# ns3-oran-customizable-db: A Tool for Building a Customizable Database for Open RAN Simulations

Felipe G. Táparo[1], Igor M. Moraes[2], and Miguel Elias M. Campista[1]
Universidade Federal do Rio de Janeiro, GTA/DEL-Poli/PEE-COPPE
Rio de Janeiro, Brazil
[2]Universidade Federal Fluminense, Midiacom/IC
Niterói, Brazil
{felipe.taparo,miguel}@gta.ufrj.br, igor@ic.uff.br

***Abstract*—An essential approach for evaluating the performance of applications designed for control and diagnosis of Radio Access Networks (RANs) is simulation. Assessing proposed solutions in simulated environments that closely approximate real-world network behavior is crucial to mitigate potential risks during deployment. This work 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 the design and integration of machine learning-based solutions into the RAN. With these enhancements, the simulator can support the creation of scenarios tailored to reinforcement learning (RL) training, leveraging performance metrics that were previously inaccessible or non-configurable within 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.**

***Index Terms*—Radio Access Networks, O-RAN, Near-RT RIC, machine learning, simulation, ns3-oran**

## I. Introduction

O-RAN (Open RAN) is an open and disaggregated Radio Access Network (RAN) architecture standardized by the O-RAN Alliance [1]. It defines open and interoperable interfaces, thereby enabling multivendor deployments and decoupling network functions from proprietary hardware and software. One of the key innovations of O-RAN is the introduction of two intelligent controllers to facilitate 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. 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 framework promotes modularity by allowing rApps and xApps to be developed and deployed as 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 the integration of machine learning (ML)-based functionalities, allowing rApps and xApps to exploit data-driven methods for intelligent RAN control and management.

The evaluation of rApps and xApps is a critical step prior to real-world deployment. In particular, ML-based solutions require extensive testing and validation, and reinforcement learning (RL)-based applications additionally demand controlled environments for both training and evaluation. Network simulation thus plays a fundamental role, offering a reproducible and customizable testbed for assessing functionality, robustness, and performance before field trials. Nevertheless, the current O-RAN research ecosystem faces limitations in simulation support.

For simulating the Open-RAN architecture, two main open-source tools are available as extensions of the ns-3 [3] network simulator: ns-O-RAN [4] and ns-O-RAN [5]. The ns-O-RAN extension connects the simulation to an external, operational Near-RT RIC [6] via a simulated E2 interface, an approach that is powerful but carries high overhead in both setup and resources. Conversely, the ns3-oran extension eliminates this dependency by implementing a complete, self-contained simulated Near-RT RIC and E2 interface within the simulator. This simulated RIC supports functionalities like xApps, a conflict mitigation module, and a shared database, allowing other simulated entities to act as E2 Nodes that send reports and receive control commands.

While the ns-3 simulator and its ns3-oran extension represent valuable tools for O-RAN research, they suffer from restricted data reporting and storage capabilities. Specifically, ns3-oran provides built-in reporting modules limited to: (i) user equipment (UE) location, (ii) packet loss statistics, and (iii) registration information in evolved Node Bs (eNBs). Consequently, the reporting and database mechanisms are tightly coupled to these predefined data models, hindering the design of general-purpose scenarios. This constraint prevents the implementation of ML-driven rApps and xApps that rely on heterogeneous performance metrics beyond the default set.

This work was partially sponsored by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) – Funding Code 001 and 88887.954253/2024-00, CNPq (310234/2025-5, 408255/2023-4, and 405940/2022-0), FAPERJ (E-26/200.380/2023), FAPESP (2023/00673-7 and 2023/00811-0), Development and Research Foundation - Fundep - Rota 2030, and our partners Stellantis and Mobway.

This work proposes an extension to ns3-oran, called "ns3-oran-customizable-db", that introduces support for customizable performance reports to overcome the above-mentioned limitations. The proposed tool generalizes the reporting and database framework by extending existing APIs and adding new functions, enabling the collection, storage, and retrieval of arbitrary data types. Importantly, all modifications maintain backward compatibility with the baseline ns3-oran, ensuring that existing simulation scripts remain functional. The usefulness of the proposed tool is demonstrated through two use cases. The first showcases the capability of generating and retrieving arbitrary metrics within the simulation environment. The second illustrates how the extended framework can be employed to train an RL-based xApp, highlighting its potential for enabling AI-native O-RAN research.

This paper is organized as follows. Section II describes the operation of ns-3 and ns3-oran. Section II presents the architecture of the proposed tool, i.e., the architecture of the proposed ns3-oran-customizable-db simulation tool. Section IV details the use cases and experimental results. Finally, Section V concludes the paper and outlines directions for future work. The source code of the proposed extension 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-adaptative), and (iv) a Dockerfile for reproducible installation of ns-3, the extension, and all required dependencies.

## II. The ns-3 Simulator and the ns3-oran Extension

ns-3 [3] is a discrete-event network simulator widely used for research in communication systems. Simulations in ns-3 are developed through C++ scripts, which are compiled and executed within the simulator environment. Being open-source and highly extensible, ns-3 supports numerous extensions that enrich its baseline functionality with protocol stacks, models, and advanced network features.

One notable extension is ns3-oran [5], developed by the U.S. National Institute of Standards and Technology (NIST). This extension introduces core elements of the O-RAN architecture into ns-3, with a focus on the Near-Real-Time RAN Intelligent Controller (Near-RT RIC). The implementation allows the simulated Near-RT RIC to exchange information with User Equipment (UE) and other RAN components through a simulated E2 interface. In the O-RAN specification, the term E2 node refers to any RAN element connected to the Near-RT RIC via the E2 interface. Within ns3-oran, both UEs and the E2 interface are simulated, enabling the Near-RT RIC to receive performance reports from E2 nodes and to send control actions back based on the analysis of the collected information.

The ns3-oran extension thus enables the creation of a simulated Near-RT RIC that interconnects with other RAN components via its E2 interface implementation. Additionally, ns3-oran incorporates a reporting and database subsystem that collects and stores performance reports from the simulated E2 nodes. To support RAN control and management functions, ns3-oran also provides a framework for implementing xApps, referred to in the simulator as logic modules. This framework allows developers to design and test Near-RT RIC applications within a controlled and fully simulated environment.

## III. The Proposed ns3-oran-customizable-db Simulation Tool

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

The added `OranAdaptativeSqlite` 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 beginning of the simulation, this component dynamically generates new tables during runtime based on the metadata contained in previously unseen custom reports. Thus, in addition to the standard tables, the new database implementation can evolve adaptively as new report types are introduced. The proposed database is also able to create new tables automatically, during the simulation, based on metadata from customized reports. Figure 1 shows the UML diagram of the `OranAdaptativeSqlite` class.

The proposed `OranAdaptativeSqlite` class provides new methods for storing and retrieving simulation data. The methods starting with "+" in Figure 1 are public and implement 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, while the second version receives a string and a number, containing the name of the table and the node ID. On the one hand, the first version returns a map containing the information of the last report inserted in the searched table, in which the key corresponds to the columns of the SQLite table and the value corresponds to the information contained in these columns. On the other hand, the second version of the method returns the last information sent by a certain node. Similarly, `GetSecondLastReport` retrieves the previous entry, enabling comparison between consecutive reports of the same type. This method also receives the name of a table and a node ID. The `GetCustomQuery` method allows the execution of arbitrary SQL queries, giving developers flexibility to extract information beyond predefined interfaces. The results of these queries are returned in structured formats, facilitating direct use in simulation scripts or external ML pipelines.

Internally, `OranAdaptativeSqlite` includes several protected methods for managing report parsing and database

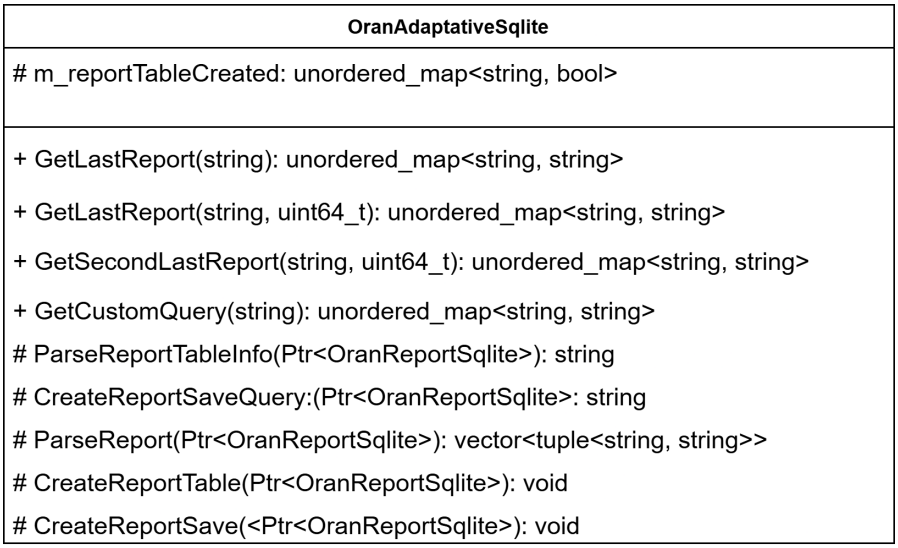


Fig. 1: The UML diagram of the implemented `OranAdaptativeSqlite` class.

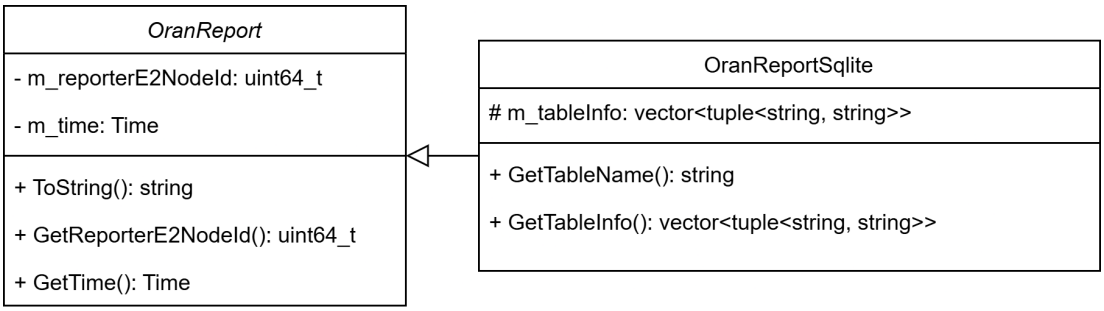


Fig. 2: The UML diagram of the implemented `OranReportSqlite` class.

updates. These methods start with a "#" in Figure 1 and are related to internal database management. For example, the `ParseReportTableInfo` method is used in the creation of new tables in the database. This method is called when a totally new custom report arrives at the Near-RT RIC. The method is then responsible for processing the information about the table type to be created for the incoming new report and thus to generate the query for the corresponding table creation. The `CreateReportTable` method executes the corresponding SQL command received from the `ParseReportTableInfo` method. The `CreateReportTable` method is then in charge of creating the table related to the new custom report. The `ParseReport` method is called whenever a report arrives at the Near-RT RIC and is responsible for processing its content to structure it according to the information database. The `CreateReportSaveQuery` method uses the `ParseReport` method to create the query string for adding information to a table upon the report's arrival. Finally, the `CreateReportSave` method sends the *query* created by the `CreateReportSaveQuery` method to the database and checks if the information was successfully added. 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 name of the destination table. This method must be overloaded so that it returns a string containing the name of the table. This method is used by the `OranAdaptativeSqlite` 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 name of the columns of the table where the data will be stored, and the type of data to be stored (e.g., `char`, `int`, `bool`); and
- `ToString`, which structures the report data to be stored in the database. This method must return the information to be stored by the report in a structured fashion.

By overloading these methods, users can seamlessly integrate arbitrary metrics and data structures into the simulation environment. More information about the implementation and

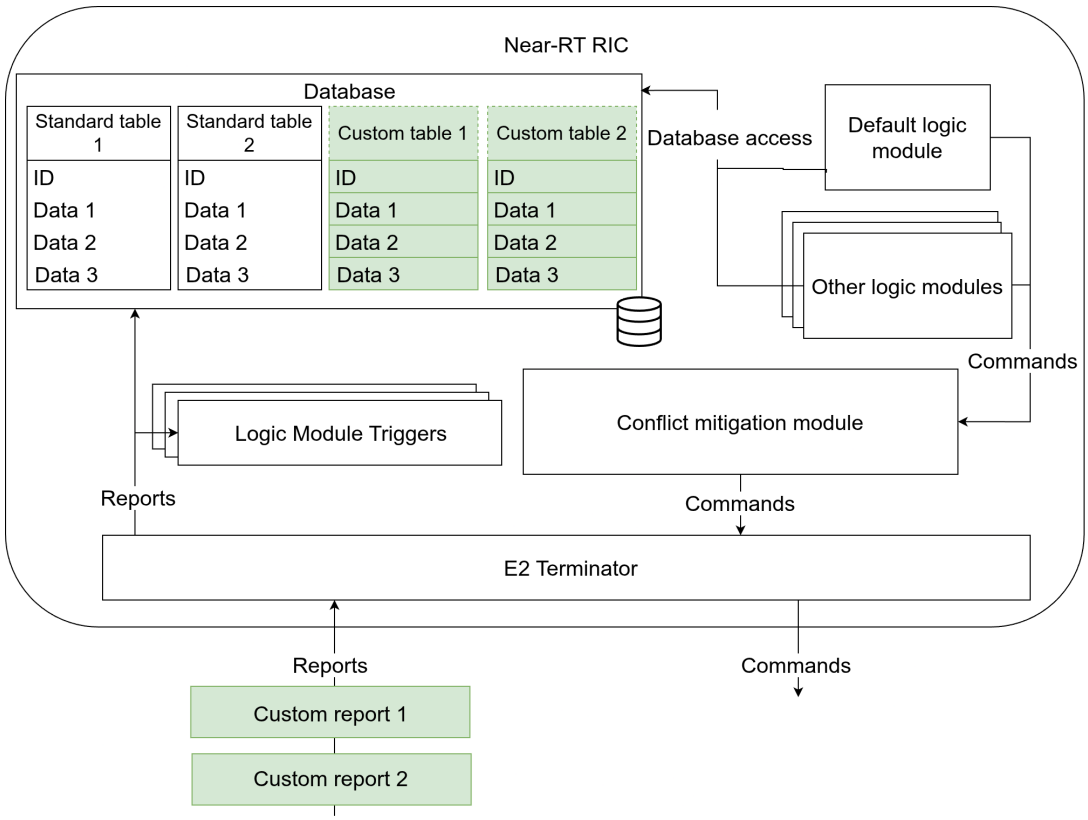


Fig. 3: Modified Near-RT RIC. In green, the custom reports and the dynamically generated tables.

use of these methods can be found in the documentation provided with the tool. Figure 2 shows the UML diagram of the `OranReportSqlite` class.

Figure 3 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 statically created at the start of the simulation, preventing the 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 green, illustrating the extended flexibility of the modified Near-RT RIC.

## IV. Use Cases

To demonstrate the applicability of the proposed extension, two representative use cases were implemented: `simple-db-example` and `rl-handover-example`.

The `simple-db-example` serves as a basic proof-of-concept for the new database system. It illustrates the creation and management of custom reports while ensuring interoperability with the standard reports already available in ns3-oran. This example validates that the extended database can handle heterogeneous report types simultaneously, enabling richer performance monitoring within the simulation.

The `rl-handover-example` represents a more advanced scenario, showcasing how the tool can support machine learning-based RAN control. In this case, the system collects performance metrics required for the training of a reinforcement learning (RL) model for handover management. This use case demonstrates how dynamically generated reports can provide the fine-grained and time-correlated data necessary to train and evaluate RL-based xApps, which are otherwise infeasible using only the default ns3-oran reporting framework.

### A. Use Case 1: Using Custom Reports

This use case demonstrates the creation of a custom report that periodically records the IPv4 address of a User Equipment (UE). The main objective is to showcase the procedure for defining custom reports and to validate the operation of the new adaptive database system within the simulation script. In addition, the example highlights the coexistence of custom reports with the standard reports already implemented in ns3-oran.

Figure 4 illustrates the simulated scenario, where a UE connected to an eNB periodically sends two types of reports to the Near-RT RIC: (i) standard reports containing location information, and (ii) custom reports containing the UE's IPv4 address. Upon reception of the custom report, the Near-RT RIC automatically generates a new table in its database based on the metadata carried in the report, demonstrating the dynamic table creation capability of the proposed system.

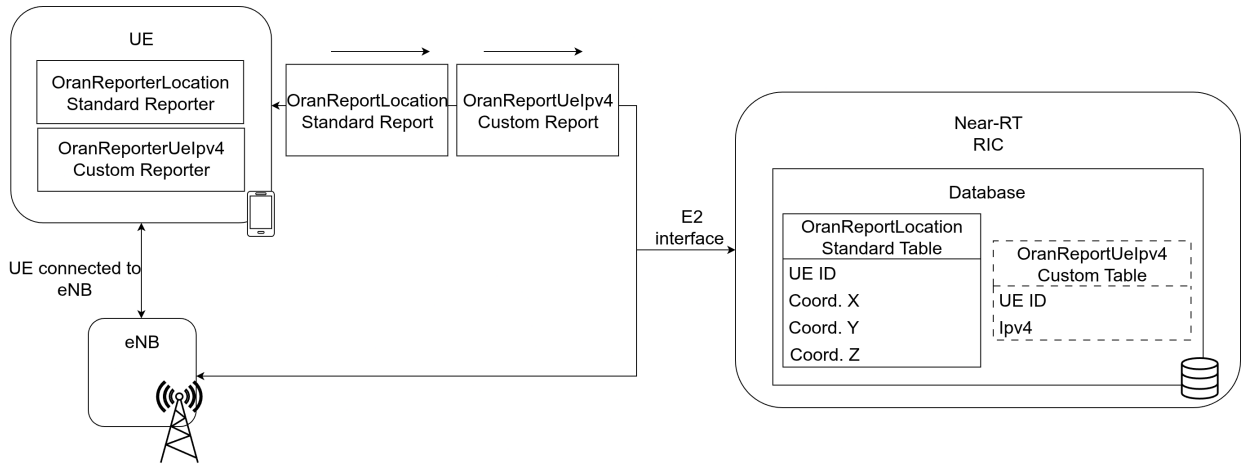


Fig. 4: Simulated scenario in `simple-db-example`.

```
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.

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. This method is used by the Near-RT RIC to retrieve the data stored in 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.

Listing 2 presents the output of the example execution, which prints on-screen the SQL queries performed by the Near-RT RIC along with their corresponding simulation times-

```
1 Query OK(101): "INSERT INTO node (nodetype) VALUES (?);"
    (1)
1 Query OK(101): "INSERT INTO noderegistration (nodeid,
    registered, simulationtime) VALUES (?, ?, ?);" (5, 1,
    1000000000)
1 Query OK(101): "INSERT OR REPLACE INTO lteue (nodeid,
    imsi) VALUES (?, ?);" (0, 1)
2 Query OK(101): "SELECT registered FROM noderegistration
    WHERE nodeid = ? ORDER BY simulationtime DESC, entryid
    DESC LIMIT 1;" (5)
2 Query OK(101): "INSERT INTO nodelocation (nodeid, x, y,
    z, simulationtime) VALUES (?, ?, ?, ?, ?);" (5, 15, 0,
    1.5, 2000000000)
2 Query OK(101): "CREATE TABLE IF NOT EXISTS UeIpv4
(entryid          INTEGER PRIMARY KEY
    AUTOINCREMENT NOT NULL, nodeid        INTEGER NOT NULL,
time      INTEGER NOT NULL, ipv4
    TEXT          NOT NULL, FOREIGN KEY(nodeid) REFERENCES
node(nodeid)                );"
2 Query OK(101): "INSERT INTO UeIpv4 (nodeid, time, ipv4)
    VALUES (5, 2000000000, "7.0.0.2");"
3 Query OK(101): "SELECT registered FROM noderegistration
    WHERE nodeid = ? ORDER BY simulationtime DESC, entryid
    DESC LIMIT 1;" (5)
3 Query OK(101): "INSERT INTO nodelocation (nodeid, x, y,
    z, simulationtime) VALUES (?, ?, ?, ?, ?);" (5, 20, 0,
    1.5, 3000000000)
3 Query OK(101): "INSERT INTO UeIpv4 (nodeid, time, ipv4)
    VALUES (5, 3000000000, "7.0.0.2");"
4 Query OK(101): "SELECT registered FROM noderegistration
    WHERE nodeid = ? ORDER BY simulationtime DESC, entryid
    DESC LIMIT 1;" (5)
4 Query OK(101): "INSERT INTO nodelocation (nodeid, x, y,
    z, simulationtime) VALUES (?, ?, ?, ?, ?);" (5, 25, 0,
    1.5, 4000000000)
4 Query OK(101): "INSERT INTO UeIpv4 (nodeid, time, ipv4)
    VALUES (5, 4000000000, "7.0.0.2");"
```

Listing 2: Exemple of logs in database for handover management use case.

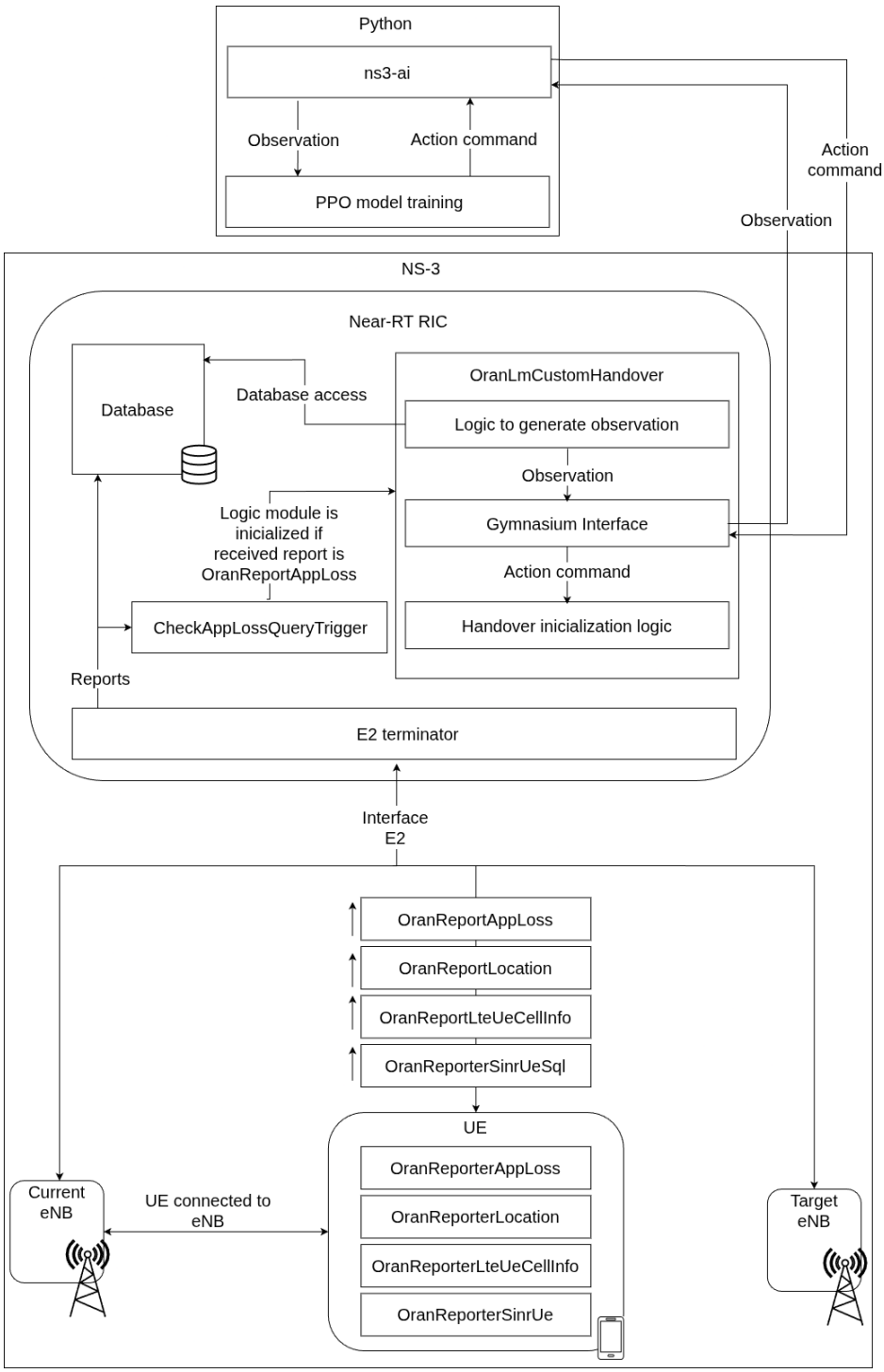


Fig. 5: Architecture of handover with reinforcement learning example.

tamps. From the logs, it is possible to observe the exact moment when the custom table is created in the database and subsequently populated with IPv4 address reports. This validates the seamless integration of custom report handling into the Near-RT RIC's database subsystem.

### *B. Use Case 2: Handover with Reinforcement Learning*

The second example demonstrates how the proposed database model can be integrated with reinforcement learning (RL) to support network control applications. In this case, a logic module is implemented to train a reinforcement learning model using the Proximal Policy Optimization (PPO) algo-

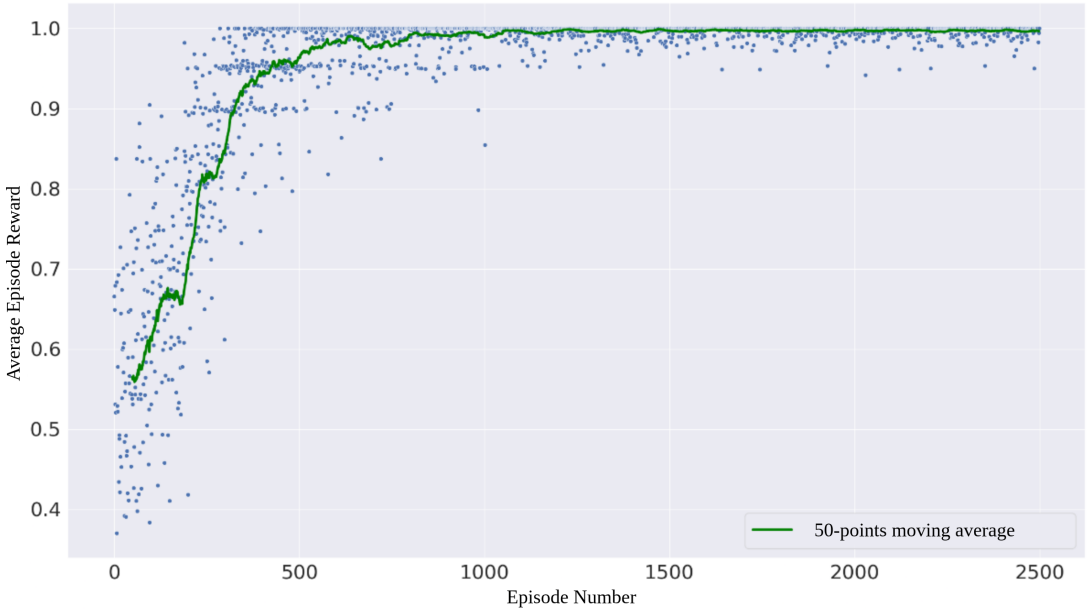


Fig. 6: Evolution of the reinforcement learning training model for handover in the use case 2 example scenario.

rithm for handover management, with the objective of minimizing packet loss. For demonstration purposes, the example simulates handover management in a scenario with one UE and two eNBs.

The two eNBs are initialized at fixed positions at opposite edges of the simulation environment. The UE is initialized at a random position between the two eNBs, with a constant velocity along the X-axis. The eNBs and the UE are initialized in collinear positions. To maximize handover occurrences, the UE's movement direction is chosen so that it always travels toward the farthest eNB. Both the UE and the eNBs are connected to the Near-RT RIC via the E2 interface. During the simulation, the UE periodically generates reports containing its spatial position, the packet loss rate observed since the last report, and the signal-to-noise ratio (SINR) of its current connection. Whenever the Near-RT RIC receives a packet loss report exceeding 10%, the logic module constructs an observation that includes packet loss rate, SINR, and the UE's distance to each eNB. This observation is forwarded to a PPO-based RL agent implemented in Python. The agent returns an action command specifying the eNB to which the UE should connect. If the chosen eNB differs from the current serving eNB, a handover is triggered. Training proceeds by repeatedly simulating this scenario, with the RL agent gradually improving its policy over successive episodes. The training process can be interrupted by the user once the model achieves satisfactory performance, after which the trained model may be used solely for inference in handover management.

This example also makes use of the ns3-ai module [7], which provides communication between ns-3 simulation scripts and external Python processes. Using ns3-ai, a Gymnasium-compatible [8] environment is defined in C++ within ns-3, enabling the simulation to act directly as an RL training environment accessible to the Python agent. The PPO algorithm was implemented using the stable-baselines3 [9] package.

After training, whenever the Near-RT RIC receives a packet loss report from a UE, the main logic module (responsible for training the machine learning model for handover management) generates an observation for the model. This observation includes packet loss rate, SINR, and the UE's distance to each eNB. The model processes this data and returns the index of the eNB the UE should connect to. If the indicated eNB matches the UE's current connection, no action is taken; otherwise, a handover process begins. Figure 5 illustrates the architecture of the implemented simulated scenario.

Figure 6 shows the evolution of the RL model during training. Each point represents the average reward obtained in one episode, where the reward function is defined as $1 - p$, with $p$ denoting the UE's packet loss rate. In the chart, each blue dot represents the mean reward observed by the network in a simulation episode, ranging from 0 (mean package loss of 100%) to 1 (mean package loss of 0%). The green line is the average value of the last 50 episodes rewards.

Table creation, report saving and retrieval times were measured for this example. Over 600 episodes, the average time to create a new table from a new custom report was $295 \pm 8$ ms. The time to save a new custom report was $224 \pm 8$ ms, which is on par with the measured time to save a standard report of package loss ($281 \pm 15$ ms). The time to retrieve custom reports was $26 \pm 3$ ms. All the measurements were made on a computer with 32 GB of RAM and an AMD 5700x3d processor, running ns-3 in a Docker container.

## V. 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 extensions, the simulator can be used as a realistic environment for training machine learning models and, in particular, reinforcement learning (RL) applications. 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. Such capabilities are particularly valuable for developing and validating ML models directly from ns-3 simulations, without the need for extensive preprocessing or external instrumentation.

Furthermore, ongoing work is extending the tool with support for integration with the 5G-LENA module [10], which implements 5G 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 applicability of the framework to next-generation RAN scenarios.

## References


[1] O-RAN Alliance, *https://www.o-ran.org/*, https://www.o-ran.org/, 2025, [Accessed 14-02-2025]. [Online]. Available: https://www.o-ran.org/

[2] M. Polese, L. Bonati, S. D'Oro, S. Basagni, and T. Melodia, "Understanding O-RAN: Architecture, Interfaces, Algorithms, Security, and Research Challenges," *IEEE Communications Surveys & Tutorials*, January 2023. [Online]. Available: https://arxiv.org/abs/2202.01032

[3] ns-3, "https://www.nsnam.org/," https://www.nsnam.org/, 2025, [Accessed 14-02-2025].

[4] A. Lacava, M. Polese, R. Sivaraj, R. Soundrarajan, B. S. Bhati, T. Singh, T. Zugno, F. Cuomo, and T. Melodia, "Programmable and customized intelligence for traffic steering in 5g networks using open ran architectures," *IEEE Transactions on Mobile Computing*, pp. 1–16, 2023.

[5] ns3-oran, "https://github.com/usnistgov/ns3-oran," 2025, [Accessed 14-02-2025].

[6] OSC, "The O-RAN Software Community (SC) Documentation," Tech. Rep., 2024, disponível em https://docs.o-ran-sc.org/en/latest/index.html.

[7] H. Yin, P. Liu, K. Liu, L. Cao, L. Zhang, Y. Gao, and X. Hei, "Ns3-ai: Fostering artificial intelligence algorithms for networking research," in *Proceedings of the 2020 Workshop on Ns-3*, ser. WNS3 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 57–64. [Online]. Available: https://doi.org/10.1145/3389400.3389404

[8] [Online]. Available: https://gymnasium.farama.org/

[9] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *J. Mach. Learn. Res.*, vol. 22, pp. 268:1–268:8, 2021.

[10] N. Patriciello, S. Lagen, B. Bojovic, and L. Giupponi, "An e2e simulator for 5g nr networks," *Simulation Modelling Practice and Theory*, vol. 96, p. 101933, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1569190X19300589