Unifying von-Neumann HPC and Neuromorphic Acceleration via the EBRAINS Research Infrastructure: A Framework for High-Performance Workflows

Krishna Kant Singh[1] Charl Linssen[1] Eric Müller[2] Eleni Mathioulaki[3]
Wouter Klijn[1] Lena Oden[4]

[1]Simulation & Data Lab Neuroscience, Jülich Supercomputing Centre, FZ Jülich [2]Kirchhoff Institute for Physics, Heidelberg University [3]ATHENA Research & Innovation Center [4]FernUniversität in Hagen

3rd Workshop on High-Performance eScience (HiPES 2026)
Euro-Par 2026, Pisa · 25 August 2026

Motivation

Can *one* workflow dispatch a neuron simulation or spiking neural network (SNN) on **multiple sites** and **heterogeneous compute** via EBRAINS-RI?

Why?

- **Data privacy** — data may not be allowed to leave the premises, so the computation must come to the data.
- **Energy efficiency** — SNNs run efficiently on neuromorphic hardware.

The vision: one notebook, multiple targets

What if one Jupyter notebook could reach HPC and neuromorphic hardware alike?

Dispatch the **same spiking network** from **one notebook** across **federated HPC sites** and a **neuromorphic substrate** — reproducible across ESD versions, with custom neuron models.

Single identity

One OIDC token for HPC and neuromorphic access.

Reproducible

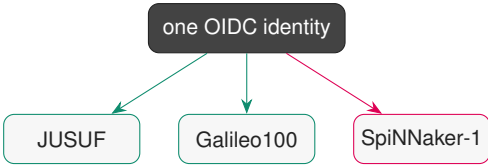
Containers bypass site-specific version drift.

Model-portable

Custom neurons written once in NESTML, compiled per backend.

EBRAINS: a federated research infrastructure for brain science

- **Federated:** many compute sites across Europe, reached through one OIDC login.
- **Software Distribution (ESD):** ∼80 curated, Spack-built packages.
- **Client libraries:** PyUNICORE for HPC, NMPI for neuromorphic — both usable from a notebook.



One login, many federated sites.

Accessible resources

One identity reaches all three

Cloud

EBRAINS Collaboratory:
JupyterLab notebooks
+ Bucket storage

where you work

HPC

von-Neumann
supercomputers

JUSUF (JSC)
Galileo100 (CINECA)

Neuromorphic

brain-inspired chips

SpiNNaker-1
(Manchester)
BrainScaleS-2 (Heidelberg)

BrainScaleS-2 shown for context — reserved for the journal extension.

What is neuromorphic computing?

Spikes are messages — event-driven, energy-efficient

- **Neurons talk in "spikes":** a message is sent *only* when a neuron fires.
- **Neuromorphic chips mirror this:** event-driven — they compute on a spike, idle otherwise.
- **Hence energy-efficient** for spiking models — the workload fits the hardware.



only fires occasionally
⇒ event-driven ⇒ low energy

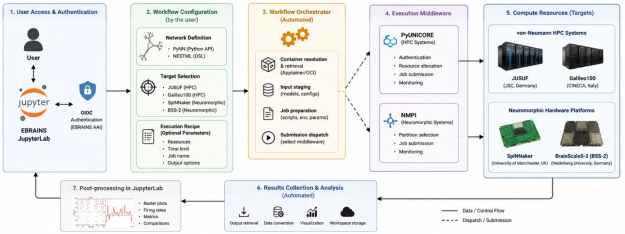
Challenges in multi-site workflows

- **Version drift.** The ESD is deployed independently across sites — the same script can meet a different software version, and can produce different output.
- **Model-portability gap.** PyNN ports *stock* models only; custom neurons must be re-implemented per backend.

Our contributions

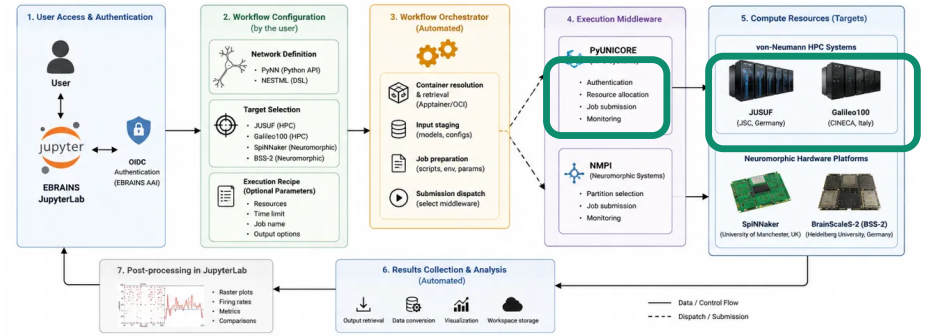
A practical, end-to-end path for hardware-agnostic workflows

1. **Multi-site workflow** — notebook → PyUNICORE → two HPC sites.
2. **Container mode** — PMIx-aware Apptainer, reproducible.
3. **Neuromorphic** — same notebook → SpiNNaker-1 via NMPI.
4. **Model portability** — NESTML: write once, compile per backend.



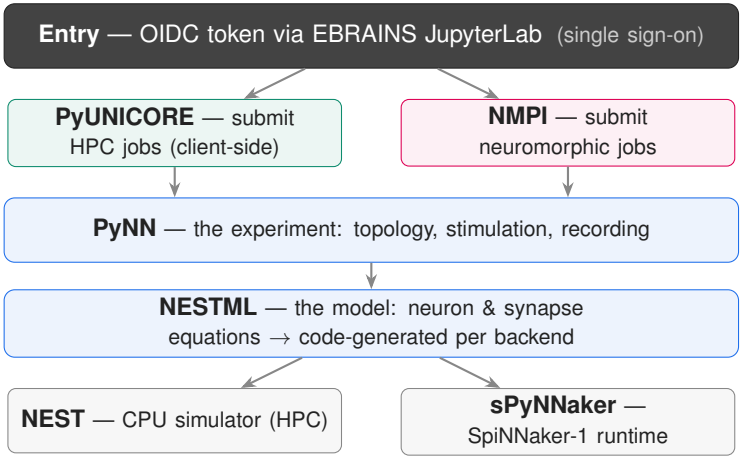
Contribution 1: Multi-site HPC

One notebook targets multiple federated HPC sites



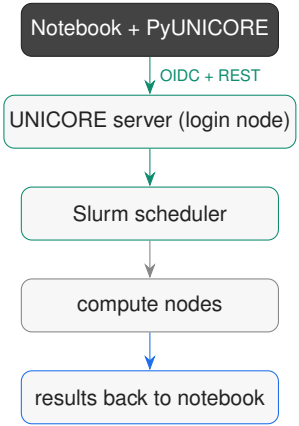
The HPC path (boxed): **PyUNICORE** submits and monitors via one OIDC token; the job description is **site-agnostic** — only the site name varies.

Used software stack (within the ESD)



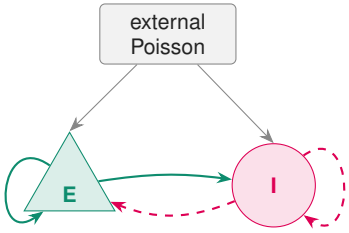
What PyUNICORE does

- **Authenticates** with the OIDC token — no SSH keys, no per-site login.
- **Talks to the UNICORE REST API** at each site's registry.
- **Uploads** the script + batch file and **submits** to the site's scheduler (Slurm).
- **Polls** status and **downloads** results — all from the notebook.

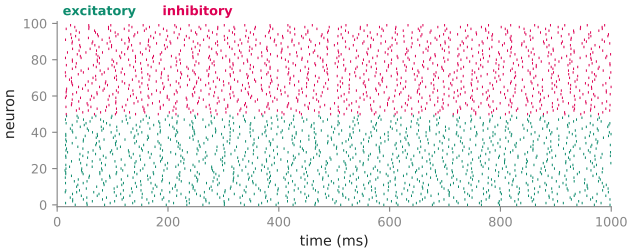


Case study: the Brunel network

A balanced random E/I network — the “hello world” of the field



4:1 E:I · sparse p=0.1 · Poisson drive.



Async–irregular regime (JUSUF)

Contribution 1: Multi-site HPC

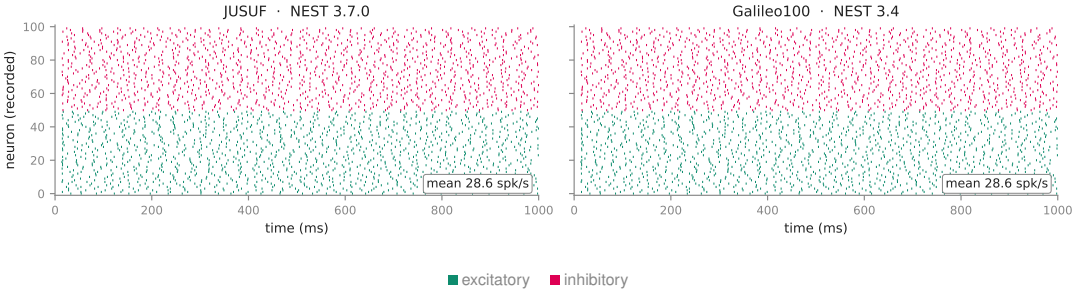
Site-agnostic submission via PyUNICORE

The **job description** is a plain dictionary — reused unchanged across sites; only the registry URL and site name vary. A FAIR, version-controllable artefact.

```
import pyunicore.client as uc
from pyunicore.credentials import create_credential
from clb_nb_utils import oauth as clb_oauth

# Declarative, site-agnostic job description
job_desc = {"Executable": "run_batch.slurm",
            "Resources": {"Nodes": 2, "Runtime": "2h
                "}}

cred = create_credential(token=clb_oauth.get_token
    ())
registry = uc.Registry(cred, "<registry-url>")
site = registry.site("<site-name>") # JUSUF / G-100

job = site.new_job(job_description=job_desc,
                   inputs=["run_batch.slurm",
                           "brunel_alpha_nest.py"])
job.poll() # block until DONE
```

Validation: HPC execution (JUSUF & Galileo100)

Cross-site reproducibility across different ESD versions



Result: same PyNN script, both sites; firing rates match to within noise — mean $\approx$ **28.6 spk/s** on *both* JUSUF (NEST 3.7.0) and Galileo100 (NEST 3.4).

Version drift: the hidden problem

Site-agnostic submission is not enough

- Each site's ESD refreshes on its own schedule, so versions diverge in practice:

  `JupyterLab ebrains/25.04` `JUSUF ebrains/24.10` `Galileo100 ebrains/23.02`

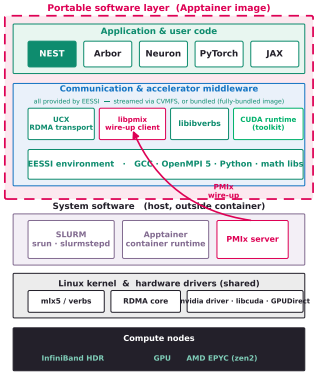
- In our runs a site's **NEST was older** than the notebook's — the *same* script can produce **different numerical output**.
- No error is raised: reproducibility can break **silently** across the workflow.

Fix: portable container execution (this work), or shared deployment via EESSI (future).

Contribution 2: containers

Pin the userspace, not the launcher

- **One Apptainer image** — NEST + Python + MPI.
- **Pull on any site;** no stack install on compute nodes.
- **PMIx** bridges the container MPI and the host launcher.



The portable container layer vs. the shared host system.

Contribution 2: containers

Two-stage submission, PMIx-aware

Two-stage UNICORE submission:

1. **Login node** — pull the pre-built image from EBRAINS Harbor (OCI via ORAS).
2. **Compute nodes** — run inside the container via `srun --mpi=pspmix`.

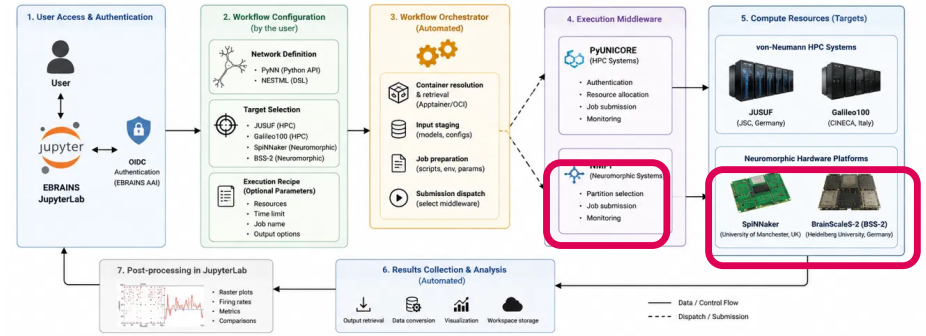
Prior work [Singh et al., HPCW'26]: **negligible overhead** vs. bare-metal on CPU clusters.

```
# Stage 1 (login node): pull image
"User precommand": "apptainer pull ..."

# Stage 2 (compute nodes): sbatch
srun --mpi=pspmix apptainer exec \
  --sharens $EBRAINS_ESD_IMAGE \
  python brunel_alpha_nest.py
```

PMIx bootstraps the containerised ranks across the host boundary.

Contribution 3: neuromorphic

Same notebook → SpiNNaker-1 via NMPI



The neuromorphic path (boxed): change the **target**, not the experiment — the same network is submitted via **NMPI**; the scientific intent is unchanged.

Contribution 3: neuromorphic

One-line change from the HPC path

NMPI exposes SpiNNaker as a remote service: upload a script + descriptor; the platform schedules it.

Swapping `platform="SpiNNaker"` for a UNICORE call is the *only* change — the network is untouched.

```python
from nmpi import nmpi_user

client = nmpi_user("EBRAINS",
    token=clb_oauth.get_token())

job = client.submit_job(
    source="brunel_alpha.py",
    platform="SpiNNaker",
    collab="my-collab")
```

HPC: `import pyNN.nest as sim` vs. NMC: `import pyNN.spiNNaker as sim`

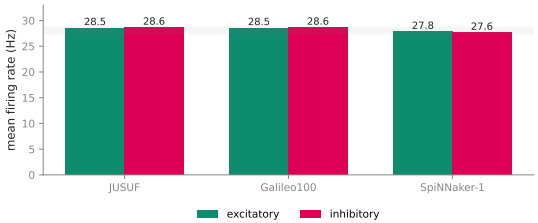
Validation: neuromorphic execution (SpiNNaker-1)

Real-time execution with a one-line change

```python
# HPC
import pyNN.nest as sim
# SpiNNaker (one-line change)
import pyNN.spiNNaker as sim
```

Runs in **biological real-time**; spikes return via NMPI, same analysis path as HPC.

Mean $\approx$ 27.7 spk/s (E 27.8, I 27.6) — within $\sim$1 Hz of HPC.

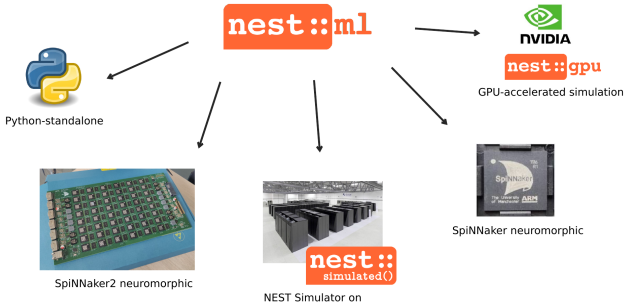


Contribution 4: NESTML — two layers of portability

Write the model once, generate code per backend

- **PyNN** → portable *experiment* (topology, protocol).
- **NESTML** → portable *model* (neuron / synapse equations).
- **Custom dynamics** are where API-only portability breaks — NESTML closes it.

Status: a **single neuron** is generated and run on both backends today; using NESTML as a full network DSL is **work in progress**.



One NESTML model → many engines. Credit: C. Linssen.

Contribution 4: NESTML

Write once, generate per backend

The `.nestml` file is *never edited* between targets — the toolchain knows the backends.

The SpiNNaker path additionally needs an ARM cross-compiler and dev stack for mapping and routing — all bundled in the ESD.

```
from nestml import generate_code

# Target: NEST (C++ module)
generate_code(model, target="NEST")

# Target: SpiNNaker (Python module)
generate_code(model, target="SpiNNaker")
```

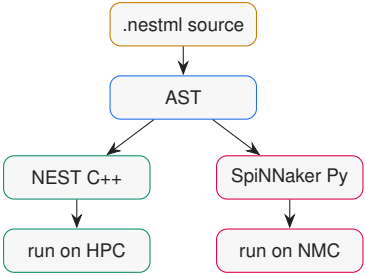
Today: a single custom neuron. Full network-level DSL use is in progress.

Validation: NESTML code generation

Custom neuron model, one source, both backends

- A single `.nestml` neuron model is written in JupyterLab.
- The notebook generates code per target: **NEST** C++ (via PyUNICORE) and **SpiNNaker** module (via NMPI).
- The `.nestml` file is **never edited** between targets.

Scope today: **single-neuron** generation & simulation on both backends; using NESTML as a full network DSL is ongoing.



What we demonstrate — and what we do not

Demonstrated

- ✓ Federated OIDC authentication
- ✓ Cross-site HPC submission (PyUNICORE)
- ✓ Zero-install container execution
- ✓ PMIx launcher / container interop
- ✓ Neuromorphic submission (NMPI)

Not claimed

- ✗ A general bitwise-identity guarantee
- ✗ Equal precision / timestep semantics
- ✗ End-to-end performance equivalence
- ✗ Energy-efficiency comparison
- ✗ Full network-level NESTML DSL

Lessons learned

- **Reproducibility is the dominant practical problem.** Federated sites deploy software independently — a site's simulator can be older than the notebook's, changing results silently. Containers fix it.
- **Authentication is the easy part.** One OIDC token reaches every UNICORE site and the NMPI gateway — the reason the workflow fits in a single notebook.
- **Portability lives at the model layer.** PyNN suffices for stock models; custom neurons need NESTML, a DSL whose compiler knows the backends.

Conclusion

One notebook, three targets, two paradigms, one pipeline

1

Unified identity

One OIDC token drives PyUNICORE and NMPI from one notebook.

2

Reproducible execution

PMIx-aware Apptainer containers eliminate version drift.

3

Model portability

NESTML: “write once, compile for NEST or SpiNNaker”.

Validated across JUSUF, Galileo100 and SpiNNaker-1 on the balanced random network.

Future work

Closing the remaining gaps

- **Streaming containers** — build container with **EESSI** / **CVMFS**.
- **Extend to BrainScaleS-2** — add the accelerated analog substrate as a third backend.
- **Energy comparison** — measure and compare energy-to-solution across HPC and neuromorphic runs.
- **NESTML as a full DSL** — extend from a single neuron to the whole network model.

Acknowledgments & references

Funding: EC Horizon 2020 — HBP SGA3 (GA 945539) • EC Horizon Europe — EBRAINS 2.0 (GA 101147319).

[1] Singh *et al.* (2026). *HPC Containers for EBRAINS.* arXiv:2603.12044 (HPCW'26).

[2] Senk *et al.* (2017). *A collaborative simulation–analysis workflow . . . using HPC.* LNCS 10164.

[3] Pedersen *et al.* (2024). *Neuromorphic Intermediate Representation.* Nature Communications 15, 8122.

[4] Linssen *et al.* (2025). *NESTML: a generic modeling language.* Frontiers in Neuroinformatics 19.

Questions?

Contact: k.singh@fz-juelich.de