Synchrotron Radiation News

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/gsrn20

The Karabo SCADA System at the European XFEL

D. Göries, W. Ehsan, G. Flucke, N. Annakkappala, V. Bondar, R. Costa, S. Esenov, G. Giovanetti, D. Hickin, I. Karpics, A. Klimovskaia, A. Mahmud, A. Parenti, P. J. S. Prafulla, A. Samadli, H. Santos, A. Silenzi, M. Smith, F. Sohn, M. Staffel, A. Garcia Tabares, J. Bin Taufik, G. Varghese, C. Youngman & S. Hauf

To cite this article: D. Göries, W. Ehsan, G. Flucke, N. Annakkappala, V. Bondar, R. Costa, S. Esenov, G. Giovanetti, D. Hickin, I. Karpics, A. Klimovskaia, A. Mahmud, A. Parenti, P. J. S. Prafulla, A. Samadli, H. Santos, A. Silenzi, M. Smith, F. Sohn, M. Staffel, A. Garcia Tabares, J. Bin Taufik, G. Varghese, C. Youngman & S. Hauf (04 Jan 2024): The Karabo SCADA System at the European XFEL, Synchrotron Radiation News, DOI: 10.1080/08940886.2023.2277650

To link to this article: https://doi.org/10.1080/08940886.2023.2277650

OPEN ACCESS

TECHNICAL REPORT

The Karabo SCADA System at the European XFEL

D. Göries, W. Ehsan, G. Flucke, N. Annakkappala, V. Bondar, R. Costa, S. Esenov, G. Giovanetti, D. Hickin, I. Karpics, A. Klimovskaia, A. Mahmud, A. Parenti, P. J. S. Prafulla, A. Samadli, H. Santos, A. Silenzi, M. Smith, F. Sohn, M. Staffel, A. Garcia Tabares, J. Bin Taufik, G. Varghese, C. Youngman, and S. Hauf

Controls Group, European XFEL GmbH, Schenefeld, Germany dennis.goeries@xfel.eu

Introduction

The Karabo Control System was developed at the European X-ray Free Electron Laser (EuXFEL) starting in 2010 and was released as free and open-source software in June 2023. Karabo is a pluggable, distributed application management system forming a supervisory control and data acquisition (SCADA) environment as part of a distributed control system. Karabo provides integrated control of hardware, monitoring, and data acquisition and can facilitate online and pipelined data analysis on distributed hardware. Services exist for access control, data logging, configuration management, and failure event recovery.

European XFEL performed a survey in 2009 that looked at other well-known systems such as Tango [1], EPICS [2], and DOOCS [3] as possible control solutions for the planned facility. At that time, neither of these systems anticipated the high data volumes produced at the European XFEL, and hence the development of Karabo, a control system tailored to the facility's control and data acquisition requirements, was started. The seamless and built-in integration of large data processing is still a distinguishing feature of Karabo, which also today, the other systems frequently only offer as an extension.

Control requirements for the European XFEL

The European XFEL is a user research facility that provides high energy, coherent X-ray pulses with unparalleled brilliance at MHz rates. The facility employs a unique X-ray beam pulse structure with up to 27,000 photon pulses per second arranged into 10 Hz trains of pulses at 4.5 MHz [4]. European XFEL instruments use state-of-the-art, large-area 2D imaging detectors capable of recording images of scattered photons produced by a single XFEL photon pulse at burst imaging rates of up to 4.5 MHz [5]. The resulting high data rates are a challenge for the data acquisition services of any SCADA system. Additionally, the high peak brilliance of the facility caters to "single-shot" experimental methods, i.e., the X-ray pulse will destroy a given sample, and statistical significance is then achieved through many repetitions. For this to be viable, the control and data acquisition system must ensure that data from many sources readily correlates in time, e.g., through a globally unique identifier and low latency data and event propagation. Finally, the European XFEL as a so-called user facility: the scientific community submits proposals for measurements, which, given enough merit and technical feasibility, are performed during multi-day "beam times" on flexible end stations supporting multiple experimental methods. The control system therefore needs to provide a high degree of flexibility to cope with the dynamic nature of the experimental setups of this operation mode.

Karabo: Fundamental concepts and architecture

Karabo is a from-scratch, in-house development of a control system that:

- is scalable and can grow alongside the facility,
- has time correlation woven into its fundamental data model,
- can process data rates of tens of GByte per second at latencies of a few 100ms,
- and caters to dynamic experimental setups that change on different time scales,

while being highly reliable and resilient to failure events. In the following, we expand on core architectural and technological choices for implementing such a system.

The core entities of the Karabo ecosystem are defined as *devices,* which are instances of pluggable software components written in either Python or C++ (see Figure 1). Devices extend the functionality of the core framework, and interface to hardware, provide system services like data logging or configuration management or coordinate other devices. Multiple devices can run within a single software process called a *device server*. The interface of a device is exposed as a self-descriptive "*Schema*" and is modifiable at run-time. It includes *commands* and *properties* of a device, as well as *attributes* of the latter, such as value limits and access restrictions. Device properties comprise both configurable parameters and read-only values, which can for instance represent hardware values.

A fundamental property is the so-called *state*. A fixed subset of states defined in the framework determines which commands and reconfigurations are currently possible for a given device. Furthermore, every device can temporarily take exclusive control of other devices,

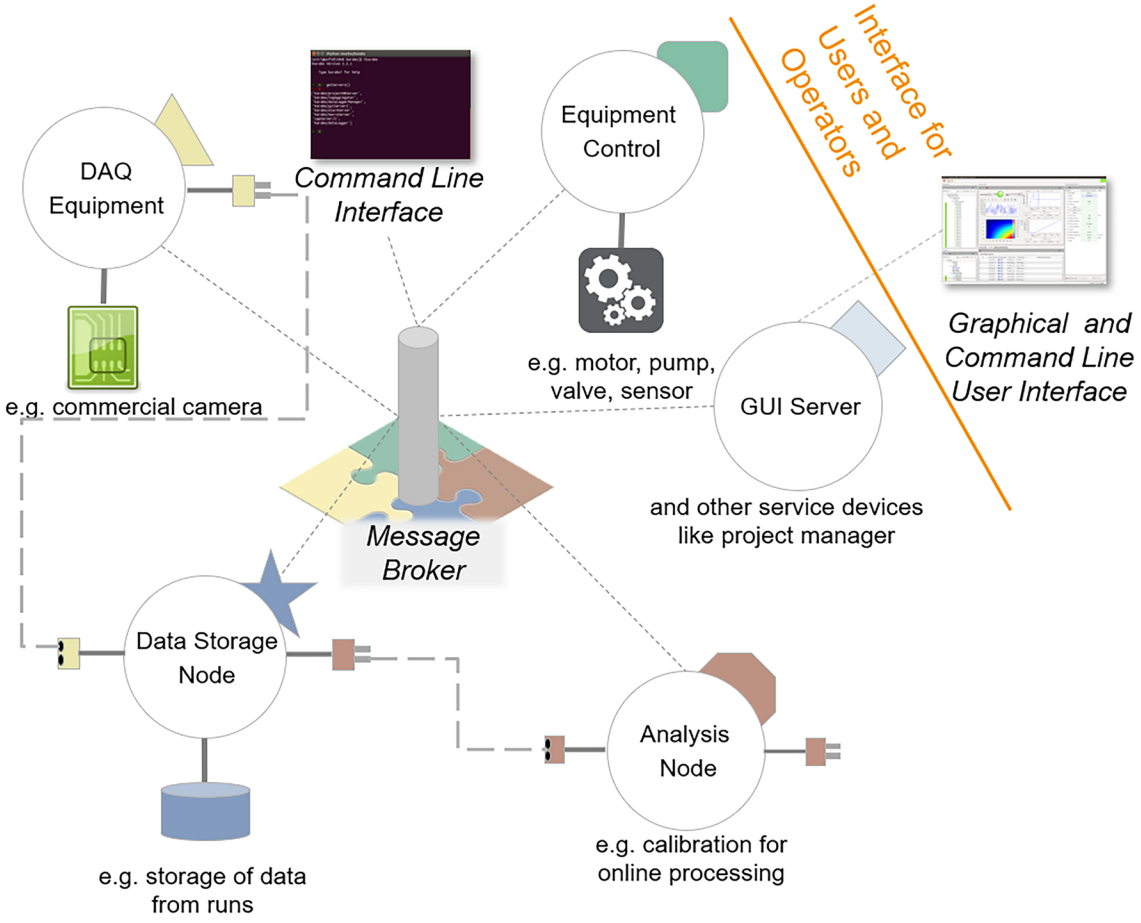

Figure 1: Fundamental architecture sketch of Karabo: a broker-based distributed and extendable system with peer-to-peer connection for high data rates.

such that only it can reconfigure the device and call its commands. This mechanism is beneficial for software that coordinates other devices, such as a beam feedback system, and wants to ensure that no other (accidental) inputs can happen.

When a device is instantiated, it broadcasts its unique *instance identifier* to the distributed system alongside supplementary information - the device is now *online*. Any device instance thus knows of every other instance at any time, resulting in a dynamic topology without the need for a central governing database. The instance information informs of interfaces a device implements, e.g., whether it complies with a standardized motor or detector interface. Using this, the topology can be filtered by functional roles, e.g., to determine which devices can participate in a *scan*. Finally, every instance announces its shutdown to all other instances - it is then *offline*.

Broker-based communication and asynchronous event-driven messaging

The distributed components of Karabo communicate by asynchronously exchanging messages via a central message broker (see Figure 1). A virtual namespace exists on the broker for every Karabo installation, referred to as a *Karabo topic,* within which instance identifiers are unique. To call instance methods throughout the distributed system, one can register them as so-called "*Slots*." Direct (unicast) slot calls support error propagation over instance boundaries from one device to another. Broadcast slot calls, i.e., multicast slots, are used to update the system topology dynamically.

Karabo implements a completely event-driven publish and subscribe signal-slot messaging pattern on top of the distributed broker messaging. By subscribing to a signal of another instance, a device does not need to poll for remote updates regularly. Instead, it is automatically informed about updates when they happen. This includes timing information which comprises a timestamp and a unique *timing identifier*. At European XFEL, the identifier reflects the train (bunch) frequency and is thus commonly referred to as the *train ID*. Outgoing updates result in a single broker message, even if many devices are subscribed to the update.

The content of any distributed message is a so-called *Karabo Hash*. This hierarchical key/value container supports element-specific attri-

bute assignment. The keys are strings and the following data types are supported as values: integers, floating points, strings, the Hash itself, vectors of all of these, the Schema and a special container for multi-dimensional arrays.

Originally, Karabo was developed using the Java Messaging Service (JMS) broker [6] using an implementation of the Open Message Queue, OpenMQ(C), library. However, the latter is no longer actively supported. Recent Karabo versions have added support for communication via the Rabbit MQ broker [7] using the AMQP protocol [8], and through this can support broker fail-over scenarios. The transition of the European XFEL installations to this technology is foreseen to be completed by the end of 2024.

Pipelines for large and fast data

Transfer of large data volumes from detectors and data processing avoids broker-based messaging and uses so-called Karabo *pipelines* instead. These pipelines are direct TCP connections between *output* and *input* channels. Serialization is optimized to avoid copies of extensive data arrays. The configurable message patterns are:

- input channels receive a copy of all data or share the data with other channels to distribute the load,
- an input channel receives data from several outputs, e.g., to collect from shared processing.

There are several ways an output channel reacts in case it sends data faster than the receiving channel(s) can process it: *drop* data, en*queue* them, or *wait* until the input channel is ready. To preview data at a reduced rate, the availability of a channel for the subsequent data item can be artificially delayed [9].

The Karabo GUI: a single extensible operator interface

Karabo provides a single graphical user interface (GUI) for control tasks (see Figure 2). This *Karabo Cockpit* is the preferred way to interact with the Karabo ecosystem. It is implemented in Python and uses the Qt [10] layout manager.

The GUI connects to so-called GuiServer devices over TCP/IP and then runs under authenticated access levels. Consequently, GUI clients do not communicate with the distributed system using the message broker, even if data transfer relies upon Karabo Hash serialization. The portability of the GUI application to different operating systems benefits from this design. TCP connections additionally facilitate an increasingly important feature: remote access to the system, using e.g., SSH tunnelling. Configuration updates are typically throttled to 2 Hz. Pipeline data is dynamically throttled to avoid saturating the client-server connection. To achieve this, the GuiServer will not forward data from a specific pipeline unless a client confirms that the previous data item has been processed. This ensures that client applications can view multiple camera screens

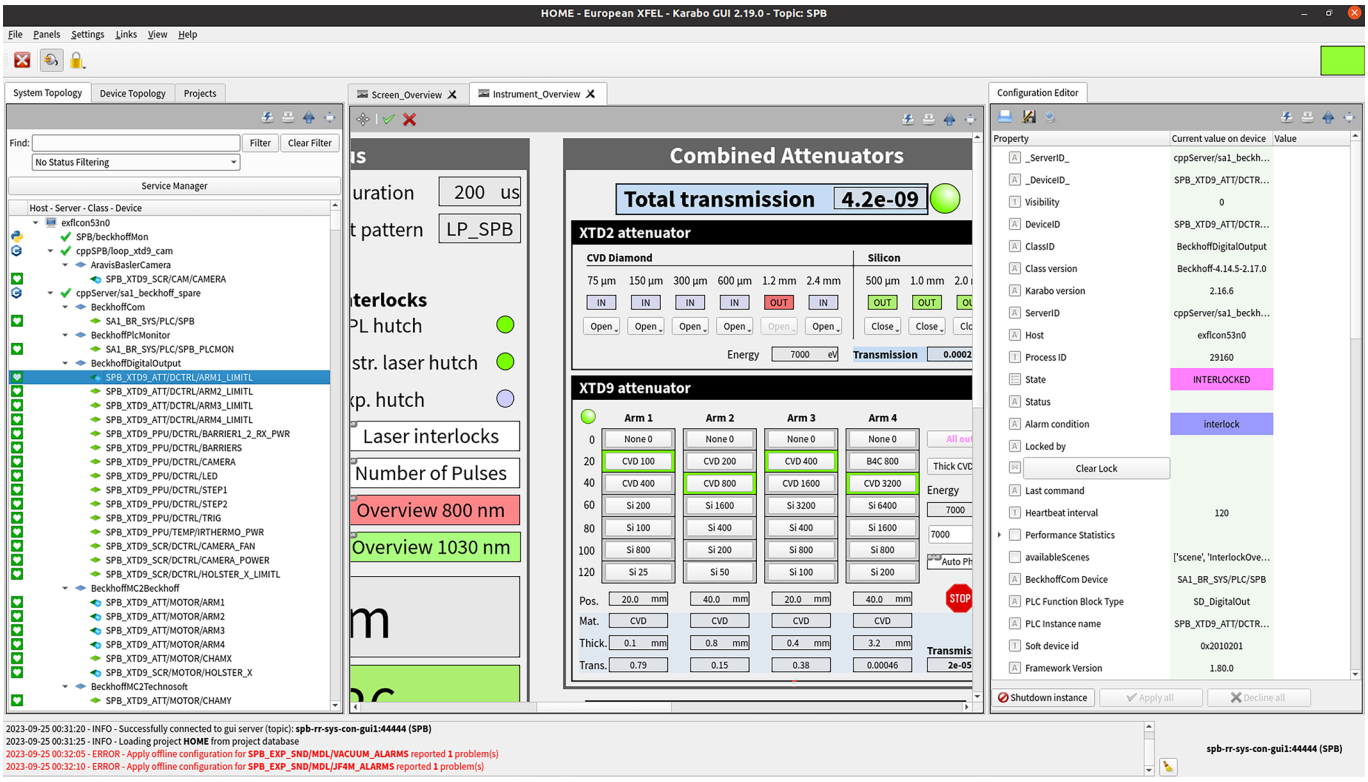

Figure 2: A connected Karabo GUI application with docked panels; left: navigation panels (system topology and projects); center: scene panels; right: configuration panel.

without being overloaded. Once a device is not of interest to the operator anymore, the client applications unsubscribe from the respective configuration updates automatically.

The GUI application features multiple panels optimized to perform different tasks. Distinct features such as projects and scene panels coexist with control system essentials, such as a *Configuration Panel* to modify the configuration of online and offline devices, and a system topology view. Karabo *projects* contain lists of devices, their associated configurations, scripting *macros*, and so-called *scenes*. Projects are stored in a non-relational database [11] and can be dynamically created as operators see fit.

By interactively dragging any device property from the Configuration Panel onto a *scene*, synoptic views are created, consisting of controllers (widgets) appropriate to the property's data type, which are added to a scene-model [12]. Operators generally interact with the control system through these scenes or the configuration panel.

With the combination of projects and scenes, operators can build versatile user interfaces without coding, and style them further in SVG graphics editors, since scenes can be translated into an SVG representation. Alternatively, the programmatic composition of scene model representations is possible, and representations of the scene model can be embedded into device code. These device-provided scenes can be accessed by double-clicking on an instance in a topology panel of the GUI.

Use cases at the European XFEL

In the above, we have mainly considered the core Karabo framework. However, the framework is not responsible for integrating hardware or implementing specific operation procedures, nor defining aspects of a scientific control system. As described above, such functionalities are provided by plug-ins called devices. These are implemented using the three application programming interfaces (APIs) the framework provides: C++, Python-bound (Bound), and Python-Middlelayer (MDL). Each API has distinguishing features:

- C++ is beneficial for high-performance applications, or for integrating third-party C or C++ libraries;
- Bound provides high-performance pipelined processing when an application additionally needs to leverage Python packages such as SciPy [13] or NumPy [14];
- The Middlelayer API is a natively "Pythonic", low-boilerplate integration option that excels in rapid development and iteration cycles.

European XFEL typically operates on a weekly schedule, with user experiments lasting 5 to 6 days. Consequently, the hardware setup at the beamlines changes frequently and results in the need to regularly interface additional hardware and define new procedures in the control system. The three APIs described above are widely used to enable this ever-changing variety of devices ranging from the integration of the previously mentioned large area detectors, to digitizers and commercial

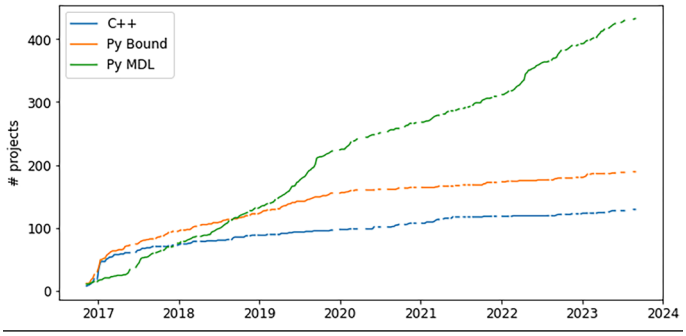

Figure 3: Number of device classes developed for the three Karabo APIs over time.

cameras, as well as programmable logic controllers (PLC) and scientific tabletop instrumentation. Base packages facilitate the implementation of standard protocols (e.g., SCPI, GenICam) or features (e.g., virtual axes). Originally scoped for procedure definitions, the Middlelayer API is increasingly used for all purposes, including to interface hardware directly (see Figure 3). A vital part of this API and its defining aspect are *device proxies*. These allow a straightforward implementation of high-level tasks which require the orchestration of other devices. In the following, we introduce examples of important devices used at the European XFEL, exemplifying the possibilities of the Karabo framework.

Data acquisition to HDF5 and event-driven to the InfluxDB time series database

Karabo keeps historical data about the devices in a topic. Properties and schemas of devices are stored whenever a device instantiates or any property or device schema changes. In consequence, the historical evolution of any property can be inspected, and a device configuration can be restored to any point in time.

Two logging backend systems exist for device data: one based on custom text files and one that uses an Influx time series database [15]. While a text file is adequate for the needs of small installations, it will not scale to the operational needs of a larger research facility. The Influx-based system addresses these limitations and supports European XFEL operations as the primary data logging backend since 2020. Each month, about 10 billion property updates are stored in the database [16] and retained for at least 3 years. Logging data is mainly accessed from the Karabo GUI Client and, in the case of Influx, from the Grafana web interface [17]. Grafana is the primary monitoring tool used by the Data Operation Center of the European XFEL and thus plays a fundamental role in daily facility operations.

The Karabo data logging subsystem is implemented as a set of common high-performance C++ devices for either backend. Multiple instances of so-called data logger devices perform data ingestion. The Influx-based system sustains an average ingestion rate of about 20 MByte per second [16].

A run-based data acquisition system (DAQ) is additionally required to perform experiments at the European XFEL. Here, a run is considered a continuous acquisition period of mostly stable experimental parameters [18]. This system stores broker-transferred information, data from pipeline connections, such as camera images and digitizer traces, and images from the bespoke 2D MHz-imaging detectors at the facility [2]. The system is mainly implemented using the C++ API and can store data rates of up to 20 GBytes per second into HDF5 files [18, 19]. Additionally, the system propagates data into the control system through its monitoring output channels.

Online detector calibration

In order to make decisions during ongoing experiments, scientists require near real-time feedback from diagnostics and detectors, including the bespoke MHz-imaging capable 2D detectors. Especially for the latter, but also for detector technologies such as Jungfrau, and ePIX, calibration of the raw detector data is necessary before decisions based on this data can be made. An online calibration pipeline, implemented in the Karabo Bound API, which utilizes GPUs is able to provide calibrated online image streams at rates of greater than 3000 megapixel images per second, exemplifying the throughput of Karabo pipeline connections. Further details can be found in [20, 21].

Failure recovery through the recovery Portal

The Recovery Portal is a Karabo device that provides an interface to retrieve device configurations for a given point in time from the Influx database and apply these configurations to a selection of devices in a topic. The user interface consists of two scenes: the comparison scene and the recovery scene.

The comparison scene allows an operator to select a point from the past and filter for device names or types for which they would like to view configurations. Past and present configurations of the chosen devices are retrieved and can be compared. The operator can use the resulting overview to determine configuration changes between the two time points. A more detailed comparison is available via a dialog, such that individual property changes can be inspected.

Through the recovery scene, the operator can retrieve device configurations from the past and additionally apply configurations to selected devices. This interface exposes the same point-in-time selection functionality and device name or type filters as for the comparison case. Once devices or groups of devices are selected for recovery, the portal applies the past configuration to the device. If a device is not online, the tool can attempt to instantiate the device first. This latter function is convenient for restarting the control system after software or facility maintenance and upgrades or in large-scale failure events such as a power cut. When the maintenance work is complete, a point-in-time just before the shutdown is selected, and all devices running then are restarted and configured as before the intervention. After recovery, a summary table indicating the success or failures of the batch reconfiguring is shown.

Integrating with other control systems – the Karabo DOOCS interface

The DOOCS (Distributed Objected-Oriented Control System) framework [3] was developed from 1992 onwards at DESY. DOOCS was initially used for test stands and was later ported to the HERA proton storage ring. Currently, it serves as the control system of the FLASH and European XFEL accelerators and will be used for the future Petra IV light source. As the accelerator control system, DOOCS monitors many diagnostics relevant to performing experiments at the European XFEL. To access this data from within Karabo, a DOOCS interface exists. The DoocsGate C++ class, based on the DOOCS library, connects to any DOOCS location and gives access to its properties as Karabo types. The class is also available in Python using a Boost Python [22] binding.

Specific Karabo devices give access to the most commonly used types of DOOCS servers, e.g, those controlling the timing boards and the X-Ray Gas Monitors (XGM). Additionally, a generic device called “DoocsMirror” enables instrument staff to seamlessly import DOOCS devices into the Karabo ecosystem without requiring involvement of experts or the need to write specific code. A list of locations and properties suffices to map properties from multiple DOOCS locations to corresponding Karabo properties. Slowly updating properties are propagated using broker messaging, while fast and large data types, such as vectors and images, are transferred through Karabo pipelines. DoocsMirror takes care of proper time information: if a DOOCS property has a non-zero train ID associated, this ID is retained as update time of the corresponding Karabo property. If DOOCS does not provide a train ID for a property, the device uses the unix epoch information supplied by DOOCS to calculate the corresponding train ID.

European XFEL has seven distinct Karabo broker topics designated for the instruments and an additional three for the photon tunnel systems. Connections between DOOCS and Karabo are established in a dedicated “DOOCS” Karabo installation, facilitating load/throughput optimization between the two control systems at a single tuning point. The device instances from this DOOCS topic are then replicated to other topics using the ‘DeviceClone’ package. This package enables the replication of a device instance from one topic to another by transmitting data from the source topic via a TCP channel to a server instance in the destination topic. Device clones can operate in a unidirectional mode, providing access to read-back properties, or a bidirectional mode, allowing slot calls and value modifications.

The Karabacon scantool

Experimental data collection at synchrotrons and free electron laser facilities frequently requires the simultaneous motion of several motors and actuators and coordinated data acquisition from various sources (cameras, pixel detectors, and digitizers). Karabacon [23] is a Middlelayer device to orchestrate such operation and is accessible using:

- a graphical user interface through which devices and parameters of a scan are defined, the execution is controlled, results are plotted, and a history of scans is tracked (see Figure 4);
- a command line interface (CLI) and macros to integrate scans in higher-level automation routines.

Karabacon can execute absolute and relative 1D and 2D step scans combining up to four motors and six data sources. A built-in motor interface enables the safe operation of many classes of Karabo devices, and a custom motor interface can be configured to support most other devices. Any scalar or vector device property or output channel can be a data source. Additionally, continuous fly scans and time scans are possible.

Extensions to the core system provide additional functionality:

- A scan history summarizes the metadata of previous scans, such as scan settings, progress, status, and basic DAQ info. The history feature can return to historic scan settings and relaunch an earlier scan.
- Other extensions modify scan steps based on user-defined transformations and define data source normalization logic.
- An aligner tool estimates essential plot characteristics (minimum, maximum, peak, valley, etc.) and moves motors to the corresponding positions.
- Scan templates store predefined scan configurations. The templates are based on a historical time point, and loading a template sets all Karabacon parameters accordingly.
- Custom scan patterns are supported by uploading, e.g., NumPy or CSV files or retrieving external pattern specifications from other devices.

Karabacon is deployed in all installations at the European XFEL and used in daily operation as a standard tool for beamline and instrument commissioning and scientific data collection.

Outlook

Over 13 years of development and 6 years of facility operation, Karabo has matured into a reliable control system that fits the requirements of the European XFEL. In June 2023, it was published as free and open-source software on GitHub.com and continues to be actively developed, both at the framework level and the numerous devices. Significant feature enhancements are being developed alongside a continu-

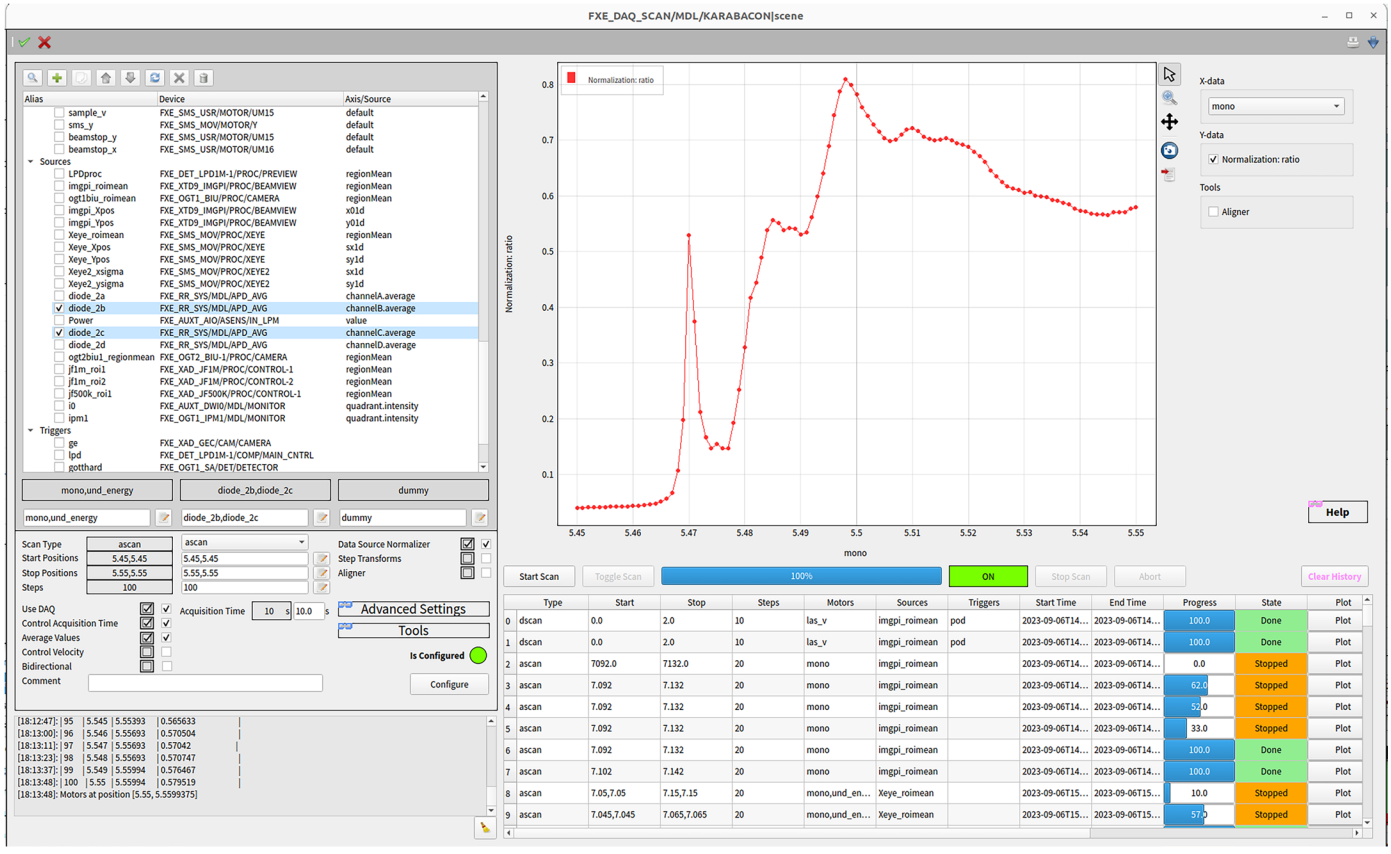

Figure 4: Scene panel of the scan tool Karabacon in use during a beam time at the FXE instrument. The motor, data source and trigger selection is on the top left corner. On the bottom left the scan parameters can be configured and on the right the scanned data can be viewed and replayed.

ous effort to address technical debt and introduce modern foundational technologies, such as for the broker or transitioning from *boost::python* to *PyBind11* [24]. A web-based user interface and REST APIs for the core system will be rolled out and expanded from 2024 onwards. This enhanced support for web technology will be complemented in 2024 by a lean authentication interface, which, while mainly aimed at auditing interactions with the system, will support authorization at the edges, e.g., the web interface or the GUI server interface. The device server logs will be moved from rolling text logs to make use of the ELK stack [25], greatly enhancing log introspection possibilities.

The Pythonic Karabo Middlelayer API is planned to be made available as a stand-alone Python package to ease implementations on other operating systems such as Windows or Raspberry Pi. The Middlelayer API's firm adherence to established Python best practices means that aside from the low-entry threshold for Python-fluent newcomers to the Karabo world, foundational large language models such as OpenAIs GPT series [26] can write Karabo device templates from prompt input alone, i.e., without additional training. AI-based documentation assistants similarly work well with Karabo's code base. We plan to expand on opportunities such integration provides for our development workflows in the future.

Conclusion

Karabo is a mature supervisory control and data acquisition system and the main user interface at the European XFEL to carry out scientific experiments. The control system has stabilized throughout the commissioning period and early operation, and performance has significantly increased.

Karabo has been tailored to the data requirements of large-scale research facilities: data can be correlated through a unique timing identifier, and high-performance data logging and acquisition systems exist. The event-driven nature of the system ensures that data traffic is minimized while significant changes are reliably propagated. The system is freely available under a mixed MPL 2.0 and GPLv3 license at www.github.com [27].

Disclosure statement

No potential conflict of interest was reported by the authors. ■

References

1. Tango Controls System, 2023, https://www.tango-controls.org/
2. Epics Controls System, 2023, https://epics-controls.org/
3. O. Hensler, and K. Rehlich, in *15th Conference on Charged Particle Accelerators*, Dec. 1996, pp. 308–315.
4. M. Altarelli, *Nucl. Instrum. Methods Phys. Res, Sect. B* **269** (24), 2845 (2011). 10.1016/j.nimb.2011.04.034
5. M. Kuster et al., *Synchrotron Radiat. News* **27** (4), 35 (2014). doi: 10.1080/08940886.2014.930809
6. M. Hapner, *Java Message Service API Tutorial and Reference: Messaging for the J2EE Platform* (Addison-Wesley Professional, Boston, MA, 2002).
7. RabbitMQ, 2023, https://www.rabbitmq.com/amqp-0-9-1-reference.html
8. S. Vinoski, *IEEE Internet Comput.* **10** (6), 87 (2006). doi: 10.1109/MIC.2006.116
9. S. Hauf et al., *J. Synchrotron Radiat.* **26** (5), 1448 (2019). doi: 10.1107/S1600577519006696
10. The Qt Company, 2023, September https://www.qt.io
11. W. Meier, in *Net. ObjectDays: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World* (Springer, Berlin, Heidelberg, Oct. 2002), pp. 169–183. doi: 10.1007/3-540-36560-5_13
12. Pydantic, 2023, September https://pydantic.dev
13. P. Virtanen et al., *Nat. Methods* **17** (3), 261 (2020). doi: 10.1038/s41592-019-0686-2
14. C. R. Harris et al., *Nature* **585** (7825), 357 (2020). doi: 10.1038/s41586-020-2649-2
15. InfluxData, InfluxDB, , Sep. 2023, https://www.influxdata.com
16. G. Flucke et al., Karabo Data Logging: InfluxDB Backend and Grafana UI, 2021.
17. M. Chakraborty, and A. P. Kundan, in *Monitoring Cloud-Native Applications: Lead Agile Operations Confidently Using Open Source Software* (Apress, Berkeley, CA, 2021), pp. 187–240.
18. European XFEL GmbH, Scientific Data Policy of European X-Ray Free-Electron Laser Facility GmbH, 2017, https://in.xfel.eu/upex/docs/upex-scientific-data-policy.pdf
19. M. Folk et al., in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, Mar. 2011, pp. 36–47. doi: 10.1145/1966895.1966900
20. P. Schmidt et al., in *Frontiers in Data Science* (CRC Press, Boca Raton, FL, Sep. 2023).
21. D. Hammer et al., in *X-Ray Free-Electron Lasers: Advances in Source Development and Instrumentation VI*, SPIE, Jun. 2023, Vol. 12581, pp. 78–84. 10.1117/12.2669491
22. B. Schäling, *The Boost C++ Libraries* (XML Press, Laguna Hills, CA, 2014).
23. European XFEL GmbH, Karabacon, Sep. 2023, https://rtd.xfel.eu/docs/scantool/en/latest/
24. W. Jakob, J. Rhinelander, and D. Moldovan, pybind11 – Seamless operability between C++ 11 and Python, 2017, https://github.com/pybind/pybind11
25. Elastic Stack, 2023, https://www.elastic.co/elastic-stack
26. OpenAI, 2023, GPT-4 Technical Report, arXiv e-print, doi:10.48550/arXiv.2303.08774
27. European XFEL GmbH, Karabo, 2023, https://github.com/European-XFEL/Karabo