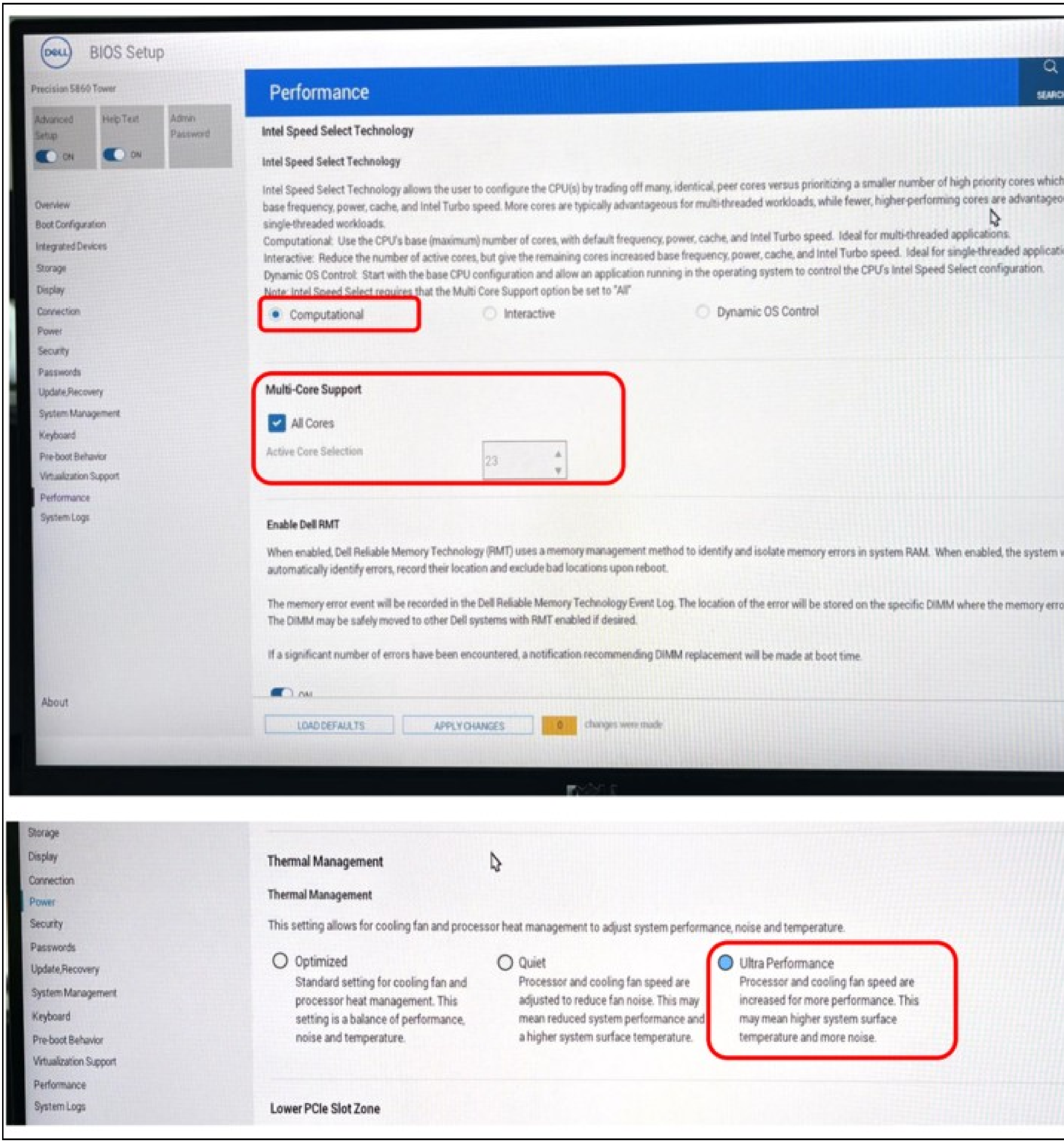
- Intel Turbo Boost Technology (Enable): This setting allows the CPU to increase its frequency above the base clock when the thermal and power budget permits. It improves processing throughput for demanding workloads such as baseband signal processing.

- Intel Hyper-Threading Technology (Disable): Turning off hyper-threading ensures that each physical core runs only one thread. This avoids unpredictable delays that occur when two threads share the

same core and switch contexts, leading to more deterministic timing in real-time applications.

Some servers may not include a direct P-state configuration menu in the BIOS. In such cases, P-states are automatically managed by the CPU and operating system to balance performance and power usage. When running real-time or 5G workloads, it is recommended to use the performance CPU governor in Linux and verify the configuration using tools like `cpupower` or `turbostat` .

The figure listed below shows additional BIOS options on the Dell Precision 5860 system. These settings focus on fully activating all CPU cores, optimizing frequency behavior, and maintaining consistent thermal performance.



Performance

Intel Speed Select Technology

Intel Speed Select Technology

Intel Speed Select Technology allows the user to configure the CPU(s) by trading off many, identical, peer cores versus prioritizing a smaller number of high priority cores which base frequency, power, cache, and Intel Turbo speed. More cores are typically advantageous for multi-threaded workloads, while fewer, higher-performing cores are advantageo single-threaded workloads.
Computational: Use the CPU's base (maximum) number of cores, with default frequency, power, cache, and Intel Turbo speed. Ideal for multi-threaded applications.
Interactive: Reduce the number of active cores, but give the remaining cores increased base frequency, power, cache, and Intel Turbo speed. Ideal for single-threaded applicati
Dynamic OS Control: Start with the base CPU configuration and allow an application running in the operating system to control the CPU's Intel Speed Select configuration.
Note: Intel Speed Select requires that the Multi Core Support option be set to "All"

● Computational ○ Interactive ○ Dynamic OS Control

Multi-Core Support

☑ All Cores

Active Core Selection 23

Enable Dell RMT

When enabled, Dell Reliable Memory Technology (RMT) uses a memory management method to identify and isolate memory errors in system RAM. When enabled, the system w automatically identify errors, record their location and exclude bad locations upon reboot.

The memory error event will be recorded in the Dell Reliable Memory Technology Event Log. The location of the error will be stored on the specific DIMM where the memory erro The DIMM may be safely moved to other Dell systems with RMT enabled if desired.

If a significant number of errors have been encountered, a notification recommending DIMM replacement will be made at boot time.

ON

Thermal Management

Thermal Management

This setting allows for cooling fan and processor heat management to adjust system performance, noise and temperature.

○ Optimized
Standard setting for cooling fan and processor heat management. This setting is a balance of performance, noise and temperature.

○ Quiet
Processor and cooling fan speed are adjusted to reduce fan noise. This may mean reduced system performance and a higher system surface temperature.

● Ultra Performance
Processor and cooling fan speed are increased for more performance. This may mean higher system surface temperature and more noise.

Lower PCIe Slot Zone

Additional BIOS optimization settings on Dell Precision 5860 Tower for high-performance operation
Each of the areas highlighted in the red boxes is explained below.

- Intel Speed Select Technology Computational Mode: This option prioritizes all CPU cores for maximum performance rather than balancing power and efficiency. Setting it to Computational ensures that each core operates at the highest available frequency during demanding workloads.
- Multi-Core Support All Cores: This option ensures that all physical CPU cores are active and available for processing. This is important for workloads like signal processing or AI inference where multiple threads are used in parallel.
- Power Thermal Management Ultra Performance: This mode increases the CPU cooling fan speed and power envelope to sustain higher frequencies for longer durations. It helps prevent performance throttling under heavy thermal load conditions. Note that in some BIOS, this option is found under Thermal Config Thermal Mode Performance, instead of under the Power menu.

After applying BIOS changes, always perform a system reboot to ensure all parameters take effect.

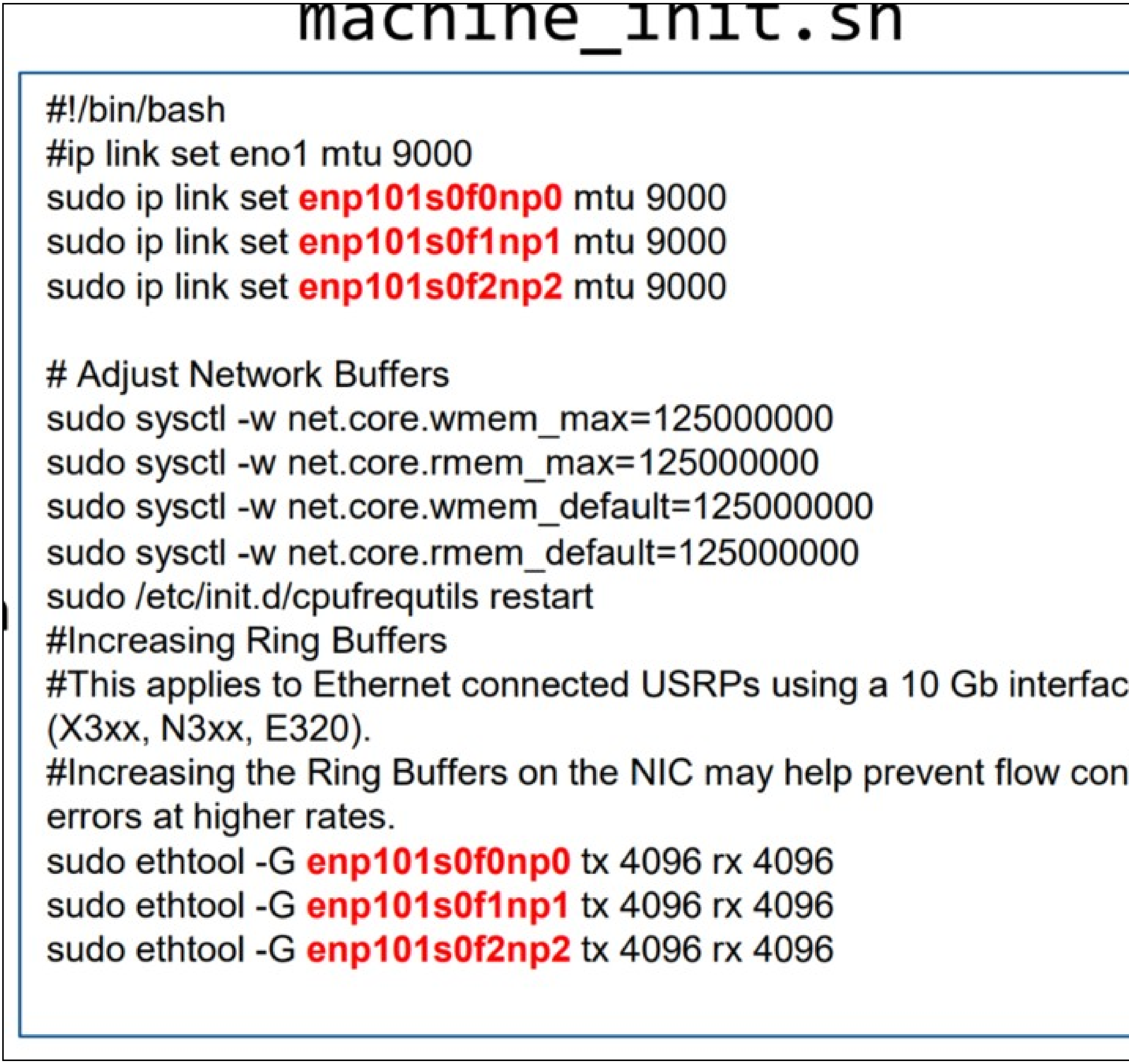
You can verify active cores and CPU frequency in Linux using the commands listed below.

```
   lscpu | grep "CPU"
   cat /sys/devices/system/cpu/online
   turbostat --Summary
```

Be sure to watch the CPU and system temperatures, especially in Performance mode, as this setting can raise internal temperatures, causing the fans to spin excessively, and creating additional noise.

To achieve the maximum data throughput between the servers and the USRP radios, the high-speed 10 Gbps and/or 100 Gbps Ethernet interfaces must be properly configured. We recommend that each user create a short shell script on every Linux server, in order to automate this setup.

This script should define the correct network interface names, adjust the MTU size (typically 9000 for jumbo frames), and configure the read/write socket buffer sizes and ring buffer parameters for optimal performance. Running this script at startup ensures that all network settings are restored after each system reboot, maintaining consistent high-speed communication between the servers and the USRP devices. Note the various network settings shown in the figure listed below.



machine_init.sh

```
#!/bin/bash
#ip link set eno1 mtu 9000
sudo ip link set enp101s0f0np0 mtu 9000
sudo ip link set enp101s0f1np1 mtu 9000
sudo ip link set enp101s0f2np2 mtu 9000

# Adjust Network Buffers
sudo sysctl -w net.core.wmem_max=125000000
sudo sysctl -w net.core.rmem_max=125000000
sudo sysctl -w net.core.wmem_default=125000000
sudo sysctl -w net.core.rmem_default=125000000
sudo /etc/init.d/cpufrequtils restart
#Increasing Ring Buffers
#This applies to Ethernet connected USRPs using a 10 Gb interfac
(X3xx, N3xx, E320).
#Increasing the Ring Buffers on the NIC may help prevent flow con
errors at higher rates.
sudo ethtool -G enp101s0f0np0 tx 4096 rx 4096
sudo ethtool -G enp101s0f1np1 tx 4096 rx 4096
sudo ethtool -G enp101s0f2np2 tx 4096 rx 4096
```

Configuration steps for optimizing high-speed Ethernet interfaces using the utility script
Identify all network interfaces by running:

```
   ifconfig
```

Go to the work directory:

```
cd $HOME/workarea
```

Open the configuration script:

```
sudo nano machine_init.sh
```

Update the interface names (e.g., `enp101s0f0np0` , `enp101s0f1np1` , etc.) to match your system, and then save and exit the file.

You can manually invoke the script with as shown below.

```
sudo $HOME/workarea/machine_init.sh
```

The script performs various optimization actions. It sets the MTU (Maximum Transmission Unit) value to `9000` for each Ethernet interface to enable jumbo frames to improve throughput, it adjusts read/write network socket buffers for higher data transfer efficiency, and it increases the ring buffer sizes on the network card to reduce packet drops at high data rates. These settings can be performed with the commands listed below. These optimizations ensure stable high-speed data transfer between USRP devices and servers, reduce latency and prevents packet loss during high-throughput streaming, and maintain deterministic network behavior for real-time applications.

```
sudo sysctl -w net.core.wmem_max=125000000
sudo sysctl -w net.core.rmem_max=125000000
sudo sysctl -w net.core.wmem_default=125000000
sudo sysctl -w net.core.rmem_default=125000000

sudo /etc/init.d/cpufrequtils restart

sudo ethtool -G enp101s0f0np0 tx 4096 rx 4096
sudo ethtool -G enp101s0f1np1 tx 4096 rx 4096
sudo ethtool -G enp101s0f2np2 tx 4096 rx 4096
```

This section describes how to verify that all Linux servers are correctly configured for maximum performance and consistent CPU frequency operation, using the commands discussed below.

Navigate to your working directory:

```
cd /home/user/workarea/
```

Run the initialization script:

```
sudo ./machine_init.sh
```

Check that all CPU cores are running at their maximum clock frequency:

```
cat /proc/cpuinfo | grep "MHz"
```

To continuously monitor the CPU clock frequency:

```
watch -n1 "grep '^[c]pu MHz' /proc/cpuinfo"
```

Verify that all CPU cores are configured to use the `performance` governor:

```
cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
```

The output should show `performance` for all CPU cores.

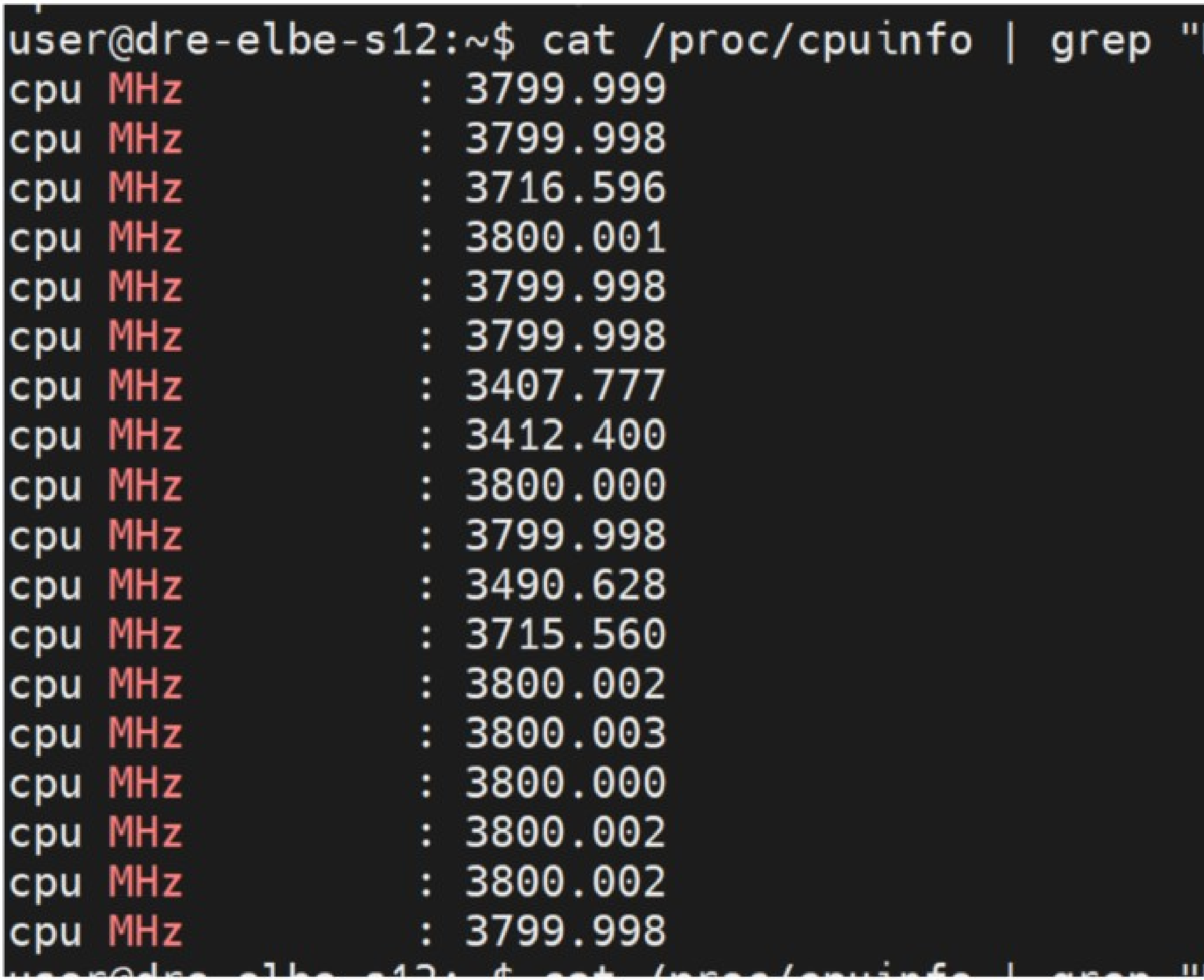
Check that Linux kernel boot parameters include power-state control options:

```
cat /proc/cmdline
```

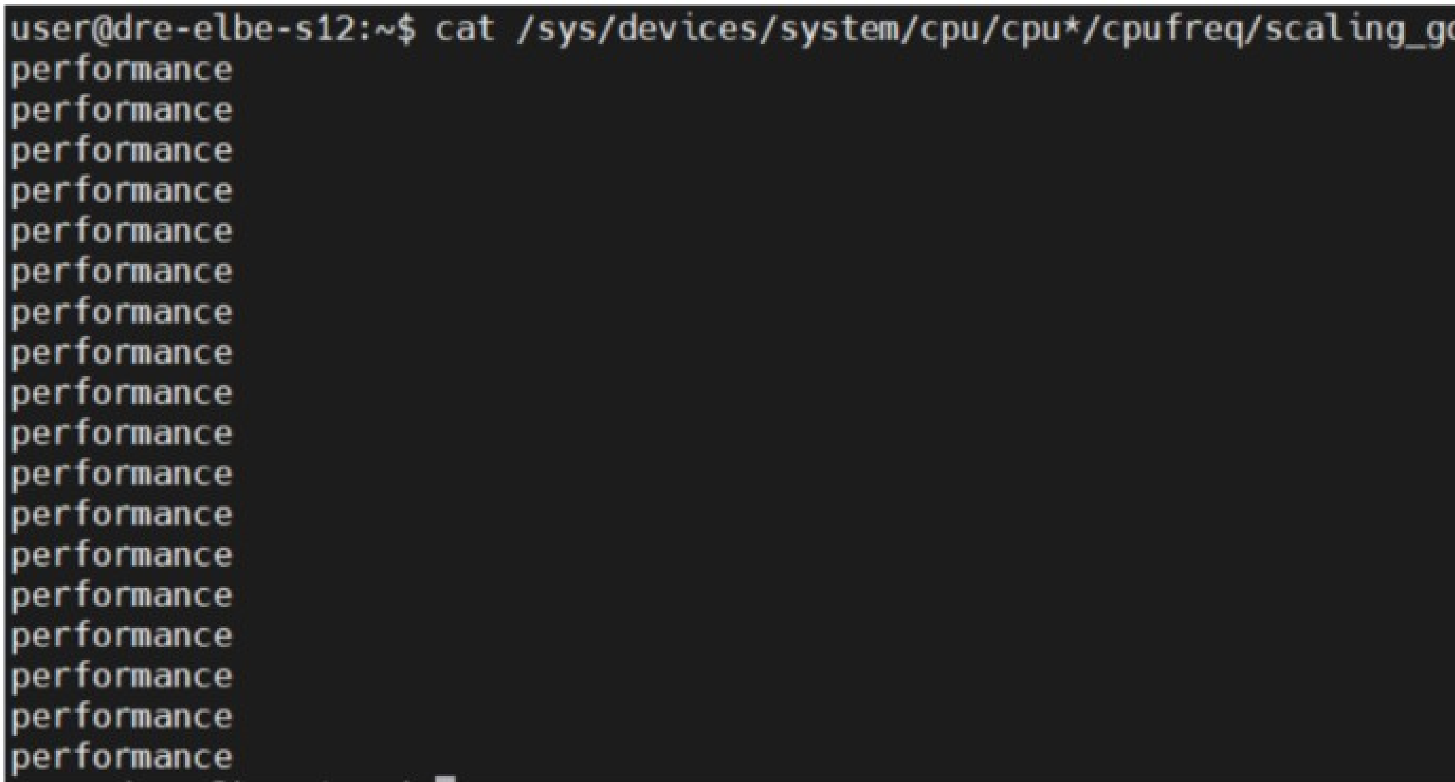
The expected output should contain:

```
intel_pstate=disable processor.max_cstate=1 intel_idle.max_cstate=0 idle=poll
```

You should see output that is similar to what is shown in the figure listed below.



```
user@dre-elbe-s12:~$ cat /proc/cpuinfo | grep "
cpu MHz             : 3799.999
cpu MHz             : 3799.998
cpu MHz             : 3716.596
cpu MHz             : 3800.001
cpu MHz             : 3799.998
cpu MHz             : 3799.998
cpu MHz             : 3407.777
cpu MHz             : 3412.400
cpu MHz             : 3800.000
cpu MHz             : 3799.998
cpu MHz             : 3490.628
cpu MHz             : 3715.560
cpu MHz             : 3800.002
cpu MHz             : 3800.003
cpu MHz             : 3800.000
cpu MHz             : 3800.002
cpu MHz             : 3800.002
cpu MHz             : 3799.998
```



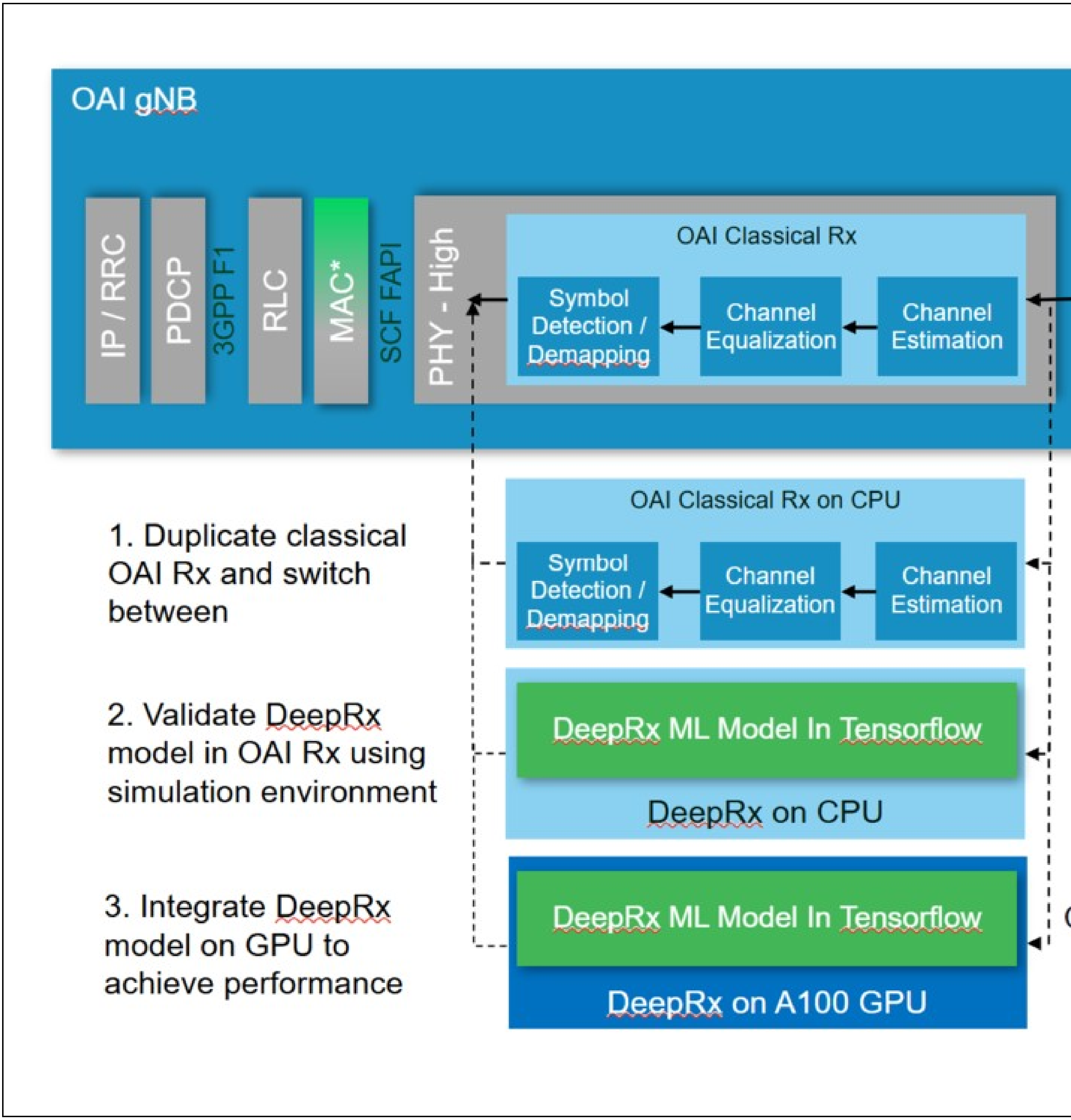
```
user@dre-elbe-s12:~$ cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_go
performance
performance
performance
performance
performance
performance
performance
performance
performance
performance
performance
performance
performance
performance
performance
performance
performance
performance
```

Verifying CPU clock frequencies, scaling governors, and power-state parameters on Linux servers
If a CPU frequency drops more than 1 GHz below the expected maximum, it indicates that power-saving or frequency-scaling features are still active.

Ensure that the BIOS options such as C-State control are disabled and that the Linux CPU governor is set to `performance` .

Reboot the system after any BIOS or GRUB configuration changes to ensure that they take effect.

The goal of this integration is to incorporate an AI/ML-based neural receiver (DeepRx) into the OAI physical layer, enabling real-time evaluation and comparison with the classical OAI receiver chain. The steps for this integration are summarized below.


OAI gNB
IP / RRC
PDCP
3GPP F1
RLC
MAC*
SCF FAPI
PHY - High
OAI Classical Rx
Symbol Detection / Demapping
Channel Equalization
Channel Estimation
OAI Classical Rx on CPU
Symbol Detection / Demapping
Channel Equalization
Channel Estimation
1. Duplicate classical OAI Rx and switch between
2. Validate DeepRx model in OAI Rx using simulation environment
DeepRx ML Model In Tensorflow
DeepRx on CPU
3. Integrate DeepRx model on GPU to achieve performance
DeepRx ML Model In Tensorflow
DeepRx on A100 GPU

Initial integration workflow for incorporating DeepRx neural receiver into the OAI test system

- Understand the OAI PHY Receiver Structure: Begin by analyzing the existing OAI receiver pipeline and identifying where the neural receiver (DeepRx) will replace or operate in parallel with the classical receiver chain.

- Duplicate and Replace Classical Rx Chain: Duplicate the traditional OAI Rx chain and replace one of the receiver instances (e.g., Rx2) with the \textbf{DeepRx Neural Receiver}. This allows direct comparison between the conventional and AI-based approaches.

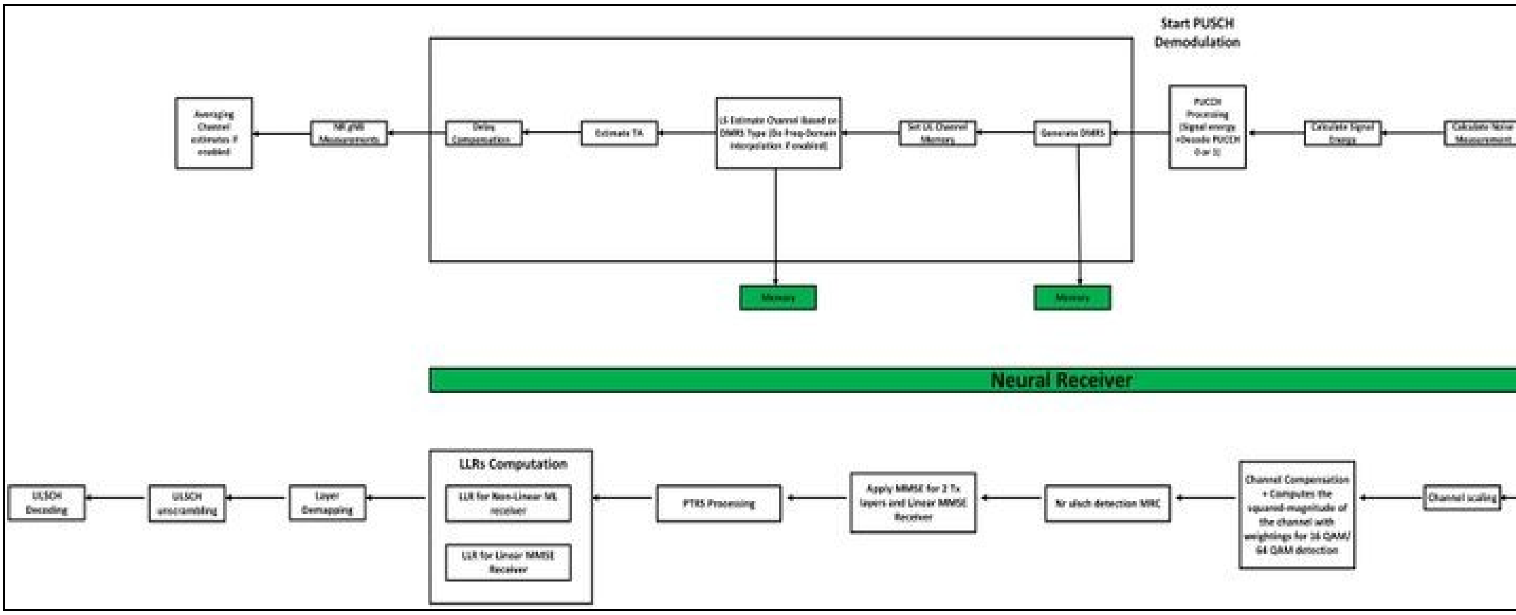
- Validate DeepRx in Simulation Environment: Validate the DeepRx model within OAI using the simulation mode:
    - ♦ Use the non-real-time OAI simulation mode on CPU for model validation.
    - ♦ Integrate the DeepRx model in the C++ OAI codebase using TensorFlow C-API.
    - ♦ Check the scaling of input and output power across the receiver chain.

- Integrate DeepRx on GPU for Real-Time Processing: After validation, move DeepRx to GPU for real-time performance evaluation:
    - ♦ Compare data transfer options between CPU and GPU (e.g., 22.6~µs latency).
    - ♦ Use proper memory configuration to maximize data transfer throughput.
    - ♦ Validate inference time of DeepRx for 2.5~MHz and 5~MHz OAI bandwidth modes.
    - ♦ Integrate DeepRx into OAI and test under different bandwidth configurations.
    - ♦ Optionally reduce DeepRx model complexity to improve inference speed.

- Compare AI Receiver and Classical Receiver: Run side-by-side experiments to compare the performance of the DeepRx neural receiver against the traditional OAI receiver chain in terms of accuracy, latency, and stability.

The integration workflow establishes a clear path for introducing AI/ML-based components into real-time wireless PHY processing. By starting with CPU-based simulation and gradually shifting to GPU-based real-time inference, the setup ensures stable validation, measurable performance gains, and reproducible benchmarking within the OAI framework.

The neural receiver (DeepRx) replaces the traditional MMSE-based detection and LLR computation modules inside the OAI physical-layer processing. This section shows the signal processing flow and how the AI model interfaces with existing OAI components.


Start PUSCH Demodulation
Calculate Noise Measurement
Calculate Signal Energy
PUCCH Processing (Signal energy +Decode PUCCH 0 or 1)
Generate DMRS
Set UL Channel Memory
LS Estimate Channel Based on DMRS Type (Do Freq-Domain interpolation if enabled)
Estimate TA
Delay Compensation
NR gNB Measurements
Averaging Channel estimates if enabled
Memory
Memory
Neural Receiver
Channel scaling
Channel Compensation + Computes the squared-magnitude of the channel with weightings for 16 QAM/ 64 QAM detection
Nr ulsch detection MRC
Apply MMSE for 2 Tx layers and Linear MMSE Receiver
PTRS Processing
LLRs Computation
LLR for Non-Linear ML receiver
LLR for Linear MMSE Receiver
Layer Demapping
ULSCH unscrambling
ULSCH Decoding

Integration of Neural Receiver (DeepRx) into the OAI receiver chain

The standard OAI receiver follows the 5G NR uplink physical layer pipeline, starting from demodulation reference signal (DMRS) generation and ending with decoding of the uplink shared channel (ULSCH). To enable AI-based inference, the DeepRx neural receiver is inserted between the channel estimation and the LLR computation modules.

The processing flow is as described below.

1. Uplink Channel Estimation: The gNB estimates the channel from DMRS symbols using a least-squares (LS) approach. The estimated channel is stored in memory for later reuse.

1. Neural Receiver Insertion: The DeepRx module is invoked after the conventional channel compensation stage. It receives preprocessed IQ samples and channel state information as inputs. The TensorFlow C-API interface is used to call the trained DeepRx model from within the C-based OAI code, allowing real-time inference directly on GPU.

1. Soft Bit (LLR) Generation: Based on the detected symbols, DeepRx outputs log-likelihood ratios (LLRs), replacing the classical linear MMSE LLR computation block. For fallback mode, the traditional MMSE-based LLR computation remains available for comparison.

1. Post-processing: After inference, the LLRs are passed to the existing OAI modules, such as Layer Demapping, ULSCH Unscrambling, and ULSCH Decoding, without any modification to higher layers.

There are several advantages to this approach.

- Seamless Integration: The DeepRx module interfaces with OAI through the TensorFlow C-API, avoiding major restructuring of the PHY code.

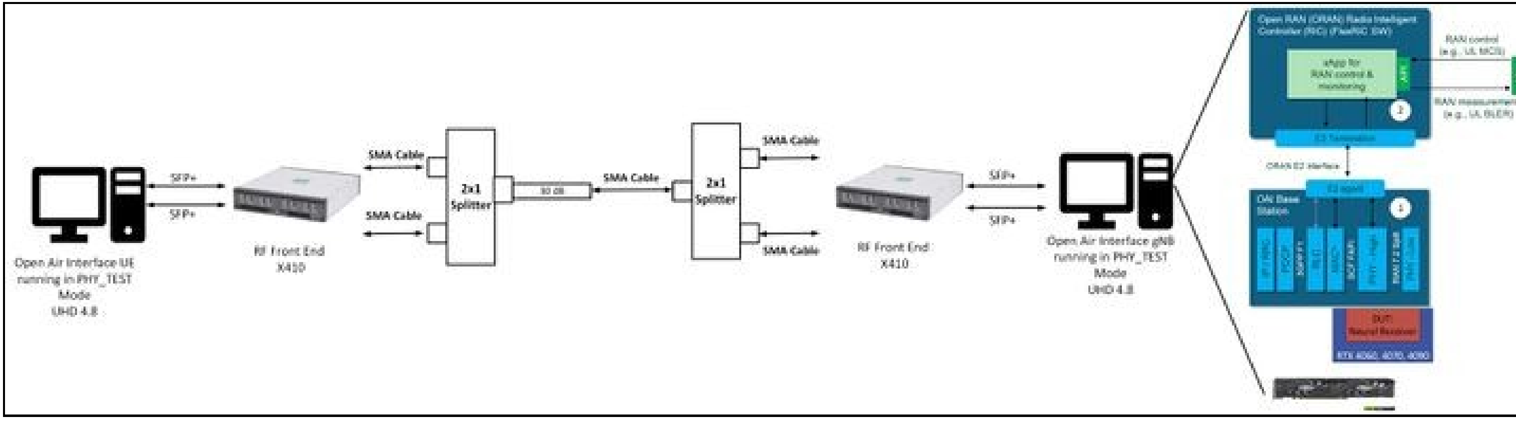
- GPU Acceleration: By running inference on the NVIDIA A100 and RTX GPUs, the system achieves real-time decoding performance for 5 MHz and 10 MHz channel bandwidths.

- Flexibility: Users can switch between neural and classical receivers through a run-time flag (`use_neural_rx=1`).

- Benchmarking: Enables direct comparison of BER/BLER and inference time between ML-based and conventional algorithms.

The DeepRx TensorFlow model is loaded once during initialization using the function </code> TF_LoadSessionFromSavedModel() </code>. Input and output tensors are pre-allocated to minimize latency. Data movement between CPU and GPU is handled through pinned memory buffers to reduce transfer overhead. The OAI logging system records inference latency, memory usage, and achieved BLER for post-analysis.

The figure listed below illustrates the complete end-to-end (E2E) setup used for integrating and evaluating an AI/ML-based neural receiver within an OpenAirInterface (OAI) 5G system. The setup includes both the OAI gNB (base station) and UE (user equipment) running in PHY TEST mode, interconnected via RF front ends (USRP X410) and managed through an Open RAN control and monitoring framework.


SFP+
SFP+
SMA Cable
SMA Cable
2x1 Splitter
30 dB
SMA Cable
2x1 Splitter
SMA Cable
SMA Cable
SFP+
SFP+
Open Air Interface UE running in PHY_TEST Mode UHD 4.8
RF Front End X410
RF Front End X410
Open Air Interface gNB running in PHY_TEST Mode UHD 4.8
Open RAN (ORAN) Radio Intelligent Controller (RIC) (FlexRIC SW)
xApp for RAN control & monitoring
API
RAN control (e.g., UL MCS)
RAN measurement (e.g., UL BLER)
2
E2 Termination
ORAN E2 Interface
E2 agent
OAI Base Station
1
DUT: Neural Receiver
RTX 4060, 4070, 4090

E2E testbed setup for integrating the Neural Receiver into the OAI system using USRP X410 and RIC control

- OAI UE and gNB: Both the OAI UE and gNB instances run in `PHY_TEST` using UHD 4.8 drivers. This configuration ensures a controlled environment focused on physical-layer data exchange, without the need for higher-layer signaling.
- RF Front Ends (USRP X410): Two USRP X410 radios are used as RF front-ends, one connected to the gNB, and the other connected to the UE. These devices handle digital-to-analog and analog-to-digital conversion, enabling over-the-air (or cabled) transmission of 5G NR signals.
- Interconnection and Attenuation: The two X410 units are connected through a series of 2×1 RF splitters and a 30 dB attenuator. This controlled attenuation emulates realistic signal propagation conditions while avoiding hardware saturation. All RF links are realized using SMA cables.
- Ethernet Connectivity: Each server connects to its respective X410 via SFP+ 10 Gbps Ethernet links, ensuring low-latency data transfer and real-time IQ data streaming.

The Software and Control Architecture is segmented into the following components.

- OAI Base Station Stack: The OAI gNB stack implements the standard 5G protocol layers (PHY, MAC, RLC, PDCP, and higher). Within the physical layer, the Neural Receiver (DeepRx) is integrated as a Device Under Test (DUT), replacing or running in parallel with the classical OAI receiver chain. The gNB server runs on Nvidia RTX GPUs (tested with RTX 4060, 4070, and 4090), leveraging CUDA for acceleration.
- Open RAN Control via RIC: A FlexRIC-based Radio Intelligent Controller (RIC) communicates with the gNB through the E2 interface. The RIC hosts xApps that enable dynamic control and monitoring of the RAN. For example, the xApp can read metrics such as uplink BLER or MCS and issue reconfiguration commands (e.g., enabling or disabling the neural receiver).
- Test Executor Entity: A Python-based Test Executor script interacts with the RIC via APIs to automate experiments. It toggles the Neural Receiver on and off, collects performance metrics (e.g., throughput, latency, BLER), and logs system responses for later analysis.

The workflow proceeds as follows:

1. The OAI gNB and UE exchange baseband IQ samples in `PHY_TEST` mode through the USRP X410s.

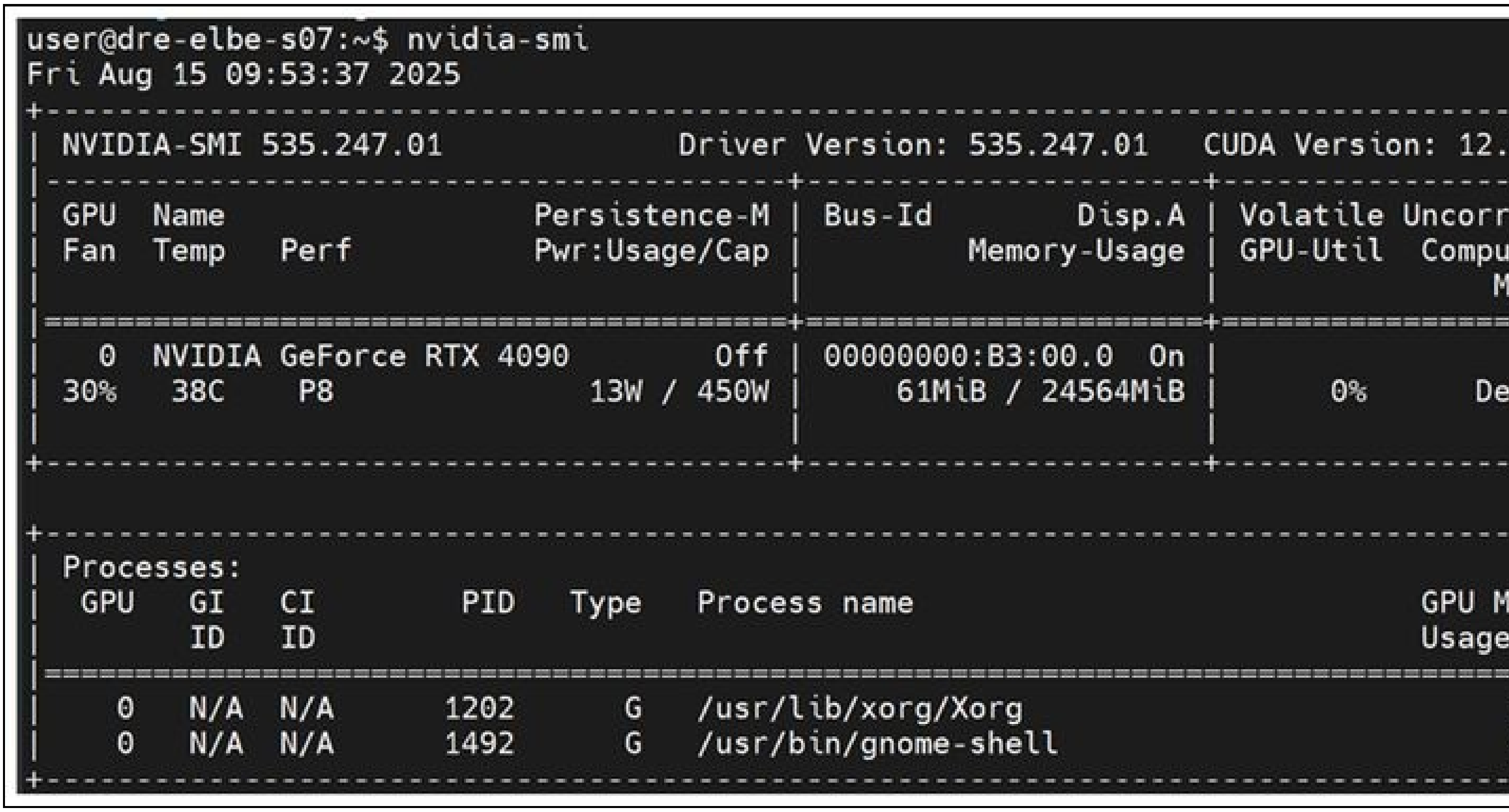
1. The neural receiver processes received symbols either on the CPU or the GPU, depending on the system configuration.

1. The RIC monitors the real-time performance and can trigger control actions through its xApp.

1. The test executor automates benchmarking, toggling the neural receiver, and recording measurements for analysis.

This modular testbed enables reproducible evaluation of AI-driven physical-layer components within a real 5G OAI environment, supporting both closed-loop and open-loop testing scenarios.

The figure listed below shows the recommended process for installing NVIDIA CUDA drivers and setting up TensorFlow with TensorRT and C-API support on Ubuntu 22.04. This procedure applies to systems equipped with Nvidia A100 or RTX 4090 GPUs.



```
user@dre-elbe-s07:~$ nvidia-smi
Fri Aug 15 09:53:37 2025
+-----------------------------------------------------------------------------
| NVIDIA-SMI 535.247.01             Driver Version: 535.247.01   CUDA Version: 12.
|-----------------------------------------+----------------------+-------------
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compu
|                                         |                      |               M
|=========================================+======================+=============
|   0  NVIDIA GeForce RTX 4090        Off | 00000000:B3:00.0  On |
| 30%   38C    P8             13W / 450W |     61MiB / 24564MiB |      0%      De
|                                         |                      |
+-----------------------------------------+----------------------+-------------


+-----------------------------------------------------------------------------
| Processes:
|  GPU   GI   CI        PID   Type   Process name                            GPU M
|        ID   ID                                                             Usage
|=============================================================================
|    0   N/A  N/A      1202      G   /usr/lib/xorg/Xorg
|    0   N/A  N/A      1492      G   /usr/bin/gnome-shell
+-----------------------------------------------------------------------------
```

Nvidia driver and CUDA installation procedure for Ubuntu 22.04
The installation steps for Nvidia device driver and CUDA are as follows.

First, update and Install GCC Tools.

```
sudo apt-get update
sudo apt install build-essential
```

Next, update the kernel headers.

```
sudo apt-key del 7fa2af80
sudo apt-get install linux-headers-$(uname -r)
```

Install the Nvidia driver with CUDA support by following the official NVIDIA installation guide available here.

Use the Ubuntu automatic detection utility.

```
sudo ubuntu-drivers list
```

Install the recommended version (535 is tested and verified). Note that version 535 automatically installs CUDA 12.2, and that newer versions such as 575 may cause unstable inference behavior.

```
sudo ubuntu-drivers install nvidia:535
sudo reboot
```

Verify the driver installation.

```
nvidia-smi
```

The output should list your GPU, such as A100 or RTX 4090, with correct driver and CUDA versions.

Next, install the CUDA Toolkit.

```
sudo apt install nvidia-cuda-toolkit
sudo reboot
```

Once the installation is complete, confirm that `nvidia-smi <code> correctly displays the GPU utilization and shows CUDA version 12.2. Check compatibility with TensorFlow 2.14 and TensorRT C-API. Validate that GPU acceleration works in user-space applications (e.g., TensorFlow or PyTorch).`

For additional detailed information, refer to the external resource "A Beginner?s Guide to NVIDIA Container Toolkit on Docker" by Umberto Junior Mele, available here.

This section describes the steps for installing TensorFlow 2.14 with CUDA, TensorRT, and the TensorFlow C-API on Ubuntu 22.04. The instructions assume that you already have CUDA 12.2 and the Nvidia drivers (version 535) installed on your system. The steps are listed below.

Create a working folder.

```
mkdir $HOME/workarea
cd $HOME/workarea
mkdir tf214
cd tf214
```

Check the Python version compatibility. TensorFlow 2.14 supports Python versions between 3.9 and 3.11.

```
python3.10 --version
```

Install and configure the Python virtual environment.

```
sudo apt install python3.10-venv
python3.10 -m venv .venv
source .venv/bin/activate
```

Upgrade <code> pip , and install relevant dependencies.

```
python3.10 -m pip install --upgrade pip
pip install numpy==1.26.0
pip install tensorflow[and-cuda]==2.14
```

Once the installation completes, verify that TensorFlow detects the GPU correctly.

```
python3.10
>>> import tensorflow as tf
>>> tf.config.list_physical_devices('GPU')
```

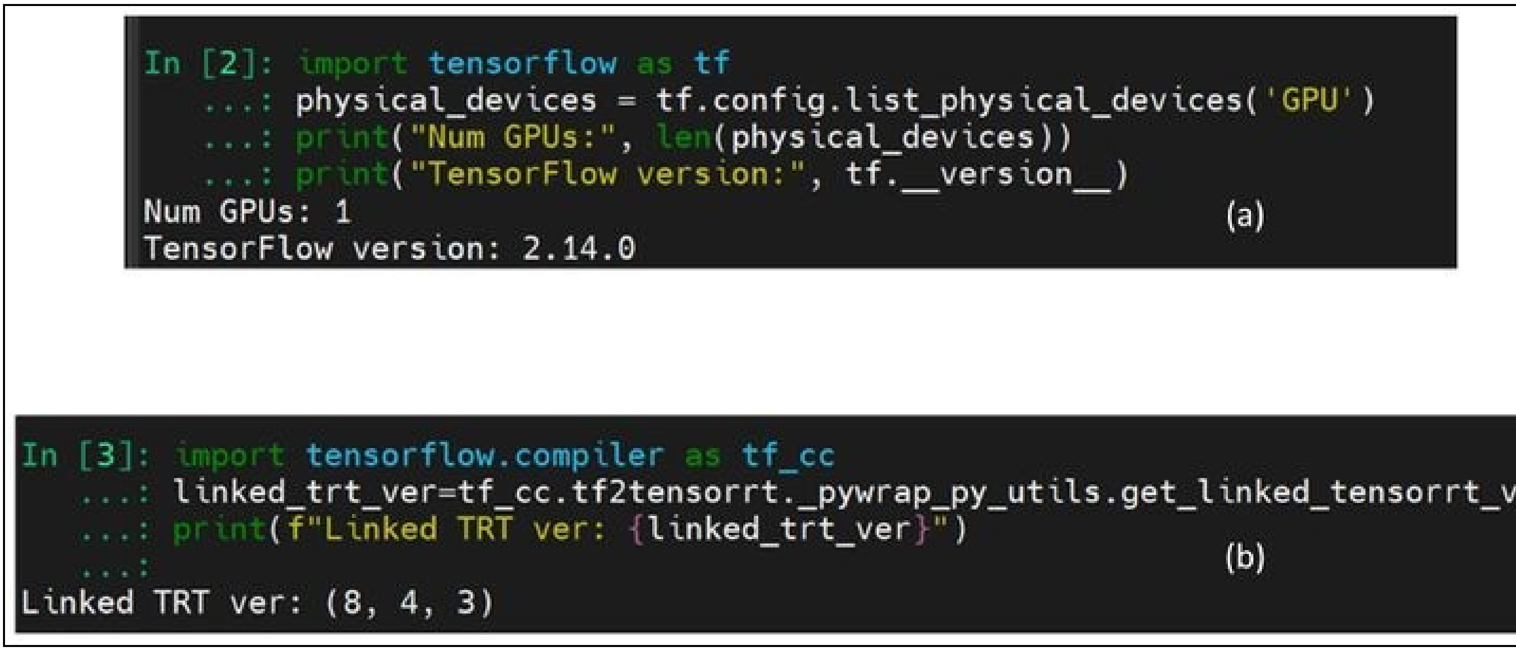
If a valid GPU such as the Nvidia RTX 4090 appears in the output, then the installation is successful.

The virtual environment (`tf214`) can be reused for future updates or experiments.

Ensure that CUDA~12.2 and cuDNN~8.9 are correctly linked before starting TensorFlow training or inference.

TensorRT support is included in TensorFlow 2.14 for accelerated inference.

After completing the TensorFlow 2.14 installation, it is important to verify that both the GPU and TensorRT are properly detected and linked within the TensorFlow environment, as shown in the figure listed below.



```
In [2]: import tensorflow as tf
   ...: physical_devices = tf.config.list_physical_devices('GPU')
   ...: print("Num GPUs:", len(physical_devices))
   ...: print("TensorFlow version:", tf.__version__)
Num GPUs: 1
TensorFlow version: 2.14.0
```

```
In [3]: import tensorflow.compiler as tf_cc
   ...: linked_trt_ver=tf_cc.tf2tensorrt._pywrap_py_utils.get_linked_tensorrt_v
   ...: print(f"Linked TRT ver: {linked_trt_ver}")
   ...:
Linked TRT ver: (8, 4, 3)
```

Verifying TensorFlow GPU and TensorRT linkage in the virtual environment}

First, activate the Python virtual environment. If the virtual environment `tf214` is not already active, then activate it first.

```
cd ~/workarea/tf214
source .venv/bin/activate
```

Next, install IPython. This is optional but recommended.

```
pip install ipython
```

Then, verify the TensorFlow GPU access. Open an IPython session and run these commands.

```
import tensorflow as tf
physical_devices = tf.config.list_physical_devices('GPU')
print("Num GPUs:", len(physical_devices))
print("TensorFlow version:", tf.__version__)
```

If `Num GPUs` is 1 or more, then TensorFlow can successfully detect the GPU.

A sample output is shown below.

```
Num GPUs: 1
TensorFlow version: 2.14.0
```

Next, verify the TensorRT version linked with TensorFlow. To confirm that TensorRT is linked correctly within TensorFlow, run the commands listed below.

```
import tensorflow.compiler as tf_cc
linked_trt_ver = tf_cc.tf2tensorrt._pywrap_py_utils.get_linked_tensorrt_version()
print(f"Linked TRT ver: {linked_trt_ver}")
```

If you see output as shown below, then TensorFlow is correctly configured with TensorRT libraries.

```
Linked TRT ver: (8, 4, 3)
```

You can exit the IPython session after successful verification using the `exit()` command.

If no GPU is detected, then recheck your CUDA driver and environment variables.

TensorRT linkage ensures optimized GPU inference during model deployment.

Next, we will install and link the TensorFlow C-API required for integration with C/C++ frameworks such as OAI. The environment paths must be configured correctly to allow the runtime to locate CUDA, cuDNN, and TensorRT libraries. In order to do this, you will need to update the `.bashrc` file in your home directory.

Append the following lines to your `$HOME/.bashrc` file. Make sure that the path matches your virtual environment (for example, `/home/user/workarea/tf214/.venv`).

```
alias sudo='sudo PATH="$PATH" HOME="$HOME" LD_LIBRARY_PATH="$LD_LIBRARY_PATH"'

# CUDA and cuDNN for TensorFlow 2.14
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/workarea/tf214/.venv/lib/python3.10/site-packages/nvidia/cuda_runtime/lib/
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/workarea/tf214/.venv/lib/python3.10/site-packages/nvidia/cublas/lib/
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/workarea/tf214/.venv/lib/python3.10/site-packages/nvidia/cufft/lib/
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/workarea/tf214/.venv/lib/python3.10/site-packages/nvidia/curand/lib/
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/workarea/tf214/.venv/lib/python3.10/site-packages/nvidia/cusolver/lib/
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/workarea/tf214/.venv/lib/python3.10/site-packages/nvidia/cusparse/lib/
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/workarea/tf214/.venv/lib/python3.10/site-packages/nvidia/cudnn/lib/

# TensorRT dependencies
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/workarea/tf214/.venv/lib/python3.10/site-packages/tensorrt/

# For PTX error handling in TensorFlow
export PATH=$HOME/workarea/tf214/.venv/lib/python3.10/site-packages/nvidia/cuda_nvcc/bin/:$PATH
```

Save the file and apply the changes.

```
source $HOME/.bashrc
```

Install TensorFlow C-API by running the following commands in sequence to download and extract the TensorFlow C-API library.

```
FILENAME=libtensorflow-gpu-linux-x86_64-2.14.0.tar.gz
wget -q --no-check-certificate https://storage.googleapis.com/tensorflow/libtensorflow/${FILENAME}
sudo tar -C /usr/local -xzf ${FILENAME}
sudo ldconfig /usr/local/lib
```

After installation, open a new terminal window, and verify that the C-API library has been linked correctly.

```
ls /usr/local/lib | grep libtensorflow
```

If both `libtensorflow.so` and `libtensorflow_framework.so` appear in the output, then the C-API is correctly installed and is ready to be used with OAI or other C-based TensorFlow applications.

Ensure that the Python virtual environment is activated whenever you run TensorFlow-related programs.

If TensorFlow or OAI fails to find the shared libraries, double-check the `LD_LIBRARY_PATH` environment variable setting.

```
echo $LD_LIBRARY_PATH
```

These settings persist across reboots, as they are set in the `$HOME/.bashrc` file.

To confirm that the TensorFlow C-API has been successfully installed and linked, create and run a simple C test program.

Create a file named `hello_tf.c` using the source code listed below.

```
#include <stdio.h>
#include <tensorflow/c/c_api.h>

int main() {
    printf("Hello from TensorFlow C library version %s\n", TF_Version());
    return 0;
}
```

Then, compile the C file and link it against the TensorFlow library, and run it.

```
gcc hello_tf.c -ltensorflow -lm -o hello_tf.o
./hello_tf
```

If the installation is correct, then the program should print the following output.

```
Hello from TensorFlow C library version 2.14.0
```

If you receive a warning such as the one shown below, then it indicates that the CUDA library paths were not set correctly.

```
tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find CUDA drivers on your machine,
GPU will not be used. Hello from TensorFlow C library version 2.14
```

To fix this, ensure that the `LD_LIBRARY_PATH` environment variable for CUDA, cuDNN, and TensorRT are added in your `$HOME/.bashrc` file, and then run the command below.

```
source $HOME/.bashrc
```

At this point, all of the components TensorFlow 2.14, TensorRT, CUDA 12.2, and the C-API, should be properly configured.

You can now compile and link C/C++ applications (such as the OAI Neural Receiver modules) directly with the TensorFlow C-API for GPU-accelerated inference.

This section explains how to build and install the USRP Hardware Driver (UHD) from source code. UHD is the open-source driver for all USRP radios and is required on both the gNB and UE systems. It is not required on the CN system. We strongly recommend building UHD from source rather than installing from binary packages to ensure compatibility and access to the latest updates. At the time of this writing, we recommend using UHD version

4.8.

Before building UHD, install all the required dependencies using the following command (for Ubuntu 22.04):

```
sudo apt update && sudo apt install -y cmake g++ libboost-all-dev libusb-1.0-0-dev libuhd-dev python3 python3-mako python3-numpy python3-r
```

Then, clone the UHD repository on GitHub, and check out the `v4.8.0.0` tag:

```
git clone https://github.com/EttusResearch/uhd.git
cd uhd
git checkout v4.8.0.0
```

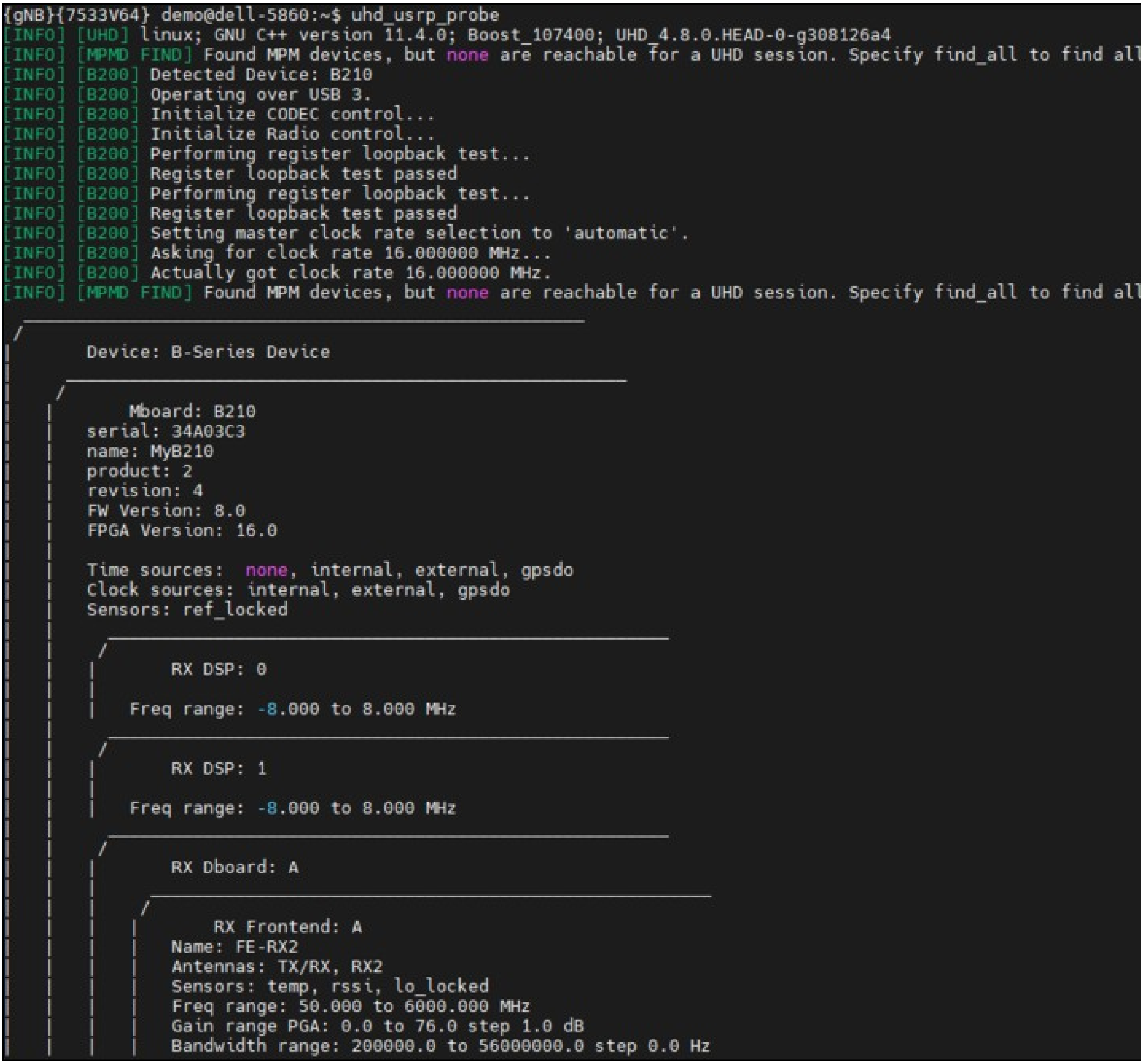
Then, build and install UHD.

```
cd host
mkdir build
cd build
cmake ../
make -j$(nproc)
sudo make install
sudo ldconfig
export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
sudo uhd_images_downloader
```

You can verify the installation by running:

```
uhd_usrp_probe
uhd_find_devices
```

For more details, and a full explanation on building and installing UHD from source code, reference the Application Note here.



```
{gNB}{7533V64} demo@dell-5860:~$ uhd_usrp_probe
[INFO] [UHD] linux; GNU C++ version 11.4.0; Boost_107400; UHD_4.8.0.HEAD-0-g308126a4
[INFO] [MPMD FIND] Found MPM devices, but none are reachable for a UHD session. Specify find_all to find all
[INFO] [B200] Detected Device: B210
[INFO] [B200] Operating over USB 3.
[INFO] [B200] Initialize CODEC control...
[INFO] [B200] Initialize Radio control...
[INFO] [B200] Performing register loopback test...
[INFO] [B200] Register loopback test passed
[INFO] [B200] Performing register loopback test...
[INFO] [B200] Register loopback test passed
[INFO] [B200] Setting master clock rate selection to 'automatic'.
[INFO] [B200] Asking for clock rate 16.000000 MHz...
[INFO] [B200] Actually got clock rate 16.000000 MHz.
[INFO] [MPMD FIND] Found MPM devices, but none are reachable for a UHD session. Specify find_all to find all
  _____________________________________________________
 /
|       Device: B-Series Device
|     _____________________________________________________
|    /
|   |       Mboard: B210
|   |   serial: 34A03C3
|   |   name: MyB210
|   |   product: 2
|   |   revision: 4
|   |   FW Version: 8.0
|   |   FPGA Version: 16.0
|   |
|   |   Time sources:  none, internal, external, gpsdo
|   |   Clock sources: internal, external, gpsdo
|   |   Sensors: ref_locked
|   |     _____________________________________________________
|   |    /
|   |   |       RX DSP: 0
|   |   |
|   |   |   Freq range: -8.000 to 8.000 MHz
|   |     _____________________________________________________
|   |    /
|   |   |       RX DSP: 1
|   |   |
|   |   |   Freq range: -8.000 to 8.000 MHz
|   |     _____________________________________________________
|   |    /
|   |   |       RX Dboard: A
|   |   |     _____________________________________________________
|   |   |    /
|   |   |   |       RX Frontend: A
|   |   |   |   Name: FE-RX2
|   |   |   |   Antennas: TX/RX, RX2
|   |   |   |   Sensors: temp, rssi, lo_locked
|   |   |   |   Freq range: 50.000 to 6000.000 MHz
|   |   |   |   Gain range PGA: 0.0 to 76.0 step 1.0 dB
|   |   |   |   Bandwidth range: 200000.0 to 56000000.0 step 0.0 Hz
```

Output from the UHD utility programs

To begin building the OpenAirInterface (OAI) gNB with Neural Receiver (DeepRx) integration, the first step is to download the appropriate OAI software branch. Two options are available: the NI custom branch with DeepRx integration; and the standard OAI codebase development branch maintained by Eurecom.

For the first option, building and using the NI codebase with the Neural Receiver DeepRx integration, follow the procedure listed below.

This codebase includes pre-integrated support for the TensorFlow-based Neural Receiver and related ML extensions.

```
cd workarea
mkdir oai && cd oai
git clone https://ni@dev.azure.com/ni/Users/_git/lti-6g-sw_oai-5g-ran
cd lti-6g-sw_oai-5g-ran
git checkout main
```

To begin building the OpenAirInterface (OAI) gNB codebase from Eurecom without Neural Receiver integrations and ML extensions, follow the procedure listed below.

```
git clone https://gitlab.eurecom.fr/oai/openairinterface5g.git
cd openairinterface5g/
git checkout develop
```

Before building the gNB, the user should edit the `build_oai` script to ensure that the installed UHD driver (from the system) is used rather than re-downloaded by the OAI build script.

```
cd cmake_targets
sudo nano build_oai
```

Locate the following lines in the script, and comment them out, as shown below.

```
if [ "$HW" == "OAI_USRP" ] ; then
  echo_info "installing packages for USRP support"
  #check_install_usrp_uhd_driver
  #if [ ! "$DISABLE_HARDWARE_DEPENDENCY" == "True" ]; then
  # install_usrp_uhd_driver $UHD_IMAGES_DIR
  #fi
fi
```

By commenting out these lines, it ensures that the OAI installation uses the existing, preinstalled UHD version (e.g., UHD 4.8) that was built and installed from source code. It prevents the OAI build script from downloading and installing an older or potentially-conflicting UHD version from a binary package. Doing this is strongly recommended for systems using Nvidia GPUs and TensorFlow acceleration to ensure driver compatibility.

After successfully cloning and setting up the OAI gNB source code, the next step is to build the Neural Receiver (DeepRx) library. This library contains the GPU-accelerated TensorFlow inference logic that will later be invoked by the OAI gNB soft-modem during runtime. To build the library, follow the following procedure.

Navigate to the TensorFlow C-API build directory inside the cloned OAI workspace.

```
cd workarea/oai/lti-6g-sw_oai-5g-ran/support_lti_6g_sw/tensorflow_c_api/neural_rx_lib/build
```

Run CMake to generate the Makefiles.

```
cmake ..
```

Then, compile the code.

```
make
```

Once the build is complete, a new executable object file named `call_test_inference.o` will be generated. This binary acts as a bridge between the OAI gNB receiver and the TensorFlow inference runtime on the GPU.

The build process links TensorFlow C-API, CUDA, and cuDNN libraries that were installed earlier. The resulting `libtest_inference.a` and `call_test_inference.o` files will later be invoked automatically by the gNB receiver when the Neural Receiver option is enabled.

The output confirms the build progress in stages, as shown below.

```
[25%] Building C object test_inference.c.o
[50%] Linking C static library libtest_inference.a
[75%] Building C object call_test_inference.c.o
[100%] Build complete
```

Ensure that no compilation errors appear, especially those related to missing CUDA paths or TensorFlow headers.

After the build finishes successfully, confirm that the generated binaries exist. Verify the you see the command output as shown below.

```
ls build
CMakeFiles  call_test_inference.o  libtest_inference.a  Makefile
```

These artifacts will be used later by the gNB process to perform real-time inference of the Neural Receiver model on the GPU.

Once the neural receiver library has been successfully compiled, the next step is to build the OAI gNB software, which integrates both the classical physical-layer receiver and the Neural Receiver extensions.

Navigate to the cloned OAI repository.

```
cd workarea/oai/lti-6g-sw_oai-5g-ran
source oaienv
cd cmake_targets
```

For the very first build of the gNB software, allow the OAI build script to install required dependencies, and execute the following command.

```
./build_oai -I -w USRP --ninja
```

The "-I" option installs all external packages and dependencies. The "-w" option adds RF board support (e.g., the USRP). The "--ninja" option enables the Ninja build system for faster compilation.

If you would like to add support for the T-Tracer, then install the required dependency. The T-Tracer utility is an extra OAI tool that provides data logging and debugging capabilities.

```
sudo apt-get install libxft-dev
```

If you would like to add support for NR Soft Scope, then install the required dependency. The NR Soft Scope utility is an extra OAI tool that provides physical-layer debugging capabilities.

```
sudo apt-get install libforms-dev
```

If the build process fails or if a clean rebuild is required, you can clear previous build artifacts with the command listed below.

```
sudo ./build_oai -c
```

The "-c" or "--clean" option erases all build files, forcing a fresh compilation from scratch.

For subsequent builds (after the initial one), omit the "-I" option.

```
# for building without Soft Scope
sudo ./build_oai --gNB -w USRP

# for building with Soft Scope
sudo ./build_oai --gNB -w USRP --build-lib nrscope --ninja
```

```
# for building with Simulation (clean + rebuild)
sudo ./build_oai -C -w SIMU -w USRP --gNB --build-lib nrscope --ninja
```

To build the complete gNB software for AI/ML-based experiments (including E2 Agent and Neural Receiver integration), use the command listed below.

```
sudo ./build_oai -w SIMU -w USRP --gNB --build-e2 --build-lib nrscope --ninja
```

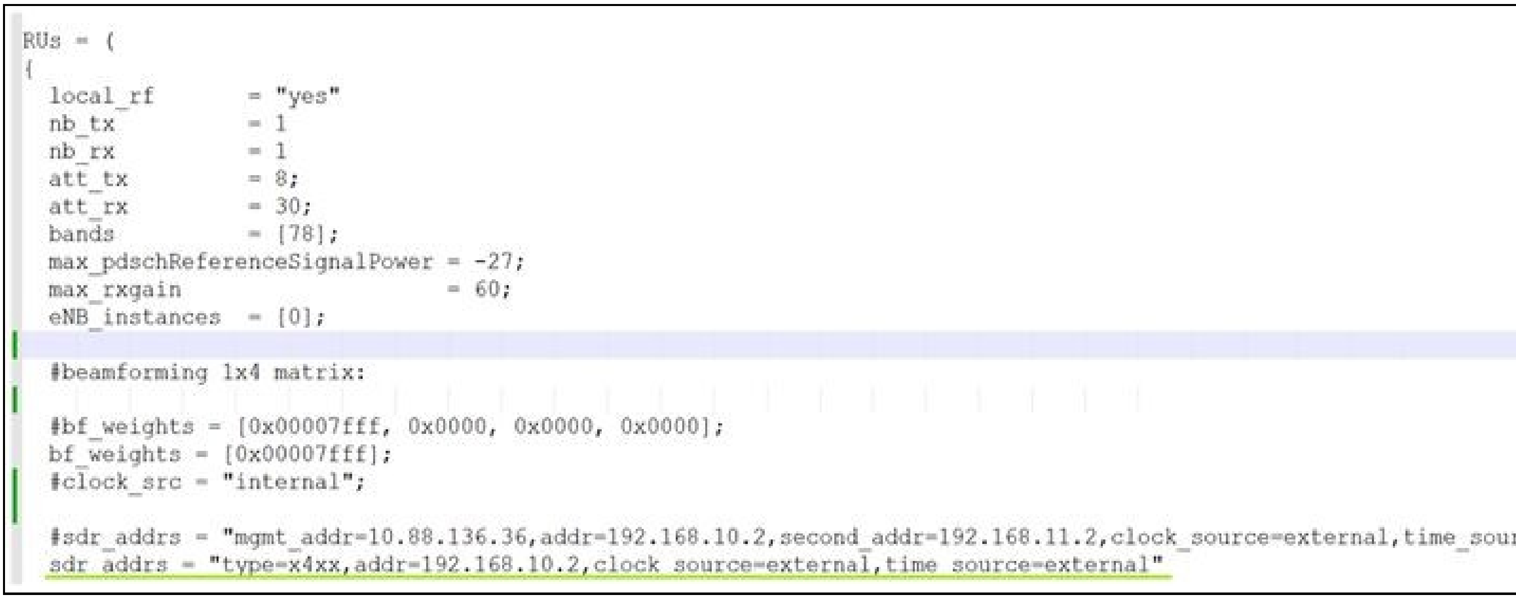
This command compiles all necessary components for:

- Running OAI gNB with the integrated Neural Receiver.
- Enabling the O-RAN E2 Agent for FlexRIC connectivity.
- Supporting both hardware (USRP) and simulated RF frontends.

After successful completion, the build output can be found in the `cmake_targets/ran_build/build` folder.

The executable `nr-softmodem` will be generated, which can now be used to run real-time experiments with the Neural Receiver.

After successfully building the OAI gNB software, the next step is to configure the USRP radio interface within the gNB configuration file. This ensures proper IP addressing, synchronization, and clock source alignment between the server and the USRP hardware.



```
RUs = (
{
  local_rf       = "yes"
  nb_tx          = 1
  nb_rx          = 1
  att_tx         = 8;
  att_rx         = 30;
  bands          = [78];
  max_pdschReferenceSignalPower = -27;
  max_rxgain                    = 60;
  eNB_instances  = [0];

  #beamforming 1x4 matrix:

  #bf_weights = [0x00007fff, 0x0000, 0x0000, 0x0000];
  bf_weights = [0x00007fff];
  #clock_src = "internal";

  #sdr_addrs = "mgmt_addr=10.88.136.36,addr=192.168.10.2,second_addr=192.168.11.2,clock_source=external,time_sour
  sdr_addrs = "type=x4xx,addr=192.168.10.2,clock_source=external,time_source=external"
```

USRP IP configuration within the OAI gNB configuration file
Open the gNB configuration file in the OAI directory.

```
nano targets/PROJECTS/GENERIC-NR-5GC/CONF/gnb.band78.sa.fr1.106PRB.1x1.usrpx410_3300.conf
```

Within the configuration file, locate the `RUs` section. Update the `sdr_addrs` field according to the connected USRP model and network topology.

\noindent Example RU configuration block:

```
RUs = (
{
    local_rf                      = "yes";
    nb_tx                         = 1;
    nb_rx                         = 1;
    att_tx                        = 8;
    att_rx                        = 30;
    bands                         = [78];
    max_pdschReferenceSignalPower = -27;
    max_rxgain                    = 60;
    eNB_instances                 = [0];
    bf_weights                    = [0x00007fff];
    sdr_addrs                     = "type=x4xx,addr=192.168.10.2,clock_source=external,time_source=external";
}
);
```

Next, open the gNB configuration file.

```
nano targets/PROJECTS/GENERIC-NR-5GC/CONF/gnb.band78.sa.fr1.106PRB.1x1.usrpx410_3300.conf
```

Change the IP address of the USRP in the `RU` section. Note that no change is required if using default IP addresses for the USRP, and it would then typically be ready to use.

```
sdr_addrs = "type=x4xx,addr=192.168.10.2,second_addr=192.168.11.2,clock_source=external,time_source=external"
```

Ensure that the `addr` field matches the IP assigned to the USRP device.

The `clock_source` } and `time_source` should both be set to `external` when using the OctoClock-g, which is recommended.

For laboratory setups without an OctoClock-G or without any GPS synchronization, these values can be set to `internal` .

The FlexRIC (Flexible RAN Intelligent Controller) is used in the OAI testbed for enabling near-RT RIC functionalities, such as RAN monitoring and control through the E2 interface. It must be built after compiling the OAI gNB with E2 agent support.

To build the OAI gNB with E2 Agent Support, follow the procedure listed below.

If not already done, build the gNB with E2 agent and RF simulation libraries.

```
sudo ./cmake_targets/build_oai -w USRP --gNB --build-e2 --build-lib nrscope --ninja
```

Next, prepare the FlexRIC Source code. The FlexRIC/E2 Agent source is included as a submodule inside the OAI repository.

```
cd lti-6g-sw_oai-5g-ran/openair2/E2AP/flexric
```

Next, follow the additional setup instructions in the `flexric/README.md` document.

Next, check and install SWIG.

First check the current SWIG version.

```
swig -version
```

If the version is below 4.1, then remove it.

```
sudo apt-get purge --auto-remove swig
```

Install all the required dependencies and build SWIG version 4.1, if needed.

```
sudo apt update
sudo apt install libpcre2-posix2 libpcre2-dev
cd ~/workarea
git clone https://github.com/swig/swig.git
cd swig
git checkout release-4.1
./autogen.sh
./configure --prefix=/usr/
make -j8
sudo make install
```

Next, check and configure the GCC version. The FlexRIC does not support GCC 11, so ensure that the system is using GCC 10 or 12.

```
gcc --version
```

If needed, install and configure a supported version of GCC.

```
sudo apt -y install gcc-10 g++-10 gcc-12 g++-12
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-10 10
sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-12 12
sudo update-alternatives --config gcc
```

Now, you can build the FlexRIC.

First, install the required dependencies.

```
sudo apt install libsctp-dev cmake-curses-gui
```

Then, build the FlexRIC itself.

```
cd lti-6g-sw_oai-5g-ran/openair2/E2AP/flexric
mkdir build && cd build
cmake ..
make -j8
sudo make install
```

You can use `make -j8` for faster parallel compilation on multi-core systems.

Ensure that SWIG version 4.1 and a supported GCC version (10 or 12) are configured before building.

The xApp gRPC Service provides a communication interface between the FlexRIC controller and external test or monitoring tools. It includes two main parts:

- Building the xApp Library (`xAppTechDemo`).
- Building the gRPC server interface for the test executor.

To build the xApp Library (`xAppTechDemo`), follow the commands listed below.

```
cd workarea/oai/lti-6g-sw_oai-5g-ran/openair2/E2AP/flexric/xAppTechDemo/
sudo mkdir build
cd build/
sudo cmake ..
sudo make
```

The library file `libxAppAll.so` should be generated under `flexric/xAppTechDemo/build/libxAppAll.so` .

Next, build the gRPC server interface. The gRPC server allows the xApp to exchange control messages and telemetry with external AI/ML test executors.

First, install gRPC and the necessary dependencies. Follow the official instructions here.

Then, run the following commands in your home directory.

```
export MY_INSTALL_DIR=$HOME/.local
sudo apt install -y build-essential autoconf libtool pkg-config
git clone --recurse-submodules -b v1.71.0 --depth 1 https://github.com/grpc/grpc
cd grpc
mkdir -p cmake/build && cd cmake/build
cmake -DgRPC_INSTALL=ON -DgRPC_BUILD_TESTS=OFF -DCMAKE_INSTALL_PREFIX=$MY_INSTALL_DIR ../..
make -j8
sudo make install
```

Next, build the gRPC interface for the xApp. Once gRPC is installed, build the interface within the OAI repository.

```
cd workarea/oai/lti-6g-sw_oai-5g-ran/openair2/E2AP/flexric/xapp_grpc_api/
sudo mkdir -p cmake/build
pushd cmake/build
sudo cmake -DCMAKE_PREFIX_PATH=$MY_INSTALL_DIR ../..
sudo make -j4
```

The compiled gRPC server binary should now be located in the `lti-6g-sw_oai-5g-ran/openair2/E2AP/flexric/xapp_grpc_api/cmake/build/grpc_ric_srv` folder.

If gRPC is already installed on your system, you can rebuild the xApp gRPC interface quickly using the commands listed below.

```
cd lti-6g-sw_oai-5g-ran/openair2/E2AP/flexric/xapp_grpc_api/cmake/build
cmake ../..
make
```

After building and configuring the OAI gNB, it can be executed in different physical layer test modes, either using the Neural Receiver, the Traditional Receiver, or the RF simulator.

If the gNB was built with FlexRIC, ensure that the FlexRIC service is running before starting the gNB.

To run the gNB using the Neural Receiver, follow the procedure listed below. Note that the Neural Receiver is enabled by default. The test executor can select between the Neural and Traditional receiver dynamically.

```
sudo ./cmake_targets/ran_build/build/nr-softmodem -O ./targets/PROJECTS/GENERIC-NR-5GC/CONF/gnb.band78.sa.fr1.106PRB.1x1.usrpx410_3300MHz.
```

--parallel-config PARALLEL_SINGLE_THREAD --phy-test -T 6 -d --MACRLcs.[0].ul_harq_round_max 1 --MACRLcs.[0].dl_harq_round_max 1 --disable_gnb_neural_rx 0

This configuration launches the gNB in Neural Receiver mode, utilizing GPU acceleration and the TensorFlow API for real-time inference.

Use this mode when validating AI-based PHY receiver integration (DeepRx or NeuralRx), or when GPU-based inference is available (e.g., RTX 4060, 4070, 4090, or A100).

To run the gNB using only the Traditional Receiver, follow the procedure listed below. This mode disables the Neural Receiver, allowing operation without GPU or TensorFlow initialization.

```
sudo ./cmake_targets/ran_build/build/nr-softmodem -O ./targets/PROJECTS/GENERIC-NR-5GC/CONF/gnb.band78.sa.fr1.106PRB.1x1.usrpx410_3300MHz.
```

The "--disable_gnb_neural_rx 1" option forces the traditional receiver to be used by default. The TensorFlow API is skipped, saving GPU resources. This mode is ideal for debugging or quick smoke tests where neural processing is not required.

The "--disable_gnb_neural_rx 0" option enables normal operation, and the Neural Receiver is active by default. If the flag is omitted, then the Neural Receiver is the default and can still be toggled by the test executor.

To run the gNB using the RF Simulator, follow the procedure listed below. In this mode, OAI runs a simulated radio connection between the gNB and UE (no real RF hardware). This is useful for initial validation and debugging before connecting physical USRPs.

```
sudo ./cmake_targets/ran_build/build/nr-softmodem -O ./targets/PROJECTS/GENERIC-NR-5GC/CONF/gnb.band78.sa.fr1.106PRB.1x1.usrpx410_3300MHz.
```

For more details about the RF Simulator, reference the OAI Simulation Mode documentation located here.

Ensure that the correct configuration file path matches the band and USRP model in use.

When testing with Neural Receiver, verify GPU initialization and TensorFlow C-API loading before runtime.

Detailed configuration steps for optimizing CPU, BIOS, and network interface performance were already covered in previous sections. These steps ensure that the testbed achieves maximum throughput and deterministic timing behavior for real-time PHY and AI workloads. In summary:

- BIOS settings should disable power-saving features and hyper-threading, while enabling Intel SpeedStep and Turbo Boost.
- Linux system tuning should include fixed CPU frequency scaling (performance governor) and restricted C-states (C0 only).
- Network interfaces connected to USRPs must be configured with MTU = 9000 for jumbo frame support and enlarged socket buffers using sysctl parameters to support high-rate IQ streaming.

All setup commands are provided in the accompanying initialization scripts machine_init.sh and ue_init.sh, which can be executed after each system reboot.

To prepare the UE side for the testbed, download the OAI UE software from either the NI repository or from the Eurecom OAI repository.

To obtain the OAI UE code from the NI repository, follow the procedure listed below.

Create a working directory.

```
cd workarea
mkdir oai && cd oai
```

Clone the NI repository.

```
git clone https://ni@dev.azure.com/ni/Users/_git/lti-6g-sw-oai-5g-ran-UE
cd lti-6g-sw-oai-5g-ran-UE
git checkout 6G_Demo_2025_w2
```

The UE version in the `main` branch may require multiple runs (2?3) for a successful first sync.

For stable and repeatable runs, you can alternatively use the following.

```
git checkout 6G_Demo_2025_w2_main
```

To obtain the OAI UE code from the public Eurecom repository, follow the procedure listed below.

```
git clone https://gitlab.eurecom.fr/oai/openairinterface5g.git
cd openairinterface5g
git checkout develop
```

After cloning the repository, navigate to the build directory.

```
cd cmake_targets
```

Edit the build script to use installed UHD drivers, and open and edit the OAI UE build script.

```
sudo nano build_oai
```

Comment out the following lines, and save the file.

```
if [ "$HW" == "OAI_USRP" ] ; then
  echo_info "installing packages for USRP support"
  #check_install_usrp_uhd_driver
  #if [ ! "$DISABLE_HARDWARE_DEPENDENCY" == "True" ]; then
  #  install_usrp_uhd_driver $UHD_IMAGES_DIR
  #fi
fi
```

This ensures that the build process uses the locally installed UHD drivers already configured and built on the machine from source code.

Once the UE source code is downloaded, the next step is to compile and build the OAI UE software stack. Follow the procedure listed below.

Navigate to the OAI directory.

```
cd workarea/oai/lti-6g-sw-oai-5g-ran-UE
```

Source the OAI environment:

```
source oaienv
```

Navigate to the build directory.

```
cd cmake_targets
```

For the very first build, install all dependencies and RF support using the "-I" option, which installs required dependencies. The "-w" option adds RF board (e.g., the USRP) support.

```
sudo ./build_oai -I -w USRP --ninja
```

For NR Soft Scope, install libforms.

```
sudo apt-get install libforms-dev
```

If rebuilding after an error or a clean start, do the following.

```
sudo ./build_oai -c
```

The "-c" or "--clean" option removes all build artifacts.

For successive builds, omit the "-I" flag and use one of the following configurations:

For the AI/ML Demo, build Without Soft Scope.

```
sudo ./build_oai --nrUE -w USRP
```

For building with Soft Scope:

```
sudo ./build_oai --nrUE -w USRP --build-lib nrscope --ninja
```

For building with RF Simulation and Soft Scope:

sudo ./build_oai -w SIMU -w USRP --nrUE --build-lib nrscope --ninja

For first-time clean builds, and for successive builds, for Simulation and SoftScope:

```
sudo ./build_oai -c -w SIMU -w USRP --nrUE --build-lib nrscope --ninja
```

The build command automatically compiles the nrUE target. You can include or omit "--gNB" depending on whether you plan to run the UE and gNB on the same or different machines. For RF simulation mode "SIMU", both gNB and UE can run on a single host without hardware.

If the build fails, always clean first:

```
sudo ./build_oai -c
```

The OAI Soft UE configurations must be verified and adjusted depending on whether the code is cloned from the NI repository or the public Eurecom repository.

You will need to adjust parameters such as "imsi", "key", and "opc" according to your test network setup.

The UE configuration file is `lti-6g-sw_oai-5g-ran/targets/PROJECTS/GENERIC-NR-5GC/CONF/ue.conf`.

Listed below is an example of the OAI UE configuration file.

```
uicc0 = {
  imsi = "208920100001101";
  key  = "0C0A34601D4F07677303652C0462535B";
  opc  = "63bfa50ee6523365ff14c1f45f88737d";
  dnn  = "oai";
  nssai_sst = 1;
  nssai_sd  = 1;
}
```

You will need to update the OAI UE configuration file, with the specific parameters for your system.

One option is to edit the configuration file directly.

```
cd lti-6g-sw_oai-5g-ran/targets/PROJECTS/GENERIC-NR-5GC/CONF/
sudo nano ue.conf
```

Then copy and paste the configuration snippet listed above, and modify the values as needed.

Another option is to copy your local UE configuration file (for example, customized IMSI or DNN) to the target UE machine in the folder `lti-6g-sw_oai-5g-ran/targets/PROJECTS/GENERIC-NR-5GC/CONF/` .

Ensure that the IMSI and authentication parameters in the OAI UE configuration file match those provisioned in the 5G Core Network (e.g., Free5GC, OAI 5G CN, or commercial core network).

For PHY Test Mode, all RRC configuration files must be provided to the OAI Soft UE.

Step 1: Run the gNB first as described in the previous section.

Step 2: Copy the following two RRC configuration files from the gNB server, in the folder `lti-6g-sw\oai-5g-ran` , to the UE machine.

The two files are:

- reconfig.raw
- rbconfig.raw

Copy those files from the gNB system to the UE system, to the folder `/home/user/workarea/rrc_files` .

If you are using a different directory, then adjust paths in the commands accordingly.

To invoke the system and run with Real RF, follow this procedure.

```
sudo ./cmake_targets/ran_build/build/nr-uesoftmodem --usrp-args "type=x4xx,addr=192.168.10.2,second_addr=192.168.11.2,clock_source=externa
```

On downlink, adjust the "rx_gain" according to the RF connection.

On uplink, adjust "tx_gain" to achieve the greatest transmit power without any saturation or signal degradation.

You can run in RF Simulator mode if no real RF channel is available. In this mode, the UE connects to the gNB over a simulated interface.

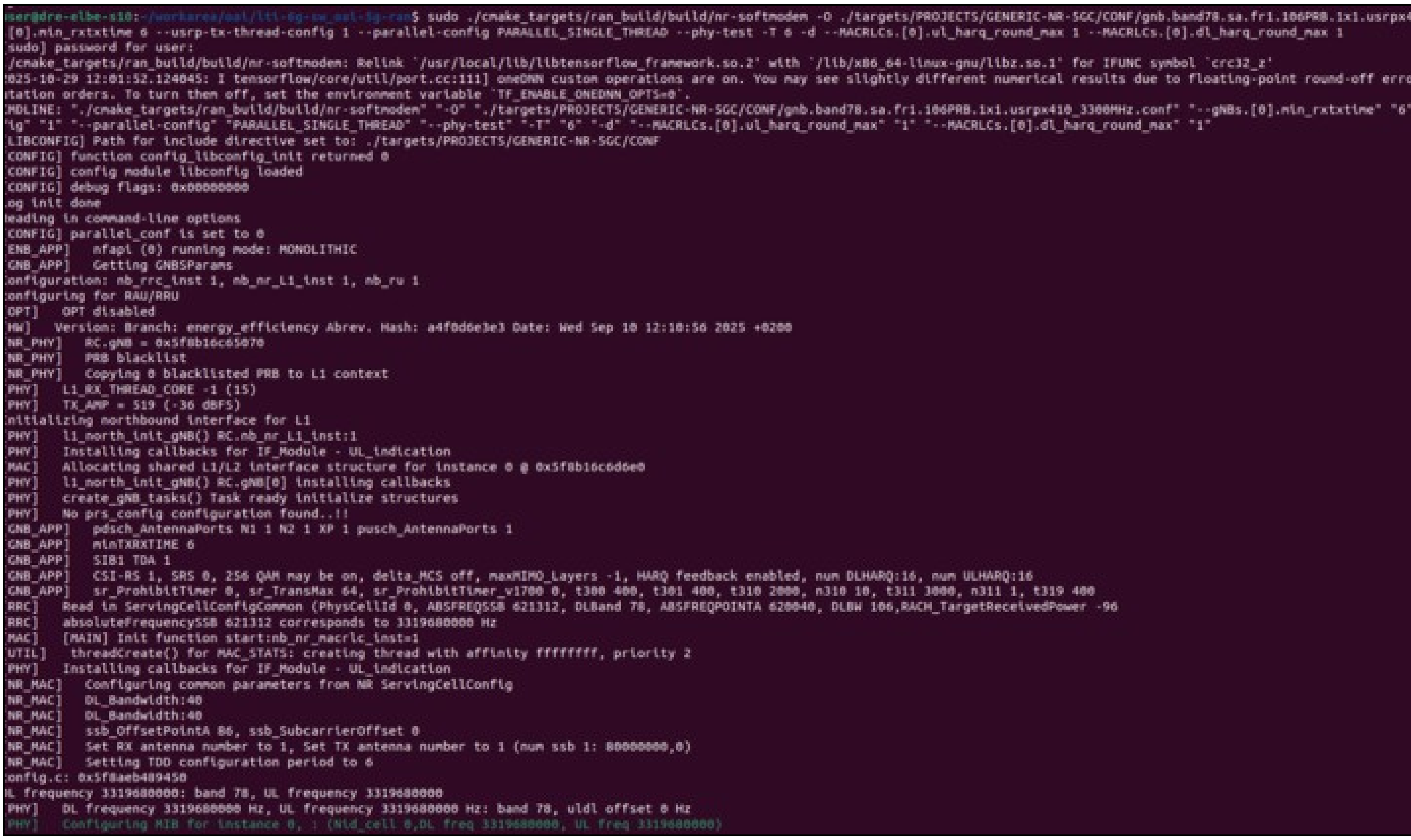
```
sudo ./cmake_targets/ran_build/build/nr-uesoftmodem --rfsim --rfsimulator.serveraddr 127.0.0.1 --phy-test --numerology 1 -O ./targets/PROJ
```

This section describes the procedure to execute the complete OAI-based 5G testbed including the gNB, FlexRIC, xApp, and the Test Executor for AI-driven neural receiver evaluation. Each component is initialized in sequence to ensure proper control, monitoring, and data exchange between the PHY, MAC, and RIC layers. The example setup corresponds to the 5G NR SA deployment in Band 78 using USRP X410 devices with UHD 4.8 drivers and TensorFlow-based inference support.

The first step in the execution pipeline is to start the OAI gNB in PHY test mode. This mode is useful for validating the physical layer, verifying the neural receiver integration, and checking RF synchronization with the connected UE or channel emulator. The gNB is launched using the nr-softmodem binary built under the LTI 6G branch with TensorFlow C-API support.

```
sudo ./cmake_targets/ran_build/build/nr-softmodem -O ./targets/PROJECTS/GENERIC-NR-5GC/CONF/gnb.band78.sa.fr1.106PRB.1x1.usrpx410_3300MHz.
```

Upon execution, the gNB initializes the baseband PHY, MAC, and RLC layers, loads the UHD driver, and detects the attached USRP X410. The system prints detailed logs confirming PHY configuration, PRB allocation, numerology, frequency parameters, and TensorFlow framework initialization. An example of a successful gNB bring-up is shown in the figure listed below.



```
ser@dre-elbe-s10:~/workarea/oai/lti-6g-sw_oai-5g-ran$ sudo ./cmake_targets/ran_build/build/nr-softmodem -O ./targets/PROJECTS/GENERIC-NR-5GC/CONF/gnb.band78.sa.fr1.106PRB.1x1.usrpx4
[0].min_rxtxtime 6 --usrp-tx-thread-config 1 --parallel-config PARALLEL_SINGLE_THREAD --phy-test -T 6 -d --MACRLCs.[0].ul_harq_round_max 1 --MACRLCs.[0].dl_harq_round_max 1
sudo] password for user:
/cmake_targets/ran_build/build/nr-softmodem: Relink `/usr/local/lib/libtensorflow_framework.so.2' with `/lib/x86_64-linux-gnu/libz.so.1' for IFUNC symbol `crc32_z'
025-10-29 12:01:52.124045: I tensorflow/core/util/port.cc:111] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off erro
tation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
MDLINE: "./cmake_targets/ran_build/build/nr-softmodem" "-O" "./targets/PROJECTS/GENERIC-NR-5GC/CONF/gnb.band78.sa.fr1.106PRB.1x1.usrpx410_3300MHz.conf" "--gNBs.[0].min_rxtxtime" "6"
ig" "1" "--parallel-config" "PARALLEL_SINGLE_THREAD" "--phy-test" "-T" "6" "-d" "--MACRLCs.[0].ul_harq_round_max" "1" "--MACRLCs.[0].dl_harq_round_max" "1"
LIBCONFIG] Path for include directive set to: ./targets/PROJECTS/GENERIC-NR-5GC/CONF
CONFIG] function config_libconfig_init returned 0
CONFIG] config module libconfig loaded
CONFIG] debug flags: 0x00000000
og init done
eading in command-line options
CONFIG] parallel_conf is set to 0
ENB_APP]   nfapi (0) running mode: MONOLITHIC
GNB_APP]   Getting GNBSParams
onfiguration: nb_rrc_inst 1, nb_nr_L1_inst 1, nb_ru 1
onfiguring for RAU/RRU
OPT]   OPT disabled
HW]   Version: Branch: energy_efficiency Abrev. Hash: a4f0d6e3e3 Date: Wed Sep 10 12:10:56 2025 +0200
NR_PHY]   RC.gNB = 0x5f8b16c65070
NR_PHY]   PRB blacklist
NR_PHY]   Copying 0 blacklisted PRB to L1 context
PHY]   L1_RX_THREAD_CORE -1 (15)
PHY]   TX_AMP = 519 (-36 dBFS)
nitializing northbound interface for L1
PHY]   l1_north_init_gNB() RC.nb_nr_L1_inst:1
PHY]   Installing callbacks for IF_Module - UL_indication
MAC]   Allocating shared L1/L2 interface structure for instance 0 @ 0x5f8b16c6d0e0
PHY]   l1_north_init_gNB() RC.gNB[0] installing callbacks
PHY]   create_gNB_tasks() Task ready initialize structures
PHY]   No prs_config configuration found..!!
GNB_APP]   pdsch_AntennaPorts N1 1 N2 1 XP 1 pusch_AntennaPorts 1
GNB_APP]   minTXRXTIME 6
GNB_APP]   SIB1 TDA 1
GNB_APP]   CSI-RS 1, SRS 0, 256 QAM may be on, delta_MCS off, maxMIMO_Layers -1, HARQ feedback enabled, num DLHARQ:16, num ULHARQ:16
GNB_APP]   sr_ProhibitTimer 0, sr_TransMax 64, sr_ProhibitTimer_v1700 0, t300 400, t301 400, t310 2000, n310 10, t311 3000, n311 1, t319 400
RRC]   Read in ServingCellConfigCommon (PhysCellId 0, ABSFREQSSB 621312, DLBand 78, ABSFREQPOINTA 620040, DLBW 106,RACH_TargetReceivedPower -96
RRC]   absoluteFrequencySSB 621312 corresponds to 3319680000 Hz
MAC]   [MAIN] Init function start:nb_nr_macrlc_inst=1
UTIL]   threadCreate() for MAC_STATS: creating thread with affinity ffffffff, priority 2
PHY]   Installing callbacks for IF_Module - UL_indication
NR_MAC]   Configuring common parameters from NR ServingCellConfig
NR_MAC]   DL_Bandwidth:40
NR_MAC]   DL_Bandwidth:40
NR_MAC]   ssb_OffsetPointA 86, ssb_SubcarrierOffset 0
NR_MAC]   Set RX antenna number to 1, Set TX antenna number to 1 (num ssb 1: 80000000,0)
NR_MAC]   Setting TDD configuration period to 6
onfig.c: 0x5f8aeb489450
L frequency 3319680000: band 78, UL frequency 3319680000
PHY]   DL frequency 3319680000 Hz, UL frequency 3319680000 Hz: band 78, uldl offset 0 Hz
PHY]   Configuring MIB for instance 0, : (Nid_cell 0,DL freq 3319680000, UL freq 3319680000)
```

OAI gNB bring-up in PHY test mode
The terminal output confirms successful initialization of the OAI gNB operating in Band 78 (3.31968 GHz, 30 KHz SCS) using the USRP X410. The TensorFlow C-API library (libtensorflow_framework.so.2) is dynamically linked, enabling the neural receiver module. The gNB configures PHY parameters such as transmission amplitude (TX_AMP = 519), numerology, and HARQ contexts, and affinitizes PHY worker threads to CPU cores for deterministic scheduling.

The gNB runs in stand-alone mode with the Neural Receiver enabled by default. In this mode, data samples from the PHY layer are passed to the TensorFlow runtime for real-time inference. Subsequent components such as FlexRIC and xApp build upon this active gNB instance to enable real-time control and monitoring.

Once the gNB is operational and synchronized, the following components can be launched in order:

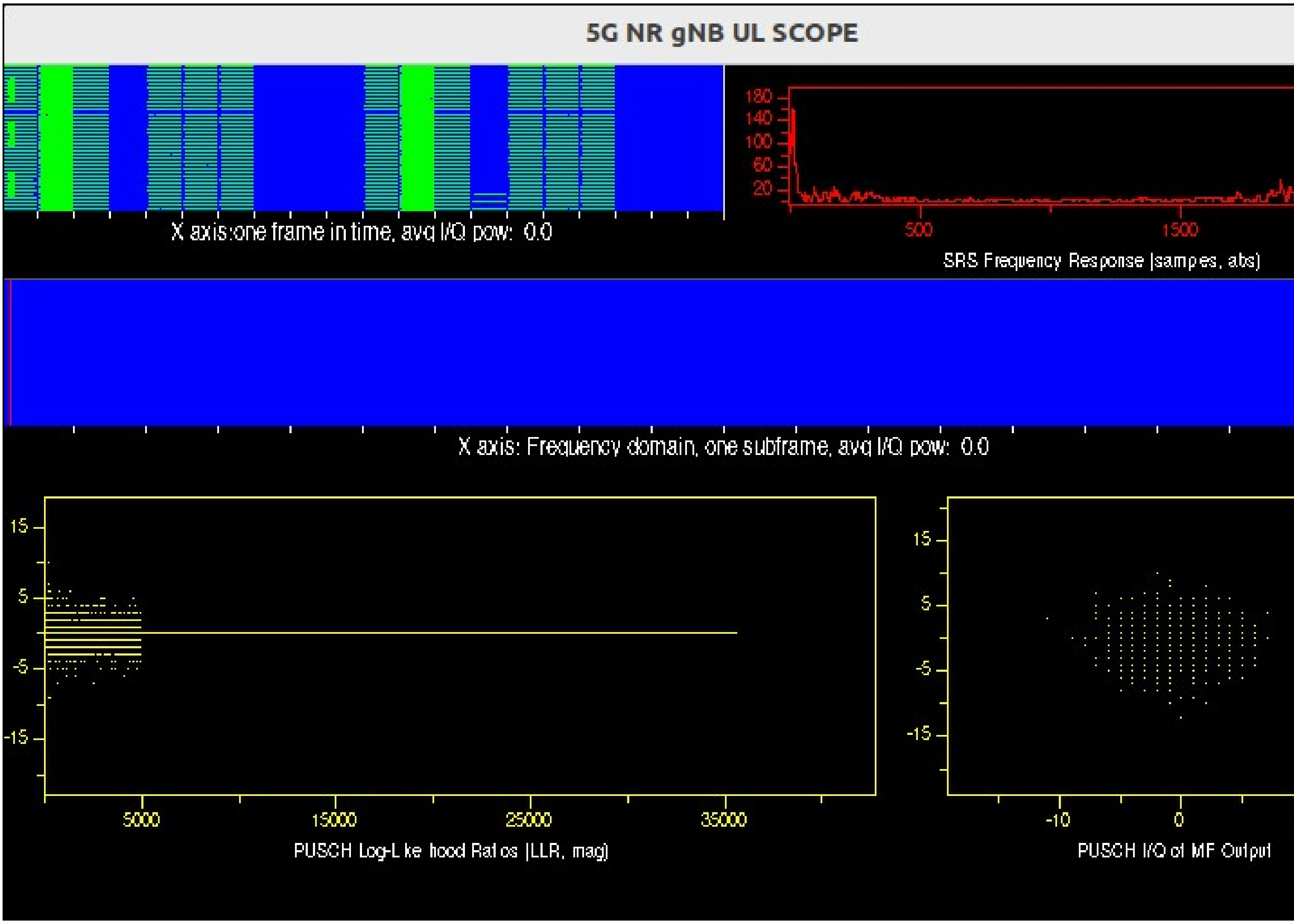
- FlexRIC Controller: Provides RIC E2 agent management and exposes E2 termination APIs for control-plane interaction.
- xApp gRPC Service: Handles AI model control, inference configuration, and feedback reporting via gRPC.
- Test Executor: Orchestrates the real-time toggling of neural receiver activation, collects metrics such as BLER and throughput, and logs inference timing.

In the following subsections, we will describe the role of each component, the startup command, and the interaction workflow with the gNB.

Once the OAI gNB is successfully initialized in PHY test mode, the user can launch the built-in UL Scope to visualize real-time uplink physical layer performance. This visualization is particularly useful for verifying RF synchronization, constellation mapping, and signal demodulation quality from the UE.

The figure shown below shows the 5G NR gNB Uplink Scope output when the UE transmits PUSCH data frames in Band 78. The scope window provides four main diagnostic panels that correspond to different domains of PHY-layer signal analysis:

- Top Left (Time-Domain Power Map): Displays the averaged received power across time frames. The presence of green-blue stripes indicates properly received PUSCH allocations, confirming active uplink transmission and symbol-level synchronization.
- Top Right (SRS Frequency Response): Shows the magnitude of the Sounding Reference Signal (SRS) across subcarriers. This helps in assessing the uplink channel frequency selectivity and verifying correct estimation of the channel response.
- Bottom Left (PUSCH LLR Values): Represents the computed Log-Likelihood Ratios (LLRs) from the demodulated PUSCH symbols. A well-distributed pattern around the horizontal axis indicates balanced bit reliability, while outliers or skewed patterns suggest distortion or incorrect MCS settings.
- Bottom Right (PUSCH Constellation Diagram): Displays the in-phase and quadrature (I/Q) points after matched filtering. The tight clustering of symbols confirms proper demodulation and synchronization. A dispersed or rotated pattern would typically indicate phase misalignment or timing offset.


5G NR gNB UL SCOPE
X axis:one frame in time, avg I/Q pow: 0.0
180
140
100
60
20
500
1500
SRS Frequency Response (samples, abs)
X axis: Frequency domain, one subframe, avg I/Q pow: 0.0
15
5
-5
-15
5000
15000
25000
35000
PUSCH Log-Likelihood Ratios (LLR, mag)
15
5
-5
-15
-10
0
PUSCH I/Q of MF Output

OAI gNB Uplink Scope Visualization
The scope illustrates real-time PUSCH reception quality and frequency-domain response at the gNB side. The upper panels depict SRS-based frequency estimation and time-domain received power, while the lower panels show LLR magnitude distribution and PUSCH constellation for uplink demodulation validation.

This visualization confirms that the PHY layer is functioning correctly and that uplink demodulation, equalization, and neural receiver integration are operating under expected conditions. In the subsequent subsection, we proceed with enabling the FlexRIC controller to monitor and manage the gNB behavior dynamically.

After verifying the gNB operation and PHY-layer performance, the next step involves launching the FlexRIC near-Real-Time (nearRT) RIC. The RIC acts as a programmable control plane entity that interfaces with the gNB through the E2 Application Protocol (E2AP), enabling external control and telemetry collection via xApps.

The FlexRIC instance is executed from the "flexric/build/examples/ric/" directory using the following command.

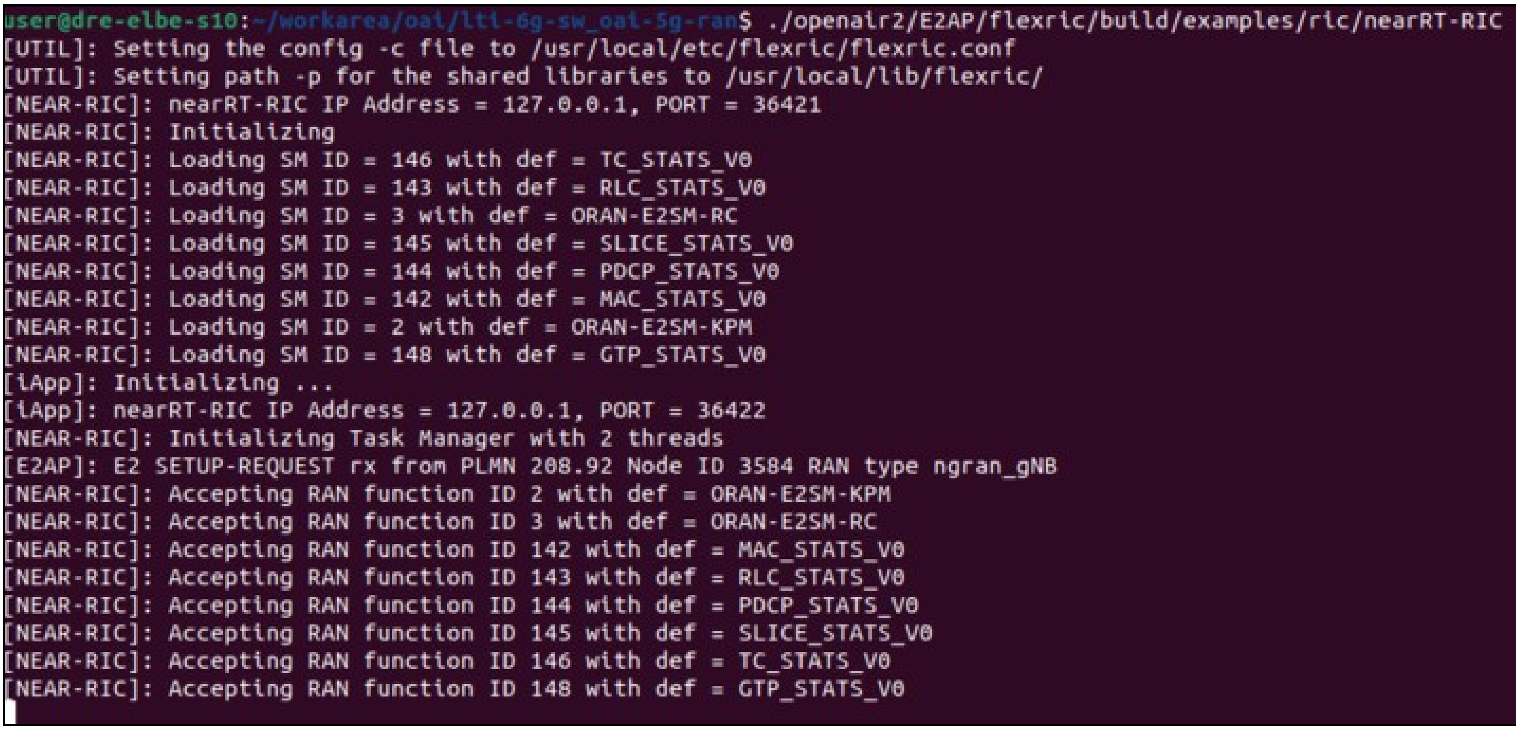
```
openair2/E2AP/flexric/build/examples/ric/nearRT-RIC
```

The figure listed below illustrates the terminal output upon successful initialization of the FlexRIC nearRT-RIC.

The startup log confirms the following items.

- Configuration Setup: The RIC loads the configuration file from "/usr/local/etc/flexric/flexric.conf" and sets up shared libraries from "/usr/local/lib/flexric/".
- IP Binding: The RIC binds to the local loopback interface at IP address 127.0.0.1 with default port numbers 36421 (RIC) and 36422 (xApp interface).
- Service Model Initialization: Multiple Service Models (SMs) are loaded successfully. Each SM corresponds to a specific control or telemetry function, for example:
- ♦
  ◊ MAC_STATS_V0: collects MAC-layer KPIs.
  ◊ RLC_STATS_V0: gathers Radio Link Control metrics.
  ◊ PDCP_STATS_V0: monitors PDCP throughput and latency.
  ◊ SLICE_STATS_V0: manages network slicing configurations.
  ◊ TC_STATS_V0: tracks traffic control performance.
  ◊ ORAN-E2SM-RC and ORAN-E2SM-KPM: implement standardized ORAN E2 Service Models for RAN Control and KPI Measurement.

- E2AP Association: The FlexRIC receives an E2 SETUP REQUEST from the gNB, confirming successful registration of all supported Service Models. Each "RAN function ID" listed (e.g., 2, 3, 143?148) corresponds to an individual SM instance.



```
user@dre-elbe-s10:~/workarea/oai/lti-6g-sw_oai-5g-ran$ ./openair2/E2AP/flexric/build/examples/ric/nearRT-RIC
[UTIL]: Setting the config -c file to /usr/local/etc/flexric/flexric.conf
[UTIL]: Setting path -p for the shared libraries to /usr/local/lib/flexric/
[NEAR-RIC]: nearRT-RIC IP Address = 127.0.0.1, PORT = 36421
[NEAR-RIC]: Initializing
[NEAR-RIC]: Loading SM ID = 146 with def = TC_STATS_V0
[NEAR-RIC]: Loading SM ID = 143 with def = RLC_STATS_V0
[NEAR-RIC]: Loading SM ID = 3 with def = ORAN-E2SM-RC
[NEAR-RIC]: Loading SM ID = 145 with def = SLICE_STATS_V0
[NEAR-RIC]: Loading SM ID = 144 with def = PDCP_STATS_V0
[NEAR-RIC]: Loading SM ID = 142 with def = MAC_STATS_V0
[NEAR-RIC]: Loading SM ID = 2 with def = ORAN-E2SM-KPM
[NEAR-RIC]: Loading SM ID = 148 with def = GTP_STATS_V0
[iApp]: Initializing ...
[iApp]: nearRT-RIC IP Address = 127.0.0.1, PORT = 36422
[NEAR-RIC]: Initializing Task Manager with 2 threads
[E2AP]: E2 SETUP-REQUEST rx from PLMN 208.92 Node ID 3584 RAN type ngran_gNB
[NEAR-RIC]: Accepting RAN function ID 2 with def = ORAN-E2SM-KPM
[NEAR-RIC]: Accepting RAN function ID 3 with def = ORAN-E2SM-RC
[NEAR-RIC]: Accepting RAN function ID 142 with def = MAC_STATS_V0
[NEAR-RIC]: Accepting RAN function ID 143 with def = RLC_STATS_V0
[NEAR-RIC]: Accepting RAN function ID 144 with def = PDCP_STATS_V0
[NEAR-RIC]: Accepting RAN function ID 145 with def = SLICE_STATS_V0
[NEAR-RIC]: Accepting RAN function ID 146 with def = TC_STATS_V0
[NEAR-RIC]: Accepting RAN function ID 148 with def = GTP_STATS_V0
```

FlexRIC nearRT-RIC Initialization
The RIC binds to the loopback IP address, loads multiple Service Models (SMs), and establishes an E2AP session with the gNB. This confirms that the control and monitoring channel between the RIC and gNB is operational.

Once initialized, the nearRT-RIC remains active to receive RAN telemetry, trigger control actions, and host various xApps that can dynamically manage gNB resources (e.g., slicing, QoE, or scheduling policies). In the next section, we proceed to configure and execute the xApp gRPC service for higher-layer interaction.

Once the FlexRIC nearRT-RIC is operational and the gNB is registered under its E2 Node list, the next step is to launch the xApp gRPC Service. This service acts as the interface between the RIC and higher-layer xApps, enabling control, monitoring, and decision-making through standardized gRPC APIs.

The xApp gRPC service is executed from the "xapp_grpc_api" build directory as follows:

```
./openair2/E2AP/flexric/xapp_grpc_api/cmake/build/grpc_ric_srv
```

The figure listed below shows the terminal output of a successful gRPC server initialization.

The sequence of operations is as follows:

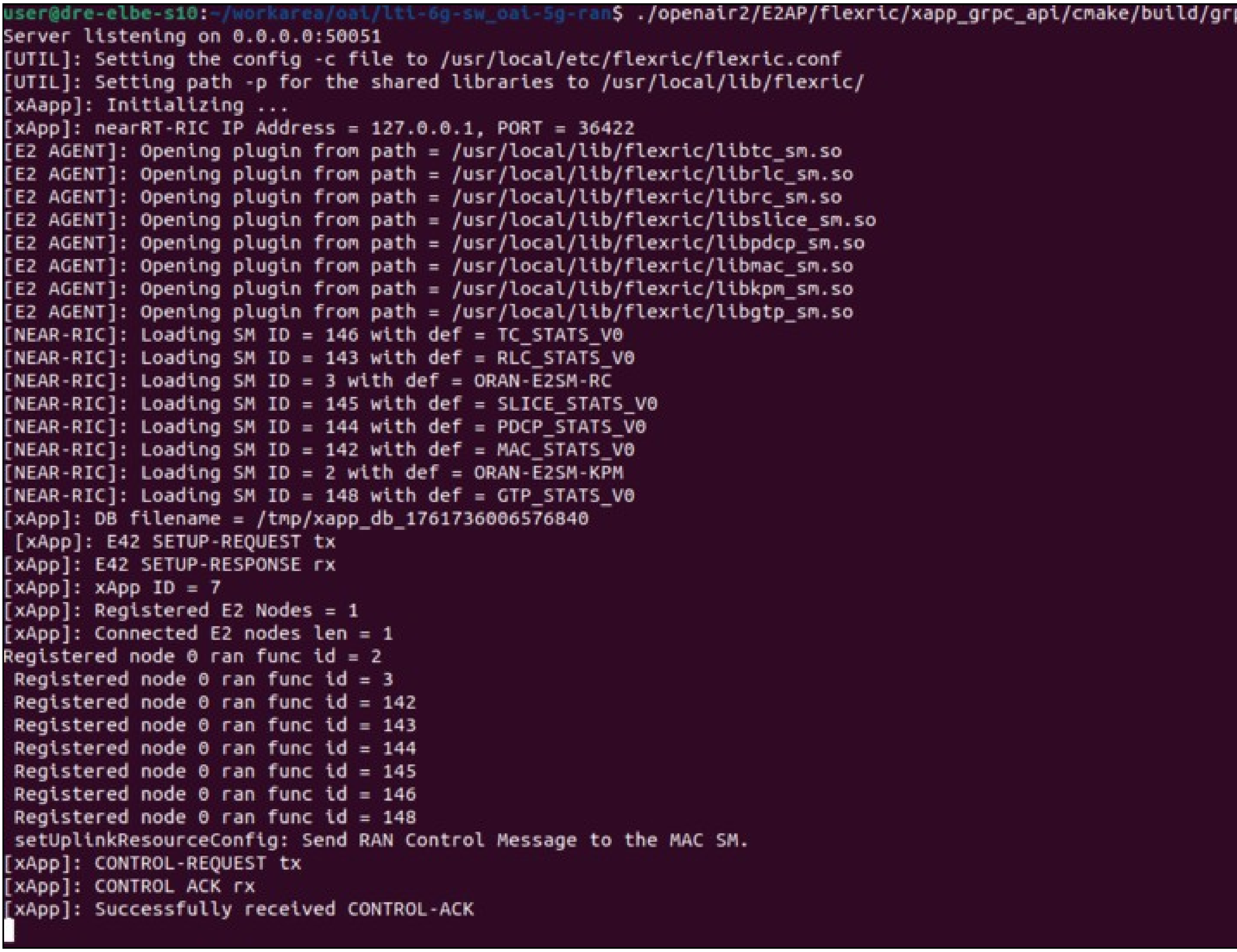
- Initialization: The server binds to the host interface at port \texttt{50051} and loads the FlexRIC configuration file from "/usr/local/etc/flexric/flexric.conf". Shared libraries corresponding to each Service Model (SM) are then dynamically loaded from "/usr/local/lib/flexric". This ensures compatibility with all registered RAN functions.

- Service Model Linking: The xApp reuses the same set of Service Models (MAC_STATS_V0, RLC_STATS_V0, PDCP_STATS_V0, SLICE_STATS_V0, TC_STATS_V0, GTP_STATS_V0, etc.) loaded by the FlexRIC. This linkage allows the xApp to monitor per-layer KPIs and issue control commands.

- E2 Node Registration: The log confirms that one E2 Node (the OAI gNB) is detected and registered with multiple RAN function IDs (2, 3, 142?148), corresponding to ORAN-compliant E2SM service models.

- Database Initialization: The xApp generates a temporary database file (e.g., \texttt{/tmp/xapp_db_1761736006576840}) to store RAN function mappings and maintain session state.

- Control Procedure: The xApp initiates a control procedure named "SetupLinkResourceConfig" to send a RAN control message to the MAC SM. The log line "Successfully received CONTROL-ACK" confirms that the control message was correctly acknowledged by the gNB via FlexRIC.



```
user@dre-elbe-s10:~/workarea/oai/lti-6g-sw_oai-5g-ran$ ./openair2/E2AP/flexric/xapp_grpc_api/cmake/build/grp
Server listening on 0.0.0.0:50051
[UTIL]: Setting the config -c file to /usr/local/etc/flexric/flexric.conf
[UTIL]: Setting path -p for the shared libraries to /usr/local/lib/flexric/
[xAapp]: Initializing ...
[xApp]: nearRT-RIC IP Address = 127.0.0.1, PORT = 36422
[E2 AGENT]: Opening plugin from path = /usr/local/lib/flexric/libtc_sm.so
[E2 AGENT]: Opening plugin from path = /usr/local/lib/flexric/librlc_sm.so
[E2 AGENT]: Opening plugin from path = /usr/local/lib/flexric/librc_sm.so
[E2 AGENT]: Opening plugin from path = /usr/local/lib/flexric/libslice_sm.so
[E2 AGENT]: Opening plugin from path = /usr/local/lib/flexric/libpdcp_sm.so
[E2 AGENT]: Opening plugin from path = /usr/local/lib/flexric/libmac_sm.so
[E2 AGENT]: Opening plugin from path = /usr/local/lib/flexric/libkpm_sm.so
[E2 AGENT]: Opening plugin from path = /usr/local/lib/flexric/libgtp_sm.so
[NEAR-RIC]: Loading SM ID = 146 with def = TC_STATS_V0
[NEAR-RIC]: Loading SM ID = 143 with def = RLC_STATS_V0
[NEAR-RIC]: Loading SM ID = 3 with def = ORAN-E2SM-RC
[NEAR-RIC]: Loading SM ID = 145 with def = SLICE_STATS_V0
[NEAR-RIC]: Loading SM ID = 144 with def = PDCP_STATS_V0
[NEAR-RIC]: Loading SM ID = 142 with def = MAC_STATS_V0
[NEAR-RIC]: Loading SM ID = 2 with def = ORAN-E2SM-KPM
[NEAR-RIC]: Loading SM ID = 148 with def = GTP_STATS_V0
[xApp]: DB filename = /tmp/xapp_db_1761736006576840
 [xApp]: E42 SETUP-REQUEST tx
[xApp]: E42 SETUP-RESPONSE rx
[xApp]: xApp ID = 7
[xApp]: Registered E2 Nodes = 1
[xApp]: Connected E2 nodes len = 1
Registered node 0 ran func id = 2
 Registered node 0 ran func id = 3
 Registered node 0 ran func id = 142
 Registered node 0 ran func id = 143
 Registered node 0 ran func id = 144
 Registered node 0 ran func id = 145
 Registered node 0 ran func id = 146
 Registered node 0 ran func id = 148
 setUplinkResourceConfig: Send RAN Control Message to the MAC SM.
[xApp]: CONTROL-REQUEST tx
[xApp]: CONTROL ACK rx
[xApp]: Successfully received CONTROL-ACK
```

xApp gRPC Service Initialization and Control Exchange
The gRPC server loads Service Models, connects to the FlexRIC nearRT-RIC, registers available RAN functions, and executes a RAN control message exchange (CONTROL-REQUEST/CONTROL-ACK) with the MAC Service Model.

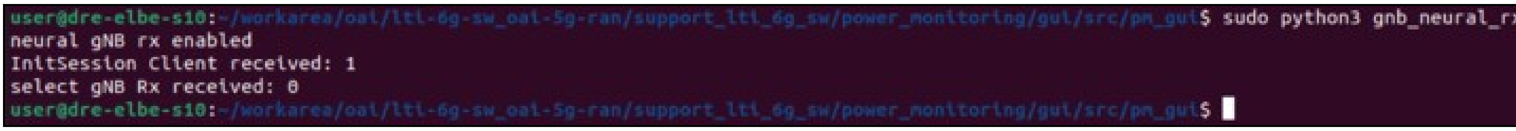
After this step, the system is ready to host intelligent xApps capable of dynamic policy enforcement (e.g., traffic steering, QoE-driven scheduling, or power control) using the gRPC interface to the FlexRIC.

After successfully launching the gNB, FlexRIC nearRT-RIC, and xApp gRPC service, the final step is to execute the Test Executor, which communicates with the Neural Receiver control module to manage inference operations and monitoring events. This executor is responsible for verifying the correct initialization of the neural receiver pipeline, establishing session communication, and enabling or disabling AI-driven reception at runtime.

The Test Executor script is implemented in Python and located in the power monitoring GUI directory:

```
cd ~/workarea/oai/lti-6g-sw_oai-5g-ran/support_lti_6g_sw/power_monitoring/gui/src/pm_gui
sudo python3 gnb_neural_rx_control.py
```

The figure listed below shows the console output after running the script.



```
user@dre-elbe-s10:~/workarea/oai/lti-6g-sw_oai-5g-ran/support_lti_6g_sw/power_monitoring/gui/src/pm_gui$ sudo python3 gnb_neural_r
neural gNB rx enabled
InitSession Client received: 1
select gNB Rx received: 0
user@dre-elbe-s10:~/workarea/oai/lti-6g-sw_oai-5g-ran/support_lti_6g_sw/power_monitoring/gui/src/pm_gui$
```

Test Executor Console Output

- Neural Receiver Initialization: The message "neural gNB rx enabled" confirms that the neural receiver module has been successfully initialized and enabled within the gNB PHY layer.
- Session Setup: The log entry "InitSession Client received: 1" indicates that the test executor has successfully established a communication session with the gNB through the neural receiver API.
- Receiver Selection: The line "select gNB Rx received: 0" denotes that the first available receiver (index 0) has been activated for neural inference. This confirms that the TensorFlow C-API-based inference library is properly linked and ready to execute model predictions during live PHY operation.

This script acts as the control entry point for toggling between the traditional and neural receivers and for visualizing power monitoring results via the integrated GUI interface.

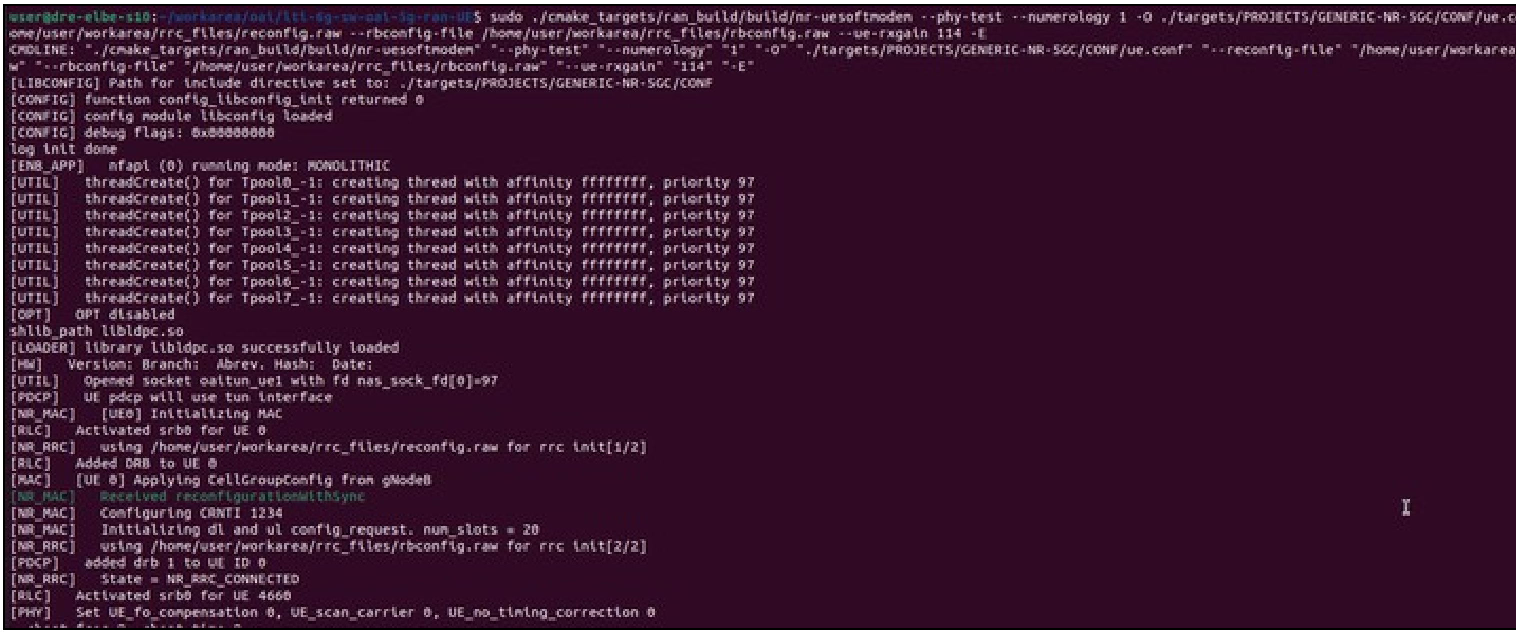
The Python-based Neural Receiver Controller initializes the inference session, enables AI-driven gNB reception, and confirms successful communication with the neural receiver interface.

Once the gNB, FlexRIC, and Neural Receiver modules are active, the next step is to launch the OAI Soft UE to complete the end-to-end 5G link setup in PHY Test Mode. The UE connects directly to the gNB without a core network (CN) and synchronizes using the provided RRC configuration files.

The UE is started using the following command:

```
sudo ./cmake_targets/ran_build/build/nr-uesoftmodem --phy-test --numerology 1 -O ./targets/PROJECTS/GENERIC-NR-5GC/CONF/ue.conf --reconfig
```

The figure listed below shows the console output corresponding to a successful UE initialization and connection.



```
user@dre-elbe-s10:~/workarea/oai/lti-6g-sw-oai-5g-ran-UE$ sudo ./cmake_targets/ran_build/build/nr-uesoftmodem --phy-test --numerology 1 -O ./targets/PROJECTS/GENERIC-NR-5GC/CONF/ue.c
ome/user/workarea/rrc_files/reconfig.raw --rbconfig-file /home/user/workarea/rrc_files/rbconfig.raw --ue-rxgain 114 -E
CMDLINE: "./cmake_targets/ran_build/build/nr-uesoftmodem" "--phy-test" "--numerology" "1" "-O" "./targets/PROJECTS/GENERIC-NR-5GC/CONF/ue.conf" "--reconfig-file" "/home/user/workarea
w" "--rbconfig-file" "/home/user/workarea/rrc_files/rbconfig.raw" "--ue-rxgain" "114" "-E"
[LIBCONFIG] Path for include directive set to: ./targets/PROJECTS/GENERIC-NR-5GC/CONF
[CONFIG] function config_libconfig_init returned 0
[CONFIG] config module libconfig loaded
[CONFIG] debug flags: 0x00000000
log init done
[ENB_APP]   nfapi (0) running mode: MONOLITHIC
[UTIL]   threadCreate() for Tpool0_-1: creating thread with affinity ffffffff, priority 97
[UTIL]   threadCreate() for Tpool1_-1: creating thread with affinity ffffffff, priority 97
[UTIL]   threadCreate() for Tpool2_-1: creating thread with affinity ffffffff, priority 97
[UTIL]   threadCreate() for Tpool3_-1: creating thread with affinity ffffffff, priority 97
[UTIL]   threadCreate() for Tpool4_-1: creating thread with affinity ffffffff, priority 97
[UTIL]   threadCreate() for Tpool5_-1: creating thread with affinity ffffffff, priority 97
[UTIL]   threadCreate() for Tpool6_-1: creating thread with affinity ffffffff, priority 97
[UTIL]   threadCreate() for Tpool7_-1: creating thread with affinity ffffffff, priority 97
[OPT]   OPT disabled
shlib_path libldpc.so
[LOADER] library libldpc.so successfully loaded
[HW]   Version: Branch:  Abrev. Hash:  Date:
[UTIL]   Opened socket oaitun_ue1 with fd nas_sock_fd[0]=97
[PDCP]   UE pdcp will use tun interface
[NR_MAC]   [UE0] Initializing MAC
[RLC]   Activated srb0 for UE 0
[NR_RRC]   using /home/user/workarea/rrc_files/reconfig.raw for rrc init[1/2]
[RLC]   Added DRB to UE 0
[MAC]   [UE 0] Applying CellGroupConfig from gNodeB
[NR_MAC]   Received reconfigurationWithSync
[NR_MAC]   Configuring CRNTI 1234
[NR_MAC]   Initializing dl and ul config_request. num_slots = 20
[NR_RRC]   using /home/user/workarea/rrc_files/rbconfig.raw for rrc init[2/2]
[PDCP]   added drb 1 to UE ID 0
[NR_RRC]   State = NR_RRC_CONNECTED
[RLC]   Activated srb0 for UE 4660
[PHY]   Set UE_fo_compensation 0, UE_scan_carrier 0, UE_no_timing_correction 0
```

OAI Soft UE Initialization and Connection
The UE successfully applies configuration from the gNB, activates SRB and DRB channels, and transitions to the "NR_RRC_CONNECTED" state, confirming successful PHY-level attachment.

- Initialization Phase: The log lines showing multiple "threadCreate()" calls confirm the creation of UE processing threads responsible for PHY and MAC operations. The message "library libldpc.so successfully loaded" indicates that the LDPC decoder (for 5G NR) is correctly linked.

- RRC Configuration and Setup: The UE applies configuration parameters received from the gNB through the lines "Applying CellGroupConfig from gNodeB" and "Added DRB to UE 0". These confirm that the UE's RLC, MAC, and PDCP layers have been successfully initialized.

- Successful Connection: The key message "Received reconfigurationWithSync" followed by "State = NR_RRC_CONNECTED" indicates that the UE has synchronized and established an RRC connection with the gNB. This marks a fully functional PHY Test mode setup, enabling uplink and downlink channel measurements and validation.

Once both the gNB and OAI Soft UE are successfully initialized, and the Neural Receiver has been enabled, then the gNB graphical interface ("nr_ul_scope") can be used to visualize the uplink (UL) channel and decoding performance in real time. This visualization is part of the OAI monitoring utilities and is automatically launched when the "--nrscope" flag is enabled during gNB execution.

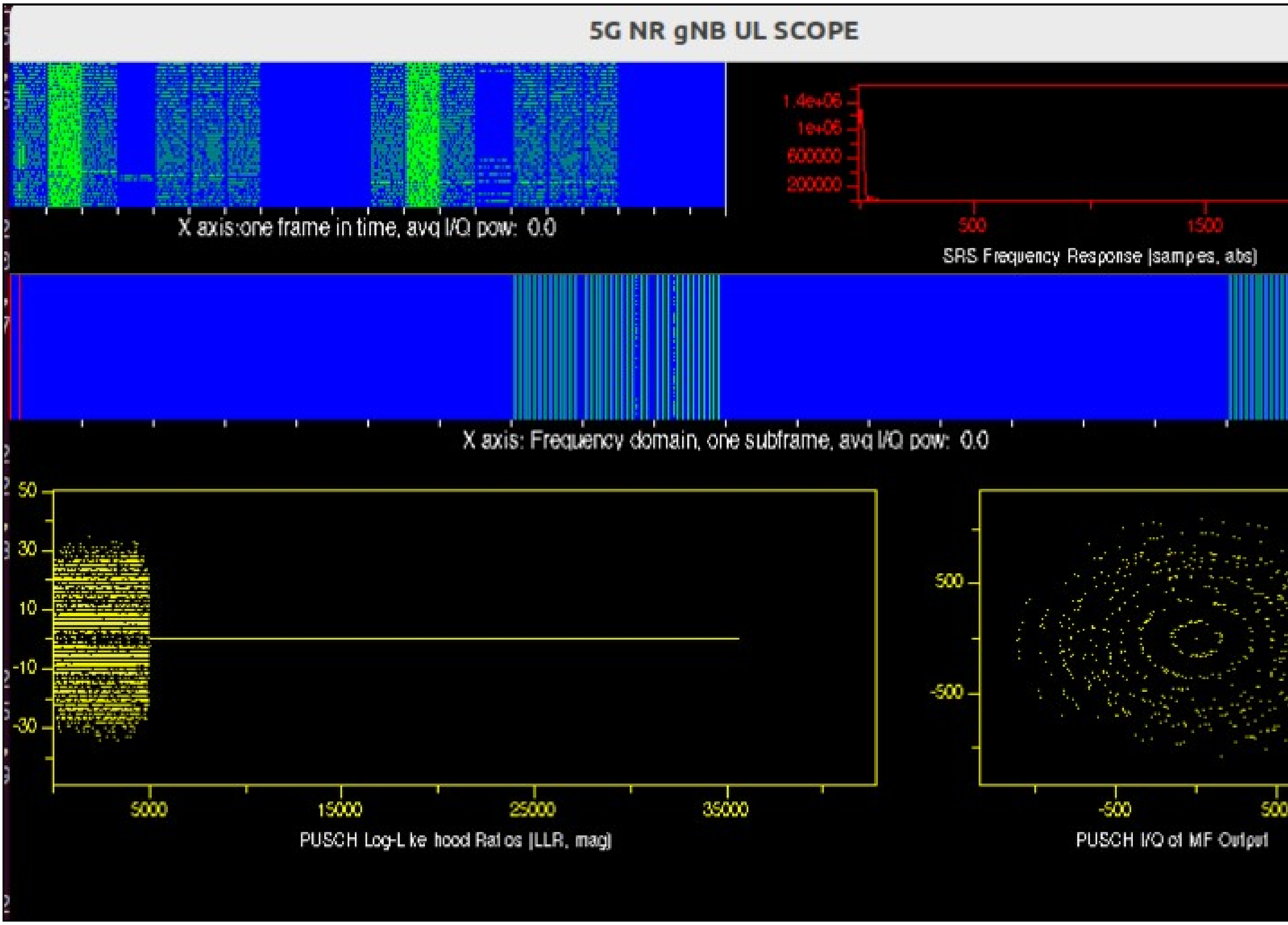
- Top Left Panel (Time Domain Signal): Displays the received uplink IQ samples over time, showing frame-wise power variations. The clear bursts correspond to PUSCH (Physical Uplink Shared Channel) transmissions from the UE.

- Top Right Panel (SRS Frequency Response): Illustrates the Sounding Reference Signal (SRS) channel frequency response used for uplink channel estimation. This provides insights into frequency-selective fading and link stability.

- Middle Panel (Frequency Domain Representation): Shows the frequency-domain mapping of received symbols, indicating the spectral occupancy of the UE signal.

- Bottom Left (Log-Likelihood Ratios (LLRs)): Presents the computed soft information before decoding, helping assess the reliability of bit decisions within the LDPC decoder. The concentration of LLR values indicates good SNR and effective demodulation by the Neural Receiver.

- Bottom Right (Constellation Plot (MF Output)): Shows the equalized modulation constellation (e.g., QPSK, 16-QAM). A clean, tight clustering of points demonstrates that the Neural Receiver has successfully performed channel equalization and interference mitigation.


5G NR gNB UL SCOPE
X axis:one frame in time, avg I/Q pow: 0.0
1.4e+06
1e+06
600000
200000
500
1500
SRS Frequency Response (samples, abs)
X axis: Frequency domain, one subframe, avg I/Q pow: 0.0
50
30
10
-10
-30
5000
15000
25000
35000
PUSCH Log-Likelihood Ratios (LLR, mag)
500
-500
-500
500
PUSCH I/Q of MF Output

5G NR gNB UL Scope with Neural Receiver Enabled
The 5G NR gNB UL Scope with Neural Receiver Enabled provides a real-time visualization of uplink IQ samples, frequency-domain response, Log-Likelihood Ratios (LLR), and modulation constellation after successful UE connection and Neural Receiver activation.

The Neural Receiver replaces traditional MMSE equalization with a learned inference model that operates on GPU, leading to improved robustness under non-linear channel distortions and interference conditions. The stability of the constellation and reduced noise dispersion in LLR plots confirm successful inference and decoding.

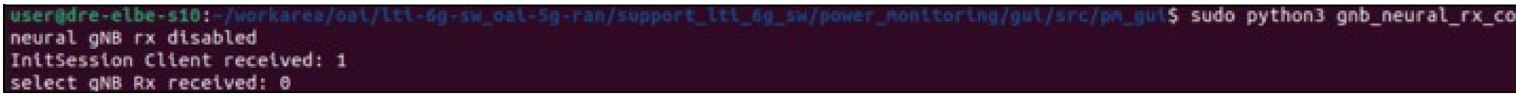
The Neural Receiver can be dynamically enabled or disabled during runtime using the "gnb_neural_rx_control.py" interface located at `support_lti_6g_sw/power_monitoring/gui/src/pm_gui` folder.

This Python-based control executor communicates with the neural inference engine on the gNB through a lightweight gRPC interface, allowing real-time activation or deactivation of the neural model.

When the Neural Receiver is "enabled", the gNB offloads channel estimation and decoding tasks to the TensorFlow-based inference library running on the GPU.

When the Neural Receiver is "disabled", the system reverts to the traditional MMSE-based receiver chain, executing all signal processing operations on the CPU.

The figure listed below shows the CLI output when the Neural Receiver is disabled. In this case, the log confirms successful initialization of the control session but indicates that inference offloading is turned off.


```
user@dre-elbe-s10:~/workarea/oai/lti-6g-sw_oai-5g-ran/support_lti_6g_sw/power_monitoring/gui/src/pm_gui$ sudo python3 gnb_neural_rx_co
neural gNB rx disabled
InitSession Client received: 1
select gNB Rx received: 0
```

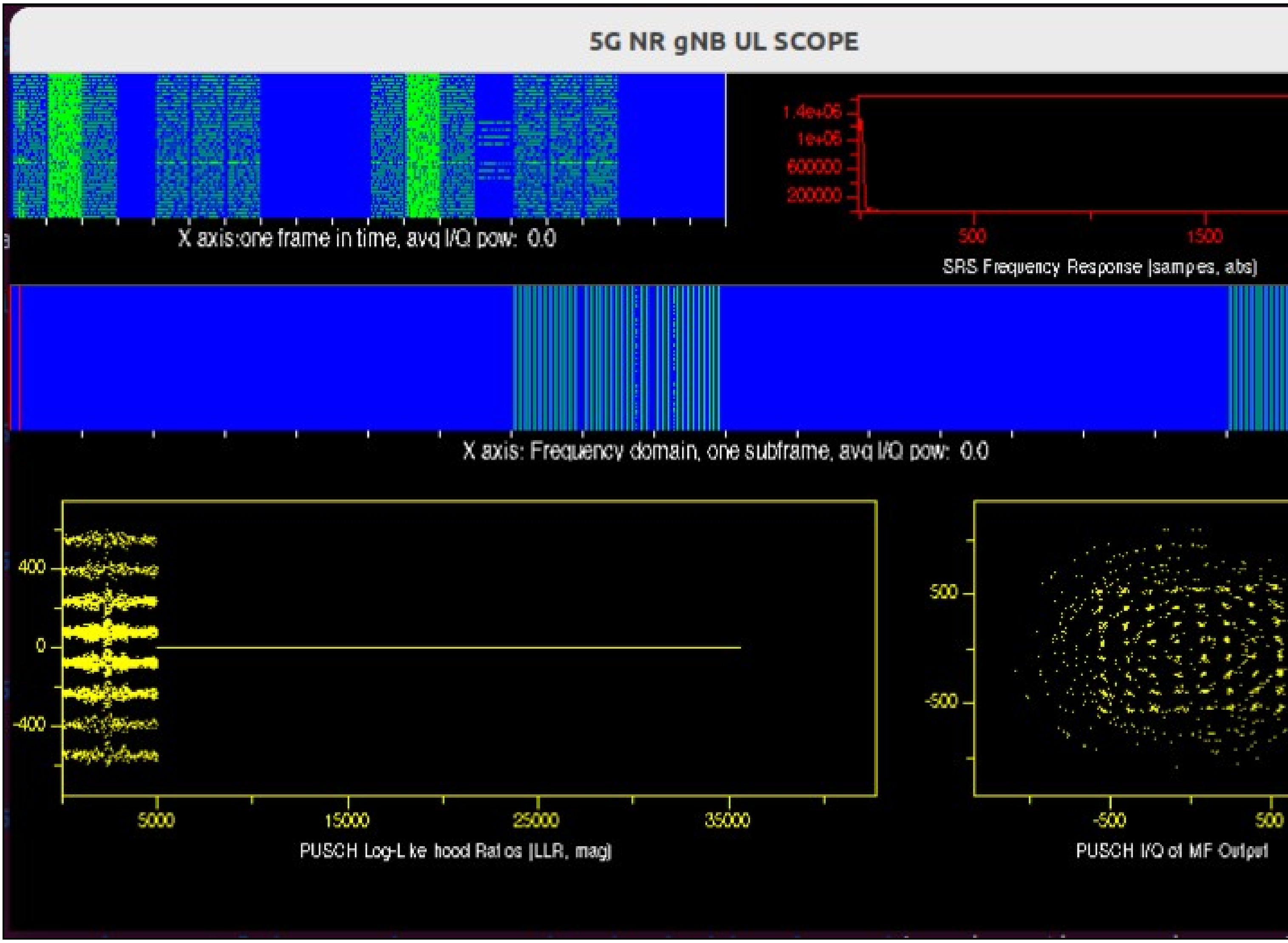
Text Executor Interface for Neural Receiver Control (Disabled Mode)
The control client connects to the gNB inference service, confirming session initialization and reporting that the Neural Receiver is disabled.

The control tool also reports the selected receiver mode ID received from the gNB (shown as "select gNB Rx received: 0"), where "0" corresponds to the traditional receiver and "1" corresponds to the Neural Receiver. This interface can be integrated into higher-level O-RAN xApps for automated receiver selection based on link quality or interference metrics.

When the Neural Receiver is disabled, the OAI gNB operates using the baseline linear MMSE receiver chain implemented in the PHY layer. In this configuration, the uplink decoding relies solely on least-squares (LS) and MMSE channel estimation without GPU-based inference. This mode serves as a reference to evaluate the performance gain of the Neural Receiver.

The figure listed below shows the 5G NR gNB UL Scope output when a UE is successfully connected using the traditional receiver. The displayed plots correspond to:

- Top left: Time-domain PUSCH signal across multiple frames.
- Top right: Frequency response of the SRS (Sounding Reference Signal), illustrating the uplink channel condition.
- Middle: Frequency-domain representation of the received frame, indicating active RBs allocated for the UE.
- Bottom left: Log-likelihood ratios (LLRs) for the decoded uplink data.
- Bottom right: PUSCH constellation after matched filtering, showing symbol dispersion before neural equalization.


5G NR gNB UL SCOPE
X axis:one frame in time, avg I/Q pow: 0.0
1.4e+06
1e+06
600000
200000
500
1500
SRS Frequency Response (samples, abs)
X axis: Frequency domain, one subframe, avg I/Q pow: 0.0
400
0
-400
5000
15000
25000
35000
PUSCH Log-Likelihood Ratios (LLR, mag)
500
-500
-500
500
PUSCH I/Q of MF Output

5G NR gNB UL Scope with Traditional Receiver
The gNB UL Scope with Traditional Receiver shows a visualization of uplink signals and constellations when the gNB operates with the standard MMSE receiver chain. The results serve as the baseline for comparison against the Neural Receiver mode.

The constellation diagram in the lower right shows moderate symbol spread, corresponding to the unassisted channel estimation performance of the classical receiver. In contrast, as seen in the Neural Receiver mode, the neural-assisted equalization improves symbol clustering and decoding reliability.

In this document, we described how the NI USRP hardware can be used for real-time AI/ML model validation in a realistic end-to-end system. Although we used the neural receiver as an example, the general workflow and methodology from design to test to a real-world deployment can also be applied to other AI/ML wireless communications. The implementation of the AI/ML reference architecture will help researchers and engineers with these tasks and challenges:

- Work from a trained AI/ML model in simulation to an initial system-level prototype.
- Validate and benchmark the AI/ML model in a real-world, system -level test environment.
- Explore how to develop low complexity yet robust ML models for real-time operation.
- Investigate the real time versus complexity versus performance trade-off.
- Generate synchronized data sets from real-time systems.
- Train and test for a broad set of scenarios and configurations.
- Automatically execute pre-selected test sequences through test entity.
- Emulate wireless scenarios using statistical channel models including mobility.
- OTA validation in end-to-end testbed environments.
- Generate synchronized data sets from real-time system.

These steps will develop more robust and reliable AI/ML models for realistic deployment conditions and help us understand the areas that AI can lead to real improvements compared to the current networks. Ultimately, it will lead to more trust and utility in AI-enhanced wireless communications systems by validating key use cases and by discovering possible unexpected and unwanted behavior.