Python Accelerator Middle layer: Architecture and Patterns

Waheedullah Sulaiman Khail
Helmholtz-Zentrum Berlin (HZB)

3rd Python Accelerator Middle Layer (pyAML) workshop at SOLEIL

February 2, 2026

Outline

Why commissioning software is hard

Motivation and high-level design goals

Core problems

- Scale: hundreds—thousands of devices; must be tested for a starting facility yet adaptable.
- Control systems differ (EPICS, Tango, proprietary) — portability issues.
- Applications written for one machine often fail on another.
- Hardware, names, and calibrations drift over time.
- Safety and reproducibility are nontrivial in live operations.
- Systems tend to overgrow — deliberate composition (modularization) is required to keep the stack maintainable.

Why commissioning software is hard

Design goals (driven by problems)

- **Control-agnosticism**: hide control-system details behind a device API.
- **Reproducibility**: store plans, rewritten device plans, and *structured execution events (event streams)*.
    - event stream = ordered records of commands, measurements, and metadata
- **Safety**: explicit policy gates, preprocessing checks, and approvals.
- **Extensibility**: add new measurements/commands without rewriting orchestration.
- **Focus on experimental orchestration:** structuring measurement plans, execution, and safety.

Why commissioning software is hard

Design motivations of `accml`

- SOLID design[a]:
  - **S** — Single Responsibility
  - **O** — Open for extension, closed for modification
  - **L** — Liskov Substitution
  - **I** — Interface Segregation
  - **D** — Dependency Inversion
- Clear info-exchange primitives: (Read)Command style language and event documents.
- Flexible integration: blend into existing control systems and computing infra.

[a] https://events.hifis.net/event/1997/contributions/14995/

Why commissioning software is hard

What needs addressing

- Multiple views of the same system (design vs device).
  1. *Design view:* lattice elements, optics intent, physics parameters.
  2. *Device view:* power supplies, BPMs, channels, hardware limits
- Different labs $\rightarrow$ different environments and naming conventions.
- Upgrades, remeasurement, optimisation $\rightarrow$ changing configurations.
- Offer a flexible "french menu": a' la carte adapters while keeping a tested core.

Yellow Pages: Configuration as a First-Class Interface

- The **Yellow Pages** provide a unified view of the accelerator configuration.
- Users can discover:
  - all devices,
  - device families,
  - logical groupings and roles.

```
# Quadrupoles used for tune correction
tune_correction_quads = yp.tune_correction_quadrupole_names()

# Horizontal steerers
horizontal_steerers = yp.horizontal_steerer_names()

# Vertical steerers
vertical_steerers = yp.vertical_steerer_names()
```

Yellow Pages: Configuration as a First-Class Interface

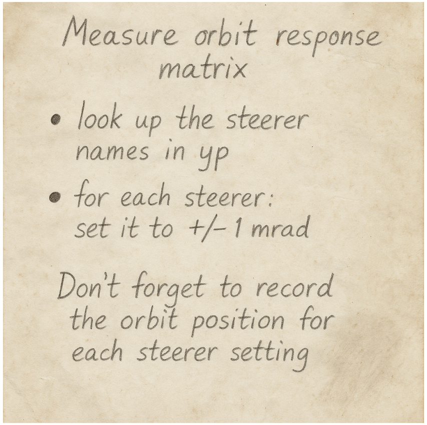
- The **Yellow Pages** provide a unified view of the accelerator configuration.
- Users can discover:
  - all devices,
  - device families,
  - logical groupings and roles.
- Configuration is exposed **independently of the control system**.
- Discovered devices and families are accessed through the **device layer** (ophyd-async), enabling uniform read and write operations.

Accelerator middle layer: Challenges (example)

Design motivations of `accml` — Orbit response measurement

Orbit response measurement (motivation)

- Basis for orbit feedback, LOCO, optics corrections.
- "Granny recipe" approach: conceptually straightforward, but real-world mapping is complex.
- Need to combine design $\leftrightarrow$ device views and translate between them.
- Lattice element $\leftrightarrow$ device: not always direct (e.g. steerer power converter $\rightarrow$ kick on a nearby element).


Measure orbit response matrix

- look up the steerer names in yp
- for each steerer: set it to +/- 1 mrad

Don't forget to record the orbit position for each steerer setting

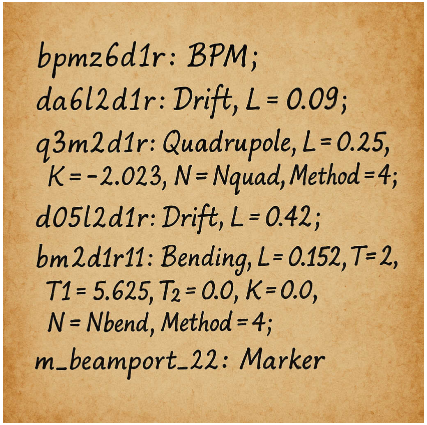
ORM measurement: "Granny recipe" style

Accelerator middle layer: Challenges (example)

Design motivations of `accml` — Orbit response measurement

Orbit response measurement (motivation)

- Basis for orbit feedback, LOCO, optics corrections.
- "Granny recipe" approach: conceptually straightforward, but real-world mapping is complex.
- Need to combine design $\leftrightarrow$ device views and translate between them.
- Lattice element $\leftrightarrow$ device: not always direct (e.g. steerer power converter $\rightarrow$ kick on a nearby element).



bpmz6d1r: BPM;
da6l2d1r: Drift, L = 0.09;
q3m2d1r: Quadrupole, L = 0.25,
K = -2.023, N = Nquad, Method = 4;
d05l2d1r: Drift, L = 0.42;
bm2d1r11: Bending, L = 0.152, T = 2,
T1 = 5.625, T_2 = 0.0, K = 0.0,
N = Nbend, Method = 4;
m_beamport_22: Marker

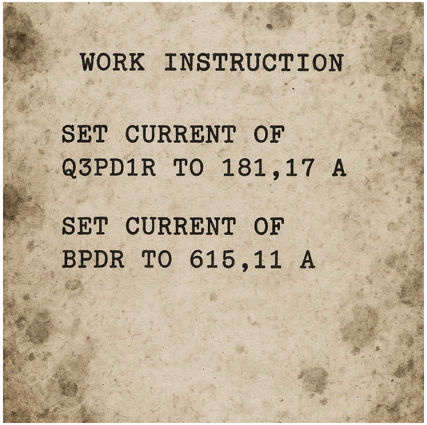
Simulation (top) and machine work instructions (bottom)

Accelerator middle layer: Challenges (example)

Design motivations of `accml` — Orbit response measurement

Take-away challenges

- Gap analysis around current working point → parameters of control models.
- Measurement plans must be FAIR:
  - **F**indable, **A**ccessible, **I**nteroperable, **R**eproducible
- Rewriting design intent to device commands (rewriter + translator + liaison) is essential.


WORK INSTRUCTION

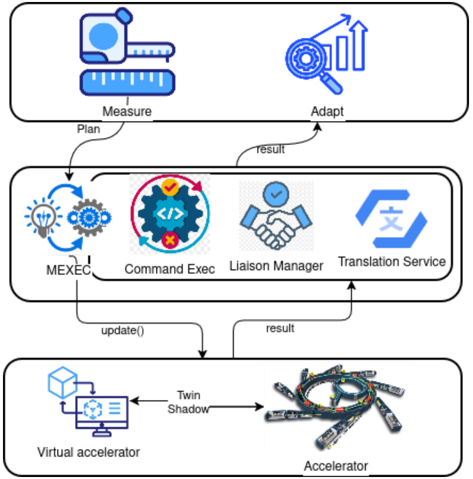
SET CURRENT OF
Q3PD1R TO 181,17 A

SET CURRENT OF
BPDR TO 615,11 A

Simulation (top) and machine work instructions (bottom)

High-Level Architecture

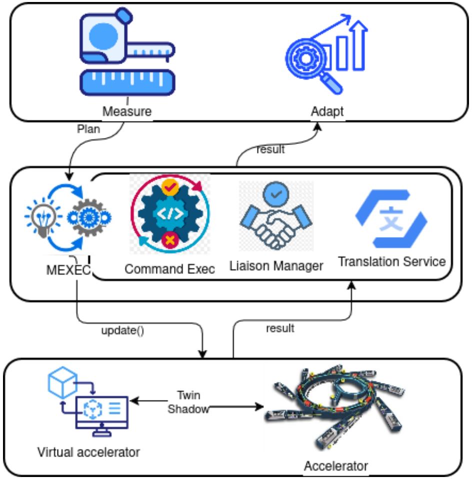
- Measurement Execution Engine – orchestrates execution, retries, sequencing, event emission
- Command Rewriter – converts design plans into device plans
- Liaison Manager – maps logical intent -> concrete devices (Whom to talk to)
- Translation Service – converts physics units <–> device units



High-Level Architecture

To solve the coupling problem, we introduce the **Measurement Execution Engine (MExec)** and strictly layered abstractions.

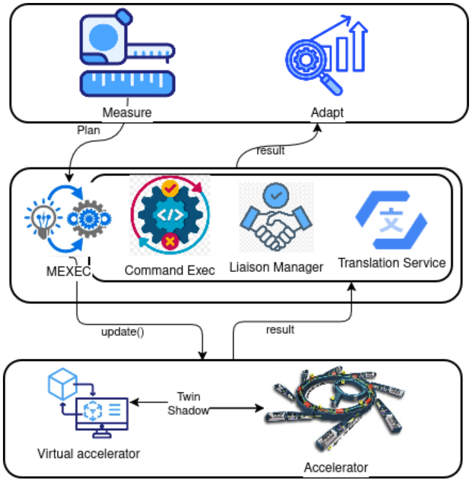
- User intent expressed as plans
- Central orchestration and execution
- Clear separation of responsibilities
- Same workflow for real, twin, and simulation
- *Produce once → store → reuse across real / twin /*



High-Level Architecture

Command rewriter

- Takes a design-level plan and produces a device-level plan.
- Uses Liaison Manager and Translation Service to rewrite commands.
- Stores both plans for traceability and auditing.



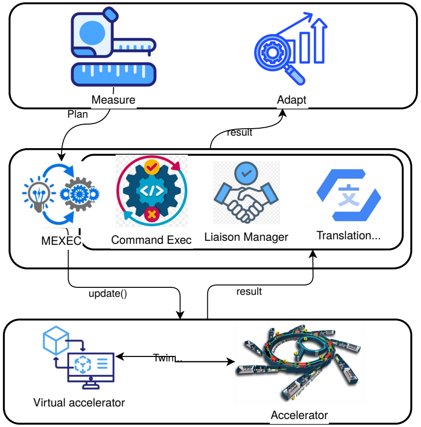
Accml solution: objects communicating by commands

Orbit response

1. produce measurement plan: in a view

Example plan (snippet)

```
# code_snippets/orm_plan.py (example)
steerer_names = yp.get("
     horizontal_steerer")
values = [0, -1e-3, 0, 1e-3]
commands = [
  Transaction([
    Command(id=name, prop="kick",
         value=val)
    for val in values
  ])
  for name in steerer_names
]
```



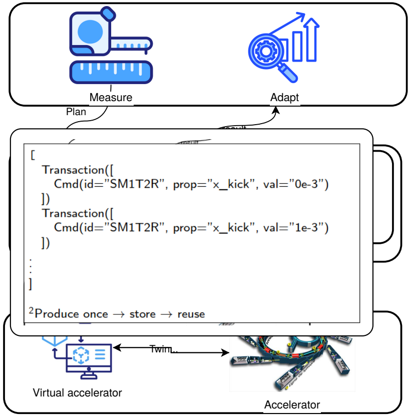
Accml solution: objects communicating by commands

Orbit response

1. produce measurement plan: in a view

Example plan (snippet)

```
# code_snippets/orm_plan.py (example)
steerer_names = yp.get("
     horizontal_steerer")
values = [0, -1e-3, 0, 1e-3]
commands = [
  Transaction([
    Command(id=name, prop="kick",
         value=val)
    for val in values
  ])
  for name in steerer_names
]
```



```
[
  Transaction([
    Cmd(id="SM1T2R", prop="x_kick", val="0e-3")
  ])
  Transaction([
    Cmd(id="SM1T2R", prop="x_kick", val="1e-3")
  ])
:
]
```

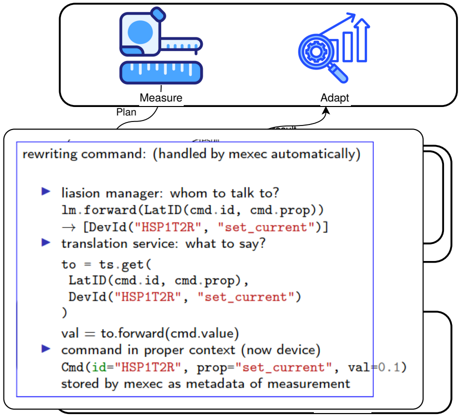
[2]Produce once → store → reuse

Produce once → store → reuse

Accml solution: objects communicating by commands

Orbit response

- 1. produce measurement plan: in a view
- 2. measurement execution engine
  - 2.1 uses command rewriter –> adapt plan if needed using liaison manager and translation service



rewriting command: (handled by mexec automatically)

- liasion manager: whom to talk to?

```
lm.forward(LatID(cmd.id, cmd.prop))
→ [DevId("HSP1T2R", "set_current")]
```

- translation service: what to say?

```
to = ts.get(
 LatID(cmd.id, cmd.prop),
 DevId("HSP1T2R", "set_current")
)
val = to.forward(cmd.value)
```

- command in proper context (now device)

```
Cmd(id="HSP1T2R", prop="set_current", val=0.1)
```

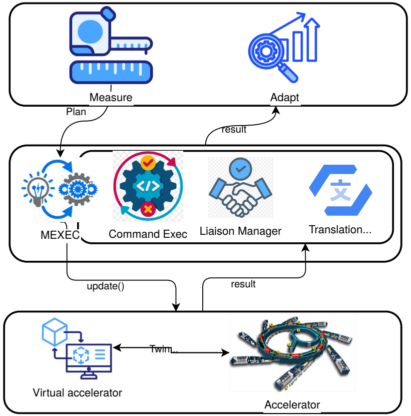
stored by mexec as metadata of measurement

Produce once ⟶ store ⟶ reuse

Accml solution: objects communicating by commands

Liaison Manager

- Bridges design-level intent (lattice elements, beam coordinates) to hardware primitives.
- Maps logical actions (e.g., "apply kick at sector X") to concrete device operations.
- Keeps topology and mapping logic centralized.

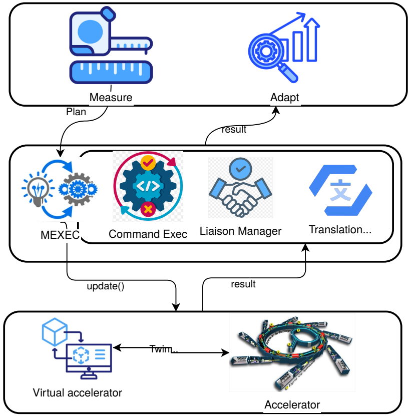


Produce once ⟶ store ⟶ reuse

Accml solution: objects communicating by commands

Translation Service

- Converts between physical units and device units (field $\leftrightarrow$ current).
- Encapsulates calibrations, cross-talk compensation, and scaling.
- Exposes forward and inverse transforms used during plan rewriting.

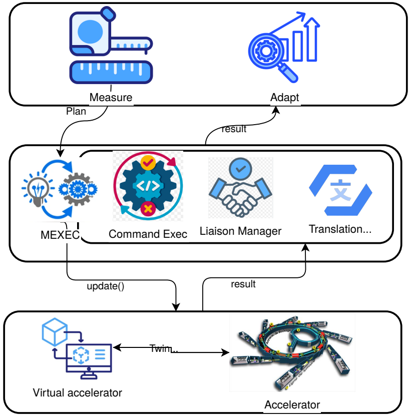


Produce once $\longrightarrow$ store $\longrightarrow$ reuse

Accml solution: objects communicating by commands

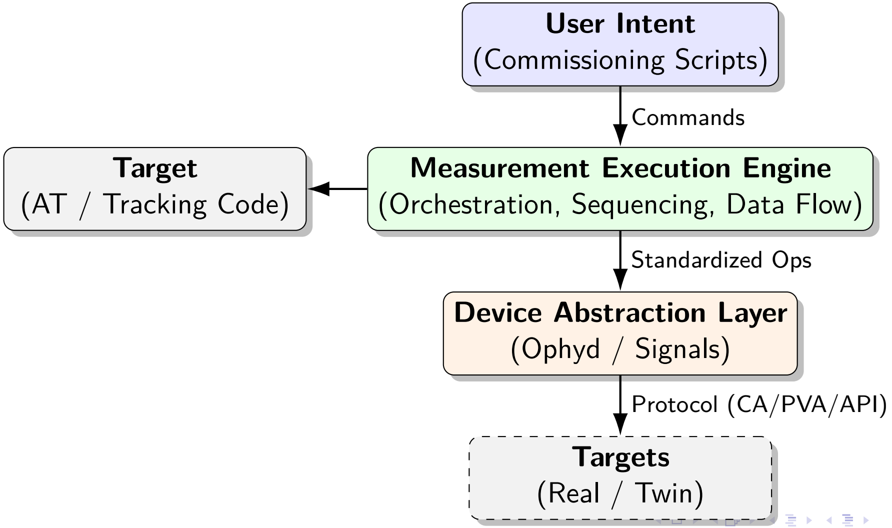
Orbit response

1. produce measurement plan: in a view
2. measurement execution engine
   1. uses command rewriter –> adapt plan if needed using liaison manager and translation service
   2. executes each set of *transactions* → backend
      reads detectors specified by read command: e.g. tune: `ReadCommand(id="tune", prop="transversal")`
3. data storage:
   - serialises commands
   - detectors → return data model → data storage serialises them



Produce once → store → reuse

High-Level Architecture



Separation of concerns

- **Client**: author of the commissioning intent (plans/commands).
- **Execution engine**: responsible for scheduling, retries, and event emission.
- **Device abstraction**: consistent API for operations (read/set/trigger).
- **Backend adapters**: concrete implementations for EPICS/Tango or simulators.

Device abstraction: essential properties

- Async-friendly: supports modern async orchestration.
- Mockable: easy to substitute soft signals for testing.
- Minimal surface area: expose only necessary operations to maintain portability.
- Deterministic semantics: define expected behaviors for set/read/trigger.

Extensibility: adding new measurements

- Define new commands rather than rewriting orchestration.
- Implement handlers and detectors at the device-binding level.
- MExec continues to schedule and ensure safe execution.

Measurement vs analysis (clear separation)

ACCML focuses on structural communication via **measurement plans**; analysis operates on structured event documents produced by MExec.

- Measurement: produce reliable event documents with metadata.
  - Commands —> Measured Data
- Preprocessing: clean and normalize raw measurements.
  - Measured Data —> Physics data
- Analysis: fit models and produce candidate setpoints or decisions.
  - Physics data —> fitted response model
- Policy: correction: decide whether to apply changes to hardware.

Key takeaways

- **Architecture matters:** separation of concerns prevents duplication and reduces risk.
- **Device abstraction and MExec** are core to running the same plan across real/twin/sim.
- **Patterns (Liaison, Translator, Rewriter, MExec)** give structure and repeatability.
- **Yellow Pages** provide a stable, discoverable configuration layer across machines.
- **Preprocessing and policy** are essential for safety and trustworthy automation.
- **Assemble the pieces deliberately:** keep a minimal, well-tested core; localise facility-specific adapters; define explicit mappings and interfaces.

Applicability beyond tune measurements

- Orbit response, chromaticity scans, optics measurements.
- Any measurement that can be expressed as commands + detectors + analysis fits this pattern.

Execution strategies: Bluesky vs non-Bluesky

- Bluesky: offers a RunEngine and an ecosystem for metadata and callbacks.
- Non-Bluesky: simpler direct execution path without the full stack.
- The architecture should support both as interchangeable execution strategies.

Final thoughts

- Treat the twin as a first-class environment for validation.
- Invest in translators, liaison mappings, and policy definitions early.
- Keep the engine and device layer minimal and well-specified.

Extra slide: SOLID principles

- **Single Responsibility (S)** Each component has one clear purpose (e.g. translation, orchestration, policy — not all mixed together).
- **Open–Closed (O)** Extend behaviour by adding components, not by modifying the core engine (new measurements, devices, or policies without rewriting MExec).
- **Liskov Substitution (L)** Real machines, digital twins, and simulators are interchangeable (the same plan must run correctly on all backends).
- **Interface Segregation (I)** Small, focused interfaces instead of large, generic ones (scripts depend only on the signals and operations they actually use).
- **Dependency Inversion (D)** High-level commissioning logic depends on abstractions, not control systems (no direct dependency on EPICS, Tango, or a specific backend).

Goal: keep commissioning software stable, testable, and adaptable as machines evolve.

Thank you!

https://github.com/python-accelerator-middle-layer/accml
https://github.com/python-accelerator-middle-layer/accml_lib
https://github.com/python-accelerator-middle-layer/accml_tango
https://github.com/python-accelerator-middle-layer/epics