Annals of Telecommunications
https://doi.org/10.1007/s12243-026-01231-3

ns3-oran-customizable-db: a tool for building a customizable database for Open RAN simulations

Felipe G. Táparo[1] · Daniel O. de Arruda[2] · Igor M. Moraes[2] · Miguel Elias M. Campista[1]

Received: 20 March 2026 / Accepted: 16 October 2026
© The Author(s) 2026

Abstract
Simulation is an essential approach for evaluating the performance of applications designed to control and diagnose Radio Access Networks (RANs). Assessing proposed solutions in simulated environments that closely approximate real-world network behavior is crucial for mitigating potential deployment risks. This article aims to extend the capabilities of the "ns3-oran" module in the "ns-3" network simulator. The new functionalities remove restrictions on the type of data that can be exchanged with the Near-RT RIC (Near-Real-Time RAN Intelligent Controller), thereby increasing flexibility for researchers and developers. The ability to collect, store, and analyze heterogeneous data is a key enabler for designing and integrating machine-learning-based solutions into the RAN. With these enhancements, the simulator can support the creation of scenarios tailored to training machine learning models, leveraging performance metrics previously unavailable or not configurable in ns-3. As a result, the proposed "ns3-oran-customizable-db" provides a more powerful simulation environment for advancing AI-driven RAN control and optimization approaches.

Keywords ns3-oran-customizable-db · Open RAN simulations · Radio Access Network (RAN)

1 Introduction

In recent years, the growth of mobile networks has enabled the emergence of a wide range of new applications with distinct characteristics. Each of these applications presents different quality-of-service requirements and network demands. For instance, video streaming typically requires high data rates while being more tolerant of latency, whereas online gaming is highly sensitive to latency but can operate with lower throughput [1]. In this context of heterogeneous and often conflicting requirements, the mobile network infrastructure must adapt to diverse application needs to improve delivered performance. However, meeting such diverse requirements remains a challenge for current mobile network architectures. These architectures often rely on proprietary equipment, which hinders interoperability between vendors and limits the network's ability to dynamically reconfigure in response to varying needs [2].

To address these limitations, O-RAN has emerged as an open, disaggregated Radio Access Network (RAN) architecture standardized by the O-RAN Alliance [3]. O-RAN defines open, interoperable interfaces, enabling multi-vendor deployments and decoupling network functions from proprietary hardware and software [4]. A key O-RAN innovation is the introduction of two intelligent controllers to enable RAN automation and optimization: the Non-Real-Time RAN Intelligent Controller (Non-RT RIC) and the Near-Real-Time RAN Intelligent Controller (Near-RT RIC). These controllers host custom logic applications, called rApps and xApps, which can execute closed-loop control actions on the RAN [5]. The Non-RT RIC operates on control loops longer than 1 s and executes rApps, while the Near-RT RIC operates on control loops between 10 ms and 1 s and executes xApps [2]. The O-RAN architecture promotes modularity by allowing rApps and xApps to be developed and deployed as

✉ Felipe G. Táparo
felipe.taparo@gta.ufrj.br

Daniel O. de Arruda
doarruda@id.uff.br

Igor M. Moraes
igor@ic.uff.br

Miguel Elias M. Campista
miguel@gta.ufrj.br

[1] Laboratório GTA/DEL-Poli/PEE-COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brazil

[2] Laboratório MídiaCom, Instituto de Computação, Universidade Federal Fluminense, Niterói, Brazil

Published online: 09 November 2026

containerized microservices, enabling third-party developers to contribute in an open-source manner. These applications cover a wide range of RAN control and management tasks, including handover optimization, interference coordination, and beamforming. Moreover, O-RAN explicitly supports integrating machine-learning-based functionalities, allowing rApps and xApps to exploit data-driven methods for intelligent RAN control and management [6, 7].

Despite the advances enabled by the O-RAN architecture, developing applications that incorporate machine learning (ML) techniques for RAN orchestration still requires extensive testing and experimentation [8, 9]. Before being deployed in real networks, such applications must undergo essential validation steps, including security and performance assessments. However, the current research landscape presents significant challenges in this regard, mainly due to the limited availability of simulators specifically designed for O-RAN controllers and the complexity associated with using existing solutions [10, 11]. This article is an extension of ns3-oran-customizable-db [12], which aims to extend the functionality of ns3-oran, one of the main open-source simulators currently available for O-RAN networks [13]. The idea is to overcome ns3-oran's main limitations and to provide a more efficient simulation environment for O-RAN applications. A new reporting acquisition system is introduced, enabling more generalized data collection and the creation of custom reports. In addition, this work extends the data storage mechanisms for the collected information by developing tools to query and store heterogeneous data types originating from multiple sources. These changes facilitate the training of machine learning models using the simulator. It is now possible to collect various data types and use them flexibly throughout the simulation, directly supporting the training process. This extended version introduces the Open-RAN architecture in Sect. 2 and provides a more comprehensive explanation of the proposed framework, including its architecture and example use cases. This includes detailed descriptions of every implemented method and a complete breakdown of the files used in the examples. Furthermore, performance benchmarks are provided for the implemented database system, comparing its scalability and memory footprint against the default ns3-oran system. Additionally, a novel example use case is introduced to demonstrate how the enhanced system can manage handovers using a custom, user-defined UE report metric.

In order to illustrate the usefulness of this work and demonstrate the capabilities of the proposed tools, three examples of their usage were developed. The first is a simple example showing how to generate custom reports for user equipment connected to a remote server. This example demonstrates the integration of the new functionalities with existing ones, using both custom reports implemented directly in the simulation script and default reports implemented in ns3-oran. The second example shows how the tool can be used in handover management by creating a custom Signal-to-Interference-plus-Noise-Ratio (SINR) report. This custom report is used as a threshold to trigger the handover process. Finally, a third example shows how the developed tools can be used to train a reinforcement learning model. In this example, data is collected from different network components and fed into a model. For this example, external tools are used to integrate a model being trained in Python with ns-3.

The remainder of this article is organized as follows. Section 2 presents key aspects of the O-RAN architecture. Section 3 discusses related work. Section 4 describes how ns-3 and ns3-oran operate. Section 5 presents the architecture of the proposed tool, namely the ns3-oran-customizable-db simulator. Section 6 presents use cases that demonstrate the simulator's effectiveness. Finally, Sect. 7 concludes this article and outlines future research directions for improving the proposed tool.

2 O-RAN architecture

Traditional RAN deployments rely on tightly integrated and vendor-specific hardware and software components, which limit interoperability and hinder innovation. To address these limitations, several research and standardization initiatives have promoted Open RAN as a key paradigm for next-generation RAN architectures [2]. Open RAN deployments rely on the principles of component disaggregation and open interfaces, adopting a white-box approach to infrastructure elements. The O-RAN architecture aims to implement these principles. The O-RAN Alliance [3], a consortium of telecommunications operators and research institutions, standardizes this architecture. The O-RAN Alliance's main objective is to develop technical specifications that standardize an open, interoperable Radio Access Network model.

Regarding the overall O-RAN architecture, the functional split of base station components is standardized under the so-called Split 7.2x option. Under this split, base stations are decomposed into three main components: the O-RAN Radio Unit (O-RU), the O-RAN Distributed Unit (O-DU), and the O-RAN Central Unit (O-CU). The O-RU implements radio frequency (RF) processing and the lower-PHY functions of the physical layer. The O-DU implements upper-PHY functions and the data-link layer, including the Medium Access Control (MAC) and Radio Link Control (RLC) protocols. Finally, the O-CU handles higher-layer functions, including Radio Resource Control (RRC), the Service Data Adaptation Protocol (SDAP), and the Packet Data Convergence Protocol (PDCP). Within the Open-RAN architecture, the O-CU is further divided into two logical subcomponents: the control

plane (O-RAN Central Unit Control Plane – O-CU-CP) and the user plane (O-RAN Central Unit User Plane – O-CU-UP).

The O-RAN architecture also includes components that enable machine-learning-based network management, namely the Service Management and Orchestration (SMO) framework and two controllers operating at different time scales: the Non-RT RIC and the Near-RT RIC. The SMO is responsible for managing machine-learning applications and enriching network information. The Non-RT RIC is hosted within the SMO and operates control loops with durations longer than one second. It supports rApps, customizable logic applications possibly developed by third parties, including open-source solutions. These rApps can implement application-specific logic and perform control actions, potentially leveraging machine learning techniques.

The Near-RT RIC, in turn, is a controller that operates control loops with time scales ranging from 10 ms to 1 s. Similar to the Non-RT RIC, the Near-RT RIC supports customizable logic applications, referred to as xApps. The Near-RT RIC communicates with O-CUs, O-DUs, and O-RAN–enabled base stations through standardized interfaces. In this work, the primary O-RAN component being simulated is the Near-RT RIC, as it enables the execution of xApps responsible for Near-RT RAN control and data collection. These components communicate through open interfaces, each serving a distinct purpose. Figure 1 illustrates the connections present in the O-RAN architecture, highlighting both O-RAN–defined interfaces and interfaces inherited from 3GPP specifications.

The A1 interface connects the Non-RT RIC to the Near-RT RIC and transmits control and management information from the Non-RT RIC to the Near-RT RIC. The O1 interface connects architectural components to the SMO, enabling orchestration and management of the system components. The O2 interface connects the SMO to an external cloud service outside the RAN (O-Cloud), allowing RAN management and monitoring through cloud-based infrastructure. The Open Fronthaul interface connects O-RUs and O-DUs within the same gNB.

The most relevant interface for this work is the E2 interface. The E2 interface connects the Near-RT RIC to the E2 nodes managed by the controller. E2 nodes are defined as RAN entities connected through the E2 interface, such as O-DUs, O-CUs, gNBs, or eNBs. The E2 interface enables the exchange of telemetry data and control commands between the Near-RT RIC and the connected nodes. For instance, in one use case presented in this work, the Near-RT RIC sends handover commands to the eNBs via the E2 interface, while the eNBs transmit telemetry data back to the controller.

3 Related work

There is a limited number of simulators specifically designed for O-RAN controllers [10, 11]. Currently, two main open-source simulation tools are available for O-RAN networks: ns-O-RAN [14] and ns3-oran [13]. Both tools are built on top of the ns-3 [15] network simulator and extend it with

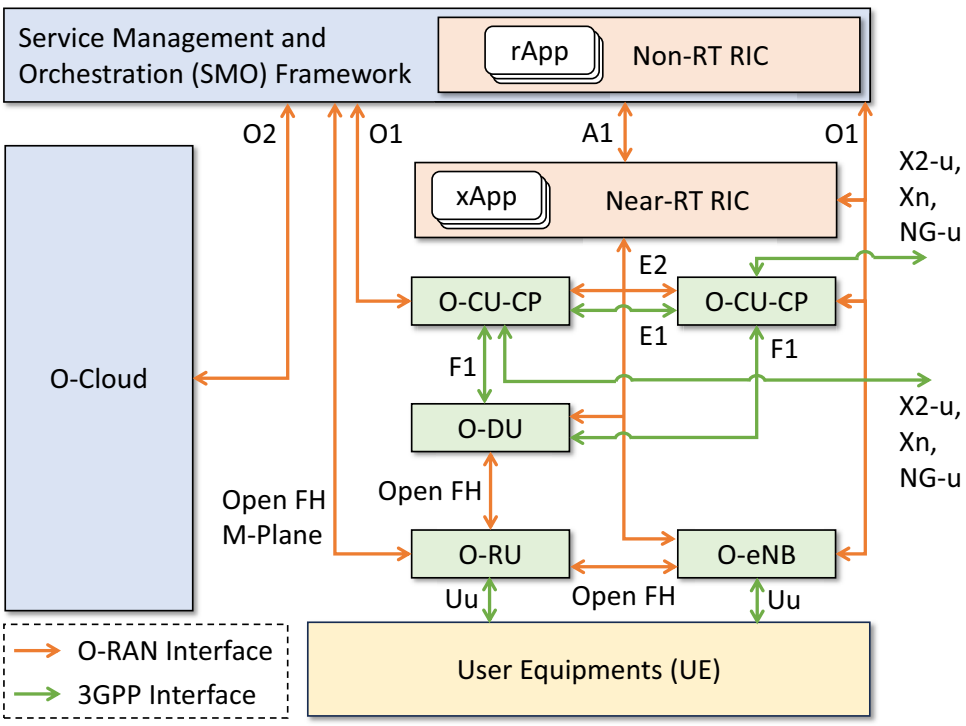


Fig. 1 The O-RAN architecture

functionalities to support experimentation with the O-RAN architecture. The former integrates a real Near-RT RIC — provided by the O-RAN Software Community [16] — running in parallel with ns-3, which imposes high computational requirements during simulation. The latter implements the O-RAN architectural components directly within ns-3, without relying on a real Near-RT RIC. Although this approach offers significant computational efficiency, the simulated Near-RT RIC implementation presents notable limitations, such as restricted options for collecting data from network elements.

Several tools have been developed to integrate artificial intelligence frameworks with ns-3. Gawłowicz and Zubow introduced ns3-gym [17], an open-source tool that integrates the ns-3 network simulator with the OpenAI Gym interface, enabling reinforcement learning algorithms to be applied to networking problems. The ns3-gym tool establishes a standardized communication interface between the simulator and reinforcement-learning agents by defining observation, action, reward, and termination spaces, allowing researchers to leverage existing RL libraries without modifying the simulator core. Its modular architecture supports both simulation-driven and real-time interactions, making it suitable for a wide range of networking scenarios. However, its architecture relies on inter-process communication between the simulator and the learning agent, which introduces synchronization overhead and may limit scalability in computationally intensive scenarios. Yin et al. proposed ns3-ai [18] as an evolution of ns3-gym. The ns3-ai tool replaces ns3-gym's socket-based inter-process communication with a shared-memory architecture, significantly reducing communication overhead and enabling much faster data exchange between ns-3 and external AI frameworks. In addition, it introduces a higher-level abstraction layer that facilitates interoperability with multiple Python-based machine learning libraries and demonstrates its effectiveness through a 5 G NR use case. Despite these improvements, ns3-ai still adopts a loosely coupled architecture in which the learning agent executes outside the simulator, requiring explicit synchronization between simulation and training processes.

Alternatively to simulators, there are tools to emulate O-RAN deployments and also stack implementations and experimental platforms. OpenAirInterface (OAI) [19, 20] provides an open-source implementation of cellular protocol stacks that can be executed as a softwarized RAN, enabling experiments with implementations closer to operational 4 G/5 G networks. OAI supports several 3GPP and O-RAN functional splits and provides interfaces such as E2 and O1 for RAN control and management, making it suitable for developing and evaluating O-RAN-compliant solutions. Its open, programmable architecture also enables integration of external intelligent controllers and xApps. Nevertheless, experiments based on OAI require substantially more computational and configuration resources than discrete-event simulation, particularly when real-time execution and software-defined radio components are involved.

Schmidt et al. introduced FlexRIC [21], an open-source software development kit (SDK) for O-RAN-compliant Software-Defined Radio Access Networks (SD-RANs). FlexRIC provides a RIC implementation, E2 service models, and support for multiple RAN platforms, enabling monitoring and control of radio network functions. It allows developers to implement customized xApps for resource management, slicing, and AI-driven network optimization while maintaining compatibility with the O-RAN E2 interface. Despite these advantages, FlexRIC primarily focuses on RIC programmability and efficient control-plane communication rather than providing an integrated environment for AI model development and training. Consequently, machine learning algorithms must be implemented and managed externally, requiring developers to handle data preprocessing, model lifecycle management, and interaction with external AI frameworks.

Colosseum [22] is a large-scale wireless network emulator with hardware in the loop that combines programmable software-defined radios, a high-fidelity RF channel emulator, and softwarized 5 G and O-RAN protocol stacks. This architecture enables researchers to reproduce realistic wireless topologies and evaluate RAN control algorithms under controlled and repeatable channel conditions. More recently, Colosseum has been positioned as an Open RAN digital twin [23], supporting the development, prototyping, and testing of AI/ML-based solutions while enabling experiments that bridge emulated and real-world networks. Despite its high fidelity and flexibility, Colosseum depends on specialized experimental infrastructure and requires substantially more computational and operational resources than software-only simulators.

Arena [24] is an open-access wireless experimentation platform designed for real-time research on sub-6 GHz 5 G and beyond networks. It consists of a ceiling-mounted grid of 64 antennas connected to programmable software-defined radios, together with a dedicated computing infrastructure, providing a reconfigurable and repeatable indoor wireless environment. Arena has been used to investigate a broad range of wireless technologies, including multi-cell 5 G networks, MIMO, spectrum sensing, and AI-based wireless applications. However, unlike simulation-based approaches, experiments on Arena are constrained by the physical topology, available radio resources, and execution time of the experimental platform.

Bonati et al. introduced OpenRAN Gym [25], an open toolbox for data collection, AI/ML development, and experimentation in O-RAN environments. The framework provides an end-to-end workflow for collecting RAN measurements, designing and training data-driven models, and deploying AI/ML-based xApps for RAN control. OpenRAN Gym com-

bines software components for RAN data collection and control with a lightweight Near-RT RIC and supports experimentation on platforms such as Colosseum and Arena. Its main strength is the integration of the AI development workflow with experimental O-RAN infrastructures, enabling solutions developed in controlled environments to be subsequently evaluated on wireless platforms. However, this experimental orientation also makes the framework dependent on the availability and configuration of the underlying wireless infrastructure. Consequently, large-scale parameter exploration, rapid experimentation, and highly controlled comparisons between different network configurations can be more difficult than in simulation-based environments. Moreover, the framework primarily orchestrates existing RAN, data collection, and AI components rather than providing a unified simulation environment in which the complete O-RAN control loop can be systematically modeled and evaluated.

The ns3-oran simulator provides an efficient O-RAN simulation environment, but its data collection capabilities are limited. Existing AI-oriented frameworks provide mechanisms to connect simulations to ML agents but do not address O-RAN-specific data acquisition and management. Experimental platforms provide realistic data, but at a higher cost. This work focuses on ns3-oran and fills this gap by extending the simulator with a new reporting acquisition system, enabling more generalized data collection and custom report creation. We also extend the data storage mechanisms for the collected information by developing tools to query and store heterogeneous data types originating from multiple sources. These changes facilitate the training of machine learning models using the simulator for AI/ML experimentation.

4 The ns-3 simulator and the ns3-oran extension

Ns-3 [15] is a widely adopted discrete-event network simulator for research in communication systems. In ns-3, network scenarios are implemented as C++ programs that are compiled and executed within the simulation environment. Because of its open-source, modular design, ns-3 can be extended with additional modules that add new protocol stacks, network models, and advanced functionality for simulating complex communication systems.

One notable extension of ns-3 is ns3-oran [13], developed by the U.S. National Institute of Standards and Technology (NIST). This extension introduces key elements of the O-RAN architecture into the simulator, with a particular focus on the Near-Real-Time RAN Intelligent Controller (Near-RT RIC). Through this extension, the Near-RT RIC can be instantiated within ns-3 and interact with simulated RAN components through an implementation of the E2 interface. In the O-RAN architecture, the E2 interface enables communication between the Near-RT RIC and E2 nodes, which correspond to RAN entities connected to the controller. Within ns3-oran, the E2 interface and associated RAN entities are simulated, enabling the controller to receive telemetry and performance reports from the network and issue control actions based on analysis of the collected data.

The ns3-oran framework therefore enables the instantiation of a simulated Near-RT RIC that interacts with other RAN components through its E2 interface implementation. In addition, the framework includes a reporting and database subsystem responsible for collecting and storing performance reports generated by the simulated E2 nodes. To enable experimentation with RAN control algorithms, ns3-oran also provides a framework for implementing xApps, referred to in the simulator as logic modules. This framework allows developers to design and evaluate Near-RT RIC applications in a controlled and fully simulated environment.

It is important to note that in the O-RAN architecture, UEs are not E2 nodes. Instead, UEs send reports first to the eNB or gNB, which then parses and forwards them to the RIC via the E2 interface. However, in the following sections, an abstraction originally implemented in the ns3-oran module is adopted, in which UEs are modeled with logical E2 terminators (via the `OranE2NodeTerminatorLteUe` class) that report directly to the RIC. The figures depicted in Sects. 5 and 6 reflect this abstraction by treating UEs as E2 nodes. This abstraction avoids the complexity of modifying the core ns-3 LTE module classes.

In addition to logic modules (xApps), the database, and the E2 interface, ns3-oran also simulates the conflict mitigation module and query triggers in the Near-RT RIC. The conflict mitigation component is responsible for handling concurrent commands generated by the logic modules. The simulator currently implements only two types of logic for the conflict mitigation component: the first does nothing, allowing all commands to execute; the other ensures that only one of the received commands acts on a given network node. Query triggers can be set up to initialize logic modules whenever a specific report arrives in the Near-RT RIC.

5 The proposed ns3-oran-customizable-db simulation tool

Originally, ns3-oran implements only a few standard report types. All information storage and retrieval methods are specific to these report types and are implemented in the classes related to the information repository. The simulator is not prepared to receive other report types, as it lacks methods generic enough to store and retrieve information beyond what is already implemented. This limits the simulator, since any

application requiring metrics beyond the few already provided becomes infeasible.

This work enhances the ns3-oran tool to support user-defined reports with any type of performance metric.

5.1 ns3-oran-customizable-db implementation

The proposed tool introduces two new core components: `OranAdaptiveSqlite` and `OranReportSqlite`. Together, these components enable the definition of custom report formats and allow the Near-RT RIC to retrieve information from the database. To integrate this reporting system, modifications were required in several parts of the simulator, most notably in the E2 interface and in the base database class.

The added `OranAdaptiveSqlite` class implements a new database subsystem based on SQLite. Unlike the default ns3-oran implementation, where all report tables are predefined and created at the start of the simulation, this component dynamically generates new tables at runtime based on metadata from previously unseen custom reports. As a result, in addition to the standard tables, the new database implementation can evolve adaptively as new report types are introduced. The proposed database can also create new tables automatically during the simulation based on metadata from customized reports. Figure 2 shows the UML diagram of the `OranAdaptiveSqlite` class.

The proposed `OranAdaptiveSqlite` class also provides a new set of methods for storing and retrieving simulation data. In Fig. 2, methods marked with "+" are public and define the interfaces for accessing the database. For instance, the first overloaded version of the `GetLastReport` method receives a string containing the name of a particular table. In contrast, the second version receives a string and a number, containing the table name and the node ID. The first version returns a map containing the most recent report inserted into the specified table, where the keys correspond to the SQLite table columns and the values correspond to the information in those columns. On the other hand, the second version of the method returns the most recent information sent by a specific node. Similarly, `GetSecondLastReport` retrieves the previous entry, enabling comparison between consecutive reports of the same type. This method also receives a table name and a node ID. The `GetCustomQuery` method allows the execution of arbitrary SQL queries, giving developers flexibility to extract information beyond predefined interfaces. Note that the user can submit invalid queries (e.g., by attempting to fetch data from a non-existent table) when using the `GetCustomQuery` method. This would result in a simulation abort with an error message detailing the failed query. It must also be noted that, because the query is entirely custom, misusing this method can return arbitrarily large results, degrading performance and potentially causing security breaches. It is assumed that the user is not using the simulator in a production database, but rather in a safe test environment, as the method `GetCustomQuery` does not have any checks or restrictions regarding the input query, sending it directly to the database without any safeguards; this can result in writes or deletion of data if misused. The results of these methods are returned in structured formats, enabling direct use in simulation scripts or external ML pipelines.

Internally, `OranAdaptiveSqlite` includes several protected methods for managing report parsing and database updates. These methods start with a "#" in Fig. 2 and are related to internal database management. For example, the `ParseReportTableInfo` method creates new tables in the database. This method is called when a totally new custom report arrives at the Near-RT RIC. The method then processes information about the table type to be created for the incoming report and generates the query for the corresponding table creation. The `CreateReportTable` method executes the corresponding SQL command received from the `ParseReportTableInfo` method. The `CreateReportTable` method then creates the table for the new custom report. The `ParseReport` method is called whenever a report arrives at the Near-RT RIC and processes its content to structure it according to the information database. The `CreateReportSaveQuery` method uses the `ParseReport` to create the query string for adding information to a table when the report arrives. Finally, the

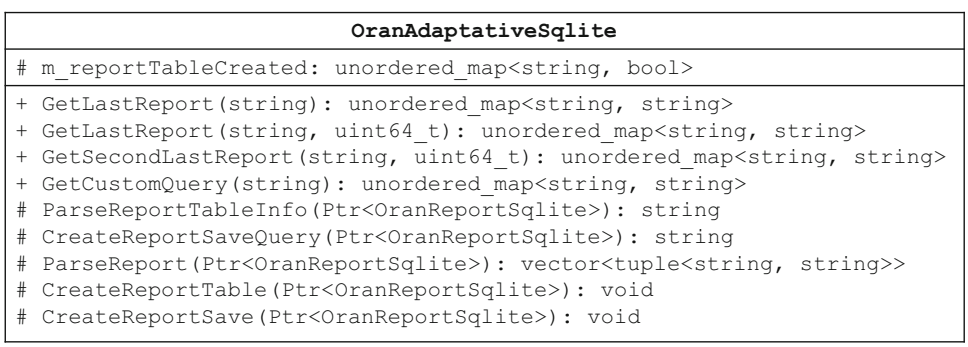


Fig. 2 The UML diagram of the implemented `OranAdaptiveSqlite` class

`CreateReportSave` method sends the *query* created by `CreateReportSaveQuery` to the database and checks whether the information was added successfully. This layered design ensures both modularity and extensibility.

The second component, `OranReportSqlite`, is implemented as a virtual class to facilitate user-defined custom reports. To extend it, developers must overload three key methods:

- `GetTableName`, which specifies the destination table name. This method must be overloaded so that it returns a string containing the name of the table. This method is used by the `OranAdaptiveSqlite` object to store the data from a customized report;
- `GetTableInfo`, which defines the schema as a vector of tuples describing column names and data types. This method must return a vector of tuples containing the names of the columns of the table where the data will be stored and the SQL type of data to be stored (e.g., `CHAR`, `INT`, `BOOL`); and
- `ToString`, which structures the report data for storage in the database. This method must return the information to be stored by the report in a structured fashion.

By overloading these three methods, users can seamlessly integrate arbitrary metrics and data structures into the simulation environment. More information about the implementation and use of these methods can be found in the documentation provided with the tool. Figure 3 shows the UML diagram of the `OranReportSqlite` class.

Figure 4 highlights the architectural differences between the baseline ns3-oran implementation and the proposed ns3-oran-customizable-db extension. In the original version, only predefined report types are supported, and tables are created statically at the start of the simulation, preventing integration of new data sources. In contrast, the proposed simulation tool supports the reception of custom reports and the dynamic creation of new tables during runtime. In the figure, custom reports and dynamically generated tables are depicted in red, illustrating the extended flexibility of the modified Near-RT RIC.

The source code of the proposed ns3-oran-customizable-db is available at https://github.com/felip-T/ns3-oran. The repository also provides (i) HTML documentation generated with Doxygen describing classes and methods, (ii) Sphinx documentation with detailed implementation notes and examples (available at https://github.com/felip-T/ns3-oran/tree/master/doc), (iii) a set of use-case examples demonstrating the tool's capabilities (available at https://github.com/felip-T/ns3-oran/tree/master/examples/examples-Adaptive), and (iv) a Dockerfile to reproduce the ns-3 installation, the extension, and all required dependencies.

5.2 Benchmarks

We run benchmarking experiments to measure the performance impact of the proposed database system. We divide the experiments into two batches. The first measures the impact of growth in simulation size, accounting for the number of simulated UEs, eNBs, and report types. In contrast, the latter measures the resource footprint of the new database system and compares it with that of the original system.

For the first batch, a scenario featuring 100 UEs and 10 eNBs is simulated. During each simulation run, each UE generates 10 reports (for a total of 1000 reports per run), and the number of distinct custom report types is configured via a CLI argument. The experiments are repeated five times to calculate confidence intervals. Figure 5a and b illustrate how the database file size on disk and the simulation memory usage scale with the number of custom report types. Figure 5a shows that the database scales linearly with the number of custom reports, with a small footprint ranging from 180 KB with 2 custom report types up to 8576 KB with a thousand different custom report types. The simulation heap memory increases slightly by about 2 MB (6%) with a thousand different report types.

In the second batch of experiments, a scenario with varying numbers of UEs (10 to 500) and eNBs (1 to 25) is simulated. In the default database system experiment, each UE sends two default reports that include location, packet loss, and cell information. To compare the adaptive database with the default system, custom reports containing the same data as the default ones are generated. Each UE transmits two reports of each type during the simulation. Figure 6a and b illustrate database disk storage and simulation heap memory as the number of UEs and eNBs scales. The difference in

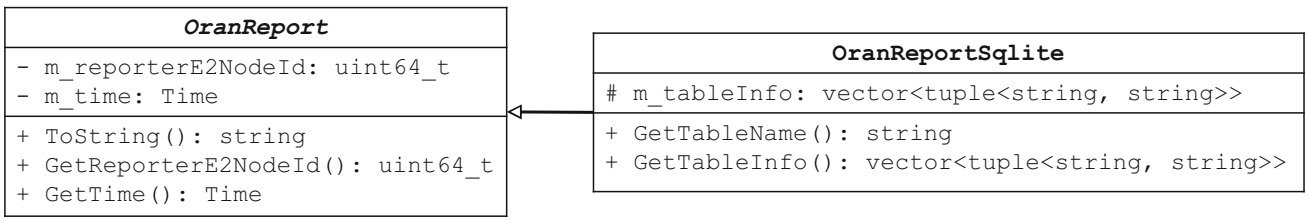


Fig. 3 The UML diagram of the implemented `OranReportSqlite` class

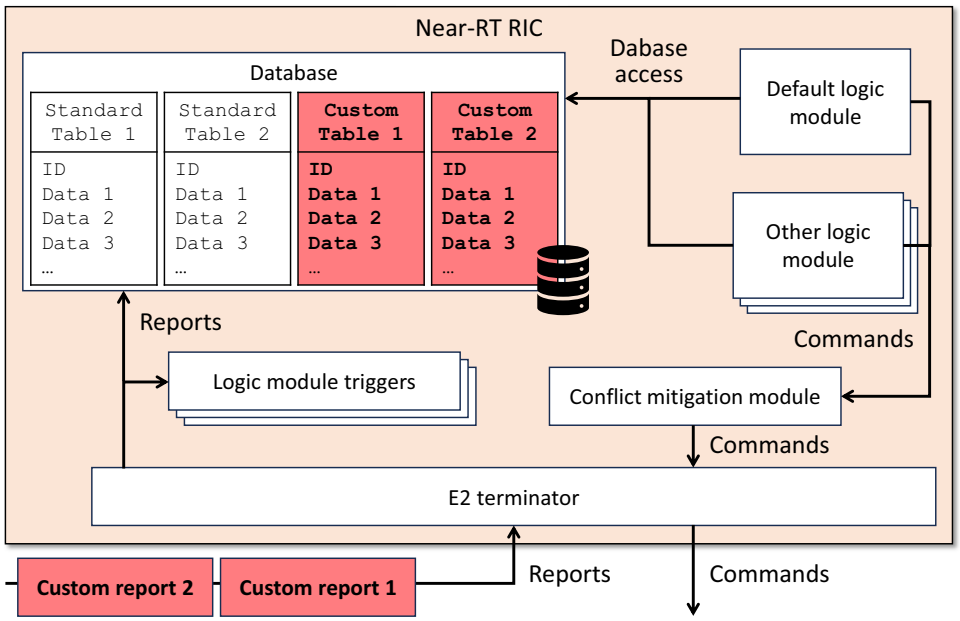


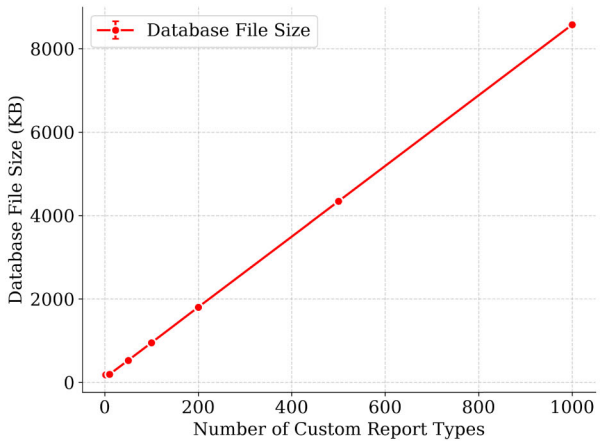
Fig. 4 Modified Near-RT RIC. Custom reports and dynamically generated tables are shown in red

database file sizes is due to the adaptive database inheriting the default database's settings for compatibility. Because all default table schemas are created before the simulation starts, the adaptive cases contain both the default and custom table schemas. Figure 6c plots the time to query the last position report for all UEs in both databases, showing similar scaling curves. Figure 6d shows the insertion time, which remained constant within the margin of error for both database systems during the simulation. Figure 6e exhibits the total end-to-end time taken per simulation run, with no perceptible difference between the standard simulator and the adaptive database within the margin of error.

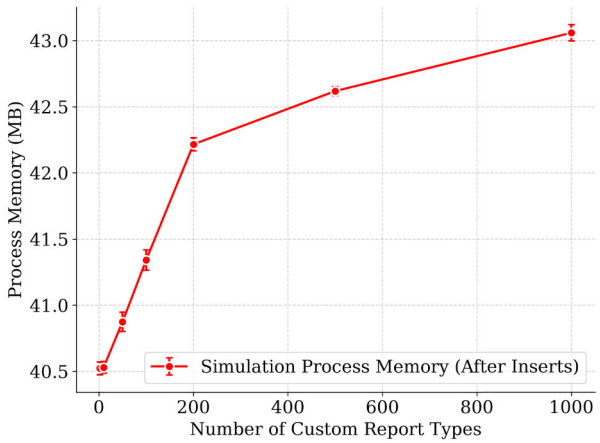
5.3 Limitations

Currently, the new database system lacks a high-level configuration helper for creating custom reports. Although the database repository itself can be configured using the default `OranHelper` provided in ns3-oran, creating and configuring custom report types in the simulation script requires implementing C++ classes manually, as discussed in Sects. 5.1 and 6.1.

Another limitation is the reliance on a local SQLite3 file for database storage. In a real-world Near-RT RIC, the database is typically implemented as a Shared Data Layer

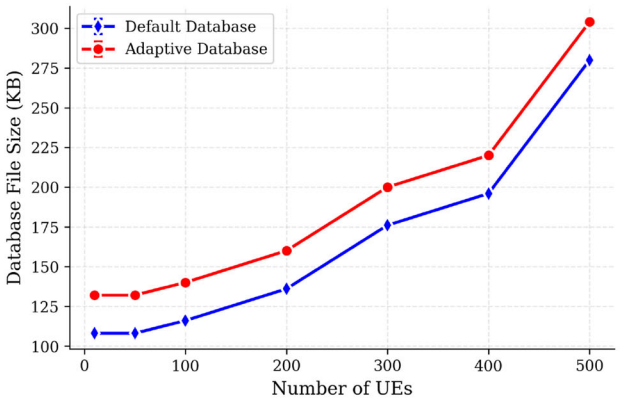


(a) Database file size on disk scales with the number of different custom reports.

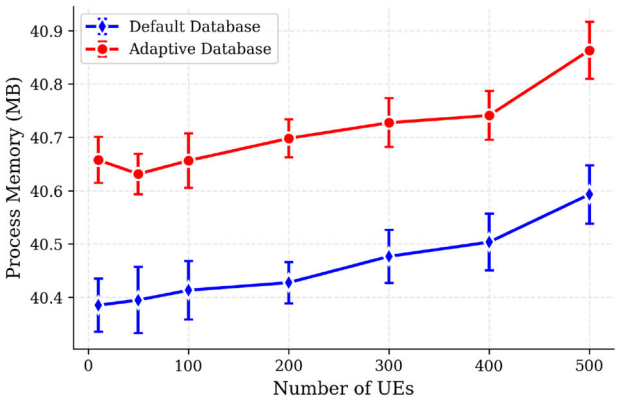


(b) Simulation process memory (heap) scales with the number of different custom reports.

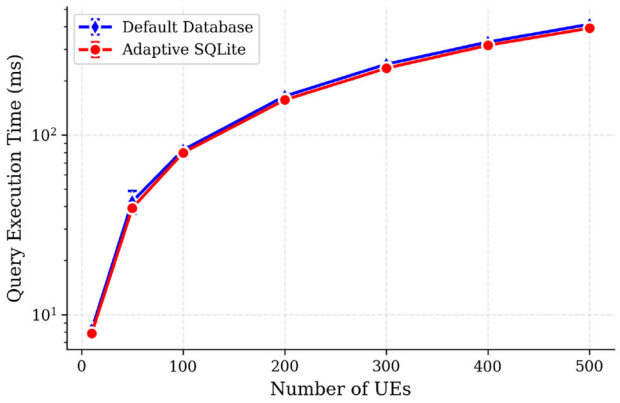
Fig. 5 Disk usage of the database (**a**) and process memory (**b**) scaling with different numbers of custom report types



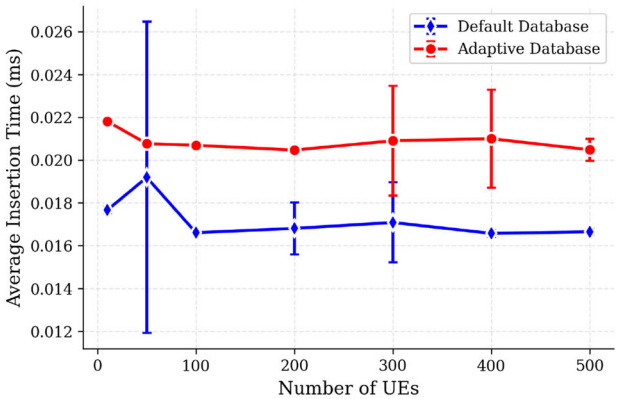
(a) Database file size on disk scaling with the number of UEs.



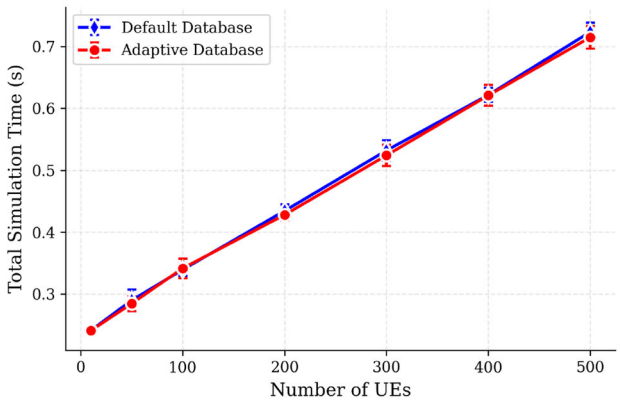
(b) Simulation process memory (heap) scaling with the number of UEs.



(c) Time for querying the last position of all UEs.



(d) Average time for inserting reports.



(e) Total simulation time.

Fig. 6 Comparison between the default database system (in blue) and the adaptive database (in red) in terms of disk file size (**a**), heap memory (**b**), query time (**c**), insertion time (**d**), and total simulation time (**e**)

(SDL) that scales and supports concurrent writes. In contrast, using a local SQLite3 file means concurrent writes are blocked. Consequently, factors such as disk I/O and storage capacity can degrade performance, posing challenges for highly concurrent simulations or scenarios in which multiple simulation scripts access the same database file.

6 Use cases

To demonstrate the operation and usefulness of the tools added to the simulator, three use cases were executed. The first is a simple example demonstrating the new data storage model alongside the standard reports that already existed. The second example demonstrates the utility of the new reporting mechanism by initiating a handover command whenever the Near-RT RIC receives a custom SINR (Signal to Interference plus Noise Ratio) report. The third example illustrates how the implemented tools can be used to create a training scenario for a reinforcement learning algorithm.

6.1 Case 1: use of custom reports

This example aims to illustrate the use of the new data storage system together with the previously existing reports. It is a simple example that simulates a Near-RT RIC with a single UE and one eNB.

The example consists of five files: `oran-report-ue-ipv4.cc`, `oran-report-ue-ipv4.h`, `oran-reporter-ue-ipv4.cc`, `oran-reporter-ue-ipv4.h`, and `simple-db-usage.cc`.

The files `oran-report-ue-ipv4.cc` and `oran-report-ue-ipv4.h` implement a custom report that stores the current IPv4 address of a user equipment. This report defines the IPv4 address as a string in the database through the `GetTableInfo` method. The report also defines the storage table name as `UeIpv4` in the `GetTableName` method. The files `oran-reporter-ue-ipv4.cc` and `oran-reporter-ue-ipv4.h` implement the reporter class logic. The reporter's purpose is to generate a report. This class is added to the user equipment and is used whenever the device prepares a report to be sent to the Near-RT RIC. Its function is to create a report of type `OranReportUeIpv4` and write data into it.

Listing 1 shows how the IPv4 custom report is implemented using the ns3-oran-customizable-db extension. In the constructor, the field "`ipv4`" is added to the `m_tableInfo` attribute, defining its corresponding SQLite type. The `GetTableInfo` method returns the schema stored in `m_tableInfo`, enabling the database to construct the appropriate table. The `ToString` method returns the data to be stored in the SQLite table in a formatted fashion. The Near-RT RIC uses this method to retrieve data from the received report. The `GetTableName` method returns the table name in which the report data is stored in the Near-RT RIC SQLite database. Together, these methods implement the interface required for handling custom report types.

Finally, the `simple-db-usage.cc` file is the main simulation script and simulates an LTE network with one eNB and one user equipment. A Near-RT RIC is also created, and both the UE and the eNB are configured. The eNB is assigned a fixed position, while the UE uses a constant-velocity mobility model. The UE is also equipped with a location reporter (which generates reports already implemented by default in ns-oran) and an IPv4 reporter (which generates custom reports as described previously). The simulation lasts for 10 s, and reports are periodically sent from the UE to the Near-RT RIC every second. During simulation execution, all SQLite database queries are printed to the terminal.

Figure 7 illustrates the example. The figure shows an eNB, a UE, and the Near-RT RIC. The UE is connected to the eNB, and both the UE and the eNB are connected to the Near-RT RIC via the E2 interface. Two reporters are installed on the UE and generate periodic location and IPv4 address reports. These reports are sent from the UE to the Near-RT RIC through the E2 interface. The Near-RT RIC contains a database with two tables. The `OranReportLocation` table, represented with a solid line, corresponds to the location reports implemented by default in ns3-oran and is initialized at the beginning of the simulation. The `OranReportUeIpv4` table, represented with a dashed line, corresponds to the custom report. This table, as well as all custom report tables, is created dynamically during the simulation when the first report of this type arrives at the Near-RT RIC.

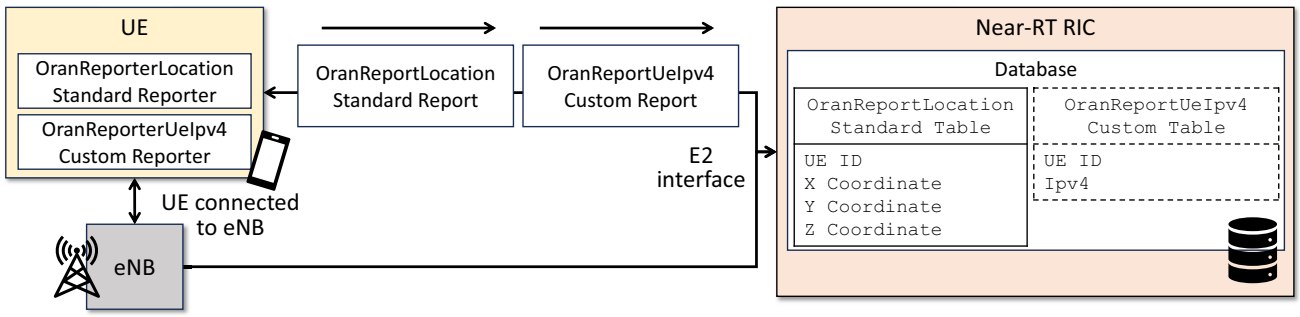


Fig. 7 Scenario simulated by the `simple-db-usage` example

```
OranReportUeIpv4::OranReportUeIpv4()
    : OranReportSqlite()
{
    m_tableInfo.emplace_back("ipv4",
    "TEXT");
}

std::string OranReportUeIpv4::
    ToString() const
{
    std::stringstream ss;
    Time time = GetTime();

    ss << "OranReportIpv4(" << "
    nodeid=" << GetReporterE2NodeId()
    << ";time=" << time.GetTimeStep()
    << ";ipv4=" << "\"" << m_ipv4 << "
    \"" << ")";

    return ss.str();
}

std::vector<std::tuple<std::string,
    std::string>> OranReportUeIpv4::
    GetTableInfo()
{
    return m_tableInfo;
}

std::string OranReportUeIpv4::
    GetTableName()
{
    return "UeIpv4";
}
```

Listing 1 Custom report class for IPv4.

6.2 Case 2: threshold handover with custom SINR report

The second example shows how custom reports can be used to control the nodes in the RAN. This example consists of two eNBs initialized 1500 m apart. A single UE is initialized near one eNB and moves towards the other at a constant speed of 50 m/s. This UE periodically sends the standard location report and a custom SINR report for storage in the Near-RT RIC. There is a custom logic module running in the Near-RT RIC that triggers a handover command whenever a SINR report below a threshold is received. To prevent the handover command from being triggered multiple times before the handover is finalized, a 2-second timeout is set between consecutive handover commands. Figure 8 illustrates the simulated scenario, including the custom SINR report and the custom handover trigger logic module.

The files `oran-report-sinr-ue.cc` and `oran-report-sinr-ue.h` implement a custom report that stores the SINR and RSRP values. This report defines a new storage table "sinr" in the `OranAdaptiveSqlite` database via the `GetTableName` method. The SINR and RSRP (Reference Signal Received Power) values are defined as `real` and `integer`, respectively, in the database via the `GetTableInfo` method. The files `oran-reporter-sinr-ue.cc` and `oran-reporter-sinr-ue.h` are responsible for implementing the reporter class logic. This reporter is installed on the UE and is responsible for capturing the current SINR and RSRP values from the PHY layer. It generates an `OranReportSinrUeSql` report to be sent to the Near-RT RIC. The files `oran-lm-sinr-handover.cc` and `oran-lm-sinr-handover.h` implement a custom logic module that runs on the Near-RT RIC. This module queries the database for the latest SINR reports. If the UE's SINR falls below a threshold (default: 30 dB), the LM triggers a handover command to the closest eNB. There is a cooldown of 2 s after a handover is triggered, preventing multiple handover commands from being issued for the same UE before the UE can perform the handover and send a new report. Finally, the `main.cc` file is the simulation script. It sets up the LTE network with two eNBs and one UE. It also initializes the Near-RT RIC with the custom SINR logic module and the Adaptive SQLite data repository. The UE is equipped with three reporters: a default location reporter, a default cell info reporter, and the custom SINR reporter. The simulation lasts for 30 s. During execution, the UE moves towards the second eNB, eventually triggering a handover when the signal from the first eNB falls below the defined threshold.

Figure 9 illustrates the example. It shows the reported UE position (green) and SINR (blue). The RIC's logic module monitors these reports and issues a handover command when the SINR drops below the 30 dB threshold. The yellow line marks when the handover is triggered. Shortly after, the blue line jumps to a higher value and continues to rise, indicating that the UE has successfully completed the handover and connected to the target eNB it is approaching.

6.3 Case 3: reinforcement learning-based handover

The third example provided with the proposed tool demonstrates how the implemented database model can be used to train a reinforcement learning model. It is important to note that this use case is intended as a proof-of-concept integration demonstration. The objective is not to propose a state-of-the-art handover algorithm or benchmark it against established heuristics, but to illustrate how the proposed database system can be used together with another plugin to train a reinforcement learning model. This example implements a logical module that trains a reinforcement learning model using *Proximal Policy Optimization* (PPO)[26] to perform handover decisions to minimize packet loss. For demonstration purposes, this example implements handover management in a simulated environment with one UE and two eNBs.

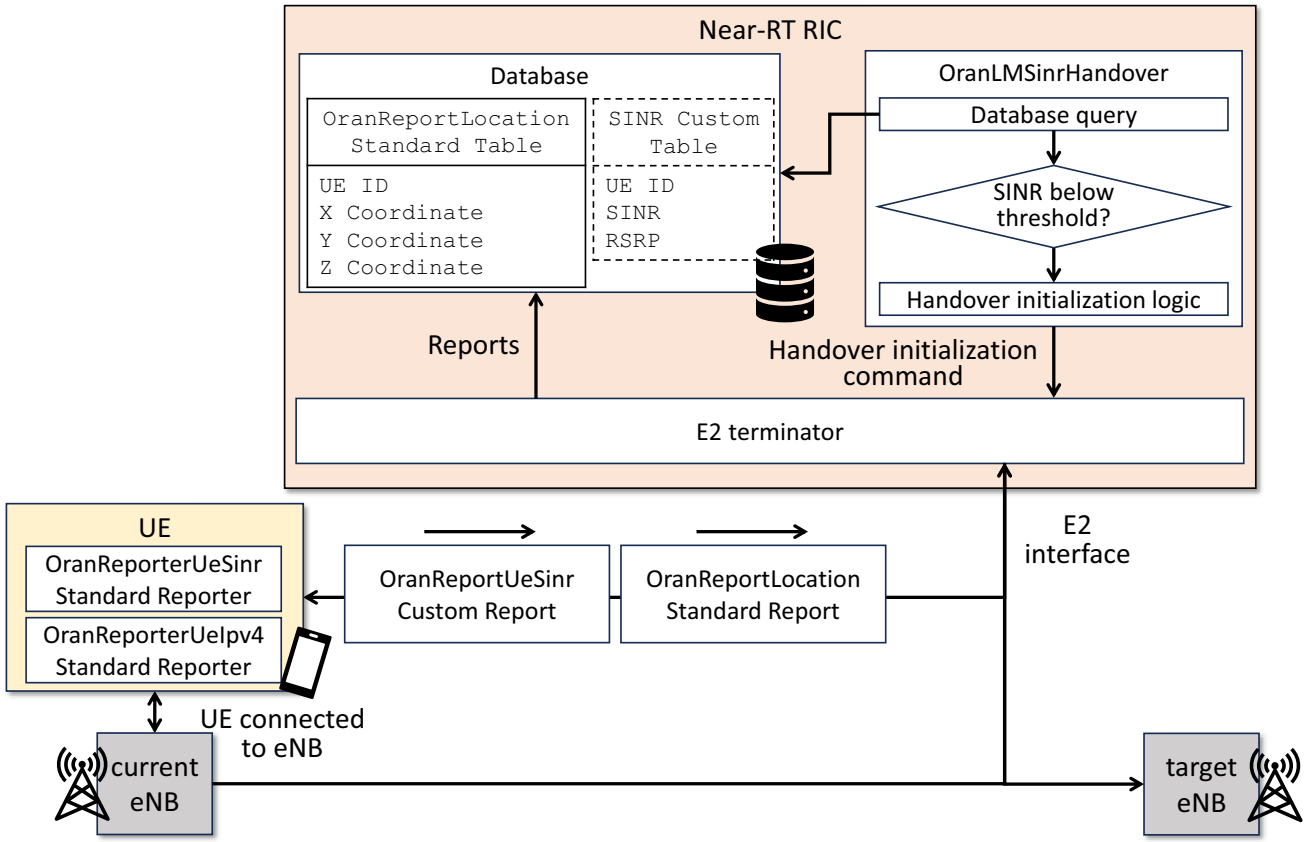


Fig. 8 Scenario simulated by the `sinr-handover` example

Additionally, this example uses the ns3-ai module [27], which enables communication between ns-3 simulation scripts and an external Python process. With ns3-ai, it is possible to define a *Gymnasium* [28] environment in C++ within ns-3 simulation scripts. Once defined, a Python script can access the environment, allowing the ns-3 simulation to act as a reinforcement learning training environment.

The eNBs have fixed positions across all simulation instances, whereas the UE initializes at a random position with a random constant speed along the X-axis. The UE speed depends on the initial position, so it always moves

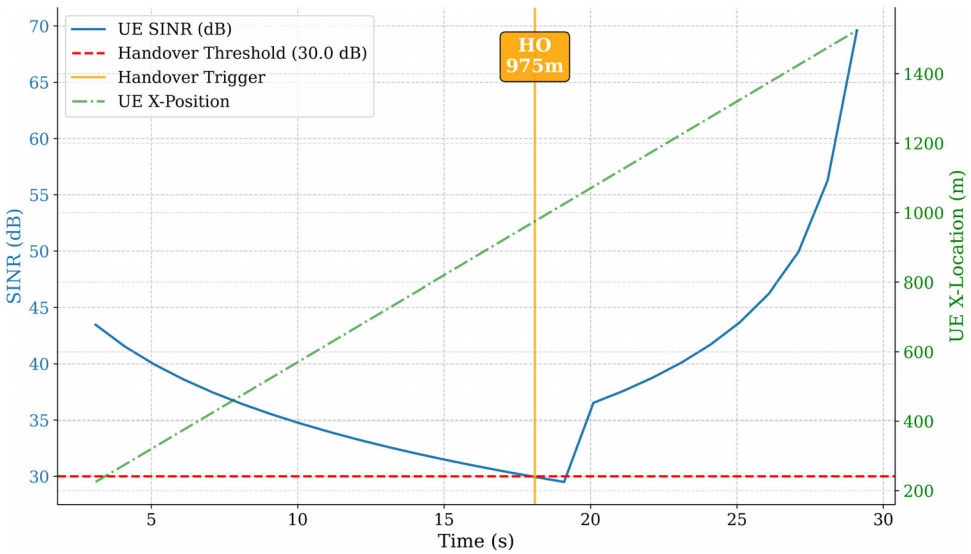


Fig. 9 Figure with UE position and corresponding measured SINR for Case 2 example. After SINR reaches the threshold, a handover is triggered

toward the farthest eNB. This configuration is designed to trigger handover events frequently. Figure 10 illustrates the UE and eNBs initialization.

The UE periodically generates reports containing its spatial position, the packet loss rate observed since the last report, and its SINR with the currently connected eNB. Whenever a packet loss report from a UE is received by the Near-RT RIC, the main logical module responsible for training the machine learning model for handover management generates an observation containing information about packet loss rate, SINR, and the distance between the UE and each eNB. The model processes this information and returns the index of the eNB to which the UE should connect. If the selected eNB is the same as the current one, no action is taken; otherwise, the system initiates a handover procedure. The number of eNBs and UEs can be easily modified through a simulation script parameter. The model aims to minimize the aggregate packet loss rate across all UEs.

The simulation is divided into ten files. The files `oran-report-sinr-ue.cc` and `oran-report-sinr-ue.h` implement the `OranReportSinrUeSql` report, which contains SINR information for user equipment. This report defines a SQLite database table containing a user equipment identifier (`ueId`), an identifier of the connected eNB (`cellId`), SINR and RSRP measurements, and the radio network temporary identifier (RNTI). The files `oran-reporter-sinr-ue.cc` and `oran-reporter-sinr-ue.h` implement a report generator applied to UEs. This generator captures physical-layer trace information from ns-3 and generates `OranReportSinrUeSql` reports with the required information. The files `open-gym.h` and `open-gym.cc` define the *Gymnasium* environment used for training. This environment implements the interfaces used for communication between the Python-based training script and the C++ simulation scenario. These files provide methods that can be accessed by the Python script to observe the simulation state and perform control actions. The provided methods include obtaining the observation and action spaces, retrieving observations, computing rewards, and sending actions.

The files `oran-logic-module-train-ml-handover.cc` and `oran-logic-module-train-ml-handover.h` implement the logical module responsible for training the handover model. This module is executed whenever a packet-loss report indicating more than 1% loss is received, or every second periodically. Its role is to generate observations to be sent to the Python training script via the interface defined in `open-gym.h`. The logical module also receives actions from the neural network and initiates the handover procedure when required. For each connected UE, observations are generated based on the most recent reports stored in the database. The reward r sent to the model is computed as $r = 1-p$ when only one UE is present, or $\sum_{i=1}^{n} \log(1-p_i + 0.01)$ when multiple UEs are present, where p is the packet loss rate of a UE and n is the number of UEs. Upon receiving an action, if the indicated eNB differs from the currently connected one, the handover procedure is initiated. The files `oran-report-trigger-apploss.h` and `oran-report-trigger-apploss.cc` define triggers for sending packet loss reports from the UE. Whenever the packet loss rate exceeds 10%, a report is sent to the Near-RT RIC. After sending such a report, the trigger is activated again only after 20 ms if packet loss remains above 10%.

Finally, the `run-handover-example.py` file defines the neural network, its training algorithm, and initializes both training and simulation. The `stable-baselines3` [29] library is used to implement the Proximal Policy Optimization algorithm. This file creates a simulation environment by initializing ns-3. Once the environment is created, the network trains for a fixed period of time, updates its parameters, and restarts the simulation. Training is performed in an infinite loop and is interrupted by the user when deemed sufficient. The model is periodically saved during training,

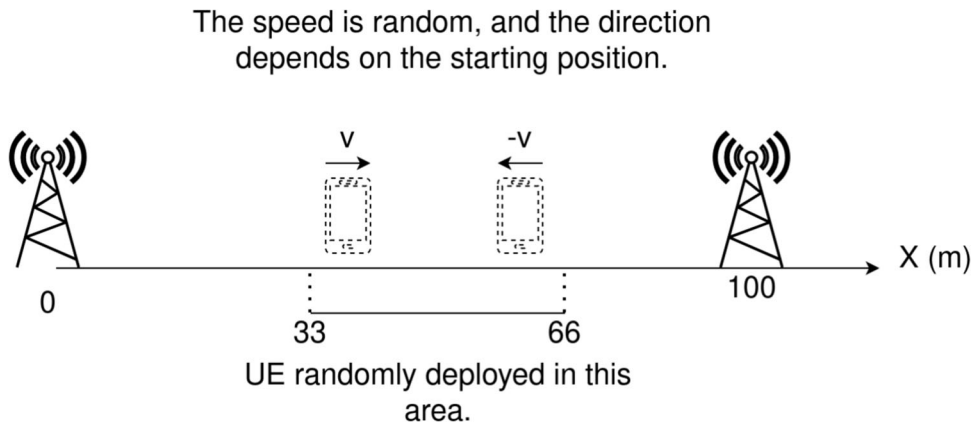


Fig. 10 Initialization of the UE and eNBs

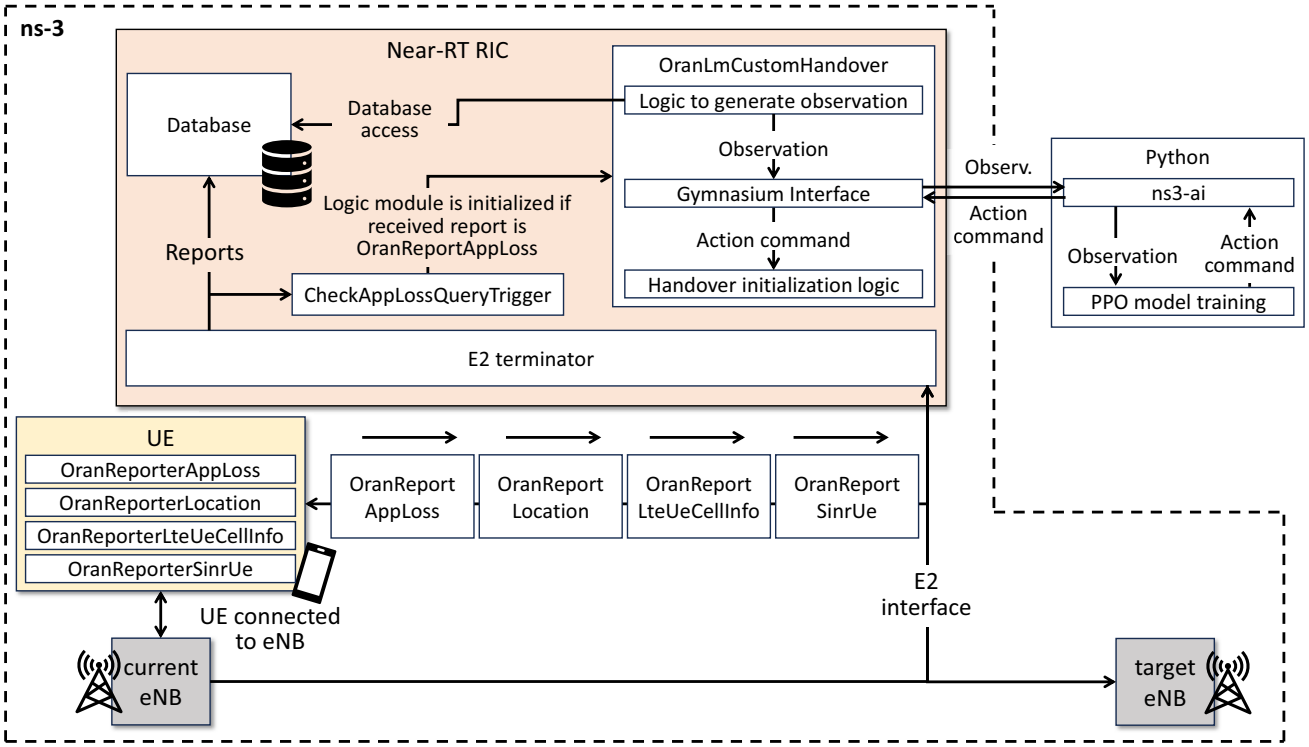


Fig. 11 Diagram illustrating the entities and communications in Example 2

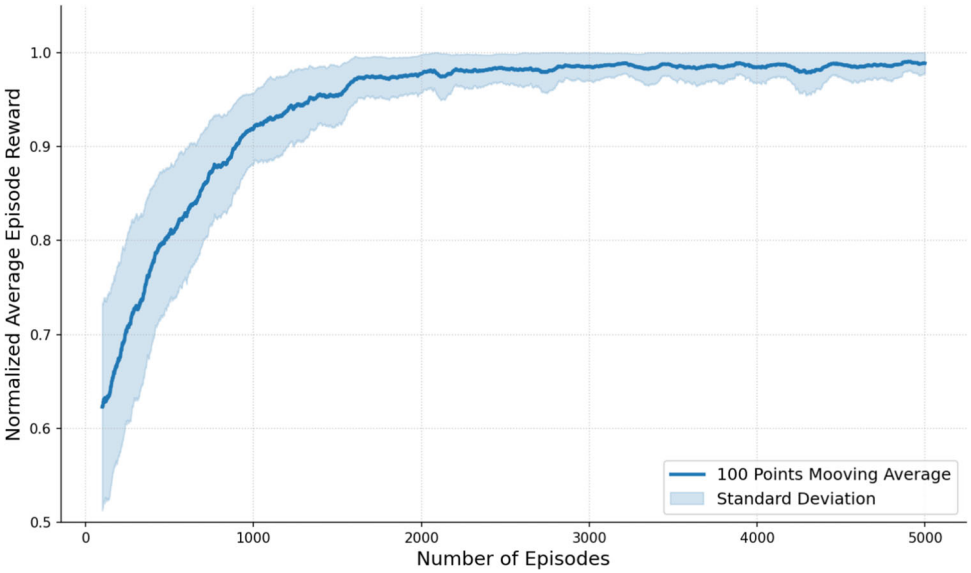


Fig. 12 Evolution of the model during training. The reward corresponds to the average packet loss rate in each simulation episode. The model was trained three times to compute the standard deviation

Table 1 PPO handover model hyperparameters

Hyperparameter	Value
Policy network	MLP with 2 hidden layers (64 units each)
Learning rate	0.0003
Rollout steps	2048
Mini-batch size	64
Optimization epochs	10
Discount factor	0.99
Entropy coefficient	0.01

and performance metrics are printed to the terminal at each iteration.

All mentioned files and a simplified script to run the simulation are openly available with the source code in the provided repository.

Figure 11 illustrates the entities and data flow in the example. In the figure, the UE periodically generates reports that are sent to the Near-RT RIC via the E2 interface. The Near-RT RIC stores all reports in the database. When a packet loss report is received, the logical module `OranLmCusstomHandover` is executed. This module accesses the database and loads the most recent packet-loss, location, connected cell, and SINR reports for the UE that triggered the module. During execution, the module sends observations containing UE data to the model being trained in a Python process via the ns3-ai module. The model generates an action command and sends it back to the logical module. If the action indicates that a different eNB is more suitable, the handover procedure is initiated.

Figure 12 shows the average reward obtained per episode during model training in the single-UE scenario. The line represents a 100-point moving average of the reward, calculated as one minus the average packet loss per episode. This figure shows how the reinforcement learning model evolves over time to minimize the packet loss ratio in a simulated, simplified scenario with one UE and two eNBs, demonstrating that the database system can, in fact, be used to store custom reports and provide data for training ML models. Table 1 lists the used hyperparameters.

7 Conclusion and future work

The developed tool enhances the design of data-driven applications for control and diagnosis in the access network. With the introduced extension, the simulator's database functionality has been improved, favoring an environment conducive to training machine learning models. The implemented database components provide users with the flexibility to collect, store, and query arbitrary metrics, enabling the construction of more complex and realistic simulation scenarios. These capabilities are particularly valuable for developing and validating ML models directly from ns-3 simulations, without extensive preprocessing or external instrumentation.

Furthermore, ongoing work extends the tool to support integration with the 5 G-LENA module [30], which implements 5 G NR functionalities in ns-3. This integration is currently available in a dedicated branch of the tool's GitHub repository and aims to broaden the framework's applicability to next-generation RAN scenarios.

Author contributions F. T., D. A., and I. M. wrote the main manuscript text, F. T. did the experiments, and F. T. and M. C. did the figures, and I. M. and M.C. reviewed the manuscript.

Funding The Article Processing Charge (APC) for the publication of this research was funded by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) (ROR identifier: 00x0ma614). This work was partially supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES – Finance Code 001 and 88887.954253/2024-00 and 88887.123038/2025-00), by CNPq (grants 310234/2025-5, 407304/2025-8, 408255/2023-4, 405940/2022-0, and 132858/2024-0), by FAPERJ (grants E-26/200.380 /2023 and E-26/210.778/2025), by FAPESP (grants 2023/00673-7 and 2023/00811-0), by RNP, and by the Research Development Foundation - Fundep - Rota 2030 together with our partners Stellantis and Mobway.

Data availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Liyanage M, Braeken A, Shahabuddin S, Ranaweera P (2023) Open RAN security: challenges and opportunities. J Netw Comput Appl 214
2. Polese M, Bonati L, D'Oro S, Basagni S, Melodia T (2023) Understanding O-RAN: architecture, interfaces, algorithms, security, and research challenges. IEEE Commun Surv Tutor 25(2):1376–1411
3. O-RAN Alliance (2025). https://www.o-ran.org/. Accessed 14 Feb 2025
4. Agarwal B, Irmer R, Lister D, Muntean G-M (2026) Open RAN for 6G networks: architecture, use cases and open issues. IEEE Commun Surv Tutor 28:2881–2924
5. Lacava A, Bonati L, Mohamadi N, Gangula R, Kaltenberger F, Johari P, D'Oro S, Cuomo F, Polese M, Melodia T (2025) dApps: enabling real-time AI-based Open RAN control. Comput Netw 269
6. de Oliveira NR, Medeiros DSV, Moraes IM, Andreonni M, Mattos DMF (2024) Towards intent-based management for Open Radio Access Networks: an agile framework for detecting service-level agreement conflicts. Ann Telecommun 79:693–706
7. Couto RS, Cruz P, Pacheco RG, Souza VMS, Campista MEM, Costa LHMK (2024) A survey of public datasets for O-RAN: fostering the development of machine learning models. Ann Telecommun 79:649–662
8. Hamdan MQ, Lee H, Triantafyllopoulou D, Borralho R, Kose A, Amiri E, Mulvey D, Yu W, Zitouni R, Pozza R, Hunt B, Bagheri H, Foh CH, Heliot F, Chen G, Xiao P, Wang N, Tafazolli R (2023) Recent advances in machine learning for network automation in the O-RAN. Sensors 23(21):8792

9. Soundrarajan R, Fiandrino C, Polese M, D'Oro S, Bonati L, Melodia T (2026) On AI verification in Open RAN. IEEE Commun Mag 64(3):39–44
10. Navarro J, Castro A, Rattaro C (2025) O-RAN testbeds: from experimental deployments to simulation frameworks. In: Wireless Days Conference (WD), pp 1–7
11. Luis Herrera J, Montebugnoli S, Scotece D, Foschini L, Bellavista P (2026) A tutorial on O-RAN deployment solutions for 5G: from simulation to emulated and real testbeds. IEEE Commun Surv Tutor 28:1709–1748
12. Táparo FG, Moraes IM, Campista MEM (2025) ns3-oran-customizable-db: a tool for building a customizable database for Open RAN simulations. In: 2025 13th Wireless Days Conference (WD), pp 1–8
13. ns3-oran (2025). https://github.com/usnistgov/ns3-oran. Accessed 14 Feb 2025
14. Lacava A, Polese M, Sivaraj R, Soundrarajan R, Bhati BS, Singh T, Zugno T, Cuomo F, Melodia T (2023) Programmable and customized intelligence for traffic steering in 5G networks using Open RAN architectures. IEEE Transactions on Mobile Computing, pp 1–16
15. ns-3 (2025). https://www.nsnam.org/. Accessed 14 Feb 2025
16. OSC (2024) The O-RAN Software Community (SC) Documentation. O-RAN Software Community, Tech. Rep. Available at https://docs.o-ran-sc.org/en/latest/index.html
17. Gawłowicz P, Zubow A (2019) ns-3 meets OpenAI Gym: the playground for machine learning in networking research. In: International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM), pp 113–120
18. Yin H, Liu P, Liu K, Cao L, Zhang L, Gao Y, Hei X (2020) ns3-ai: fostering artificial intelligence algorithms for networking research. In: Workshop on Ns-3 (WNS3), pp 57–64
19. Kaltenberger F, Melodia T, Ghauri I, Polese M, Knopp R, Nguyen TT, Velumani S, Villa D, Bonati L, Schmidt R, Arora S, Irazabal M, Nikaein N (2025) Driving innovation in 6G wireless technologies: the openairinterface approach. Comput Netw 269
20. Villa D, Khan I, Kaltenberger F, Hedberg N, da Silva RS, Maxenti S, Bonati L, Kelkar A, Dick C, Baena E, Jornet JM, Melodia T, Polese M, Koutsonikolas D (2025) X5G: an open, programmable, multi-vendor, end-to-end, private 5G O-RAN testbed with NVIDIA ARC and openairinterface. IEEE Transactions on Mobile Computing
21. Schmidt R, Irazabal M, Nikaein N (2021) FlexRIC: an SDK for next-generation SD-RANs. In: International Conference on Emerging Networking EXperiments and Technologies (CoNEXT), pp 411–425
22. Bonati L, Johari P, Polese M, D'Oro S, Mohanti S, Tehrani-Moayyed M, Villa D, Shrivastava S, Tassie C, Yoder K, Bagga A, Patel P, Petkov V, Seltser M, Restuccia F, Gosain A, Chowdhury KR, Basagni S, Melodia T (2021) Colosseum: large-scale wireless experimentation through hardware-in-the-loop network emulation. In: IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)
23. Polese M, Bonati L, D'Oro S, Johari P, Villa D, Velumani S, Gangula R, Tsampazi M, Robinson CP, Gemmi G, Lacava A, Maxenti S, Cheng H, Melodia T (2024) Colosseum: the open ran digital twin. arXiv:2404.17317
24. Bertizzolo L, Bonati L, Demirors E, Al-Shawabka A, D'Oro S, Restuccia F, Melodia T (2020) Arena: a 64-antenna SDR-based ceiling grid testing platform for sub-6 GHz 5G-and-beyond radio spectrum research. Comput Netw 181
25. Bonati L, Polese M, D'Oro S, Basagni S, Melodia T (2023) Open-RAN Gym: AI/ML development, data collection, and testing for O-RAN on PAWR platforms. Comput Netw 220:109502
26. Schulman J, Wolski F, Dhariwal P, Radford A, Klimov O (2017) Proximal policy optimization algorithms. [Online]. Available: arxiv:1707.06347
27. Yin H, Liu P, Liu K, Cao L, Zhang L, Gao Y, Hei X (202) Ns3-Ai: fostering artificial intelligence algorithms for networking research. In: Proceedings of the 2020 Workshop on Ns-3 (WNS), pp 57–64
28. Farama Foundation: Gymnasium: an API standard for reinforcement learning with a diverse collection of reference environments. [Online]. Available: https://gymnasium.farama.org/
29. Raffin A, Hill A, Gleave A, Kanervisto A, Ernestus M, Dormann N (2021) Stable-Baselines3: reliable reinforcement learning implementations. J Mach Learn Res 22:268:1–268:8
30. Patriciello N, Lagen S, Bojovic B, Giupponi L (2019) An E2E simulator for 5G NR networks. Simul Model Pract Theory 96:101933

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.