ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THPD053

BLUESKY QUEUE SERVER FOR BEAMLINE CONTROL AT NSLS-II

D. Gavrilov, T. Caswell, A. Sligar, M. Rakitin
NSLS-II, Brookhaven National Lab, Upton, NY, USA

© Content from this work may be used under the terms of the CC BY 4.0 licence (© 2025). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

Abstract

Bluesky is a Python-based framework for experiment orchestration that is widely used at synchrotron facilities around the world. Queue Server (QS) is an essential component of Bluesky software stack that supports high-level functionality, such as control over the environment for executing Bluesky plans, enqueueing plans, executing and managing the plan queue, monitoring and controlling running plans, etc. The functionality is exposed via comprehensive set of APIs, which are designed to support wide range of workflows. QS is successfully used in applications, such as GUI-based and remote control, AI-driven and multimodal experiments.

INTRODUCTION

Bluesky [1] is a Python-based open-source software developed at NSLS-II as part of the effort to implement standard approach to experiment control at the beamlines. Source code for all components of Bluesky software stack is available from Bluesky Collaboration GitHub organization [2].

The components of Bluesky stack are shown in Fig. 1. The purpose of the Ophyd layer at bottom of the stack is to represent controlled devices as Ophyd objects, i.e. instances of Python classes with standard interface. The interface hides the underlying device implementation and allows standard access to the device. Ophyd devices may communicate with hardware directly or using higher level protocols, such as EPICS, REST API etc. The Bluesky layer supports execution of step-by-step logic of the experiments and communicates with the hardware using Ophyd objects. The experiment logic is expressed in the form of Bluesky Plans executed by Run Engine, which is part of Bluesky package.

Ophyd and Bluesky packages are Python libraries that provide components for developing Ophyd objects and experimental plans and Bluesky Run Engine for execution of the plans. In the traditional Bluesky workflow, plans were executed by manually running commands in IPython environment. While being the most flexible and convenient for development purposes, this workflow was found not intuitive by many beamline users. Several custom GUI applications were also developed to support workflows for selected beamlines. Those applications are running the control and GUI code in the same process and suffered from reliability issues.

Queue Server (QS) was recently added to the stack as part of transitioning to service-based architecture for experiment control. The purpose of QS is to implement server-client framework, in which execution of Bluesky plans on the server is fully controlled by client software using rich set of APIs. The client software may include GUI applications, user-developed scripts, autonomous agents, etc. Strong separation between the clients and the server performing low-level control operations increases reliability and robustness of the system. The architecture implements security features, including authentication and access control.

QUEUE SERVER OVERVIEW

Components of Queue Server

QS consists of the following components: Run Engine (RE) Manager [3], Bluesky QS API Python package [4], Bluesky HTTP Server [5]. Also, Bluesky-Widgets package [6] includes a number of PyQt widgets for controlling QS.

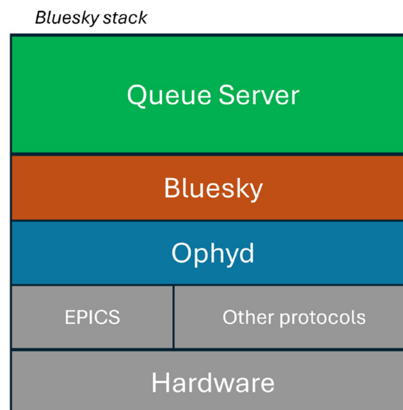

Figure 1: Components of the Bluesky Stack.

RE Manager is an execution engine for Bluesky plans. It is the core component of QS and can be started as a service or as a standalone application. It also supports complementary functions, including simple queueing of plans, execution of Python scripts and functions, etc. RE Manager exposes a rich set of APIs to client applications. Bluesky HTTP Server implements REST API for securely accessing RE Manager and exposing the QS on a wider network. The HTTP Server is easily extensible with custom routers to implement workflow-specific logic. Bluesky QS API is a Python package for communicating with QS using REST API via the HTTP server or directly with the RE manager. The package implements features for efficient communication with QS, such as local data caching, status monitoring, etc. Both thread-based and *async* versions of the API classes are available. Bluesky-Widgets package provides developers with general purpose PyQt widgets for controlling QS. The package also contains a generic *queue-monitor* application, which is useful for monitoring of QS in simple workflows.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THPD053

API-Based Workflow Using Queue Server

All functionality of QS is accessible using API requests. Multiple clients may connect to QS simultaneously. The workflow for API-based control is shown in Fig. 2. RE Manager consists of two main modules: the Manager and the Worker. The Manager is responsible for processing API requests from clients, queueing of plans and creating and maintaining the plan execution environment. The Worker runs in a separate disposable process, which is created, closed or terminated by the Manager in response to API requests from clients. After the Worker process is created in response to *environment_open* request from a client, the startup code is loaded into plan execution namespace. The startup code contains instances of Ophyd devices, Bluesky plans and Run Engine. RE Manager can load startup code represented as an IPython startup directory, Python script or installed Python mode.

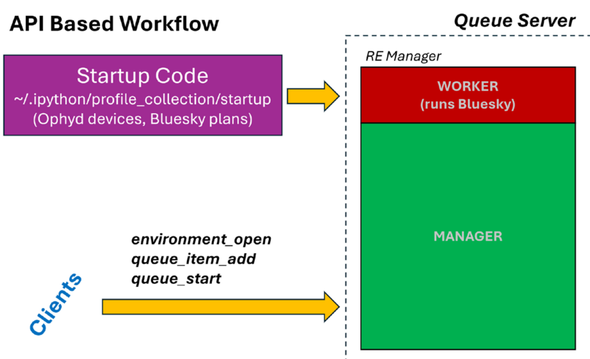

Figure 2: API-based workflow using Queue Server.

Depending on configuration, RE Manager creates plan execution environment. For debugging and development the IPython kernel mode is provided. When enabled, the in-process IPython kernel is created by the worker and the startup code is loaded as part of the kernel initialization. External applications, e.g. Jupyter Console, may connect directly to IPython kernel and access the namespace. This feature is useful for interactive Bluesky plan development or troubleshooting.

Once the startup code is successfully loaded, RE Manager is ready to execute Bluesky plans. In a typical workflow, the plans are placed into a queue, which is maintained by the Manager. For example, a *queue_item_add* API adds a plan at specified position in the queue. The queue is mutable and can be modified at any time.

Queue execution is started by a client by sending *queue_start* API request. Once the queue is started, execution of the queue until all plans in the queue are completed, queue is stopped by a client or one of the plans fails. As plans are executed, they are removed from the plan queue and added to plan history. The history items contain additional information, such as the completion status, list of run IDs produced by the plan, traceback for failed plan etc.

RE Manager supports additional features for supporting special workflows. Those features include special queue modes: the queue may be configured to execute a set of plans in the loop, ignore failed plans or start automatically once a plan is added to an empty queue. RE Manager also supports API for executing individual plans (bypassing the queue), uploaded Python scripts, or functions defined in the startup code.

Clients can monitor the state of the server by polling the status of RE Manager. The status information may be used to monitor the progress of execution of the queue and the running plans. The status also indicates when large data structures, such as plan queue or plan history, are changed at the server. The data structures can be downloaded by the clients only when they are updated at the server, thus reducing consumed network bandwidth. The monitoring features provide sufficient information for intelligent client software, such as autonomous agents, to observe execution of the experiment and configure the server for future steps as the new data become available.

Queue Server API Overview

RE Manager supports rich set of 45 API functions described in detail in the documentation [7]. The functions combined in 12 groups are shown in Fig. 3.

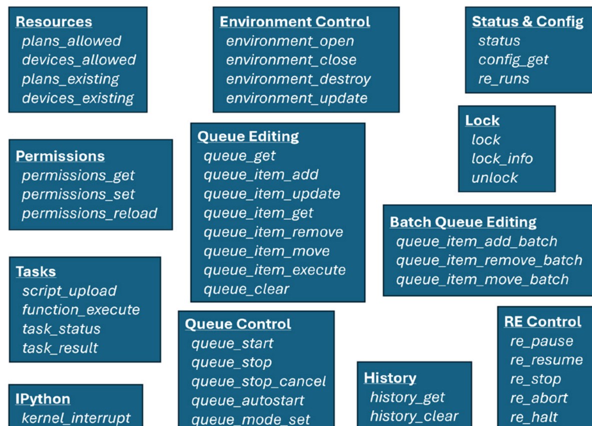

Figure 3: Queue Server API separated into groups.

Environment Control APIs allow clients to open, close or 'destroy' the environment. The environment can be orderly closed only when it is in *idle* state, i.e. no plans are executed. If the environment becomes unresponsive because of a bug in user software (Bluesky plan), RE Manager can be reset by terminating the worker processes using *environment_destroy* API.

APIs in the **Resources** group return descriptions of plans and devices extracted from plan execution namespace. The descriptions can be used by clients for validation of the plans before the plans are submitted to the queue. Clients may request a list of all existing plans or devices from the namespace or only the plans and devices that the particular user group is allowed to access.

APIs in the **Permissions** group allow to upload, download or reload the configuration that defines user group permissions. The permissions are used by RE Manager to generate lists of allowed plans and devices.

Queue Editing APIs are used to manipulate items in the plan queue. The API allow adding, removing, modifying (update) or moving items within the queue. The operation of clearing the queue removes all items from the queue.

© Content from this work may be used under the terms of the CC BY 4.0 licence (© 2025). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THPD053

© Content from this work may be used under the terms of the CC BY 4.0 licence (© 2025). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

The *queue_item_execute* API executes a single plan bypassing the queue. The **Batch Queue Editing** operations for adding, removing and moving items are 'atomic', i.e. they cannot be interrupted by other API calls, and they modify the queue only if operation succeeds for every item in the batch.

Queue Control APIs allow clients to start and stop execution of the queue. The operation of stopping the queue does not stop currently running plan, which is allowed to run to completion. The request to stop the queue remains pending until then and can be cancelled. Additional APIs allow setting the queue mode and enabling queue autostart mode.

Run Engine (RE) Control APIs allow clients to interrupt and resume execution of a currently running plan. The commands are passed directly to the Run Engine. A client can pause a currently running plan by sending *re_pause* API request with options of immediate or deferred pause. A paused plan can be resumed or stopped (the options are stop, abort, halt). If a plan is stopped, then execution of the queue is also stopped.

History APIs allow clients to download contents of the plan history and clear the plan history. The *history_clear* API can be used to trim the history to the desired size by passing the size or UID of one of the history items with the request.

APIs from the **Status & Config** group allow clients to request RE Manager status (*status* API), read the manager configuration (currently *config_get* API returns connection info for IPython kernel) or the list of runs produced by the currently running plans (*re_runs* API).

APIs from **Lock** group allow clients to temporarily lock certain functions of RE Manager with the password. The lock can be applied to operations on the environment, on the queue or on both. The environment operations include all API that may start or affect execution of the code in the worker environment. The queue operations include all operations that modify contents of the queue. Locked operations are still accessible to the clients if matching password is passed with API requests.

In addition to plans, RE Manager supports API for execution of Python scripts and functions (**Tasks** group). Clients can use the *script_upload* API to upload Python script (as a text). The script is then executed in the plan execution environment. The API can be used to add new plans and devices to the environment. The *function_execute* API allows to execute functions already defined in the environment. Tasks can be started in the main thread of the worker or in a separate background thread. Background tasks do not block the main thread and can run concurrently with plans.

Finally, the *kernel_interrupt* API in the **IPython** group is used to send interrupt to running IPython kernel to stop currently running cell.

HTTP Server provides the matching set of REST endpoints, allowing access to all RE Manager functionality. The REST endpoints are named consistently with RE Manager API, e.g. *environment_open* API is accessible using */environment/open* endpoint, *queue_item_add* API - using */queue/item/add* endpoint. HTTP Server also supports additional API for authentication and access control.

DEPLOYMENT CONFIGURATIONS

The Queue Server can be deployed in two configurations shown in Fig. 4. Configuration (A) is used for small systems with the server and all clients sharing the same local network or running on the same computer. The clients communicate with RE manager directly over 0MQ. Communication over 0MQ may be encrypted with fixed key, but no authentication or access control is supported. This configuration is suitable for small experimental setups, where restricting physical access to equipment provides sufficient security.

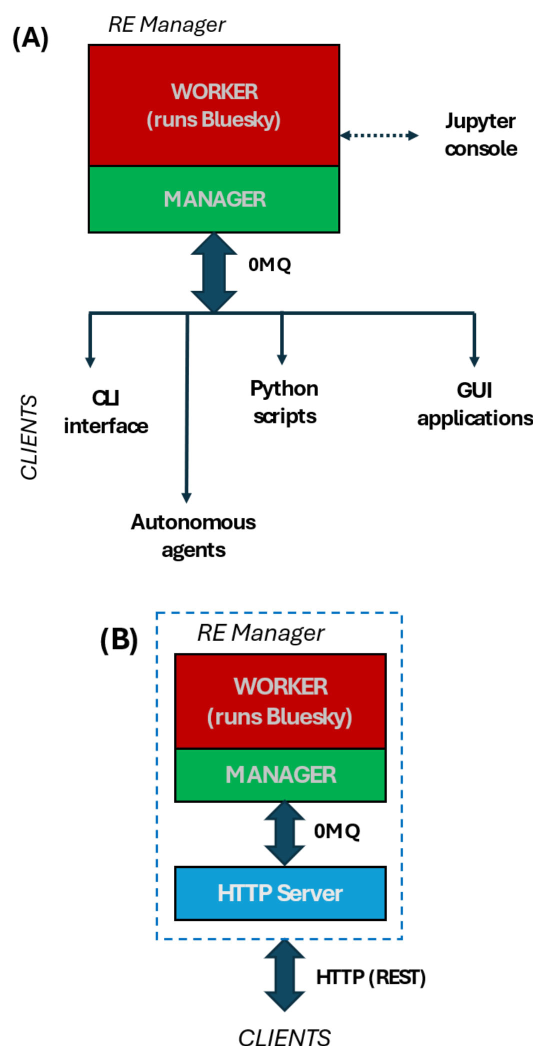

Figure 4: Deployment configurations for Queue Server: (A) direct communication with RE Manager over 0MQ for development and testing; (B) communication with RE Manager via HTTP Server using REST API suitable for secured production deployments.

In configuration B, local and remote clients communicate with HTTP Server using REST API. The HTTP server supports security features, including authentication and fine-grained access control, which restricts user access to

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THPD053

predefined sets of APIs. The HTTP Server forwards API requests to RE manager over an encrypted 0MQ channel. The HTTP Server also performs caching of large data structures, such as lists of available plans and devices, to minimize load on RE Manager.

QUEUE SERVER PYTHON API

Clients communicate with RE Manager by sending JSON API requests. The Bluesky Queue Server API package [4] is designed to provide Python application developers with convenient access to all API, avoiding low-level formatting of JSON requests. The package also provides additional features that simplify programming, such as built-in local caching of downloaded data, *wait_for* functions, etc. The API package works consistently over HTTP and 0MQ, allowing development of the code compatible with both protocols. REST API also includes support for authentication and access control. The package provides both thread-based and *async* versions of the API.

The example code in Fig. 5 demonstrates how the request to start execution of the queue can be sent to the Queue Server using the API package. After the request is submitted, the script is waiting for RE Manager to switch to *idle* state, which happens after all plans in the queue are completed or the queue execution is stopped. Both thread-based and *async* versions are shown in the example.

```
# Synchronous code (HTTP, 0MQ)
RM.queue_start()
try:
    RM.wait_for_idle(timeout=120)  # Wait for 2 minutes
    # The queue is completed or stopped, RM Manager is idle
except RM.WaitTimeoutError:
    # Processing of the timeout error

# Asynchronous code (HTTP, 0MQ)
await RM.queue_start()
try:
    await RM.wait_for_idle(timeout=120)  # Wait for 2 minutes
    # The queue is completed or stopped, RM Manager is idle
except RM.WaitTimeoutError:
    # Processing of the timeout error
```

Figure 5: Code example using QS Python API package.

CLIENT APPLICATIONS

While developers are encouraged to include support for QS in their custom applications, two generic clients are included with the package.

The *qserver* CLI tool is installed as part of *bluesky-queueserver* package [3]. The tool allows sending commands to QS directly from command line and provides access to most of the APIs. While controlling the server from CLI is not practical during user operations, it is still a useful tool for evaluation of QS or troubleshooting of the system. The commands for adding a plan to the queue and starting execution of the queue and the output to those commands are shown in Fig. 6.

The *queue-monitor* GUI application is installed as part of *bluesky-widgets* package [6]. The application includes widgets for controlling all basic functions of QS and support for most of the experimental tasks. It can also be used for monitoring RE Manager alongside custom applications optimized for particular workflows. The screenshot of the main window of *queue-monitor* application is shown in Fig. 7.

```
$ qserver queue add plan '{"name": "count", "args": [["det1",
"det2"]], "kwargs": {"num": 10, "delay": 1}}'
10:04:49 - MESSAGE:
{'item': {'args': [['det1', 'det2']],
          'item_type': 'plan',
          'item_uid': '0aa7f1be-3923-4d67-ba7b-b19d26ec6291',
          'kwargs': {'delay': 1, 'num': 10},
          'name': 'count',
          'user': 'qserver-cli',
          'user_group': 'primary'},
 'msg': '',
 'qsize': 1,
 'success': True}
$ qserver queue start
10:04:50 - MESSAGE:
{'msg': '', 'success': True}
```

Figure 6: *qserver* CLI tool.

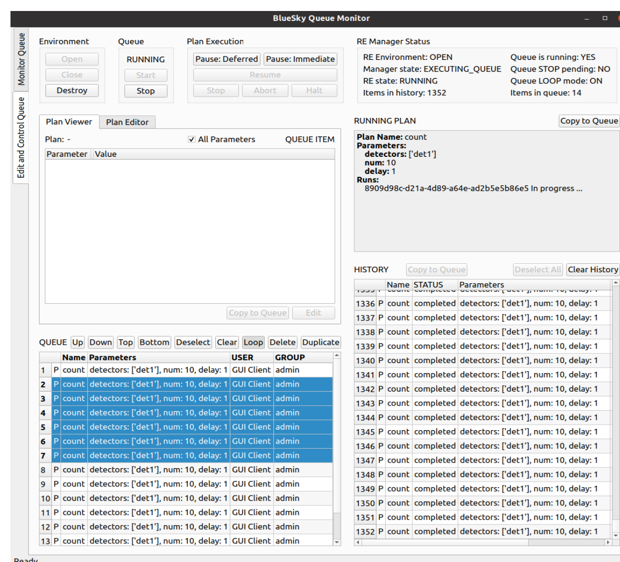

Figure 7: *queue-monitor* GUI application.

CONCLUSION

QS is an extension of the Bluesky software stack that facilitates service-based architecture of the control system. QS was extensively tested at NSLS-II beamlines and proved to be a useful tool for remote and autonomous control, including multimodal experiments [8]. It was demonstrated that separating plan execution engine into a separate service significantly improves reliability of GUI-based control by forcing strong separation between GUI and control code. QS is currently used or evaluated by other facilities, including ALS, SLAC, APS, BESSY, Australian Synchrotron and PLS-II.

The research described herein is Fundamental Research as defined in the EAR (15 CFR §734.8) or Part 810 (10 CFR §810.3), as applicable, and as described in the USD (AT&L) memoranda on Fundamental Research, dated May 24, 2010, and on Contracted Fundamental Research, dated June 26, 2008.

REFERENCES

[1] D. Allan, T. Caswell, S. Campbell, and M. Rakitin, "Bluesky's Ahead: A Multi-Facility Collaboration for an a la Carte Software Project for Data Acquisition and

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2025). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THPD053

Management," *Synchrotron Radiat. News*, vol. 32, no. 3, pp. 19–22, May 2019. doi:10.1080/08940886.2019.1608121

[2] Bluesky Collaboration, GitHub, https://github.com/bluesky/.

[3] Bluesky Queue Server, Python library. GitHub, https://github.com/bluesky/bluesky-queue-server

[4] Bluesky Queue Server API, Python library, GitHub, https://github.com/bluesky/bluesky-queue-server-api

[5] Bluesky HTTP Server, Python library, GitHub, https://github.com/bluesky/bluesky-httpserver

[6] Bluesky Widgets, Python library, GitHub, https://github.com/bluesky/bluesky-widgets

[7] Bluesky Queue Server documentation, https://blueskyproject.io/bluesky-queue-server/.

[8] P. Maffettone *et al.*, "Self-driving multimodal studies at user facilities", Jan. 2023, arXiv:2301.09177. doi:10.48550/arXiv.2301.09177

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2025). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.