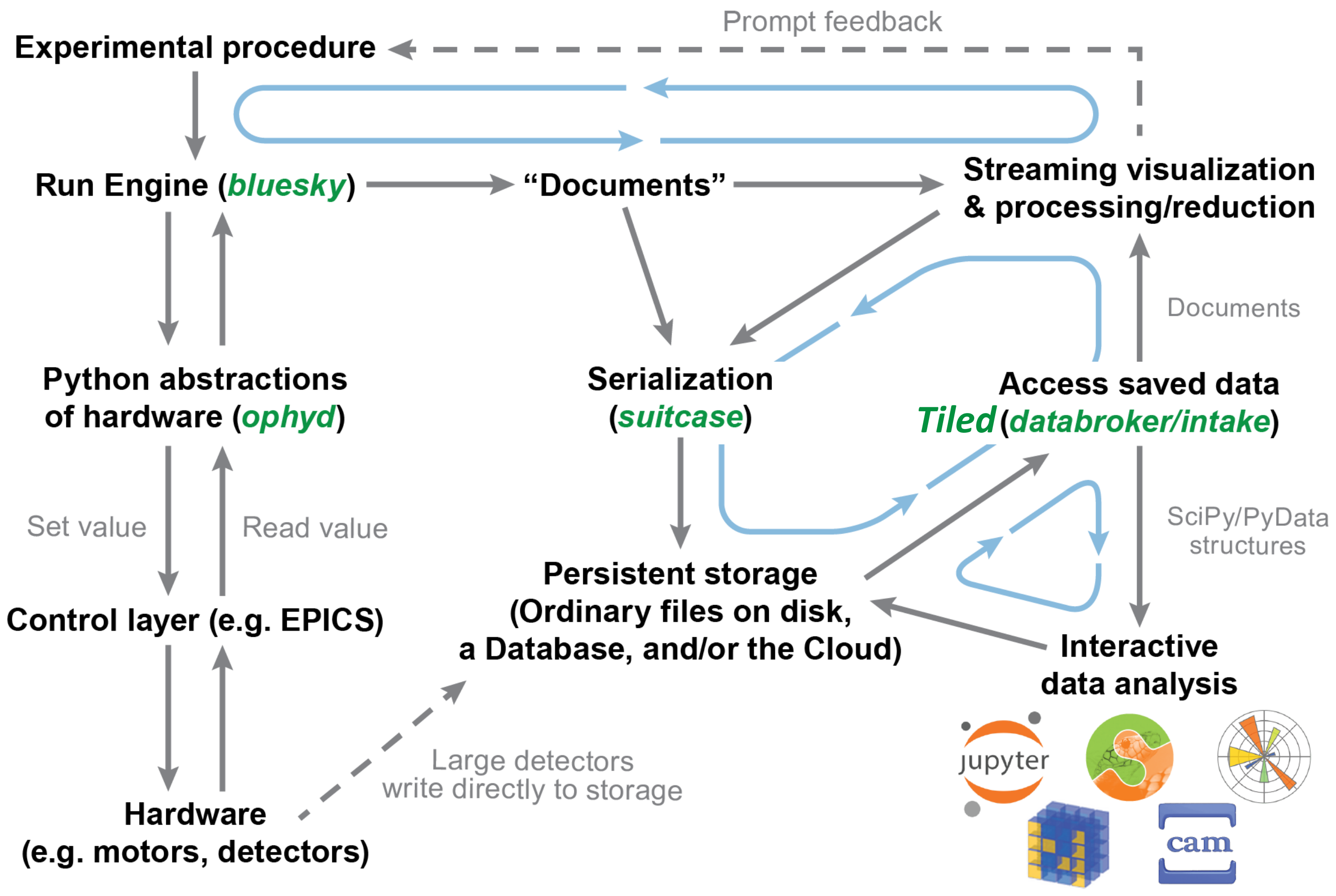
HZB Helmholtz Zentrum Berlin

Experiments Orchestration with Bluesky

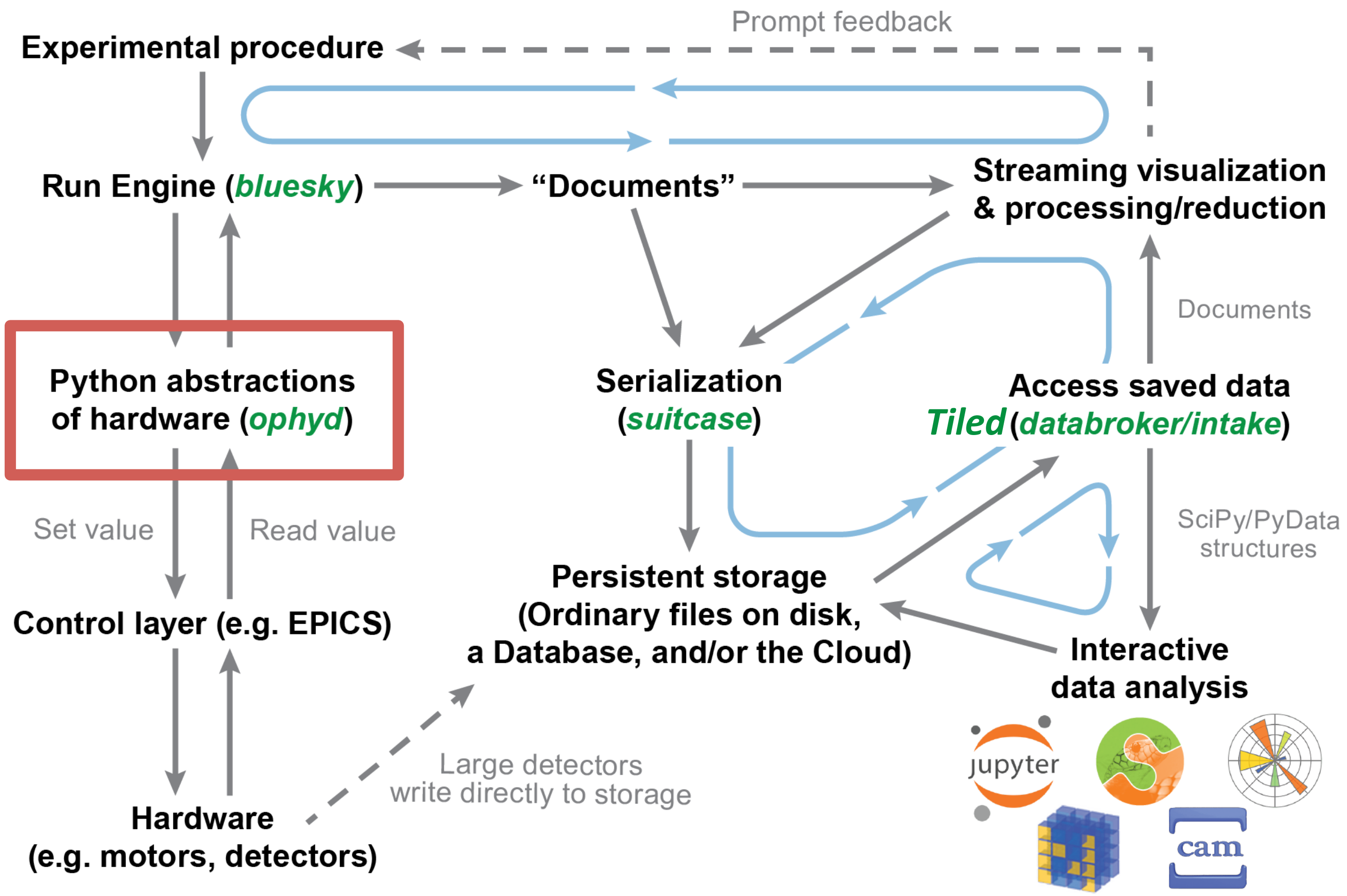
Luca Porzio - Marcel Bajdel

EPICS Collaboration Meeting – Spring 2024
Pohang (South Korea)

Architecture Overview



Architecture Overview



ABSTRACTION LAYER WITH OPHYD

- **Ophyd** puts the control layer (e.g. EPICS, TANGO, serial protocols, HTTP, …) behind a **high-level interface**. It keeps device-specific details contained.
- **Group individual signals** into logical "Devices" to be configured and used as one unit.
- Assign signals and devices **human-friendly names** that propagate into **metadata**.
- **Categorize signals** by "kind" (primary reading, configuration, engineering/debugging).

```
epics> dbl

TEST:Left-Mtr-X
TEST:Left-Mtr-Y
TEST:Right-Mtr-X
TEST:Right-Mtr-Y
```

```python
from ophyd import Device, Component, EpicsSignal

# Here we group signals into a Device
class XYStage(Device):
    x = Component(EpicsSignal, 'Mtr-X')
    y = Component(EpicsSignal, 'Mtr-Y')

# and connect to multiple instances
# of that device.
left_stage = XYStage('TEST:Left-', name='left_stage')
right_stage = XYStage('TEST:Right-', name='right_stage')
```

INTERACT WITH DEVICES

- **Read, Describe, Set or Subscribe** to single signals or Devices.

```
from ophyd import Device, Component, EpicsSignal,
EpicsSignalRO

class RandomWalk(Device):
    x = Component(EpicsSignalRO, 'x')
    dt = Component(EpicsSignal, 'dt')

random_walk = RandomWalk('random_walk:', name='random_walk')
random_walk.wait_for_connection()

random_walk.x.read()

random_walk.describe()

status = random_walk.dt.set(2)
```

DEVICE STATUS OBJECT

- Ophyd **Status** objects signal when some potentially-lengthy action is complete.
- A Status object is created with an associated **timeout**.
- The recipient of the Status object may add callbacks that will be notified when the Status object completes.
- The Status object is marked as completed successfully, or marked as completed with an error, or the timeout is reached, whichever happens first.

ADD COMPLEX BEHAVIORS

- Implement coordination across multiple PVs, such as a setpoint PV and a readback PV, in order to know when a process is done.

```
class Decay(Device):
    """
    A device with a setpoint and readback that decays exponentially toward the
setpoint.
    """
    readback = Component(EpicsSignalRO, ':I')
    setpoint = Component(EpicsSignal, ':SP')
    done = Component(EpicsSignalRO, ':done')

    def set(self, setpoint):
        """
        Set the setpoint and return a Status object that monitors the 'done' PV.
        """
        status = DeviceStatus(self.done)

        # Wire up a callback that will mark the status object as finished
        # when the done signal goes from low to high---that is, a positive edge.
        def callback(old_value, value, **kwargs):
            if old_value == 0 and value == 1:
                status.set_finished()
                self.done.clear_sub(callback)

        self.done.subscribe(callback)

        # Now 'put' the value.
        self.setpoint.put(setpoint)

        # And return the Status object, which the caller can use to
        # tell when the action is complete.
        return status


decay = Decay('decay', name='decay')
status = decay.set(135)
```

USE STANDARD CLASSES

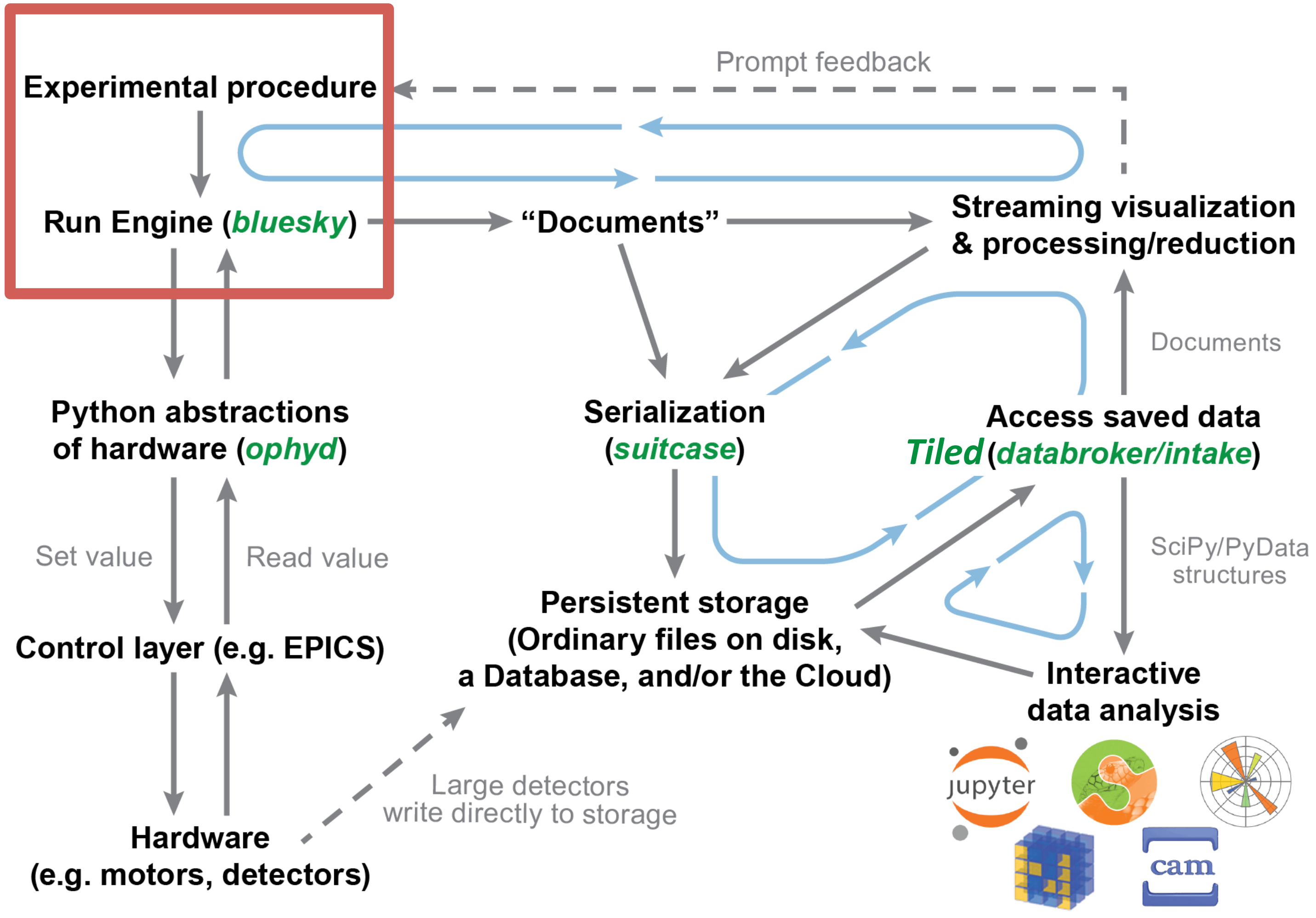
- The pattern of **readback, setpoint and done** is pretty common, so *ophyd* has a special Device subclass (*PVPositioner*) that writes the set() method for you if you provide components with these particular names.

```
from ophyd import PVPositioner

class Decay(PVPositioner):
    """
    A device with a setpoint and readback that decays
exponentially toward the setpoint.
    """
    readback = Component(EpicsSignalRO, ':I')
    setpoint = Component(EpicsSignal, ':SP')
    done = Component(EpicsSignalRO, ':done')
    actuate = Component(EpicsSignal, ...)  # the "Go" button

def callback(status):
    print("DONE:", status)

decay = Decay('decay', name='decay')
status = decay.set(140)
status.add_callback(callback)
```

Examples

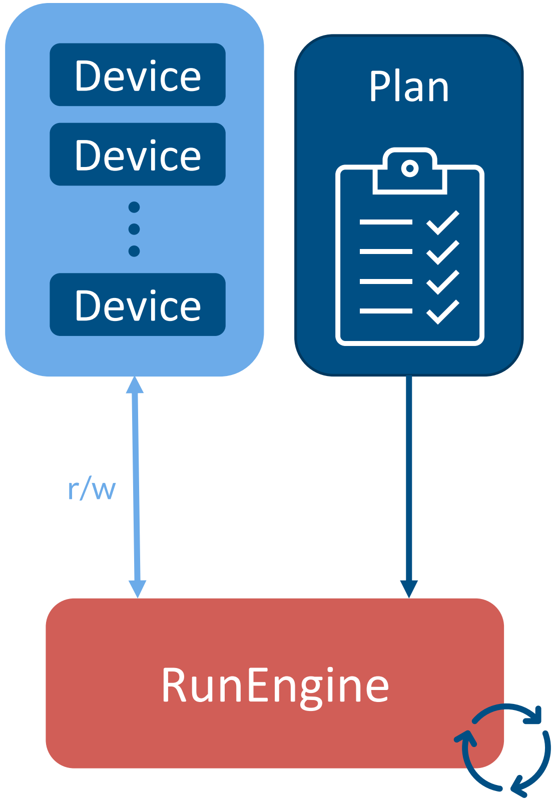
- Single PVs
- Ophyd/Group Signals into Devices
- Ophyd/Complex Behaviours

Architecture Overview



BLUESKY RUN ENGINE

- *Bluesky* encodes an experimental procedure as a **plan**, a sequence of atomic instructions. The **RunEngine (RE)** is an interpreter for plans.
- The **RE** lets us focus on the logic of our experimental procedure while it handles important technical details consistently:
  - it communicates with hardware
  - monitors for interruptions
  - organizes metadata and data
  - coordinates I/O
  - ensures that the hardware is left in a safe state at exit time.



PLANS

- They represent **experimental procedures**.
- A plan tells the *RunEngine* how to interact with Devices.
- A variety of pre-assembled plans are provided (e.g. *scan*, *count*).

```python
from ophyd.sim import det, motor
from bluesky.plans import count, scan

# a single reading of the detector 'det'
RE(count([det]))

# five consecutive readings
RE(count([det], num=5))

# five sequential readings separated by a 1-second delay
RE(count([det], num=5, delay=1))

# a variable delay
RE(count([det], num=5, delay=[1, 2, 3, 4]))

# Take readings forever, until interrupted (e.g., with Ctrl+C)
RE(count([det], num=None))

# Scan motor from -10 to 10, stopping at 15 equally-spaced points
# along the way and reading det.
RE(scan([det], motor, -10, 10, 15))
```

PLANS

- Pre-assembled plans are built from smaller "*plan stubs*".
- We can mix and match the "stubs" and the "pre-assembled" plans to create **custom procedures**.

```
from ophyd.sim import det, motor
from bluesky.plans import scan
from bluesky.plan_stubs import mv

def sweep_exposure_time(times):
    "Multiple scans: one per exposure time setting."
    for t in times:
        yield from mv(det.exp, t)
        yield from scan([det], motor, -10, 10, 5)

motor.delay = 0
RE(sweep_exposure_time([0.01, 0.1, 1]))
```

- *Bluesky* is not tied to *ophyd* or *EPICS* specifically: **any Python object may be used**, so long as it provides the specified methods and attributes that *Bluesky* expects.

PLANS

- Pre-assembled plans are built from smaller "*plan stubs*".
- We can mix and match the "stubs" and the "pre-assembled" plans to create **custom procedures**.

```
from ophyd.sim import det, motor
from bluesky.plans import scan
from bluesky.plan_stubs import mv

def sweep_exposure_time(times):
    "Multiple scans: one per exposure time setting."
    for t in times:
        yield from mv(det.exp, t)
        yield from scan([det], motor, -10, 10, 5)

motor.delay = 0
RE(sweep_exposure_time([0.01, 0.1, 1]))
```

Plans are implemented as *generators.*
Read more here:
https://blueskyproject.io/bluesky/appendix.html

- *Bluesky* is not tied to *ophyd* or *EPICS* specifically: **any Python object may be used**, so long as it provides the specified methods and attributes that *Bluesky* expects.

Examples

- Single PVs
- Ophyd/Group Signals
- Ophyd/Complex Behavious
- Hello Bluesky

INTERRUPTIONS

- The *RunEngine* capture the **SIGINT** (Ctrl+C) signal and it can be safely interrupted and resumed.
- Plans can provide **checkpoints**, indicating a place where it is safe to resume after an interruption.
- Suspension can be **interactive** (using SIGINT), **planned** (incorporated into a plan) or **automated** (using an agent in background).

Interactive Suspension/Resume

Command	Outcome
Ctrl+C	Pause soon.
Ctrl+C twice	Pause now.

Command	Outcome
RE.resume()	Safely resume plan.
RE.abort()	Perform cleanup. Mark as aborted.
RE.stop()	Perform cleanup. Mark as success.
RE.halt()	Do not perform cleanup — just stop.
RE.state	Check if ‘paused’ or ‘idle’.

INTERRUPTIONS

- The *RunEngine* capture the **SIGINT** (Ctrl+C) signal and it can be safely interrupted and resumed.
- Plans can provide **checkpoints**, indicating a place where it is safe to resume after an interruption.
- Suspension can be **interactive** (using SIGINT), **planned** (incorporated into a plan) or **automated** (using an agent in background).

```
import bluesky.plan_stubs as bps

def pausing_plan():
    while True:
        yield from some_plan(...)
        print("Type RE.resume() to go again or RE.stop() to stop.")
        # marking where to resume from
        yield from bps.checkpoint()
        yield from bps.pause()
```

Planned suspension

INTERRUPTIONS

- Automated pausing can be achieved by making use of **suspender agents**.
- The agent monitors some condition and, if it detects a problem, it suspends execution. When it detects that conditions have returned to normal, it gives the *RunEngine* permission to resume.

```
from ophyd import EpicsSignal
from bluesky.suspenders import SuspendFloor

beam_current = EpicsSignal('...PV string...')

# pause when beam_current <= 2
# resume when beam_current >= 3
sus = SuspendFloor(beam_current, 2, resume_thresh=3)
RE.install_suspender(sus)
```

Automated suspend/resume

ASYNCHRONOUS HAL: OPHYD-ASYNC

- **Asynchronous** Signal access, simplifying the **parallel control** of multiple Signals.
- Built with **`asyncio`** (**`async/await`** syntax)
- Better library support for splitting the logic from the hardware interface to avoid complex class hierarchies.
- It was written with the aim of implementing **flyscanning** in a generic and extensible way.

```
epics> dbl

TEST:Left-Value
TEST:Left-Mode
TEST:Right-Value
TEST:Right-Mode
```

```
class Sensor(StandardReadable, EpicsDevice):
    """A demo sensor"""

    value: A[SignalR[int], PvSuffix("Value"), Format.HINTED_SIGNAL]
    mode:  A[SignalRW[EnergyMode], PvSuffix("Mode"), Format.CONFIG_SIGNAL]

left_sensor = Sensor('TEST:Left-', name='left_sensor‘)
right_sensor = Sensor('TEST:Right-', name=,right_sensor')
```

Using async code makes it possible to do the "put 3 PVs in parallel, then get from another PV"

OPHYD-ASYNC PROTOCOLS

- To make a simple device, you need to subclass from the **StandardReadable** class, create some **Signal** instances, and optionally implement other suitable **Bluesky Protocols** like **`bluesky.protocols.Movable`** or **`bluesky.protocols.Readable`**

```
class DemoMotor(EpicsDevice, StandardReadable, Movable, Stoppable):
    """A demo movable that moves based on velocity."""

    # Whether set() should complete successfully or not
    _set_success = True
    # Define some signals
    readback: A[SignalR[float], PvSuffix("Readback"), Format.HINTED_SIGNAL]
    velocity: A[SignalRW[float], PvSuffix("Velocity"), Format.CONFIG_SIGNAL]
    units: A[SignalR[str], PvSuffix("Readback.EGU"), Format.CONFIG_SIGNAL]
    setpoint: A[SignalRW[float], PvSuffix("Setpoint")]
    precision: A[SignalR[int], PvSuffix("Readback.PREC")]
    # If a signal name clashes with a bluesky verb add _ to the attribute
    # name
    stop_: A[SignalX, PvSuffix("Stop.PROC")]

    async def stop(self, success=True):
        self._set_success = success
        await self.stop_.trigger()
```

USAGE WITH THE RUN ENGINE

- Define a context maneger (`with init_device()`)
- Instantiate the devices
- Execute plans using RunEngine

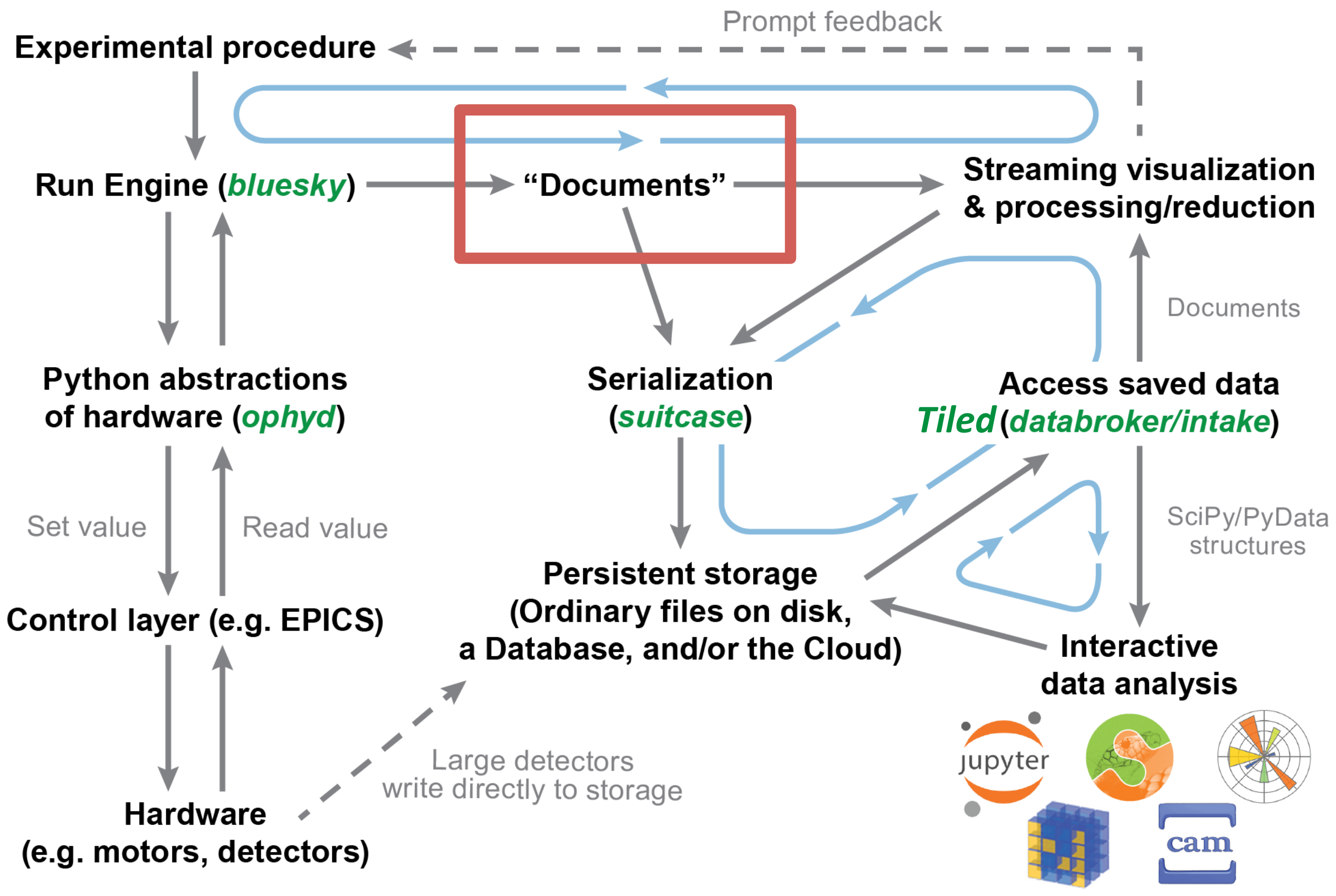
```
from ophyd_async.core import init_devices
with init_devices():
    rw_horiz = RandomWalk("random-walk:horiz:", name="rw_horiz")
    rw_vert = RandomWalk("random-walk:vert:", name="rw_vert")

RE(count([rw_horiz.x], num=3, delay=1))
```

You can use Ophyd sync and Ophyd_async devices in the same plan.

Examples

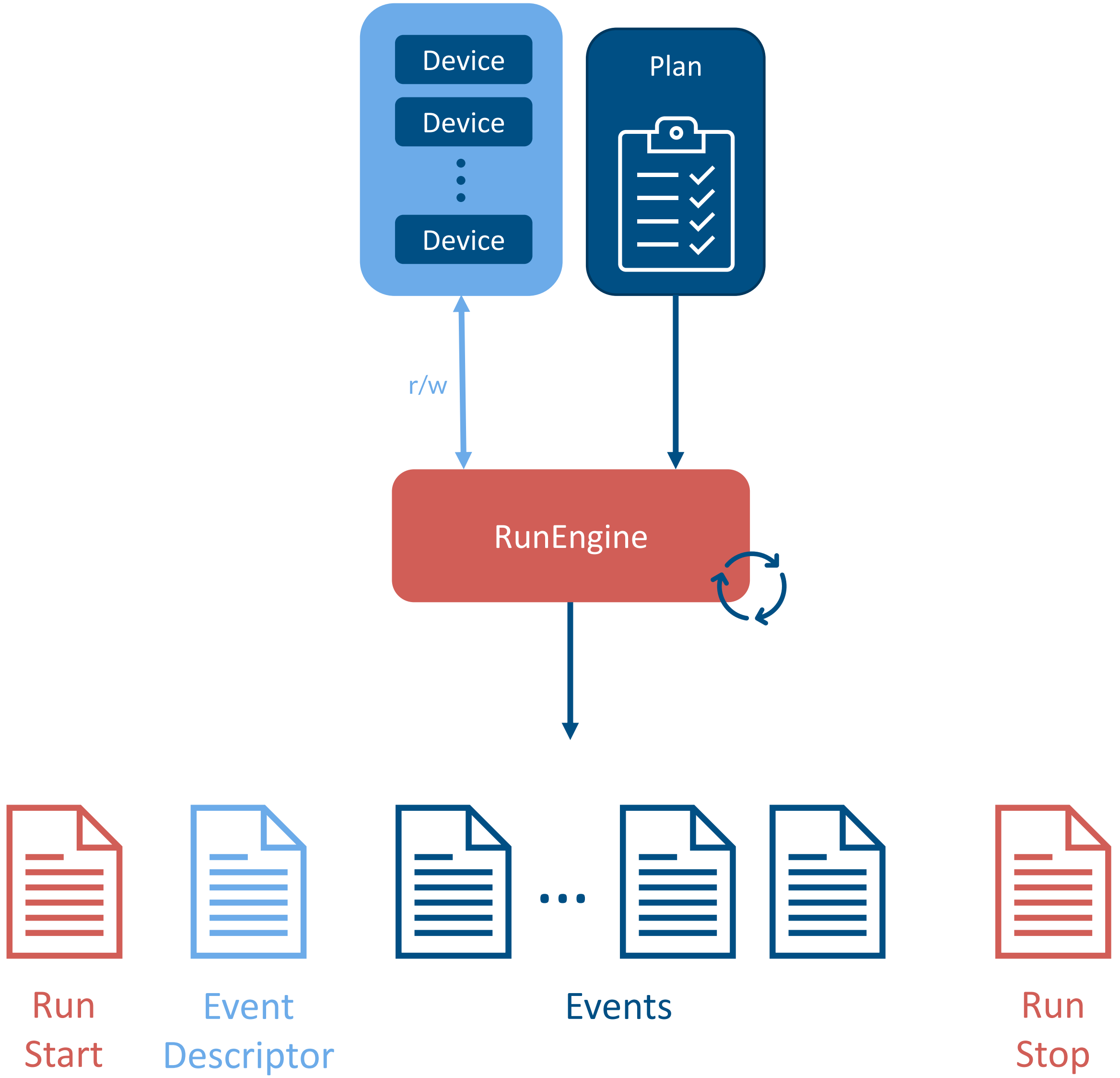
- Ophyd_Async/Group Signals

Architecture Overview



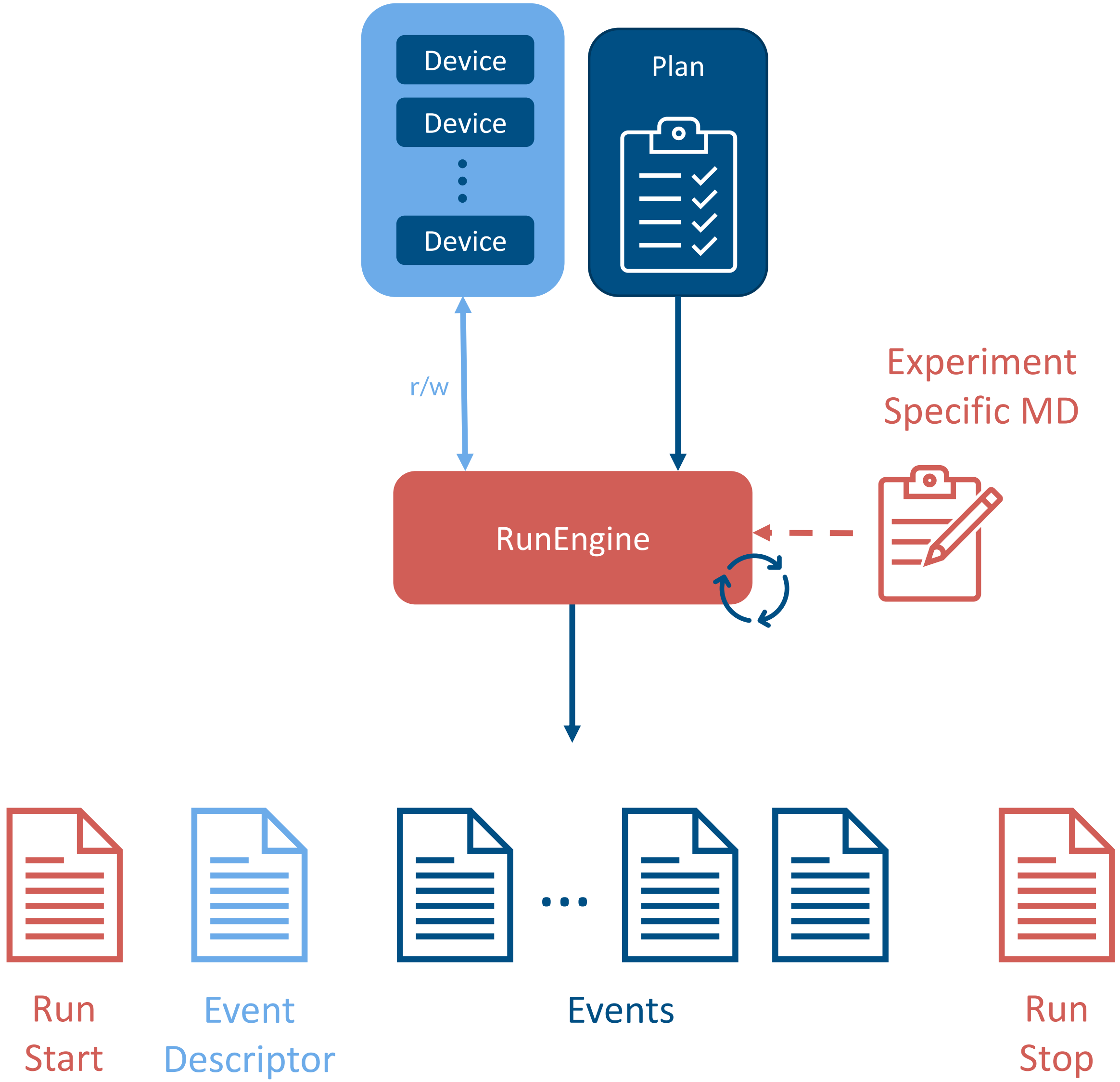
DOCUMENTS & METADATA

- Enable better research by recording rich metadata alongside measured data for use in later analysis.
- All of the metadata and data generated by executing the plan is organized into **Documents**, which are created by the **RunEngine**.
- Documents in each run are:
    - **Run Start**: metadata known at the start of the run.
    - **Event Descriptor**: schema for the data in the Event + hardware configuration
    - **Event**: actual measurements.
    - **Run Stop**: metadata known only at the end of the run.



DOCUMENTS & METADATA

- There are some things that we know **a priori** before doing an experiment. They are good candidates for inclusion in the Start Document.
- Some information are **experiment specific** and should be included in a single run.



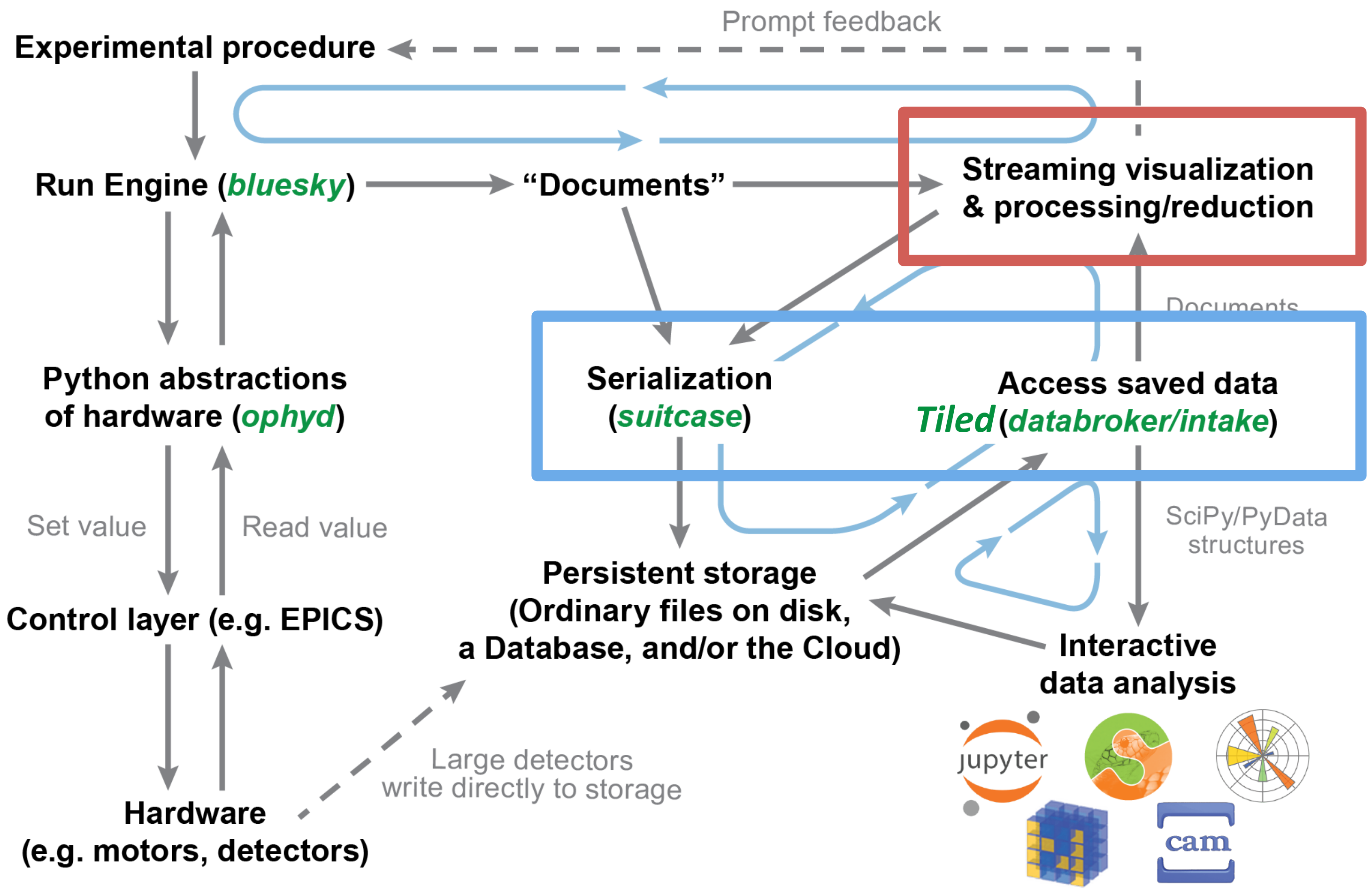
ADDING METADATA

- For each run, the *RunEngine* automatically records: *time*, *ID*, *plan name* and *plan type.*
- Additional metadata can be added **interactively** (for one plan run), **persistently** (for repeated use and/or between sessions).
- Allowed data types are: *strings*, *numbers*, *tuples*, and (nested) *dictionaries.*

```
# only valid to this run
RE(plan(), sample_id='A', purpose='calibration', operator='Luca')

# through a plan with the "md" parameter
def my_plan():
    yield from count([det], md={'purpose': 'calibration'})  # one
    yield from scan([det], motor, 1, 5, 5, md={'purpose': 'good data'})  # two
    yield from count([det], md={'purpose': 'sanity check'})  # three

# reuse metadata on all plans by adding to RE.md
RE.md['proposal_id'] = 123456
RE.md['project'] = 'fusion reactor'
RE.md['dimensions'] = (5, 3, 10)

# link metadata to a directory of files for use between sessions
from bluesky.utils import PersistentDict
RE.md = PersistentDict('some/path/here')
```

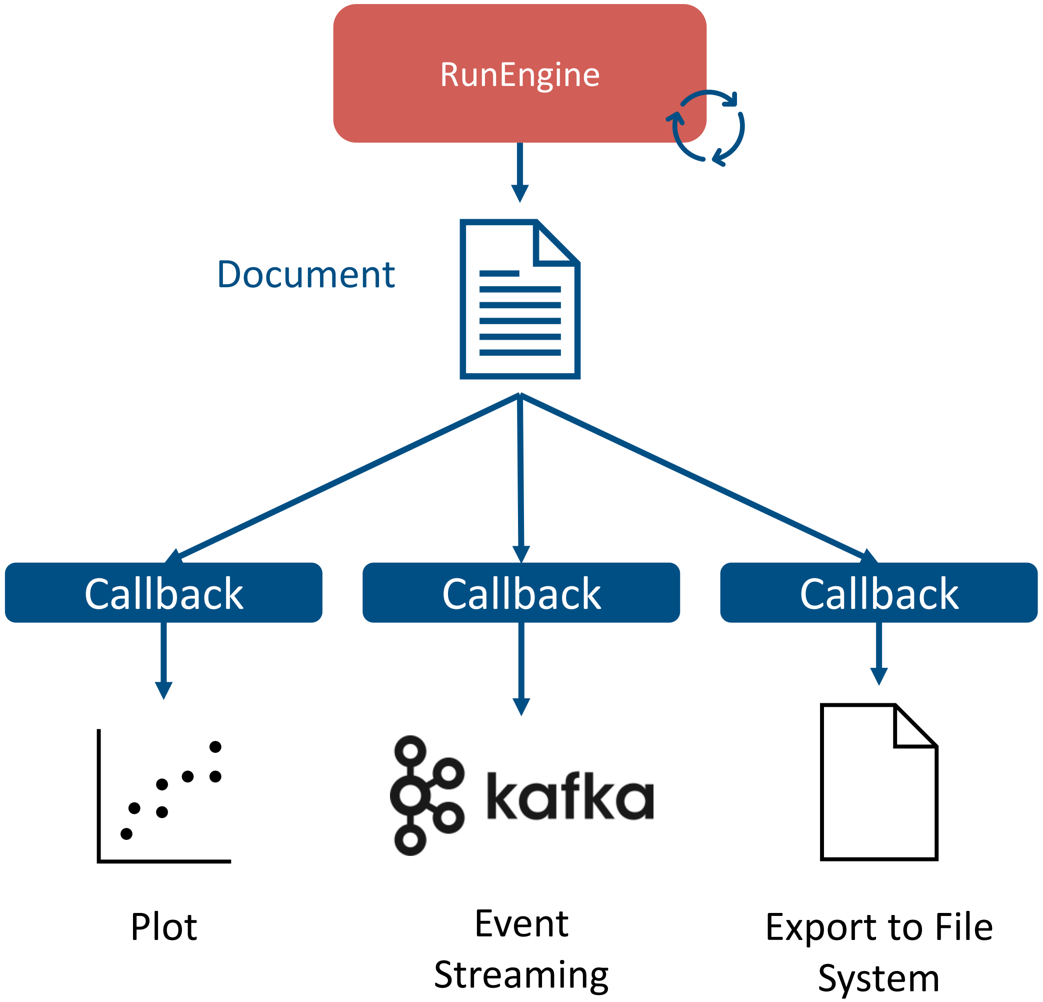
Examples

Architecture Overview



DATA PROCESSING USING CALLBACKS

- Each time a new Document is created, the *RunEngine* passes it to a list of functions. These functions ("**callbacks**") can be used to store the data to disk, print a line of text to the screen, add a point to a plot, or even transfer the data to a cluster for immediate processing.



DATA PROCESSING USING CALLBACKS

- A callback is like a self-addressed stamped envelope: it tells the *RunEngine*, “When you create a Document, send it to this function for processing.”
- Callbacks can be invoked **interactively** (specific to a run) or **persistently** (applied to every plan run).

```
from bluesky.plans import count
from ophyd.sim import det

RE(count([det]), print)

# subscribe the RE to the callback "cb"
# to run it persistenlty
RE.subscribe(cb)
```

VISUALIZATION AND EXPORT

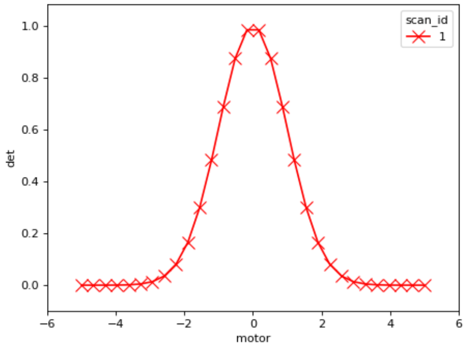
- Pre-assebled callbacks are available in *Bluesky* for plotting, fitting and exporting.

- Examples are:
    - *LivePlot* (plot scalars)
    - *LiveFit* (perform non-linear least squared best fit)
    - *FileWriter* (export to FileSystem)
    - Elog
    - Telegram …

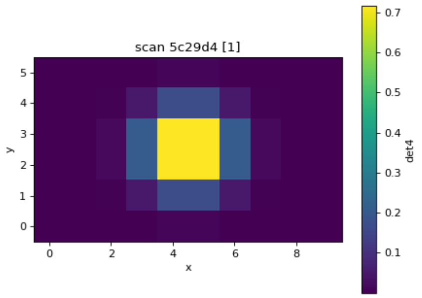
```
from ophyd.sim import det, motor
from bluesky.plans import scan
from bluesky.callbacks import LiveTable

dets = [det]
RE(scan(dets, motor, 1, 5, 5), LiveTable(dets))
```

```
+-----------+------------+------------+----------------+------------+
|   seq_num |       time |      motor | motor_setpoint |        det |
+-----------+------------+------------+----------------+------------+
|         1 | 19:29:51.1 |      1.000 |          1.000 |      0.607 |
|         2 | 19:29:51.2 |      2.000 |          2.000 |      0.135 |
|         3 | 19:29:51.3 |      3.000 |          3.000 |      0.011 |
|         4 | 19:29:51.4 |      4.000 |          4.000 |      0.000 |
|         5 | 19:29:51.5 |      5.000 |          5.000 |      0.000 |
+-----------+------------+------------+----------------+------------+
generator scan ['81631d0a'] (scan num: 1)
```

LivePlot



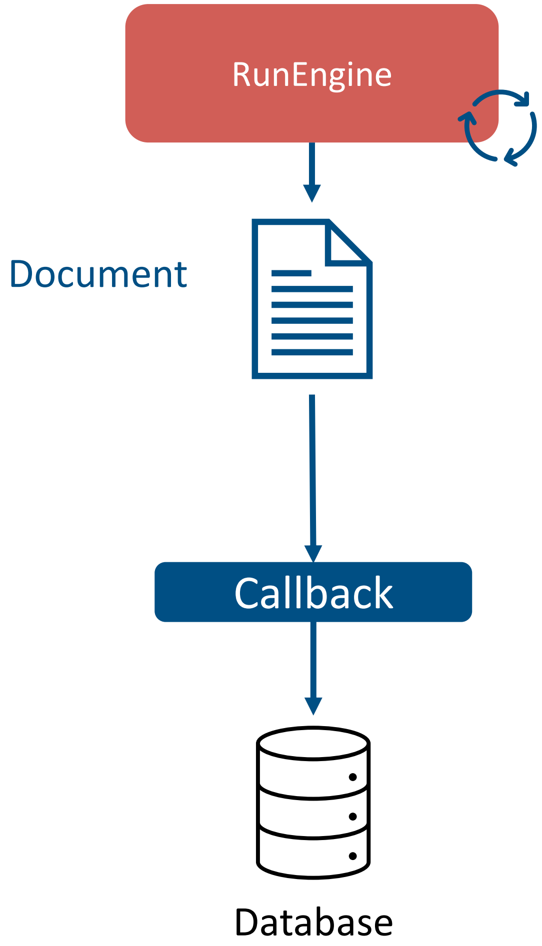
LiveGrid



Examples

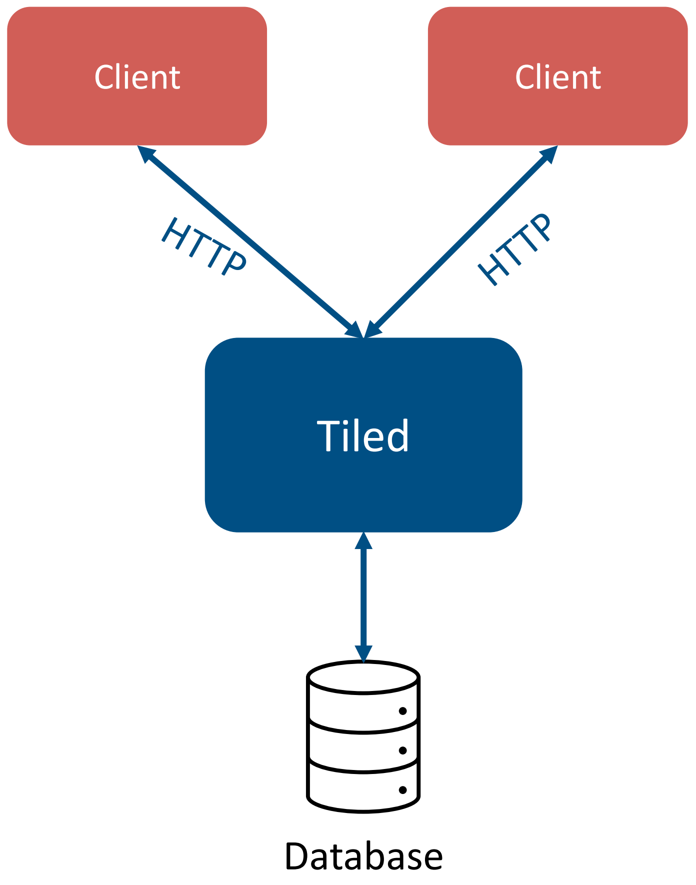
DATA ACCESS

- Documents are usually saved in database for **long-term storage**.
- **Data access** is a key factor in scientific research.
- Effective data access accelerates workflows and allows better production of scientific results, enabling **FAIR Data** (*Findable, Accessible, Interoperable, and Re-usable*).



DATA ACCESS WITH TILED

- Tiled is a **data access** service for data-aware portals and data science tools.
- Tiled's service can sit atop databases, filesystems, and/or remote services to enable **search** and **structured, chunkwise access to data** in an extensible variety of appropriate formats, providing data in a consistent structure regardless of the format the data happens to be stored in at rest.



DATA ACCESS WITH TILED

- Client can use *Tiled* **to read and write** data from storage.

```
from bluesky.callbacks.tiled_writer import TiledWriter

tiled_client = from_uri("http://127.0.0.1:8000")
tw = TiledWriter(tiled_client)
RE.subscribe(tw)

RE(count([noisy_det], num=10, delay=1, md={"Operator": "John"}))

last_data = tiled_client.values().last()
```

DATA ACCESS WITH TILED

- Client can use *Tiled* **to read and write** data from storage.
- **Search** over metadata using the extensive set of supported **queries**.

```
>>> client['short_table'].metadata
DictView({'animal': 'dog', 'color': 'red'})

>>> client['long_table'].metadata
DictView({'animal': 'dog', 'color': 'green'})

>>> client['structured_data'].metadata
DictView({'animal': 'cat', 'color': 'green'})

# etc.
```

```
>>> client.search(Key("Operator") == "John").search(Key("shape") != "circle"))
>>> client.search(FullText("hello"))
```

Examples

- HZB/Tiled

BONUS: IPYTHON MAGICS

- *Ipython* integrated into Jupyter is an interactive python interpreter. It has a very useful feature called "**magics**".
- Magic commands act as **convenient functions** where Python syntax is not the most natural one.
- They help scientists to write instructions in a more clean way and less prone to syntax errors.
- *Bluesky* comes with a set of useful magics.
- It is possible to create **custom magics** and load them into the environment.

```
from bluesky.magics import BlueskyMagics
get_ipython().register_magics(BlueskyMagics)

from ophyd.sim import motor1

%mov motor1 42
```

is equivalent to

```
from ophyd.sim import motor1
from bluesky.plan_stubs import mv

RE(mv(motor1, 42))
```

If IPython's 'automagic' feature is enabled, IPython will even let you drop the % as long as the meaning is unambiguous:

`%mov motor1 42` → `mov motor1 42`

Examples

SUMMARY: BLUESKY IN A NUTSHELL

- **Live, Streaming Data:** Available for inline visualization and processing.
- **Rich Metadata:** Captured and organized to facilitate reproducibility and searchability.
- **Experiment Generality:** Seamlessly reuse a procedure on completely different hardware.
- **Interruption Recovery:** Experiments are "rewindable," recovering cleanly from interruptions.
- **Automated Suspend/Resume:** Experiments can be run unattended, automatically suspending and resuming if needed.
- **Pluggable I/O:** Export data (live) into any desired format or database.
- **Customizability:** Integrate custom experimental procedures and commands, and get the I/O and interruption features for free.
- **Integration with Scientific Python:** Interface naturally with *numpy* and Python scientific stack.

Questions?