Supplementary Information

S1: Runtime Performance Characteristics:

AutodiDAQt makes assumptions about the data acquisition task to simplify the acquisition runtime. Significantly, because AutodiDAQt implements the acquisition runtime as a set of asynchronous tasks running on a single process, AutodiDAQt assumes that the reads from the instrumentation are IO-bound rather than CPU-bound. Although this is not a strict assumption, as the actor for an instrument can talk to another process which it sets up during the application's startup, circumventing this assumption requires the end user to take care of any multitasking concerns arising out of a partial adoption of multiprocessing.

Despite this constraint, the AutodiDAQt's runtime is very low overhead, as can be verified by running the profiling benchmarks included in the source repository. Benchmarks are always machine-dependent, but on the plain consumer hardware available at the time of publication, the overhead per experimental configuration ("point") is on the order of 200 µs when running an acquisition generating synthetic data from a 250 px by 250 px virtual CCD. As AutodiDAQt is not intended for applications which need to operate instruments in closed-loop control or collect data in real time, an overhead of less than one millisecond per point makes use of the multiprocessing unnecessary for most experiments. AutodiDAQt achieves this level of performance by running UI repainting infrequently, using the Qt event loop in place of the standard library event loop, and by performing essentially no data bookkeeping other than memory allocation during an experimental run. All the data collation and transformation are deferred to a separate process once an experiment is complete.

S2: Product Space and Sum Space Structures:

The design philosophy underpinning AutodiDAQt's conception of instruments and data acquisition modularity is that data acquisition hardware defines a mathematical space, with each piece of hardware exposing zero or more dimensions for control, and one or more dimensions for acquisition. Every control dimension is an acquisition dimension because we can always ask to record where we are in configuration space. By viewing hardware in this way, we can think of data acquisition tasks in terms of a product and sum structure.

By product, we mean that we can think of the product of two data acquisition programs P and P' which record over the spaces $S = \mathrm{span}(\{s_i\})$ and $S' = \mathrm{span}(\{s_j'\})$. Then, the acquisition program $P \times P'$ consists of recording over the configuration space $T = \mathrm{span}(\{s_i\} \cup \{s_j'\})$. The simplest way to construct this acquisition program is just to traverse the program P' inside the acquisition loop of P. To concretize this idea, suppose for an ARPES measurement that P consists of measuring a single ARPES cut at different temperatures so that $\{s_i\} = [s_{\mathrm{Temperature}}]$ and P' consist of measuring a 2D cut

of a Fermi surface at constant photon energy by scanning one angular degree of freedom θ, so that $\{s_j'\} = [s_\theta]$. Then, $P \times P'$ consists of recording 2D cuts of the Fermi surface across different temperatures, with $\{t_i\} = \{s_i\} \cup \{s_j'\} = [s_{\text{Temperature}}, s_\theta]$.

By sum, we mean that if at each point in the configuration space S, program P (P') records a vector of data $r \in R$ ($r' \in R'$), then the program $P + P'$ consists of recording data according to what P and P' require independently $(r, r') \in R \times R'$. If we consider a simple ARPES measurement where P records the photoelectron spectrum r and P' records the photocurrent from the sample, then $P + P'$ consists of measuring and recording the photoelectron spectrum and the integrated photocurrent at each point of the acquisition program. The sum product structures for these simple acquisition programs are shown diagrammatically in **Error! Reference source not found.**.

The advantage of this modularity is that it becomes transparent to determine how to construct useful acquisition routines over complicated control and acquisition hardware by compositing simple, one-dimensional acquisition programs. This motivates composing the acquisition sequence from primitives in addition to separating the task of describing the acquisition sequence, the heart of the experiment, from the details of building robust and fluent DAQ programs.

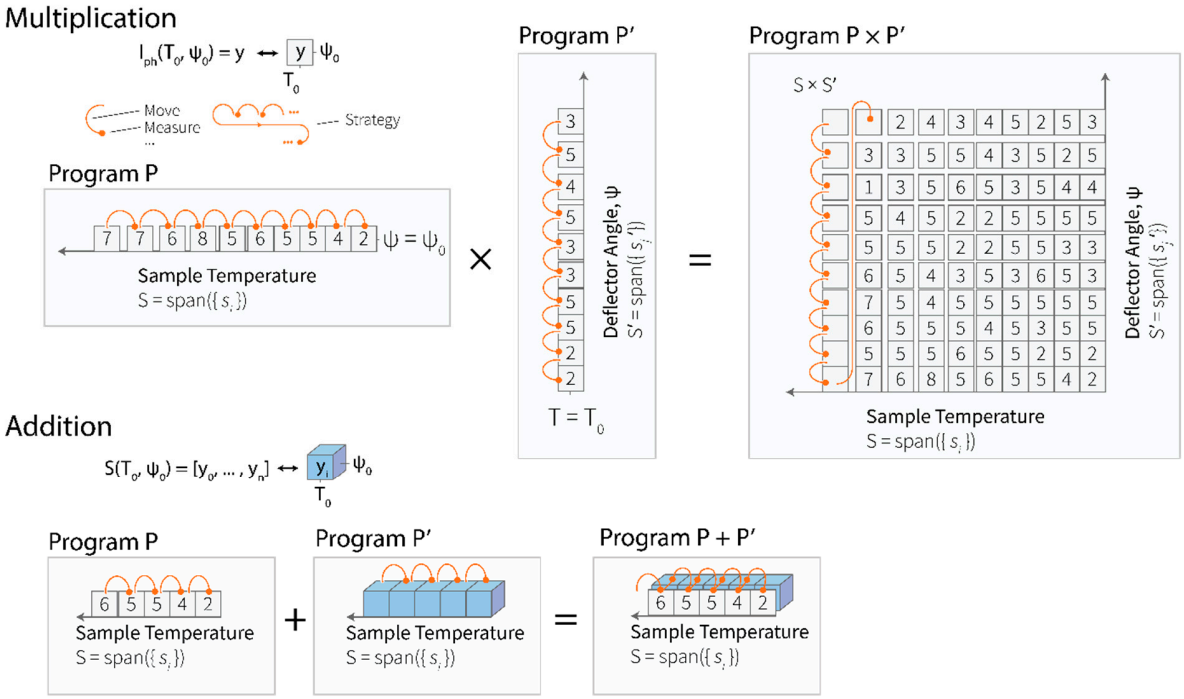


Figure S1. Products (top) and sums (bottom) of configuration spaces and programs. The diagrams above show how AutodiDAQt can use products and sum structures to composite acquisition programs. (Top) in the top row, two one-dimensional acquisition programs P and P′ acquire over spaces S and S'. Their product consists of

iterating over the Cartesian product $S \times S'$. There is some ambiguity in terms of the order of traversal over the product space. One option is to run P' in the inner loop of P, which is what is shown in the first row. Other strategies exist depending on what is desirable in the context. This can be configured using different kinds of product structures over S and S' such as randomly acquiring a point from the grid, acquiring them in an alternating ascending and descending fashion—if the motion in S' is expensive, for instance—and using space filling curves to sample coarsely in the higher dimensional space first. (Bottom**)** in the bottom row, the sum of two programs is shown instead. The second program differs to the first in that it records $S(T, \psi)$, the entire ARPES spectrum, instead of just the photocurrent $I_{ph}(T, \psi)$. The sum of the programs records the union of the data by iteratively moving to configurations (T_0, ψ_0), recording the data $I_{ph}(T_0, \psi_0)$ required by P and then recording the data $S(T_0, \psi_0)$ required by P' before continuing to the next required configuration (T_1, ψ_0).

A large collection of complete example applications is distributed and described alongside the AutodiDAQt source [examples module]. These examples cover topics ranging from axis and instrument specification [manual_axis.py], creating additional UI components to run alongside the main acquisition UI [ui_panels.py], acquisition interlocks [scanning_interlocks.py] and the API for rapidly programming acquisition applications [scanning_experiment_revisited.py], logical coordinate transforms and virtual axes [computed_axis.py], in addition to many other topics.

A representative example is shown and discussed below, along with the user interface that it generates and a sample data output, to better inform the reader as to how AutodiDAQt simplifies common DAQ programming tasks.

Scanning over configuration: After defining the axes and experimental degrees of freedom, the simplest DAQ operation is to collect data inside the volume defined by these degrees of freedom.

```
from autodidaqt import AutodiDAQt,
Experiment
from autodidaqt.mock import
MockMotionController,
MockScalarDetector
from autodidaqt.scan import scan

dx =
MockMotionController.scan("mc").stag
es[0](limits=[-10, 10])
dy =
MockMotionController.scan("mc").stag
es[1](limits=[-30, 30])

read_power = {"power":
"power_meter.device"}

class MyExperiment(Experiment):
    scan_methods = [
        scan(x=dx, name="dx Scan",
read=read_power),
        scan(x=dx, y=dy, name="dx-dy
Scan", read=read_power),
    ]

AutodiDAQt(
    __name__,
    {},
    {"experiment": MyExperiment},
    {"mc": MockMotionController,
"power_meter": MockScalarDetector},
).start()
```

Figure S2: **Code Listing 1**—example acquisition program for scanning a motion controller while recording from a power meter.

In Figure S2 or Code Listing 1, we see the full program source for a simple acquisition program which uses the mock instrument definitions provided by AutodiDAQt. Scan axes are defined in the lines starting *dx* = and *dy* =. Scan axes are defined relative to a named instrument (here, "mc" for "Motion Controller") and a path to a specific axis (here, index 0 of "stages" and index 1 of "stages" for each of *dx* and *dy*). This definition gives a full path to the axis which should be controlled when referencing the scan direction *dx*/*dy* and they are equivalent to universal resource identifiers//mc/stages/0/ and //mc/stages/1/. Limit configurations specify how far each axis can safely be moved in its units during an acquisition.

Then, in the experiment definition, these scan directions *dx* and *dy* are combined into two acquisition programs "dx Scan" and "dx-dy Scan" which scan over the x direction (//mc/stages/0) or both the x and y directions, respectively. Each of these acquisition programs will acquire data by reading from the

instrument “power_meter.device” (equivalently, //power_meter/device/) and storing readings under the label “power”. Internally, the scan function which we are using to generate the acquisition programs “dx Scan” and “dx-dy Scan” takes the Cartesian product of the control variables and the union of the read variables to build an acquisition program from “dx” and “dy”, which are essentially specifications of acquisition programs which affect only one hardware degree of freedom (see the above section on *Product Space and Sum Space Structures*).

In Figure S3, we show the running DAQ program corresponding to Code Listing 1 (Figure S2). AutodiDAQt synthesizes the UI controls for each of the scan modes “dx Scan” and “dx-dy Scan” as we requested, with the controls for “dx-dy Scan” being currently visible in the acquisition window on the right. Most of this rightmost window is taken up with output variable and control variable visualizations. The currently selected tab shows the value history for the hardware at URI //mc/stages/0/ which, per a prior discussion, corresponds to the x direction in the scan definition. It has a staircase structure because in the requested scan configuration (shown), the inner loop consists of rastering y in 51 steps between 0 and 10.

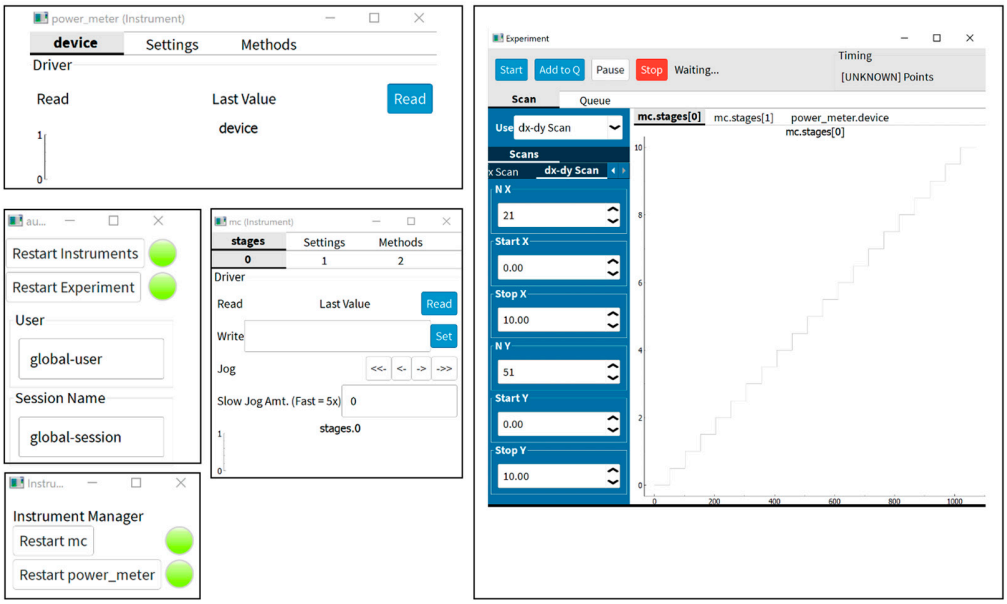


Figure S3: Generated UI and DAQ program for ***Error! Reference source not found.***. The right-hand side is the main acquisition window which shows the configuration options for the selected scan mode, as well as plots for each of the control and output acquisition variables. On the left, monitor panes and controls for each instrument are shown. For instruments with control axes, these axes are provided with a plot of the control variable history, direct write and read controls, and jog controls to manually adjust positioning.

The resulting data of running this scan with the configuration depicted can be seen in Figure S4. We can see output variables x and y with appropriate lengths for the requested acquisition sequence as well as the 2D scalar output *power* with dimensions (x, y). In addition to the collated format which carries the high-level semantics for the requested acquisition sequence, the raw acquisition sequence is stored so that it can be introspected for any

irregularities in the future. This is one of many ways in which AutodiDAQt supports automatically providing defensive and reproducible scientific experiments without domain knowledge or software expertise on the part of the scientist programming the data acquisition system.

xarray.Dataset

► Dimensions: (**x**: 21, **y**: 51)

▼ Coordinates:

x (x) float64 0.0 0.5 1.0 1.5 ... 9.0 9.5 10.0

```
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,  5.5,
        6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5, 10. ])
```

y (y) float64 0.0 0.2 0.4 0.6 ... 9.6 9.8 10.0

```
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ,  1.2,  1.4,  1.6,  1.8,  2. ,  2.2,
        2.4,  2.6,  2.8,  3. ,  3.2,  3.4,  3.6,  3.8,  4. ,  4.2,  4.4,  4.6,
        4.8,  5. ,  5.2,  5.4,  5.6,  5.8,  6. ,  6.2,  6.4,  6.6,  6.8,  7. ,
        7.2,  7.4,  7.6,  7.8,  8. ,  8.2,  8.4,  8.6,  8.8,  9. ,  9.2,  9.4,
        9.6,  9.8, 10. ])
```

▼ Data variables:

power (x, y) float64 ...

```
array([[4.295106, 5.689163, 3.316228, ..., 6.110956, 4.773731, 5.690863],
       [4.92123 , 5.38519 , 5.301642, ..., 3.989704, 4.618874, 4.545255],
       [4.475149, 3.827245, 4.878145, ..., 4.712234, 4.771796, 5.536446],
       ...,
       [6.766935, 4.850957, 5.951787, ..., 3.24675 , 4.488549, 5.39566 ],
       [6.604045, 5.736134, 5.074479, ..., 5.38638 , 6.288686, 4.454095],
       [5.135886, 4.999819, 4.7088  , ..., 6.534573, 4.7834  , 6.165107]])
```

x-values (x, y) float64 ...

y-values (x, y) float64 ...

► Attributes: (0)

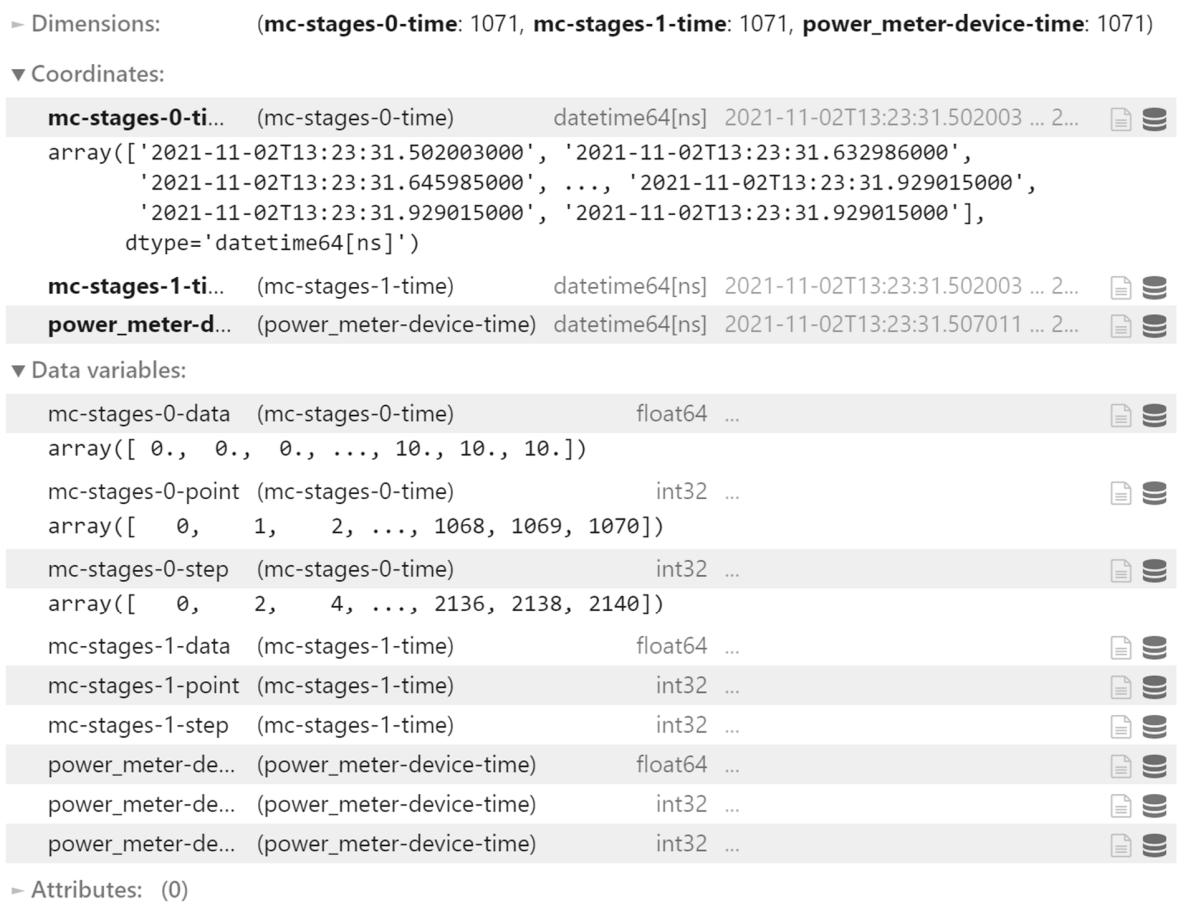


Figure S4: Output data for the DAQ program shown in **Error! Reference source not found.** when executed with the parameters and configuration depicted in **Error! Reference source not found.**. **(Top)** Each of the control variables x and y has an entry in the output file, and the recorded data *power* is a two dimensional scalar function (array) of the variables (x,y). **(Bottom)** In addition to the collated data format which has the high-level semantics of the data acquisition operation we hoped to perform, AutodiDAQt also retains a structured command log which records timings and locations of control axes and read values at each point (entry in the collated format) and step (sequence of reads or motions performed together) in the acquisition sequence. A metadata log and application log, both not shown here for brevity, are also retained with each output.

In Figure S3, we show the running DAQ program corresponding to **Error! Reference source not found.**. AutodiDAQt synthesizes the UI controls for each of the scan modes "dx Scan" and "dx-dy Scan" as we requested, with the controls for the "dx-dy Scan" being currently visible in the acquisition window on the right. Most of this rightmost window is taken up with the output variable and control variable visualizations. The currently selected tab shows the value history for the hardware at URI //mc/stages/0/ which, as per a prior discussion, corresponds to the x direction in the scan definition. It has a staircase structure because in the requested scan configuration (shown), the inner loop consists of rastering y in 51 steps between 0 and 10. The resulting

data of running this scan with the configuration depicted can be seen in Figure S5. We can see output variables x and y with appropriate lengths for the requested acquisition sequence as well as the 2D scalar output *power* with dimensions (x, y). In addition to the collated format which carries the high-level semantics for the requested acquisition sequence, the raw acquisition sequence is stored so that it can be introspected for any irregularities in the future. This is one of many ways in which AutodiDAQt supports automatically providing defensive and reproducible scientific experiments without domain knowledge or software expertise on the part of the scientist programming the data acquisition system.

AutodiDAQt provides value records and timestamps for each step and point of the acquisition sequence. A point in the acquisition sequence corresponds to a single entry in the collated data, whereas a step corresponds to a set of motions or reads from the hardware which are performed concurrently. By examining the step and point counters "mc-stages-0-point" and "mc-stages-0-step", we can see that there are twice as many steps as points for this acquisition sequence. This is because to collect any given piece of data, we first move to the desired point (x,y) on each even step before then reading *power* on each odd step. For a more complicated acquisition program, there may be no simple correspondence between the step and point number. For instance, this may occur if multiple steps of an axis are required to perform the backlash compensation of a motor or multiple temperature steps are used to approach a target temperature for the PID controller's stability.

In addition to the acquisition record provided, AutodiDAQt keeps application logs in the JSON format with timestamps which can be correlated to points in the raw acquisition record to cross-correlate data irregularities with detailed event logs.

S4: AutodiDAQt Application Structure:

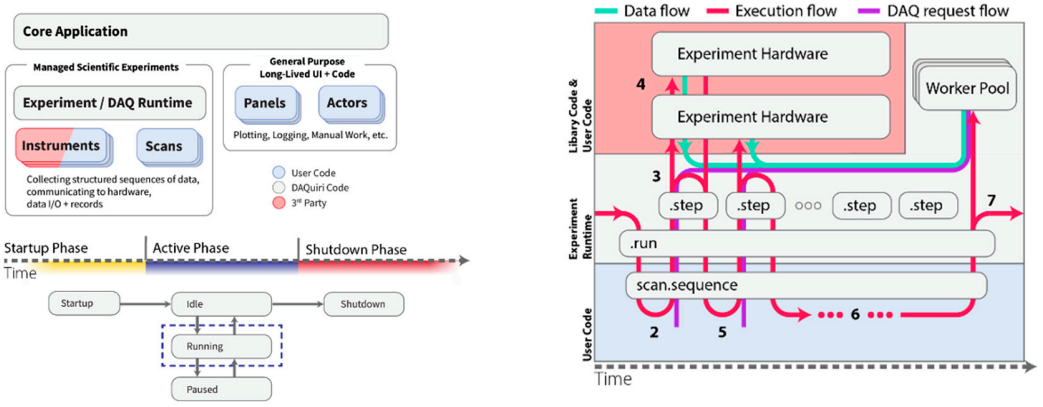
Data acquisition programs must operate reliably in a nearly real-time paradigm and tolerating faults to errors in user code. Traditionally designed monolithic data acquisition programs are particularly brittle to faults in user code because there is no distinguishing barrier between the code written by users or scientists, and the code required to support the healthy operation of its instrumentation. AutodiDAQt makes three defensive choices regarding the application's structure to reduce its fragility.

The first is that AutodiDAQt adopts a central monitor process which is responsible for UI and communications, while all other code runs with some degree of isolation, either in coroutines, threads, or another process. This makes it simpler for AutodiDAQt to recover gracefully from hardware and software failures. Each instrument driver is isolated and can schedule any necessary code to run in its own coroutine without having to rely on the user code to perform these tasks repetitively and in a timely fashion for a safe operation.

The second design decision is in preventing the user acquisition code from directly operating against the hardware. Instead, the acquisition steps are only orchestrated in the user code but are executed entirely in the library code which can be expected to have a higher reliability. This also makes it

transparent to a user to test data acquisition programs without physical hardware attached which aids in the safe and rapid development of data acquisition programs. This situation shown in Figure S5b tracks the execution and DAQ request/command flow through the process of running an acquisition program. A detailed description of the control flow during the course of an experiment is described in the caption to Figure S5b.

The third design decision is in adopting state machines to coordinate the experiment and acquisition flow and remote command flow. A very simplified version of the internal state machine for running acquisition experiments is shown in the bottom of Figure S5a, which also depicts how these internal states map onto the terminology for the data acquisition program's lifecycle. On the side of reliability, this improves the testability of the data acquisition software by facilitating creating well-defined state conditions for the software to be tested in. Indirectly, the reliability is also increased by facilitating user understanding via software specifications in terms of well-defined states.



(**a**) (**b**)

Figure S5. Structure of the AutodiDAQt Framework and Experiment Control Flow. (**a**) Granular component hierarchy with significant elements of the framework. Actors and panels provide an actor-based computing model for long lived computations and tasks, and their associated UI. The experiment runtime provides high-level abstractions for communicating with instruments and collecting data from them by using declaratively programmed acquisitions. The diagram at the bottom of the figure is a simplified state diagram for the state machine inside the experiment runtime showing transitions between different phases in the application lifecycle and between running and idle states of the experiment. (**b**) Control flow between user and library code during the collection of a single run of data. Red arrows indicate the control flow inside the program, with cyan arrows for collected data. User code generates a description of the desired acquisition steps (purple) which is handled concurrently by the experiment runtime (forking red arrows) which performs the data collection, deals with data collection and storage, and ensures synchronization before further acquisition steps are performed. Chronologically during a run, (1) the experiment enters the running state; (2) it requests a description of what to do from the user's acquisition; and (3) it performs a step of the experiment by distributing the request to

appropriate hardware. (4) The hardware moves, reads, or does both and returns the data back to the runtime where it is collated into raw events and structured records. After all tasks in the step have been completed, the experiment asks for another point (5) from the user's acquisition continuing for as long as it is fed motion or acquisition steps (6). (7) The complete record of acquisition steps, their timing, and the raw and structured data are passed to a worker pool to be saved and the runtime is again immediately available for further runs and acquisition tasks.

S5: AutodiDAQt Asynchronous Programming Model:

Data acquisition is a fundamentally asynchronous task, but there are many programming models which can be invoked to handle application programming in this asynchronous environment. For AutodiDAQt, we operate in an augmented actor model where synchronization happens via message passing between actors and a central monitor. Message passing coordination is required to facilitate remote command execution, a central feature required to provide a tight integration with data analysis software. The necessity of applying this application paradigm between AutodiDAQt and remote acquisition planning also motivated applying this approach internally to coordinate between the monitor and the independent components. In AutodiDAQt, each independent component has a coroutine with represents the ego of that component. This coroutine reads from an associated inbox of messages, can send messages to other components, and can perform any independent work required of that component. The collection of this coroutine, the inbox, and the associated private state maps well onto the actor model and, therefore, can be called an actor (see also A5a).

In AutodiDAQt, each independent piece of hardware is associate with its own actor and additional separate actors exist for the monitor and the experiment abstraction. If needed, users can define additional actors to take up responsibilities such as logging, communication with status dashboards at larger facilities, or running auxiliary user interfaces to provide cameras and diagnostics, as required by the needs of the data acquisition system. Using separate actors—as discussed in ***Error! Reference source not found.***—provides application robustness by isolating concerns across the messaging passing boundary. In the event of serious errors, the monitor can attempt to restart the failed component or else can inform each other actor of the failure and the requirement to safely shutdown so that the issue can be addressed.

S6: Details and AutodiDAQt Type System Forwarding and Remote Acquisition: a constrained remote application programming interface is furnished based on an exported, extensible remote type system. This type of system covers the datatypes required to specify all the parameters for data acquisition procedures, as well as the data that they produce at each point in the configuration space. This is necessary so that a remote data acquisition planner, like the one bundled in the AutodiDAQt remote, can update the acquisition parameters before dispatching a data acquisition request and can interpret the data which are being acquired so that it can be collated and understood by the data analysis system. AutodiDAQt currently has no plans to support third party software in the place of the AutodiDAQt receiver. As

a result, the communication API and remote procedure call format over the message broker are the implementation details. However, the remote RPC commands are very simple—amounting to lightweight JSON RPC—and all the commands are available in the shared AutodiDAQt common module.

The type of system itself must be serializable to the wire format adopted by AutodiDAQt—which is flexible, although JSON is used by default—so that it can be provided to the remote acquisition planner when a remote attaches to the data acquisition system. Each exported type is associated with a unique ID. Once the type of system has been exported, values are sent over the wire in a container format specifying the unique type of ID of the value and the serialized contents of that value. This permits arbitrary data to be packed on the side of the data acquisition suite before being unambiguously unpacked at the remote planner/data analysis side so that it is immediately available, even before the data has been fully collected.

Technology Choices and Software Ecosystem: the NumPy [45] array format and numeric programming ecosystem have over the last decade become the tool of choice for scientific programming, with a broad adoption across diverse scientific disciplines. Although it has formed the backbone for scientific computing in recent years, the NumPy array format emphasizes general purpose operations over discipline-specific semantics. For this reason, extensions to the algorithms used on these formats, as in SciPy [46], and to the formats themselves have been developed. AutodiDAQt leverages xarray [47]—an extension of the NumPy format and the tabular data format Pandas [48]—to provide coordinate aware and unitful representations of the acquired data. To provide efficient on-disk representations of the large multidimensional datasets which are produced by ARPES, for instance, AutodiDAQt adopts the zarr [49] compressed array format, with additional extensions to serialize to additional formats which may be required in specific scientific disciplines.

For inter-process and remote communication, AutodiDAQt uses NNG (nanomsg-next-generation) for message passing between the main data acquisition process and a remote acquisition process. A description of how a command language is established between the main process and the remote process is discussed briefly in ***Error! Reference source not found.***.

Author Contributions: A.L. and C.H.S. initiated and directed this research project. C.H.S. developed the data acquisition program and applied it to nanoXPS and pump and probe ARPES experiments. A.L. and C.H.S. wrote the manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Director, Office of Basic Energy Sciences, Materials Science and Engineering Division, of the U.S. Department of Energy, under Contract No. DE-AC02-05CH11231, as part of the Ultrafast Materials Science Program (KC2203).

Data Availability Statement: The data that support the finding of this study are available from the corresponding author upon request.

Conflicts of Interest: The authors declare that they have no competing interest.