Improved DX on the development of O-RAN Near-RT RIC E2 Interface Solutions

Mathias Santos de Brito
Next Generation Netowrks
Techinishce Universität Berlin
Berlin, Germany
mathias.santos.de.brito@tu-berlin.de

Thomas Magedanz
Next Generation Netowrks
Techinishce Universität Berlin
Berlin, Germany
thomas.magedanz@tu-berlin.de

Marius Corici
NGNI
Fraunhofer FOKUS
Berlin, Germany
marius-iulian.corici@fokus.fraunhofer.de

***Abstract*—The O-RAN alliance is pushing the development of openness and disaggregated RAN solutions. Its OAM architecture proposed components to support applications intended to monitor, manage, and control the RAN. The so-called RICs are the core components for delivering intelligent solutions. The Near-RT RIC communicates with the RAN components via the E2 interface, designed using the ASN.1 interface definition language, commonly used in the telecom domain. ANS1c is a tool to generate code in C based on the ASN.1 definitions as input. This work proposes a wrap solution to the code generated by ASN1c to facilitate the development of E2-based solutions, aiming to increase the engagement of developers to build Near-RT solutions, especially newcomers. This paper describes the methods used to better DX and evaluates its performance compared to the vanilla code generated by ASN1c. The results show a considerable overhead compared to the original ASN1c; on the other hand, this exchange significantly improves the developer experience, paving the way for more efficient and engaging development processes.**

***Index Terms*—5G, Mobile Networks, Open RAN, O-RAN, E2AP**

I. Introduction

ASN.1 [1] is an IDF (Interface Definition Language) very popular in different fields like security and telecommunications. The language is used to describe in detail the elements of a protocol, from the datatypes to the messages to be exchanged. Complex datatypes and relationships can be defined, creating a well-defined and structured protocol.

In the context of the O-RAN [2] alliance that uses ASN.1 to describe the E2 interface between communication between E2 Nodes and the Near-Real Time Radio Intelligent Controller (Near-RT RIC) [3], responsible for enabling control loops within a time window of 10ms-1s. The protocol is based on two main pillars: the E2 Application Protocol (E2AP) [4] and the E2 Service Models (E2SM) [5], both specified using ASN.1.

There are many tools to generate source code from ASN1 definitions in different programming languages; in the open-source world, the most popular choice is the ASN1c [6] compiler. In addition, there are many forks of ASN1c that add specific enhancements. For newcomers, the entire concept of ASN.1, along with the C language and the complexity of the generated code, can drastically increase the onboarding time of developers in projects and the risk of introducing common errors to the C language, such as memory leaks.

In the context of the design and implementation of E2 control loops for improving the performance of 5G networks in the German project CampusOS, it was conceived and applied a wrapping technique to reduce the complexity of ASN1c based code and speed up the development of the E2 solution.

The solution consists of using design patterns [7] to wrap the generated structures from ASN1c while associating, to the wrappers, operations for encoding, decoding, printing, and string conversion using function pointers, making it easier to operate over the types. The wrappers are designed to reduce the need for dynamic memory allocation and reduce the risks of memory leaks.

We demonstrate the efficiency of the solution by wrapping the generated ASN1c structures for the E2 Application Protocol (E2AP) and compare both approaches, highlighting the impact on Developer eXperience (DX) [8], decreasing the learning curve, and allowing developers to focus on innovative ideas around the proposed problem instead of the framework used.

As C code is easily integrated into many languages, we also investigate how the proposed approach behaves on integration with the go language using the CGO embedded go tool to integrate the encoder / decoder generated by ASN1c.

It is essential to verify the overhead added since one of the main advantages of using ASN.1 is for performance reasons, in this context, we conducted a series of measurements to verify the performance of the proposed solution compared to the original code generated by ASN1c.

Reducing the overhead of developers on projects based on ASN.1 is a must in the telecommunications field, since it is extensively used; in this work, we demonstrate that using the ASN1c open source solution along with the right design choices can drastically reduce the time for onboarding developers and increase code output.

The rest of this paper is organized as follows: In Section I, we discuss the current work in the literature on the topic. Section II describes the methodology used to plan and identify the best approach to develop the proposed solution; in Section III, we bring details of the implementation. In Section IV, we describe the results of the performance measurements that compare the raw ASN1c code with the proposed solution. In Section V, we conclude, and after that we discuss the future

979-8-3315-2096-0/25/$31.00 ©2025 IEEE

work in Section VI.

This work aims to provide a design pattern for encapsulating generated ASC1c code and a library around the O-RAN E2AP protocol. Although wrapping the ASN1c generated code is not an uncommon task, they are commonly done in the scope of projects. An example can be seen in the repository *e2-interface* of the E2 simulator and E2 Termination [9] from the O-RAN Software Community (O-RAN SC). They use the raw ASN1c to provide high-level functions, which are usually mainly used in the project. On the other hand, we focus on having a library that can be used to speed up E2-related development, e.g. near-RT RICs.

It is important to compare the solutions built around the E2AP protocol here, and for that, a focus on existing Near-RT RICs is necessary. Flexric [10] uses a wrapping technique that helps to abstract away some details of the ASN1c-generated code. However, we focus on DX, and besides the wrappers, we use some aspects of the builder pattern to allow for easy creation and population of the structures.

Another solution is the μ-Onos Near-RT , which takes an approach to converting the ASN.1 specification to Google Protobuf files and uses the Go language to implement their solution. We, on the other hand, use the code generated by ASN1c and the C language. Despite also using Golang in a second approach, we take a different approach by mapping the developed Wrapper using CGO.

Commercial near-RT RICs have recently been advertised, for example, Juniper RIC [11], but lack of access to their code does not allow for a precise evaluation of their approach. However, the lack of release of E2AP libraries makes it clear that their focus is on internal solutions.

The two mentioned projects are the two most popular open source RICs, and we see that despite having some RICs already released, they all create their own code around the E2AP specification specifically for their solution. In this regard, we take a different approach to making ASN.1 generated code more easily readable and easier to use for developers.

DX is a relatively recent research area and is still lacking a general accepted standard taxonomy, metrics and evaluation processess; however, different studies and methodologies are proposed [8]. The authors in [12] propose a conceptual framework defining three dimensions of DX, cognition, affect, and conation. In this work, we focus on the cognition dimension as defined in [12]. This dimension is related to the perception of developers of objects and processes on the activities carried out, this includes the tools used.

II. Methodology

ASN1c, like the very nature of ASN.1, relies on a hierarchical structure when generating C code, making use of structs and unions to generate part of the code to represent the ASN.1 data types. However, the features of the C language are not enough to generate intuitive and easy code, especially when mapping features from ASN.1 such as *choices* and constraints. The inexistence in C of any mechanism of reflection/introspection tends to bring even more complexity to the generated code. In fact, part of the ASN1c generated code can be credited to the lack of modern features in C, not ASN1c.

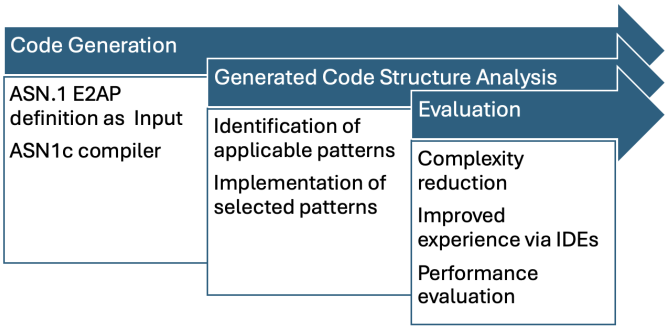

Fig. 1. Project methdoology for identifying the best approach for abstracting the complexity of the ASN1c generated code.

When working with existent code, understanding it is key to bringing development up to speed, but not only that, creating abstraction layers that make sense and feel natural to developers can increase developers' output, which is crucial in a project. Abstraction layers are an old topic in computer science and programming, and the literature is rich. However, it is essential to understand how these concepts are being applied in the context of ASN.1 C generated codes, since we are focused on open source tools.

In the first phase, we analyzed the generated code, the generated types, operations, and their relationship. Once then, the existing design patterns could be analyzed. The nature of the process is creational; we want to create messages to be sent over the network. On the other hand, we want to guarantee that basic operations are part of the structures and that additional functionalities are added, which is a behavioral problem. In summary, we followed the process depicted in Figure 1.

To evaluate our solution's reduced complexity, we consider the following metrics: code that is easier to read, write, and understand; improved and precise hints of autocompletion from IDEs; and minimal performance overhead.

The ASN.1 definitions used are those of O-RAN E2AP. We used the versions used by Wireshark to implement support for E2AP, and they are available in the project repository.

Generating the source code using ASN1c is not trivial when the ASN.1 structures are too complex. Numerous forks are available, each of them implementing features that will eventually be incorporated into the main repository. However, at the point of writing this paper, one version often used and referenced in some O-RAN SC documentation is the one from Nokia (https://github.com/nokia/asn1c).

For the implementation, one of the goals was not to use any external dependency, so the resulting code should be more portable and generate less overhead, but not at the cost of providing simplicity. Looking at the code in the list 1, we can have an idea of how the ASN1c generated code is used.

```
1 void main() {
2     E2setupFailure failure = {0};
3
4     // Allocate memory for the protocol IEs
```

```
5     E2setupFailureIEs_t *ie1 =
6         (E2setupFailureIEs_t*)calloc(
7             1, sizeof(E2setupFailure_t));
8     E2setupFailureIEs_t *ie2 =
9         (E2setupFailureIEs_t*)calloc(
10            1, sizeof(E2setupFailureIEs_t));
11
12    // Set the ProtocolIEs ie1 and ie2
13    ie1->id = ProtocolIE_ID_id_TransactionID;
14    ie1->criticality = Criticality_reject;
15    ie1->value.present = \
16        E2setupFailureIEs__value_PR_TransactionID;
17    ie1->value.choice.TransactionID = 1;
18
19    ie2->id = ProtocolIE_ID_id_TimeToWait;
20    ie2->criticality = Criticality_ignore;
21    ie2->value.present = \
22        E2setupFailureIEs__value_PR_TimeToWait;
23    ie2->value.choice.TimeToWait = 3;
24
25    // Add IEs to the list of ProtocolIEs
26    ASN_SEQUENCE_ADD(&(failure.protocolIEs.list), ie);
27    ASN_SEQUENCE_ADD(&(failure.protocolIEs.list), ie);
28 }
```

Listing 1. Partial E2 Setup Failure message using solely the generated code form ASN1c.

The main takeaway from analyzing the code and the ASN.1 definition for E2AP while considering how ASN1c handles the code generation is that it is clear that the lines from 13-16 and 19-21 will always look the same, since a ProtocolIE is a general structure we need to inform with the type it will hold. These are details that we can abstract away from the developer.

Two common design patterns were considered by evaluating the options for abstracting the original C structs: the Builder pattern and the Decorator/Wrapper pattern. The first resolves the problem of reducing the complexity of initialize and populating a the ASN1c structures with values. The second to encapsulate the ASN1c structure and share common operations between different types. reffig:patterns

In the C language we can rely on function pointers to have a behavior similar to object-oriented languages and implement the patterns in a similar being the desired result shown in listing 2.

```
1  void main() {
2      E2SetupFailureWrapper e2SetupFailure = \
3          newE2SetupFailure();
4
5      e2SetupFailure
6          .setTransactionId(&e2SetupFailure, 1)
7          .setTimeToWait(&e2SetupFailure, 3);
8
9      e2SetupFailure.asn.decode(
10         e2SetupFailure.type,
11         e2SetupFailure.asnStruct
12     );
13     // for the above operation optmal would be
14     // e2SetupFailure.decode(e2SetupFailure)
15 }
```

Listing 2. Abstracting away from the static details from the developer and using a call chaining.

The two examples in Listings 1 and 2 show how the DX can be improved in terms of abstracting away parameters that should have the same values for every instance.

In C, we need to take into account memory management. Some design decisions can impact DX by requiring the developer to constantly allocate and deallocate memory, increasing the possibility of introducing memory leaks in the code. We need to find a balance, and the code in Listing 2 in lines 9-14 shows the current solution and what would be ideal. This is important for onboarding new developers not used to the concept, while reducing the risks of introducing bugs in the code.

After some iterations and evaluation of the different approaches, we proposed and implemented a C library called TubE2 and a Go version by mapping the C code with the CGO feature from Golang. However, in this paper, we focus on the C version.

III. Implementation

The implementation of the TubE2 library is based on the following requirements:

- Must introduce minimal overhead;
- Must not introduce external dependencies;
- Must reduce the complexity of creating and initializing an ASN1c structure with values;
- Should not minimize the risk of memory leaks;
- Should make it easier to use ASN1c operations, e.g., encode/decode.

A. Core C Macros

To achieve our objectives, a mix of the Wrapper and Builder patterns was chosen based on the fact that the first can hide the details of the underlaying code while allowing derived types to add specific behavior, and the second reduces the complexity of creating and populating instances with values.

Two core C macros were established for this purpose, and they are shown in Listing 3.

```
1  #define ASN1C_WRAP(asn_struct_type) \
2      asn_struct_type _structPtr; \
3      ASN1CStructWrapper asn; \
4      asn_TYPE_descriptor_t asnType;
5
6
7  #define ASN1C_WRAPPER_INIT( \
8      asn_struct_type, \
9      asn_DEF_type, \
10     wrapper_type_name \
11 ) ({\
12     wrapper_type_name wrapper = {0}; \
13     wrapper.asnType = asn_DEF_type; \
14     wrapper.asn = newASN1CStructWrapper(asn_DEF_type); \
15     wrapper;\
16 })
17
18 ASN1CStructWrapper newASN1COpsWrapper(
19     asn_TYPE_descriptor_t type
20 ) {
21     ASN1CStructWrapper wrapper;
22     wrapper.validate = asn1cOpsWrapper_validate;
23     wrapper.encode = aasn1cOpsWrapper_encode;
24     wrapper.decode = asn1cOpsWrapper_decode;
25     wrapper.toString =  asn1cOpsWrapper_toString;
26     wrapper.print = asn1cOpsWrapper_print;
27     wrapper.free = asn1cOpsWrapper_free;
28     return wrapper;
29 }
```

Listing 3. Core macros to be used for wrapping the ASN1c structs.

The `ASN1C_WRAP` macro, in lines 1-4, is used in each wrapper *struct* created to encapsulate the generated ASN1c, which provides mandatory fields for the solution to work properly. The `ASN1C_WRAPPER_INIT` is used in constructor

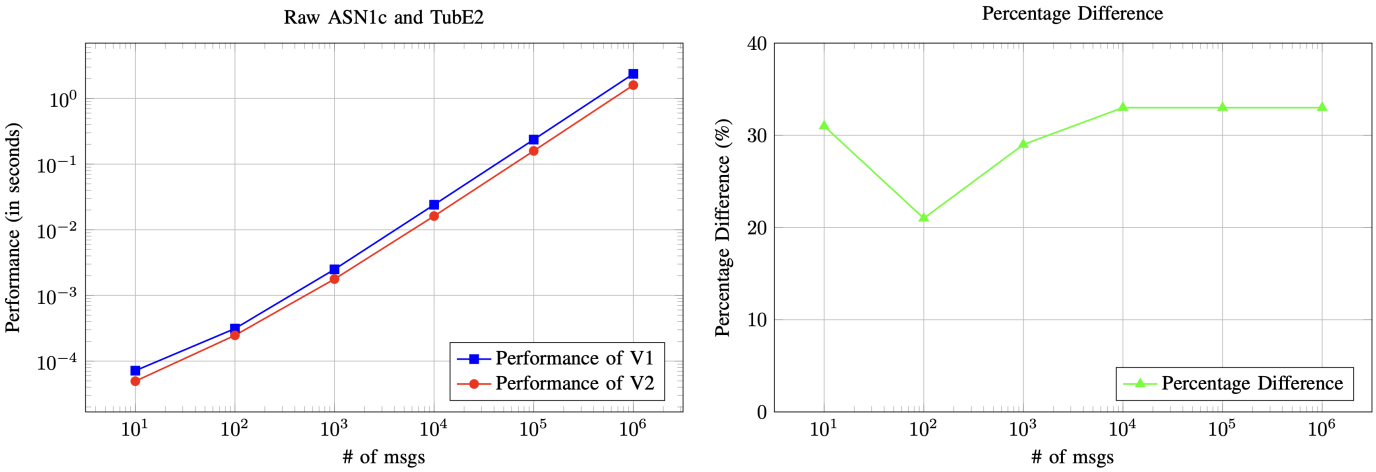

Fig. 2. The performance on the number of generated E2 Setup Request messages per second using plain ASN1c generated code (V1) compared to our solution (V2) on the left, and the difference in percentage of the number of generated messages on the right.

functions to properly initialize the `asn` wrapper's attributes. These attributes hold pointers to common operations from the ASN1c generated code, and they are set by the function `newASN1CopsWrapper` called in `ANS1C_WRAPPER_INIT`.

In the Listing 4, we wrap the ASN1c struct generated to represent the E2AP Setup Request procedure and an example of how to use it.

```
1  struct E2SetupRequestWrapper_t {
2      ASN1C_WRAP(E2setupRequest_t)
3      E2SetupRequestWrapper (*setTransactionId)(
4          E2SetupRequestWrapper *w,
5          long id
6      );
7      E2SetupRequestWrapper (*setE2NodeId)(
8          E2SetupRequestWrapper *w,
9          GlobalE2NodeIdWrapper nodeId
10     );
11     E2SetupRequestWrapper (*setRanAddRanFunctionList)(
12         E2SetupRequestWrapper *w,
13         RanFunctionListWrapper list
14     );
15     E2SetupRequestWrapper (*setComponentConfigAddition)(
16         E2SetupRequestWrapper *w,
17         E2NodeComponentConfigAddtionListWrapper list
18     );
19 };
```

Listing 4. Core macros to be used for wrapping the ASN1c structs.

The struct wrapper is expected, in addition to the wrapped type, to have pointers to some of the ASN1c operations that set the attributes of the encapsulated ASN1c *struct*, for example, `setTransactionId`, `setE2NodeId`, `setRanAddRanFunctionList`, and finally `setComponentConfigAddition`. They receive as a parameter the reference to the action instance being handled and the wrapper with the value to be set in the case of complex types or the value directly in the case of primitive types. These are simple functions that basically transfer the content of the struct passed as a parameter to the proper attribute of the wrapped struct.

Note that all the function pointers are expected to return the wrapper itself so that we can set the other attributes using function/method chaining. The reason for the setter functions is that for some attributes, the attribution is complex and requires more than one operation, this is the case of attributes derived from ASN1 *choice* types. In the end, we have the result shown in Listing 1.

IV. Results

Improvements in DX on the cognitive dimension [12] require the developer to be comfortable with the platform, techniques, processes, and procedures, as well as the required skills. Although quantifying satisfaction would require an extensive study, our results shows a reduction on ~46% in line of code (LoC) and the abstraction of four low-level static assignments of the AS1c generated structures. Also, naming conventions used highlight the final intention of the operations and use the builder design pattern to chain the calls of the functions, which helps the developer through auto-completion from Integrated Development Environments (IDE). The solution also improves the readability and self-documentation of the code.

Our approach for development does not add the need for an additional library or increase the complexity of memory management, resulting in a safe and easy-to-use extension to the generated ASN1c code. We carried out some tests to verify the difference between the performance of a pure ASNC1c code and the wrapped version. The test consists of creating an E2 Setup Request variable, populating it and freeing the memory. The messages are not sent over the network. We vary the number of messages and verify how long it takes for the process to finish. The results are shown in figure 2.

The tests were run on an x86 machine with an Intel(R) Core(TM) i7-4980HQ CPU @ 2.80GHz processor, with 16 gigabytes of Random Access Memory (RAM) and Ubuntu 22.04.4 LTS as the operating system. The ASN1c compiler used was the ASN.1 Compiler v0.9.29, Nokia fork commit *8b8028a80aa8f5545dfcaa9d357150ac0c475f70*. To generate C types, encoders and decoders with ASN1c, the command shown in Listing 5 was used.

```
asn1c -fcompound-names -gen-PER -findirect-choice \
-fincludes-quoted -fno-include-deps -pdu=all -D \
<e2ap-definition-files-list>
```

Listing 5. Command used and parameters used to generate the ASN1c code.

The TubE2 shows a performance around 30% below the performance of the pure ASN1c when generating messages. This impact is mainly due to the creation of the wrapper structures, additional function calls, and, consequently, context exchanges. However, more research is needed, especially on how the values are assigned *by copy* compared to *by reference*. However, the volume of messages is relatively high, with the TubE2 generating one million messages in around 2.34 seconds.

One positive aspect of our library — further investigation should be carried out to verify if it is impacting performance — is the fact that no impact on memory management has been added. The user needs just to instantiate the wrapper instead of the raw ASN1c struct.

In addition, the setter functions help the developer identify what data types they need without the need to navigate through the ASN.1 specification or the AS1Nc-generated code. IDEs can hint at which wrapper type is needed to set the value. Despite this also happening in the ASN1c code, the level of detail and the number of types to be handled are many, as shown in the listing 1.

V. Conclusion

ASN1c generated code for large ASN.1 specifications, e.g. the O-RAN E2AP and other E2 specifications, takes much work. Most projects will abstract away the details, but mainly for internal use. We are proposing a general library using the E2AP protocol. It abstracts unnecessary details from developers working on higher levels, letting them focus more on the solution.

We have experienced an increase in productivity and more maintainable code in E2AP-based applications. However, this came at an exchange in performance, where we achieved only 70% of the performance observed when using raw ASN1c.

Memory management is also a positive aspect of our solution, and we kept it as simple as possible to avoid unnecessary allocations and potential memory leaks.

In fact, only a few E2AP-based solutions are available, be they near-RT RIC, E2 Terminations, or related SDKs. One of the potential reasons is the need for an easy-to-use library or abstraction. This project aims to fill this gap and provide the community with an intuitive way to handle ASN.1 generated code.

Finally, onboarding new programmers in the ASN.1-based protocols when using TubE2 can be faster, and the time required to ship new functionalities is reduced.

VI. Future Work

After some initial profiling, we concluded that the performance of our solution could be improved. In this first version, the focus was on DX, and the design took preference over the performance. However, we consider that achieving 70% of the performance is enough to bring it to production. We want to improve this value, and work is being carried out.

Another aspect to consider is the use of setter functions to carry details of the ASN1 specifications, and an example is considering the use of the prefixes `setm` and `seto` to indicate mandatory and optional fields. Once again, the priority is to improve the development experience.

The Golang mappings are under development and provide almost all the features available in the C code. However, due to the Golang features, we can improve DX even further.

We are investigating processes to automate the generation of the mappings. It is a well-defined and structured process and the development of tools for that purpose is feasible.

Acknowledgment

This research and development project is funded by the German Federal Ministry of Economics and Climate Action (BMWK) within the funding measure ”5G campus-networks” under the funding code 01MC22002 and supervised by the DLR Projektträger — Division Society, Innovation, Technology at the German Aerospace Center. We thank for the opportunity to work on this project.

References

[1] G. Neufeld and S. Vuong, “An overview of asn. 1,” *Computer Networks and ISDN Systems*, vol. 23, no. 5, pp. 393–415, 1992.

[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*, vol. 25, no. 2, pp. 1376–1411, 2023.

[3] H.-T. Thieu, V.-Q. Pham, A. Kak, and N. Choi, “Demystifying the near-real time ric: Architecture, operations, and benchmarking insights,” in *IEEE INFOCOM 2023-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2023, pp. 1–8.

[4] O. Alliance, “O-ran e2 application protocol (e2ap) 5.0,” *O-RAN.WG3.E2AP-R003-v05.00*, 2020.

[5] ——, “O-ran e2 service model (e2sm), ran control 6.0,” *O-RAN.WG3.E2SM-RC-R003-v06.00*, 2020.

[6] L. Walkin, “asn1c: The asn.1 compiler,” 2023, accessed: 2023-10-01. [Online]. Available: https://github.com/vlm/asn1c

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design patterns: Abstraction and reuse of object-oriented design,” in *ECOOP’93—Object-Oriented Programming: 7th European Conference Kaiserslautern, Germany, July 26–30, 1993 Proceedings 7*. Springer, 1993, pp. 406–431.

[8] A. Razzaq, J. Buckley, Q. Lai, T. Yu, and G. Botterweck, “A systematic literature review on the influence of enhanced developer experience on developers’ productivity: Factors, practices, and recommendations,” *ACM Computing Surveys*, vol. 57, no. 1, pp. 1–46, 2024.

[9] O.-R. S. Community, “Ric platform e2,” 2023, accessed: 2023-10-01. [Online]. Available: https://github.com/o-ran-sc/ric-plt-e2

[10] R. Schmidt, M. Irazabal, and N. Nikaein, “Flexric: an sdk for next-generation sd-rans,” in *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 411–425. [Online]. Available: https://doi.org/10.1145/3485983.3494870

[11] J. Networks, “Juniper ric,” 2023, accessed: 2023-10-01. [Online]. Available: https://www.juniper.net/us/en/solutions/mobile-service-provider/5g/ric.html

[12] F. Fagerholm and J. Münch, “Developer experience: Concept and definition,” in *2012 International Conference on Software and System Process (ICSSP)*, 2012, pp. 73–77.