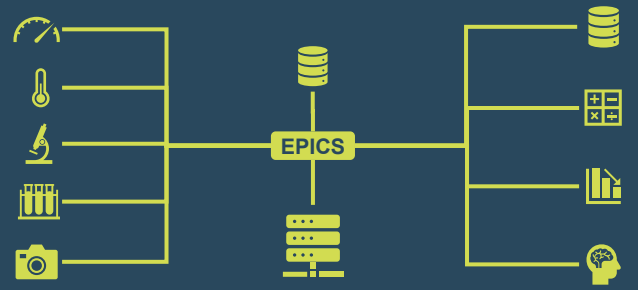
PP&B – Computer Support Group

FRITZ-HABER-INSTITUT
MAX-PLANCK-GESELLSCHAFT

EPICS COLLECTOR

A PYTHON-BASED MODULAR EPICS COLLECTOR FOR ON-DEMAND PV RECORDING



Simeon D. Beinlich[1,*] and Heinz Junkes[1]

[1] PP&B – Computer Support Group
Fritz-Haber-Institut der Max-Planck-Gesellschaft, Berlin, Germany

*beinlich@fhi.mpg.de

PP&B – Computer Support Group

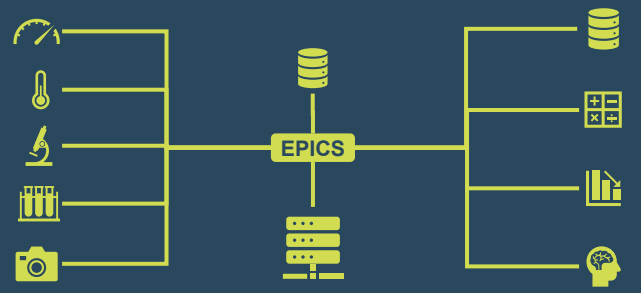
FRITZ-HABER-INSTITUT
MAX-PLANCK-GESELLSCHAFT

EPICS COLLECTOR

A PYTHON-BASED MODULAR EPICS COLLECTOR FOR ON-DEMAND PV RECORDING

EPICS-IN-EPICS-OUT DATA RECORDING & PROCESSING?

SIMPLE PYTHON-BASED MODULAR EPICS "DEVICES"



Simeon D. Beinlich[1,*] and Heinz Junkes[1]

[1] PP&B – Computer Support Group
Fritz-Haber-Institut der Max-Planck-Gesellschaft, Berlin, Germany

*beinlich@fhi.mpg.de

ORIGINAL IDEA: EPICS COLLECTOR

Record data without leaving EPICS

- Record one or multiple PVs on-demand
- Serve history again as PV (array)
- Perform low-level tasks such as:
  - upload to Data-Management-System,
  - export to file,
  - minor processing
  - add non-EPICS data.
- Allow users to run collectors:
  - on their local machine,
  - on a remote host,
  - or both.

… like a ‘large COMPRESS record’ …

Why?

- Users can record:
  - *what they need*
  - *when they need it*
- Direct (live) access to recorded data via PVs
- No additional interfaces / APIs / protocols / data structures for accessing data.
- Treat PV Data and PV History “in the same manner”.
- Modular, distributable, scalable, extendible, …

How?

- Create a ‘Collector Device’ via Python + P4P
- Run it on local machine or on remote sever

~~ORIGINAL~~ IDEA: EPICS COLLECTOR & PROCESSING

Why not do the same for:

- *Data Processing*
- *Data Analysis*
- *Data Visualization*
- *AI/ML Feedback?*
- *…*

… like a 'large CALC record' …

Why?

- Users can process:
  - *what they need*
  - *when they need it*
- Direct (live) access to processed data via PVs
- No additional interfaces / APIs / protocols / data structures for accessing data.
- Treat all Data "in the same manner".
- Modular, distributable, scalable, extendible, …

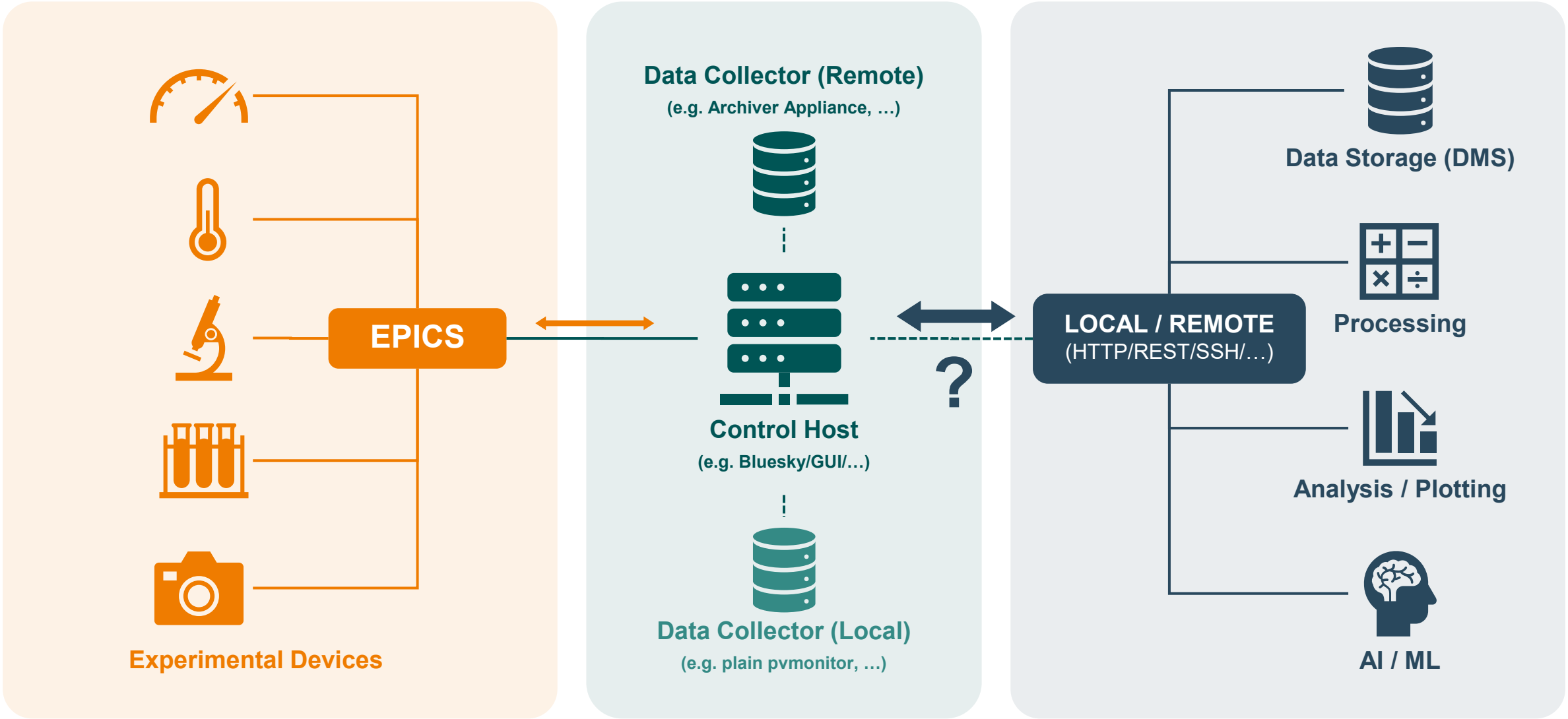
GOAL

Users can 'devicify' processing steps and integrate them directly into their setup.

How?

- Create a 'Processing Device' via Python + P4P
- Run it on local machine or on remote sever

HOW TO ORCHESTRATE DEVICES & COMPLEX PROCESSING?



Orchestrate Devices via EPICS

Orchestrate Local/Remote Processing?

HOW TO ORCHESTRATE DEVICES & COMPLEX PROCESSING?

How to …

… distribute processing?

… communicate with/between processing hosts?

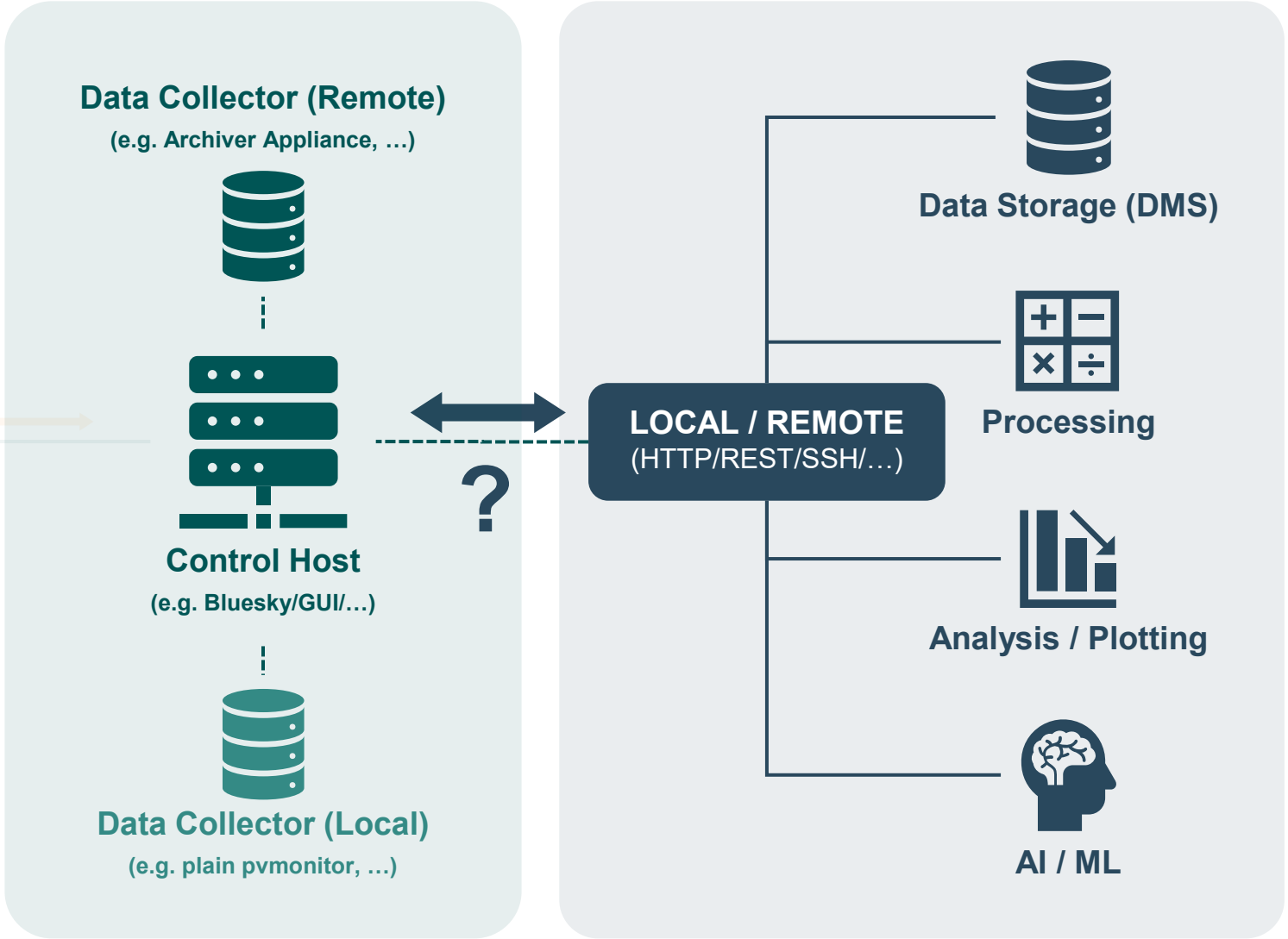
… transfer data?

… handle errors/alarms?

… extend processing?

… feed-back data for closed loops?

… get live-updates?



Orchestrate Local/Remote Processing?

USE EPICS / PVACCESS?

How to …

… distribute processing? ✓

… communicate with/between processing hosts? ✓

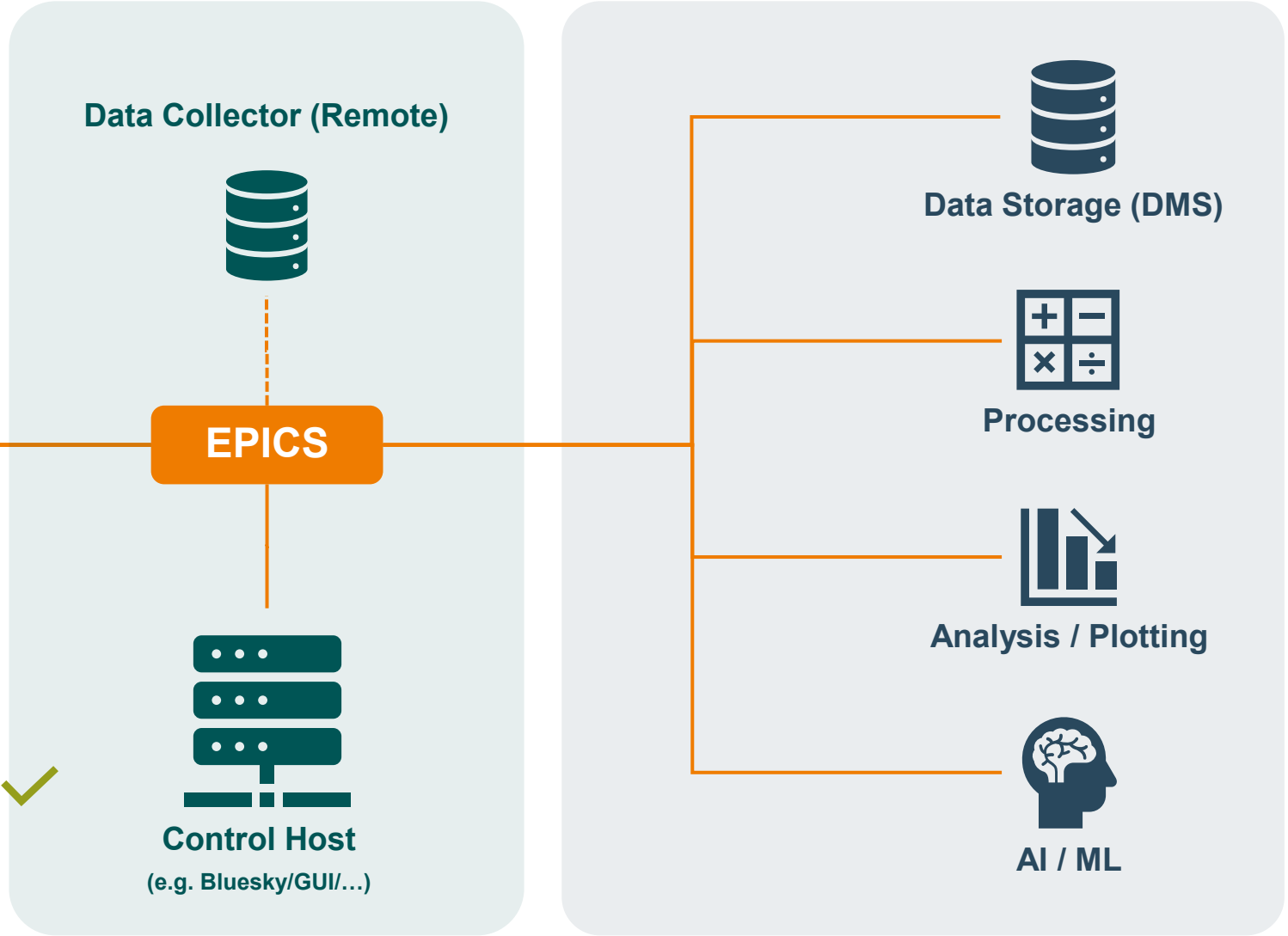
… transfer data? ✓

… handle errors/alarms? ✓

… extend processing? ✓

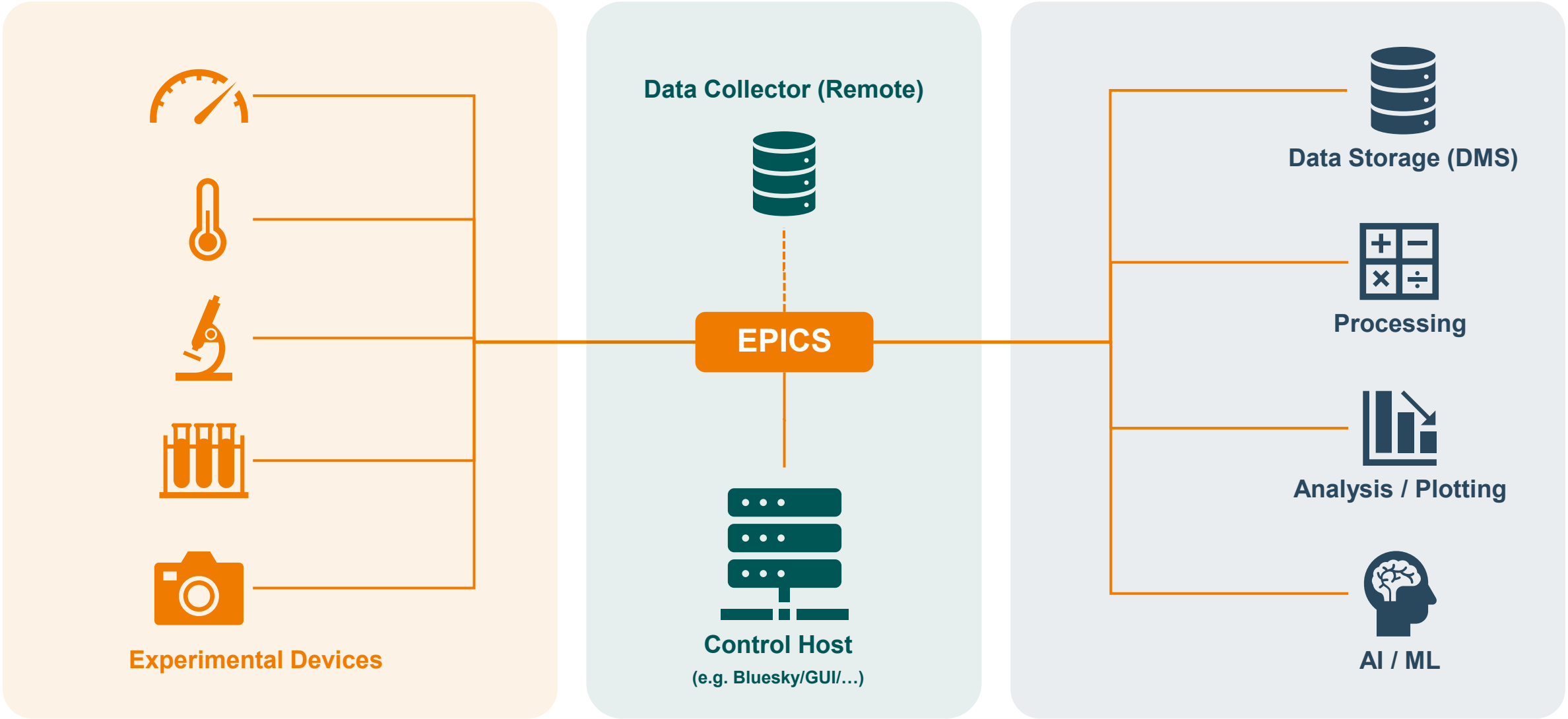
… feed-back data for closed loops? ✓

… get live-updates? ✓



Orchestrate Processing via PVACCES

USE EPICS / PVACCESS?

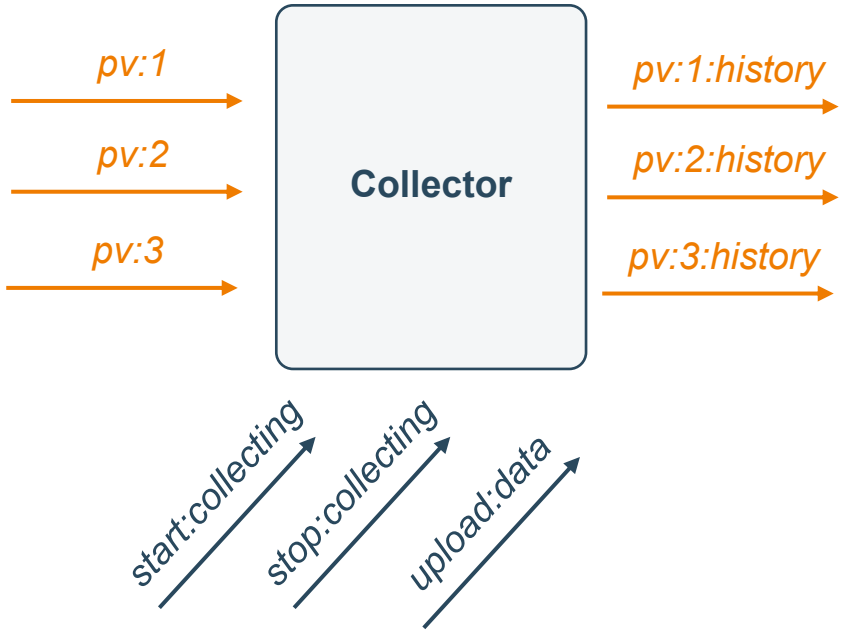


Orchestrate Devices via EPICS

Orchestrate Processing via PVACCES

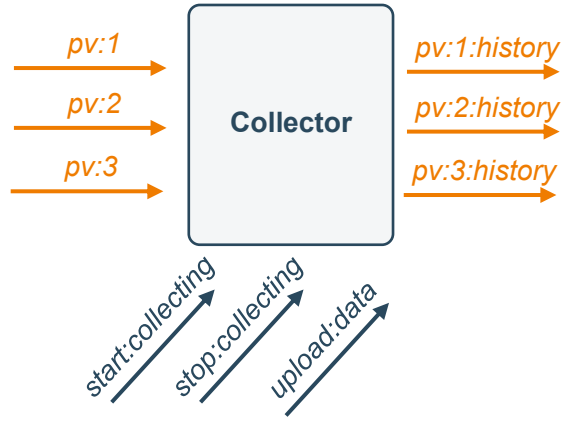
EPICS COLLECTOR “DEVICE”

- Records one or multiple PVs and serves them as array PVs
- Python + PVAccess (P4P client + server)
- Optional easy to use Python client
  (Todo: Ophyd-interface)
- Additional functionalities
  (start, stop, upload, to-hdf5, …)



EPICS COLLECTOR "DEVICE"

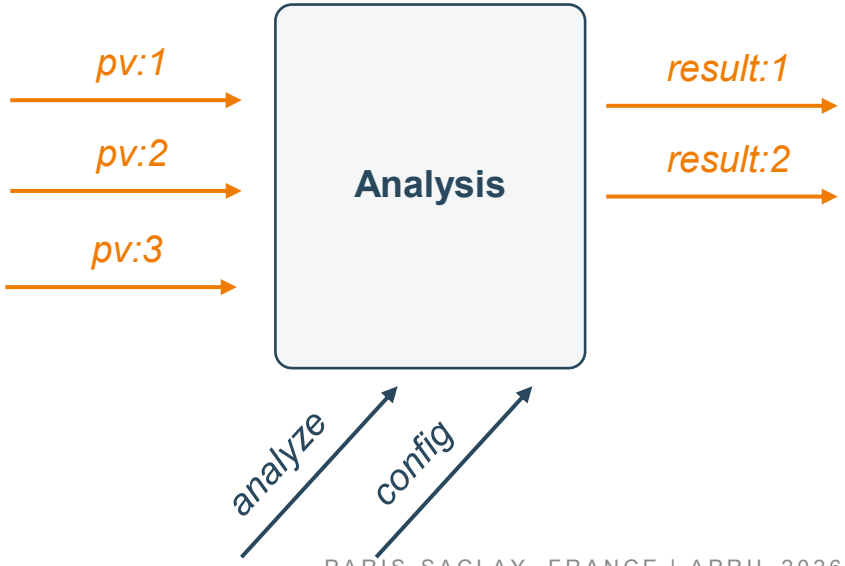
- Records one or multiple PVs and serves them as array PVs
- Python + PVAccess (P4P client + server)
- Optional easy to use Python client
  (Todo: Ophyd-interface)
- Additional functionalities
  (start, stop, upload, to-hdf5, …)



EPICS "PROCESSING DEVICE"

For Processing / Analysis / Visualization / ML …

- Get data from PVs
- Do something
- Serve results as PVs



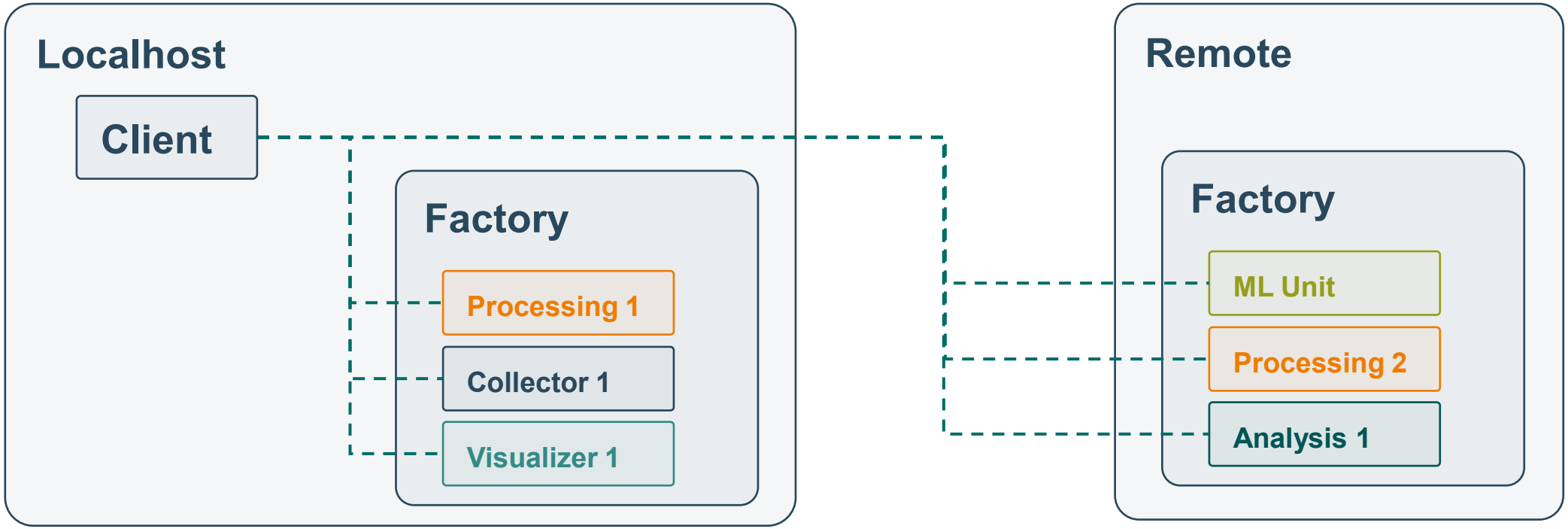
FACTORY DESIGN

How to run multiple devices on local & remote hosts?

Users can:

- Create local & remote devices on-demand
- Switch between local & remote devices

→ Modular, scalable, & **distributable**



WORK IN PROGRESS

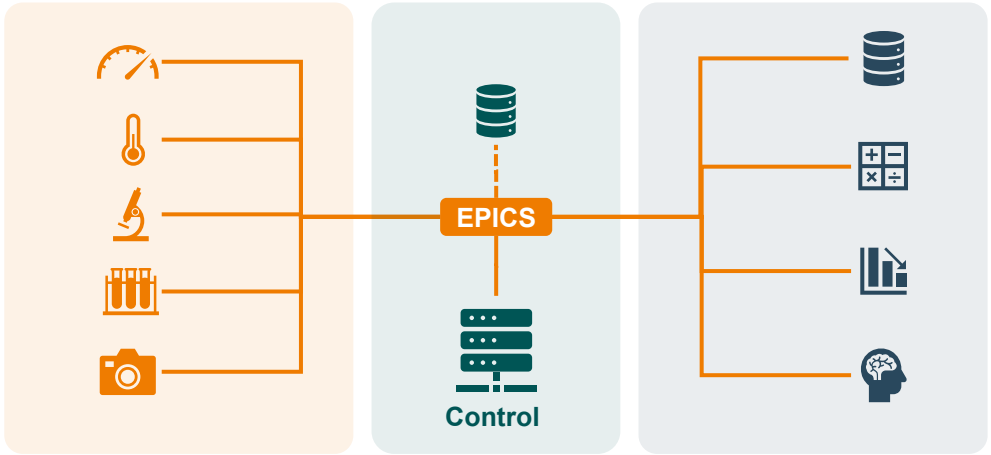
Gitlab Repo: TBA

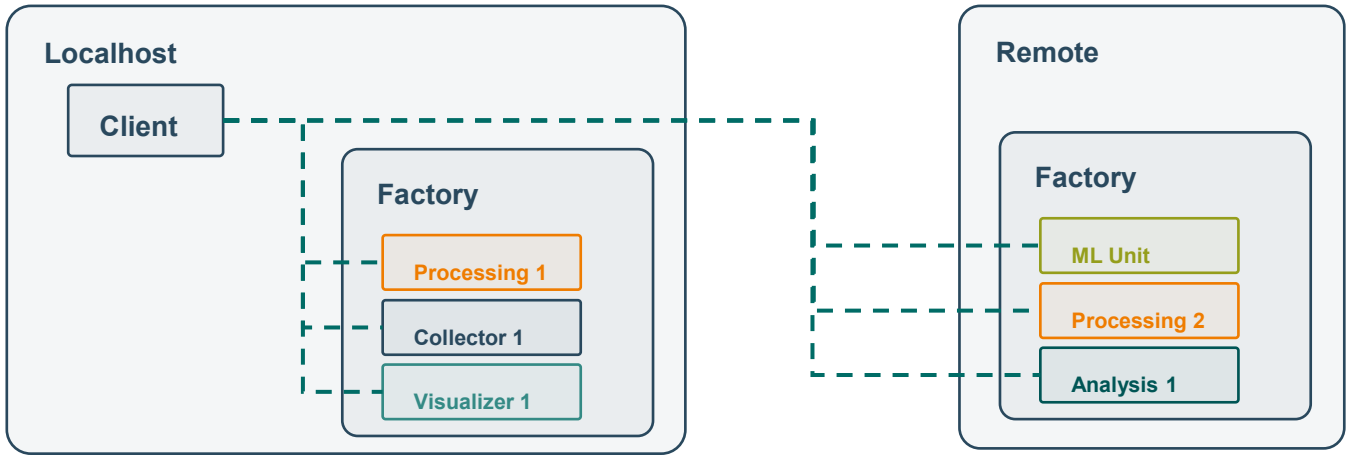
Thanks!

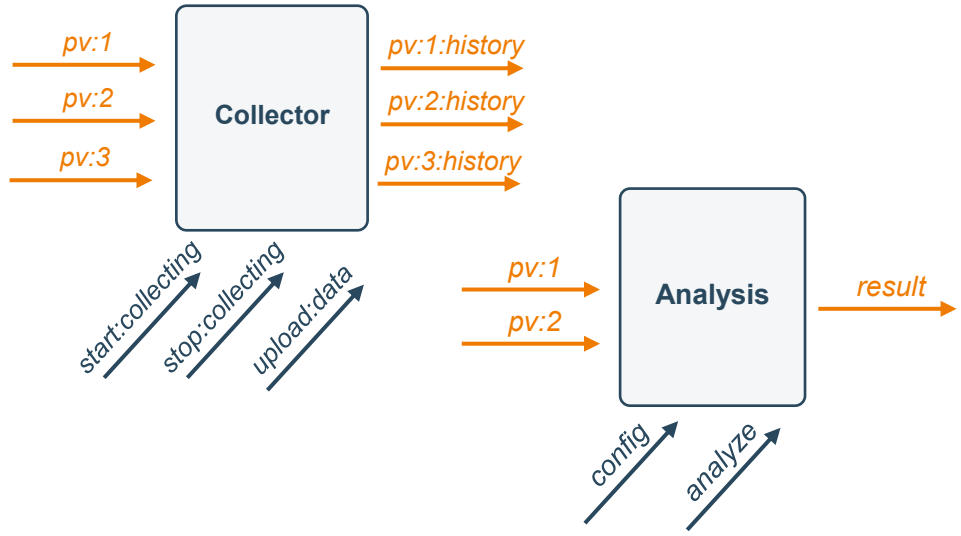
GOAL: “Base”-Toolset for Devices/Factories/Clients

- Similar approaches already in use?
- Experiences with ‘devicifying’ collector/analysis/processing steps?
- Other approaches in use to orchestrate post-acquisition steps in unified way?

→ *beinlich@fhi.mpg.de*







APPENDIX: EPICS-IN-EPICS-OUT FOR ANALYSIS AND PROCESSING?

Chances

- Use existing automation tools for entire loop
  (e.g. bluesky plans)
- Use existing control tools (e.g. Phoebus GUI, Alarm handler, …)
- No “translation” between EPICS and other protocols & data structures
  (e.g. REST API, MQTT, HDF5, CSV, …)
- Services can communicate directly
  (without a control host)
- Distribute anything to prevent Control Host overload
  (“Labview Syndrom”)

Troubles

- Wrap existing processing/analysis procedures into “Devices”? Use an ‘Adaptor’?
- Access / security / network boundaries in EPICS?
- “Common Data Science” does not ‘speak’ EPICS
- When using P4P: PVA only
- Complicated custom PV structs or many ‘primitive’ PVs?
- RPCs vs. PV callbacks?
- Handling custom structs in EPICS is complicated (?)
- More difficult than just running a script on the control host.
  → Need to simplify “devicification” as much as possible

Alternatives for orchestrating data acquisition, analysis & processing?

- Message-broker-based communication for post-EPICS steps (e.g. MQTT) + some data transfer mechanism?
- REST, SSH, 3rd Party orchestration tools, … ? …

APPENDIX: THE METADEVICE BASE CLASS

class MetaDevice:

Base class for devices to make "devicification" as simple as possible.

Default PVs:

- TYPE, STATE, UUID, CONFIG, INFO, ENABLE, CLOSE

Custom PVs:

- Helpers for adding input/cmd/output PVs
- Helpers for simple callback/monitor definition

Device States

- State machine enforcing legal state transitions (created -> ready -> running …)

Start/Stop/Configure mechanisms

Factory Registration

P4P Server & Client handling

- Unique prefix, custom PVS/PVA conf, …
- Close PVs, monitors, …, on device closing

Simple Logging & Error handling

… ? …

```
@register_device_type
class MyDevice(MetaDevice):

    def __init__(self, ...):

        super().__init__(...)

        # add input/output/cmd PVs:
        self.my_pv = self.add_pv(„my:pv“, on_put=self.on_my_pv_put)

        # ... Other INUT/OUTPUT/CMD PVs etc. ...

    def configure(self, config_val):
        # do some configuration ...
        self.state = DeviceState.ready

    def on_my_pv_put(self, value):
        # do something with the new value ...
        self.my_pv.post(value)

    # ... other pv handlers ...

    def start(self):
        # special things to do when device gets enabled ...

    def stop(self):
        # special things to do when device gets disabled ...

    def close(self):
        # special things to do when device gets closed ...
```

APPENDIX: USING A DEVICE

Via PVs (shell, p4p client,…).

- Create
- Configure
- Enable
- Use
- Disable
- (Enable, Use, Disable, …)
- Close

Device Clients (work in progress)

- Simple ‘pure-Python’ interaction
- One client for each device type
- (Direct) Ophyd/Bluesky support
- Factory Client:
  - Create devices on local/remote factories
  - Get corresponding clients back

```python
import epics_meta_devices as emd

# start directly or via a factory (see next slides)
ping_device = emd.devices.collect.Collector('mycollector')
# ... everything else is done via PVs ...
```

```python
import p4p.client.thread

ctxt = p4p.client.thread.Context('pva', nt=False)

# configure PVs to collect:
ctxt.put('mycollector:CONFIG’, {
    'input_pvs': ["dummy:circle:x", "dummy:circle:y"]
})

# enable
ctxt.put('mycollector:ENABLE', True)

# use it (e.g. here: retrieve recorded data):
history = ctxt.get("mycollector:OUT:dummy:circle:x").history
# List of ‘original’ PVs: List[p4p.Value]
# [Value(-0.9455185755993191), Value(-0.93969262078591), ...]

[v.value for v in history]
# v is idential to running ctxt.get("dummy:circle:x")
# [-0.9455185755993191, 0.93969262078591, ...]

# disable the device
ctxt.put('mycollector:ENABLE‘, False)

# close the device
ctxt.put('mycollector:CLOSE‘, True)
```

APPENDIX: THE DEVICE FACTORY

class DeviceFactory:

MetaDevice that allows to dynamically create / delete other MetaDevices.

PVs

- factory:CREATE – {device_type, device_prefix} – creates device
- factory:DELETE – {device_prefix} – stops and removes device
- factory:DEVICES – list of currently running devices
- factory:DEVICETYPES – list of available device classes (~“plugins”)

Device Creation (CREATE callback)

- One multiprocessing.Process per child device ('spawn' context for p4p)
- Process exits when: device.state == closed or close_event + informs cleanup

Device Bookkeeping:

- Process, close_event (factory tells the child to stop), exit_event (child tells the factory it has finished).

Device Deletion (DELETE callback)

- Sets close_event which allows device to close cleanly.

Cleanup (thread listening to exit events)

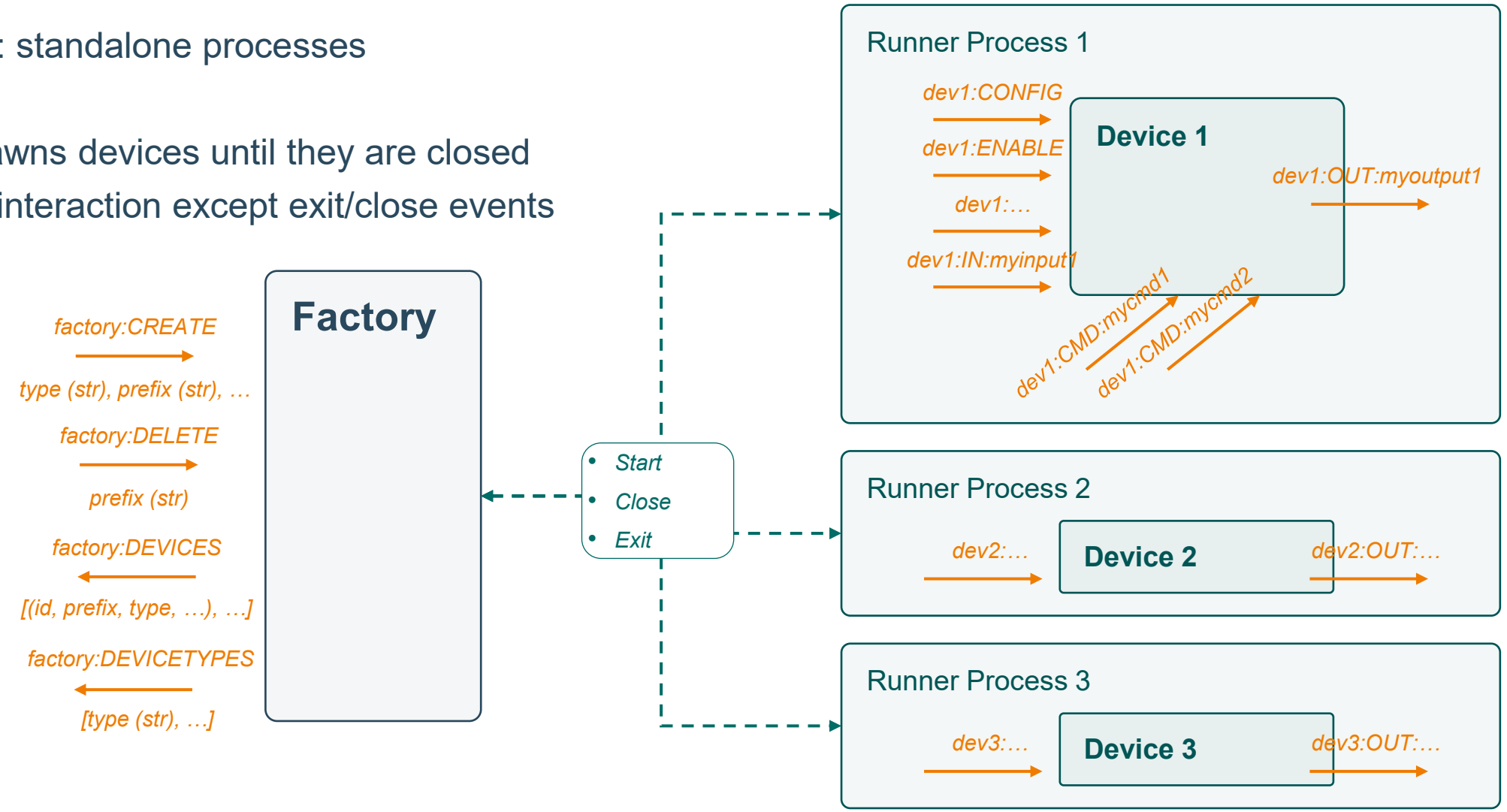
- Removes bookkeeping, processes are cleaned up on exit by construction

```
import epics_meta_devices as emd

# start a new factory
factory = emd.factory.DeviceFactory('myfactory')
# ... everything else is done via PVs ...
```

```
import p4p.client.thread

ctxt = p4p.client.thread.Context('pva', nt=False)

# maybe there will be some things to configure ...
ctxt.put('myfactory:CONFIG', {'configured':True})

# enable the factory
ctxt.put('myfactory:ENABLE', True)

# get device types
ctxt.get('myfactory:DEVICETYPES`)
#> Value(['collect.Collector', 'ping.Ping’])

# create device
ctxt.put('myfactory:CREATE`, {
   'device_type': 'collect.Collector',
   'device_prefix`: 'mycollector`
})

# then, at some point: delete device
ctxt.put('myfactory:DELETE`, {
   'device_prefix`: 'mycollector`
})
```

APPENDIX: DEVICE FACTORY

- Devices: standalone processes
- Factory:
  - Spawns devices until they are closed
  - No interaction except exit/close events



APPENDIX: FORMER EPICS COLLECTOR V1

EPICS Collector

Python – Based

- On-demand recording (~~24/7~~)
- Create, Start, (Use), Stop, Store

TL;DR mode

- Simple to use
- Minimal installation
- Minimal/no config

Distribute

- Local / remote collecting (remote control via REST-API)

→ Dedicated collector hosts

- Serve recorded data

→ Remote analysis/visualization

Extend

- Custom file formats/structures
- Custom DMS
- Custom Processing (local/remote)
- Custom “other” data
- …

```
import epics_collector as ec

client = ec.CollectorManagement()

coll = client.add_collector(
    [„circle:x“, „circle:y“],
)

xs = coll.get(„circle:x“).history
ys = coll.get(„circle:y“).history

...

coll.to_hdf(„test.h5“)
coll.close()
```

```
rcoll = client.add_collector(
    [„circle:p“, „circle:r“],
    type=„remote“,
    url=„collect.example.com“,
)
ps = rcoll.get(„circle:p“).history

...
```