ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUPD110

Bluesky NeXus: A SOLUTION FOR NEXUS-COMPLIANT DATA ACQUISITION IN Bluesky*

D. Tomecki[1,†], M. Bajdel[1], D. Burke[2], A. Dillmann[1], E. Lott[1], S. R. Patel[1], L. Porzio[1], W. Smith[1], S. Vadilonga[1]
[1]Helmholtz-Zentrum Berlin für Materialien und Energie GmbH, Berlin, Germany
[2]Deutsches Elektronen-Synchrotron DESY, Hamburg, Germany

Abstract

Modern scientific experiments require rich, standardized metadata to ensure that data is Findable, Accessible, Interoperable, and Reusable (FAIR). The *NeXus* format, a hierarchical data standard widely used in neutron, x-ray, and muon science, provides a robust framework for organizing such metadata. However, automating its integration into data acquisition workflows remains a challenge.

We present Bluesky NeXus, a Python package that enables automated, standards-compliant *NeXus* file generation within *Bluesky*—a modular data acquisition framework deployed at synchrotron and neutron facilities. Users of the package define the desired *NeXus* device structure—including groups, datasets, and attributes—using human-readable YAML configuration files. These are validated with *Pydantic*, a Python library for schema validation, ensuring consistency with *NeXus* definitions.

Bluesky NeXus captures both static metadata (e.g., instrument configuration) and dynamic measurements, consolidating them into a complete *NeXus* file automatically archived with each experiment. It integrates with the containerized *Bluesky* environment used at BESSY II, supporting a wide range of experimental setups.

Developed as part of the ROCK-IT project, Bluesky NeXus simplifies the creation of FAIR-compliant metadata and enhances the reproducibility and interoperability of scientific data.

INTRODUCTION

Efficient data collection and storage are essential for ensuring reproducibility, accessibility, and interoperability in modern scientific research. Large-scale facilities such as synchrotrons and neutron sources rely on complex instrumentation that produces vast amounts of data. The *Bluesky* framework [1] is widely used to manage experimental workflows, enabling researchers to design and execute scans while capturing structured metadata. However, storing this information in a consistent, standards-based format remains challenging, particularly when multiple instruments contribute to a single experiment.

The *NeXus* data format [2], built on the HDF5 (Hierarchical Data Format) standard, provides a hierarchical, self-describing structure for organizing, sharing, and preserving scientific data. Integrating *NeXus* with *Bluesky* runs ensures that both metadata and measurements are stored in a standards-compliant manner, enabling seamless retrieval and promoting reproducibility across research environments.

Capturing the configuration of all participating instruments—such as detectors, motors, and environmental sensors—is essential. A structured *NeXus* file preserves this information alongside experimental results, supporting downstream analysis and aligning with Findable, Accessible, Interoperable, and Reusable (FAIR [3]) data principles.

This paper introduces Bluesky NeXus [4], a software package developed as part of the ROCK-IT [5] project that automates *NeXus* file generation within the *Bluesky* framework. It employs a *Pydantic*-based [6] mechanism to define and validate *NeXus* structures, ensuring compliance with international conventions while supporting user-defined metadata extensions. By standardizing storage and automating metadata capture, Bluesky NeXus strengthens data integrity and facilitates collaboration across experimental facilities.

OBJECTIVE

The primary objective of this work is to design and implement a robust, extensible mechanism for generating a *NeXus*-compliant file from a *Bluesky* run. The resulting file serves as a structured, self-describing representation of experimental data, in alignment with community standards in neutron, X-ray, and muon science. Our goal is to serialize the data while ensuring it adheres to the hierarchical structure and naming conventions defined by the *NeXus* format.

The file structure is defined to include two principal groups: `NXinstrument` and `NXcollection`, both of which are *NeXus* base classes [7]. This hierarchical layout is illustrated in Fig. 1, which shows an exemplary NeXus file structure including four instruments.

The `NXinstrument` group is central to our implementation. It encapsulates all relevant information about the experimental devices involved in the plan, including configuration, structure, and associated metadata. This group must conform to the appropriate *NeXus* base class definitions and reflect the physical and logical organization of the instruments as accurately as possible.

By contrast, the `NXcollection` group plays a supplementary role. It serves as a metadata archive, storing unaltered copies of the *Bluesky* start and stop documents. These documents provide high-level contextual information about the run—such as timing, user annotations, and global parameters. Since the `NXcollection` group involves a straight-

* Work funded by ROCK-IT (Remote, Operando Controlled, Knowledge-driven, and IT-based) project.
† Corresponding author: daniel.tomecki@helmholtz-berlin.de

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUPD110

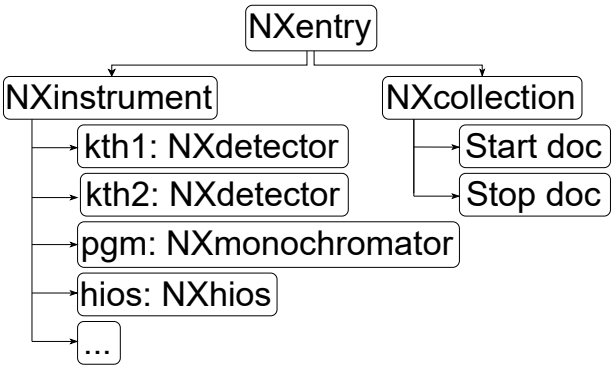

Figure 1: Illustration of the *NeXus* file structure.

forward copy operation without complex transformation or validation, it is not discussed further in this paper.

CHALLENGES IDENTIFIED

In the following, we focus on the challenges associated with extracting information from the instruments involved in the plan and structuring it to conform to the expected *NeXus* format. In particular, we ensure that the extracted data is organized within the `NXinstrument` group according to *NeXus* standards. The instruments in the plan are represented by device instances of *Ophyd* [8] classes, which serve as interfaces for interacting with hardware components in the *Bluesky* framework. During our analysis, we identified several challenges in mapping these instances to *NeXus* device structures based on *NeXus* base class definitions. These challenges arise from significant variability in the structure and naming conventions of common and beamline-specific *Ophyd* device classes.

At the BESSY II [9] facility in particular, this variability is pronounced: we work with a heterogeneous set of device class definitions that lack unified naming conventions or structural consistency in their components and attributes. This diversity makes it impractical to enforce large-scale refactoring of device classes. Therefore, any solution must be capable of adapting to the existing landscape without imposing disruptive changes.

The main challenges identified are as follows:

1. Mapping a device instance to its corresponding *NeXus* base class definition.
2. Selecting only the relevant components or attributes, since not all defined elements in a device class are applicable to *NeXus*.
3. Resolving naming inconsistencies between device components and *NeXus* attributes—for example, using `en` or `enrg` instead of the standardized `energy`.
4. Addressing structural mismatches between internal device hierarchies and *NeXus* expectations. For instance, a diffraction order might be represented as `mono.diff_order` in the device, while the *NXmonochromator* [10] base class expects a nested structure like `mono.GRATING.diffraction_order`.
5. Performing unit conversions where necessary—for example, converting energy values from electronvolts (eV) to kiloelectronvolts (keV) to comply with *NeXus* conventions.

ADDRESSING THE CHALLENGES

The challenges described above, and in particular the heterogeneous device class landscape at BESSY II, motivated our decision to adopt *Pydantic*, a Python library for data validation and settings management based on type annotations.

Pydantic is widely used to define and enforce data structures in a clear and declarative way. It automatically validates input data, performs type conversions when necessary, and generates informative error messages, making it particularly well-suited for data modeling tasks.

Instead of imposing structural or naming changes on device classes, *Pydantic* allows us to preserve them in their current form while still producing strictly compliant *NeXus* structures. In this way, *Pydantic* serves as a flexible adapter that reconciles local naming schemes and hierarchies with the standardized definitions expected by the *NeXus* base classes, bridging the gap between the flexibility of local device definitions and the strict requirements of *NeXus*.

Role of the Pydantic Schema

1. Maps a device instance to a *NeXus* base class.
2. Selects relevant components from an *Ophyd* device that correspond to attributes defined in the *NeXus* base class.
3. Renames components to match *NeXus* conventions (e.g., `en` → `energy`).
4. Constructs arbitrarily nested attribute structures consistent with *NeXus* specifications.
5. Defines transformation functions (e.g., unit conversions) to produce *NeXus*-compliant values.

The *Pydantic* schema ensures consistency between *Ophyd* device classes and the *NeXus* base classes. The application of *Pydantic* schemas effectively addresses all the challenges discussed in the previous section.

Each *Ophyd* device class is associated with a specific *Pydantic* schema. The assignment is managed through a decorator, which parses key-value pairs from the *Pydantic* schema provided in the form of a YAML-formatted string and assigns the resulting dictionary to a class attribute named `nx_schema`.

Role of the Pydantic Model

1. Validates the schema, ensuring that all required *NeXus* groups and fields are present.
2. Enforces structured data modeling to maintain consistency with the target *NeXus* layout.
3. The model instance enables the deserialization of validated data into a dictionary format for *NeXus* file generation.

For each device instance, a *Pydantic* model is instantiated using model name specified in the schema and the schema

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUPD110

content as input. Validation occurs upon instantiation, guaranteeing conformity of the schema to the model. Once instantiated, the model can be deserialized into a dictionary representation of the validated schema, which can be seamlessly integrated into the `open_run` message, corresponding to the start document in the *Bluesky* event model. If a device class lacks a corresponding *NeXus* base class, a general *Pydantic* model—`NXgeneralModel()`—is used, which imposes no constraints on the validated schema associated with that device class.

IMPLEMENTATION STRATEGY

The implementation of *NeXus* file generation is structured around two major phases: the *pre-run stage*, which performs necessary preparations before data acquisition begins, and the *post-run stage*, which finalizes and writes the structured data after the *Bluesky* data collection process completes. This two-stage approach allows for a clear separation between metadata initialization and data population, aligning well with the structure of the *Bluesky* data acquisition framework. The first stage is implemented using a *Bluesky* preprocessor that prepares and injects static metadata prior to the run, while the second stage is supported by a *Bluesky* callback that finalizes the data and writes the *NeXus* file upon completion of the run. Figures 2 and 3 illustrate the data flow for each of these phases.

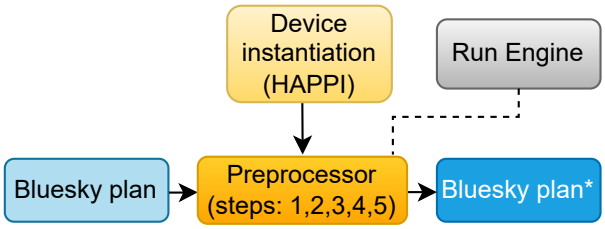

Figure 2: Illustration of the pre-run flow.

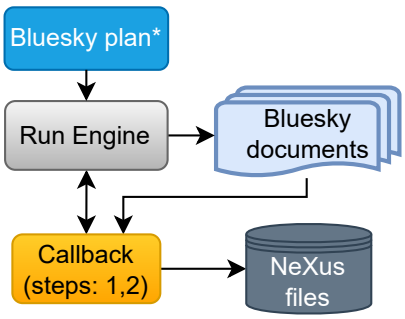

Figure 3: Illustration of the post-run flow.

Pre-run Actions Performed by the Preprocessor

The pre-run stage is initiated prior to the actual execution of the experimental plan. Its primary role is to analyze the devices involved and construct a skeleton *NeXus* file structure that is ready to be populated with static metadata and later dynamic data. The specific tasks performed in this stage include:

1. **Device detection:** Automatically detect all *Ophyd* device instances that participate in the experimental plan. This is essential for establishing which hardware components need to be represented in the *NeXus* file.
2. **NeXus structure generation:** For each detected device, generate a corresponding *NeXus* structure that adheres to the standards defined by the relevant *NeXus* base classes. This includes organizing components into nested groups and assigning correct attribute names based on the schema.
3. **Static metadata assignment:** We use the `HAPPI` [11] (Hierarchical Access to Persistent Parameters for Instruments) library to instantiate `Ophyd` devices and to assign static metadata to an arbitrary device instance. This static metadata is then used to populate predefined placeholders — identified by the prefix `$pre-run-md` — in the *NeXus* structure, ensuring consistent and accurate representation of each device's attributes within the data hierarchy.
4. **Device metadata structure generation:** Construct a consolidated structure that contains all static metadata for the devices involved in the plan. This metadata is assigned to the device instances as described in previous item.
5. **Run metadata injection:** Inject both the *NeXus* structure and the device metadata structure into the `open_run` message, which corresponds to the start document in the *Bluesky* event model. The device metadata structure is inserted under the name `device_md`, and the *NeXus* structure under the name `nexus_md`.

Post-run Actions Performed by the Callback

Once the experimental plan completes, the system enters the post-run stage. The primary objective in this phase is to complete the *NeXus* structure by filling in dynamic data fields with actual measurements and time-dependent values recorded during the run. Key activities include:

1. **Dynamic data population:** Populate previously defined placeholders in the *NeXus* structure with values extracted from the event and descriptor documents generated during the *Bluesky* run. These placeholders, whose names begin with the prefix `$post-run`, may include sensor readings, motor positions, or any other time-resolved data acquired throughout the experiment. Data for device components with the attribute `kind='config'` is retrieved from the configuration dictionary in the descriptor document. Data for device components with the attribute `kind='hinted'` or `kind='normal'` is retrieved from the `data_keys` dictionary in the descriptor document and from the associated event documents. Data is initially retrieved from the primary event or descriptor documents. If the component name is unavailable there, it is subse-

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUPD110

quently retrieved from the baseline event or descriptor documents.
If only a subset of a device's schema-defined components is used in a plan, the device should be subscribed to the baseline. This ensures that the unused components are included in the baseline descriptor, enabling replacement of all placeholders associated with the device in the schema YAML file.

2. **File serialization and writing:** The finalized *NeXus* structure is serialized and written to disk using the `h5py` [12] library. The resulting output file includes:
   - An `NXinstrument` group named `instrument`, which contains all validated and fully populated device information structured according to the *NeXus* schema.
   - An `NXcollection` group named `run_info`, which stores copies of the run metadata, specifically the start and stop documents. These provide contextual information about the experiment, such as run time, configuration parameters, and user annotations.

This staged approach ensures that the generated *NeXus* files are both standards-compliant and complete, containing all necessary metadata and measured values in a structured and reproducible format. Furthermore, the division between pre-run and post-run responsibilities allows for modularity in implementation and easier maintenance or extension of the pipeline in the future.

SCHEMA MAPPING EXAMPLE

This section defines a simplified monochromator and demonstrates how it is represented in a corresponding *Pydantic* schema. The code for both the device class definition (Fig. 4) and the schema (Fig. 5) is explained through comments following each figure. The `Mono` class includes two components: `en` (a simulated axis representing energy) and `d_ord` (a signal representing diffraction order). It also defines a `read_attrs` list, which is used internally by *Ophyd* to control readouts but is not relevant to the NeXus schema. Note that this is a minimal working example intended for demonstration purposes only. A production-ready implementation would typically include additional groups, fields, or attributes.

```
from ophyd import Device, Signal
from ophyd import Component as Cpt
from ophyd.sim import SynAxis

@NxSchemaLoader(Mono_nxschema)
class Mono(Device):
    en = Cpt(SynAxis, name="en")
    d_ord = Cpt(Signal, name="d_ord")
    read_attrs = ["en.readback"]
```

Figure 4: Definition of the device class `Mono`.

Comments on the Code Lines in Fig. 4

5: Class decorator that assigns a YAML-formatted string `Mono_nxschema` to the `Mono` device class.
7: *Ophyd* component representing energy. Note that the name `en` does not match the expected name `energy` as defined in the *NeXus* base class *NXmonochromator*.
8: *Ophyd* component representing the diffraction order. Note that the name `d_ord` does not match the expected name `diffraction_order` as defined in the *NeXus* base class *NXgrating* [13].
9: The attribute `read_attrs` is not relevant to *NeXus* and is therefore not included in the schema defined in Fig. 5.

```
Mono_nxschema: str = """
nx_model: NXmonochromatorModel
nxclass: NXmonochromator
energy:
  nxclass: NX_FLOAT
  value: $post-run:en
  dtype: float64
  attrs:
    units: "keV"
GRATING:
  nxclass: NXgrating
  diffraction_order:
    nxclass: NX_INT
    value: $post-run:d_ord
    dtype: int32
description:
  nxclass: NX_CHAR
  value: $pre-run-md:description
  dtype: str
"""
```

Figure 5: *Pydantic* schema represented as a YAML-formatted string.

Comments on the Code Lines in Fig. 5

The `nxclass` values such as `NX_FLOAT`, `NX_INT` and `NX_CHAR` refer to standard *NeXus* data types, which define the expected type of each field in the schema according to the [NXDL type system] [14].

1: `Mono_nxschema` is a name of the variable that stores the *Pydantic* schema as a YAML-formatted string.
2: Specifies the name of the *Pydantic* model used to validate the schema.
3: `nxclass`: Specifies the name of the *NeXus* base class used to create *NeXus* group.
4: `energy` is the name of the field to be created within the `NXmonochromator` group. Note that this name corresponds to the field name defined in the *NeXus* base class *NXmonochromator*.
5: `nx_class`: Specifies the attribute (`NX_FLOAT`) to be associated with the `energy` field.
6: `value`: Specifies the placeholder for the value of the `energy` field, which is populated with data during post-run dynamic data processing. Note that the name `en`

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUPD110

used in the placeholder `$post-run:en` corresponds to the component name in the device class `Mono`.

7: `dtype`: Specifies datatype of the field `energy` as defined in the *NeXus* base class *NXmonochromator*.

8-9: *attrs*: Introduces the specification of the attributes associated with the `energy` field, in accordance with its definition in the *NeXus* base class *NXmonochromator*.

10: `GRATING` is the name of the group to be created within the `NXmonochromator` group. Note that this name corresponds to the value defined in the *NeXus* base class *NXmonochromator*.

11: `nx_class`: Specifies the attribute (`NXgrating`) to be associated with the `GRATING` group.

12: `diffraction_order` is the name of the field to be created within the `GRATING` group. Note that this name corresponds to the value defined in the *NeXus* base class *NXgrating*.

13: `nx_class`: Specifies the attribute (`NX_INT`) to be associated with the `diffraction_order` field.

14: `value`: Specifies the placeholder for the value of the `diffraction_order` field, which is populated with data during post-run dynamic data processing. Note that the name `d_ord` used in the placeholder `$post-run:d_ord` corresponds to the component name in the device class `Mono`.

15: `dtype`: Specifies datatype of the field `diffraction_order` as defined in the *NeXus* base class *NXgrating*.

16: `description` is the name of the field to be created within the `NXmonochromator` group. Note that this field is not defined in the *NeXus* base class *NXmonochromator*.

17: `nx_class`: Specifies the attribute (`NX_CHAR`) to be associated with the `description` field.

18: `value`: Defines the placeholder for the value of the `description` field, which is populated with data during pre-run data processing, as explained in "Static metadata assignment" under the '*Pre-run Actions Performed by the Preprocessor*' subsection (item 3). Note that the name `description` in the placeholder `$pre-run:description` corresponds to an attribute of the device instance metadata.

LOGGING

To facilitate debugging and ensure traceability of operations, a configurable logger can be initialized. This logger supports both console and file outputs, each with independently configurable log levels. To efficiently manage disk usage, log rotation is implemented and customizable, allowing specification of the number of backup files and their maximum sizes. This setup provides robust monitoring of the data processing pipeline and simplifies troubleshooting during both development and production phases.

FUTURE WORK

While the current implementation demonstrates the feasibility and advantages of schema-driven *NeXus* file generation, several directions for further development have been identified.

First, our approach currently extracts data exclusively from the *baseline* and *primary* event streams of the *Bluesky* event model. Extending support to an arbitrary number of event streams would make the solution more flexible and applicable to a wider range of experimental setups.

Second, the current implementation writes to the *NeXus* file only those elements that are explicitly defined in the schema. While this guarantees full control and compliance with *NeXus* specifications, it becomes impractical for devices that either expose a large number of components or have individual components containing extensive metadata—such as an `EpicsMotor`. A more scalable solution would involve introducing mechanisms to automatically include all information contained in the *Bluesky* documents, while still allowing explicit schema definitions.

Finally, the current workflow generates the *NeXus* file only at the end of an experimental run. For experiments where immediate feedback is essential, it would be advantageous to support incremental writing during execution, i.e., writing per event as the experiment proceeds. Such a capability would not only enable near real-time analysis of ongoing results but also minimize the risk of data loss in long-running experiments, where delaying file writing until the end could result in losing all collected data if a failure occurs.

CONCLUSION

We have presented Bluesky NeXus, a modular approach for generating NeXus-compliant files within the Bluesky data acquisition framework. By using YAML-based, Pydantic-validated schemas, the system ensures structural consistency with NeXus standards while accommodating diverse device configurations.

The two-stage workflow—pre-run for static metadata and post-run for dynamic data—provides a clear separation between initialization and data capture, enabling reliable, reproducible, and FAIR-compliant data storage. This schema-driven approach allows domain scientists to define NeXus structures declaratively, without requiring deep programming expertise, and integrates seamlessly into existing Bluesky workflows at large-scale facilities.

Additional details on application usage and implementation can be found in the project's README file.

REFERENCES

[1] D. B. Allan, "The Bluesky Project: A Multi-Facility Collaboration for Data Acquisition and Management", presented at ICALEPCS'19, New York, NY, USA, Oct. 2019, paper WESH3001, unpublished.

[2] M. Könnecke *et al.*, "The NeXus data format", *J. Appl. Crystallogr.*, vol. 48, no. 1, pp. 301–305, Jan. 2015. doi:10.1107/s1600576714027575

[3] M. D. Wilkinson *et al.*, "The FAIR Guiding Principles for scientific data management and stewardship", *Sci. Data*, vol. 3, no. 1, p. 160018, Mar. 2016. doi:10.1038/sdata.2016.18

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUPD110

[4] bluesky_nexus, https://codebase.helmholtz.cloud/hzb/bluesky/core/source/bluesky_nexus

[5] R. Müller *et al.*, "Experimental Data Taking and Management: The Upgrade Process at BESSY II and HZB", in *Proc. ICALEPCS'23*, Cape Town, South Africa, Oct. 2023, paper MO2AO04. doi:10.18429/JACoW-ICALEPCS2023-MO2AO04

[6] Pydantic, https://docs.pydantic.dev

[7] NeXus base classes, https://manual.nexusformat.org/classes/base_classes/index.html

[8] Ophyd, https://github.com/bluesky/ophyd

[9] Helmholtz-Zentrum Berlin, https://www.helmholtz-berlin.de/index_en.html

[10] NeXus base class: NXmonochromator, https://manual.nexusformat.org/classes/base_classes/NXmonochromator.html#nxmonochromator

[11] HAPPI - Heuristic Access to Positions of Photon Instruments, https://pcdshub.github.io/happi

[12] HDF5 for Python, https://www.h5py.org/

[13] NeXus base class: NXgrating, https://manual.nexusformat.org/classes/base_classes/NXgrating.html#nxgrating

[14] NXDL Data Types and Units, https://manual.nexusformat.org/nxdl-types.html