Designing a Cross-Platform Reproducibility Plugin System for Neuromorphic Tools

Neuromorphic computing platforms vary widely (from software simulators like Brian2 or NEST to hardware like Loihi or SpiNNaker), which makes it challenging to **reproduce experiments across different systems**. Even with a solid theoretical foundation for neural dynamics, the *heterogeneity of neuromorphic software and hardware stacks* means results are often hard to replicate on another platform [1]. To address this, we can design a **flexible plugin/API layer** that each framework can adopt to support a common *"executable reproducibility"* standard. Below, we outline design patterns for such plugin systems, examine relevant examples from other domains, and recommend strategies for cross-framework compatibility, packaging, and sandboxing to ensure **standardized execution and reproducibility**.

Plugin Architecture Patterns in Scientific Software

A well-designed plugin system defines clear **extension points** in the core software, allowing new modules to be added without modifying the core. This is common in scientific tools that need flexibility:

- **Jupyter Kernels (Language Plugins):** Jupyter decouples the user interface from execution by using a **standard message protocol** for kernels. Front-ends (notebooks, IDEs) communicate via JSON over ZeroMQ to a kernel process that executes code [2]. Thanks to this design, *supporting a new language* in Jupyter simply means implementing the kernel API in that language – no changes needed in Jupyter itself. This plugin-like architecture has enabled dozens of languages to plug into the same interface [3]. The key lesson is to define an *agnostic protocol* or interface so that different backends (here, programming languages) can all respond to the same commands.

- **OpenMM's Modular Plugins:** OpenMM, a molecular simulation library, allows new features (e.g. novel force calculations or hardware backends) to be added as **dynamic plugin libraries**. These are placed in an OpenMM `plugins` directory and loaded at runtime by the framework [4]. Each plugin implements a defined API (registration functions, etc.) so OpenMM can discover and use it [5]. This pattern shows the value of a *modular, runtime-loaded* plugin: neuromorphic simulators could similarly load "execution plugins" that conform to a standard interface for running experiments. The core framework would provide hook points (like OpenMM's `registerKernelFactory` calls [5]) where plugins register themselves.

- **PyTorch Hardware Backends:** In machine learning, frameworks like PyTorch support extensibility by allowing new hardware or accelerator backends via a plugin interface. For example, PyTorch's XLA integration uses the OpenXLA API to let *TPUs and other custom devices* execute PyTorch models with minimal code changes [6]. This is analogous to neuromorphic contexts: a plugin system could enable different neuromorphic chips or simulators to execute a given model through a unified API. The **dispatcher pattern** in PyTorch (registering custom operators for a new device) shows how a core system can call into plugin-provided implementations depending on context.

- **Other Extensible Tools:** Many scientific tools use plugins for flexibility. For instance, image analysis platforms allow **plugins for new file formats or algorithms**, and data processing frameworks use plugin modules for custom I/O or computation. The general design pattern is to **expose stable extension points** (APIs or configuration hooks) and document them clearly so that contributors can create modules that "plug in" seamlessly. In our case, the extension point would be the *execution of a neuromorphic experiment* – the plugin interface should define how an external module receives a model or script and produces standardized results.

Key design principles emerge from these patterns: define a clear interface/protocol for the plugin, keep it language-agnostic (e.g. use common standards like JSON messaging or CLI arguments), and ensure the core system treats plugins as *first-class citizens* (e.g. auto-discover them, call standard init functions, etc.). This allows new neuromorphic platforms to be "slotted in" with minimal changes to the overall infrastructure.

Cross-Framework Compatibility Strategies

To make a single experiment executable on **multiple neuromorphic backends**, we should introduce an abstraction layer. Several successful examples from computational neuroscience and AI highlight how abstraction facilitates reproducibility:

- **Common APIs like PyNN:** *PyNN* is a Python API that lets researchers define spiking neural network models in a *simulator-independent* way [7]. Code written with PyNN can run **unmodified on different simulators** (NEST, NEURON, Brian2, etc.) or even certain neuromorphic hardware [8]. This works because each backend provides an adapter implementing the PyNN API. The result is a *unified interface*: one script can be executed on multiple platforms for cross-verification [9]. Adopting a similar common API in our plugin system would greatly enhance portability. For example, we could define a standard set of calls (to create neurons/synapses, run simulation, retrieve outputs) that each framework's plugin must implement. Indeed, PyNN has become *"the most widespread common interface"* bridging software simulators and neuromorphic hardware [10] – a testament to how a well-designed API can foster compatibility.

- **Standard Model Descriptions (NeuroML & NIR):** Another approach is using a *common model format* as an intermediate representation. **NeuroML**, an XML-based language for neural models, was developed to allow model sharing across tools. Models in NeuroML can be run on multiple simulators, enabling *cross-simulator validation* and reuse of model components [11] [12]. In practice, researchers converted models to NeuroML and observed *similar behavior across five different simulators*, proving that a standardized description improves reproducibility [12]. More recently, the **Neuromorphic Intermediate Representation (NIR)** was proposed as a unified instruction set for spiking models. NIR defines common primitives and abstractions that *bridge differences between neuromorphic implementations*, making it possible to reproduce the same model across **7 simulators and 4 hardware platforms** [13]. These examples suggest that our plugin system could leverage a common representation: for instance, the plugin API might accept a model described in a standard format (like PyNN, NeuroML, or NIR) and translate it to the platform's native configuration. This *translation layer* means experiment descriptions aren't tied to one framework's quirks, aiding both portability and consistency.

- **Language-Agnostic Interfaces:** To be truly flexible, the reproducibility API should be accessible from any language. One option is to implement the plugin interface as a **command-line tool or service**. For example, each neuromorphic framework could provide an executable (or a container entrypoint) that accepts a standardized input (JSON/YAML config, or a script name) and produces results in a standard format (e.g. a CSV or NeuroML/SONATA output). A researcher or the journal's system (JENS) could then invoke any platform's plugin through the same CLI syntax. This approach was used in some workflow systems (common **CLI wrappers** for different tools) and ensures that even if one platform is in Python and another in C++, they both can be driven uniformly. Another approach is a lightweight **RPC protocol** (even something like a REST API or ZeroMQ messaging similar to Jupyter) that plugins can implement to receive experiment run commands. The goal is *minimal invasiveness* – frameworks shouldn't need a full rewrite, just a thin adapter layer. If many neuromorphic tools are already Python-based, a **Python plugin interface** could be natural (using Python entry points to discover plugins). But to cover those that aren't Python, sticking to a process/ CLI boundary (which any language can interact with) might be better for language agnosticism.

- **Adapter/Bridge Pattern:** Internally, each plugin can act as an **adapter** between the standard API and the framework's native API. This follows the *Bridge design pattern*: the plugin translates generic calls (e.g. "load model", "set parameters", "run simulation for T milliseconds") into the specific library calls or hardware instructions of that platform. As an example, one could write a `NestPlugin` that takes a network description (maybe in PyNN or NEST's native SLI/Python) and executes NEST, or a `LoihiPlugin` that takes a NxSDK or Lava script and runs it on Loihi hardware, but in both cases the *external interface* (methods and outputs) are the same. This separation makes the system *extensible*: when a new neuromorphic framework comes along, one only needs to implement the adapter for it, without affecting existing ones. In practice, the PyNN project showed this can be done incrementally – they ported existing model code to PyNN gradually, replacing simulator-specific pieces with PyNN calls one by one and verifying the behavior stays the same [14]. That suggests we can wrap existing simulators piecewise, ensuring the plugin doesn't demand a complete overhaul of the codebase.

Cross-framework compatibility is crucial for an executable standard: it means a published experiment could be run on a different simulator or hardware to validate results, simply by switching out which plugin is invoked. Adopting common APIs or formats (like PyNN or NIR) in the plugin design will greatly aid this goal, as it has in other domains (analogy: ONNX for neural networks allowed models to move between frameworks [15]).

Packaging, Metadata, and Versioning for Reproducibility

Having a common interface is one side of reproducibility; the other is ensuring the *environment* and *execution context* are consistent. Here, we leverage best practices in packaging and metadata:

- **Containerization of Environments:** The use of containers (Docker for general use, Singularity for HPC) has become a gold standard for reproducible science. Containers bundle the OS, dependencies, and libraries so that software runs identically on any machine [16]. For our plugin system, each neuromorphic framework could provide a **Docker/Singularity image** with the framework installed and the plugin interface implemented. Running an experiment then might boil down to launching the container with the experiment specification mounted. The Journal of Executable Neuromorphic Science (JENS) could even require authors to submit a container image (or

Dockerfile) along with their plugin code. This guarantees that *the same software versions* are used when reviewers or readers execute the experiment. As one example, the *Singularity* container system was explicitly developed to bring reproducible environments to scientific computing – allowing researchers to "easily copy and execute complete environments on other platforms" [17]. By using such containerized plugins, we ensure that even years later, the neuromorphic code will run as originally intended, unaffected by OS updates or dependency changes.

- **Reproducibility Metadata:** Along with the code, we should include **metadata** that describes the execution context. Each plugin could automatically output a metadata file containing information like: framework name and version, plugin version, OS details, dependency list (e.g. pip freeze or conda environment), and hardware specifics (if relevant, e.g. Loihi board version). This metadata might be embedded in the container (labels or README) and also produced at runtime for record-keeping. For instance, a plugin could have a method `get_environment_info()` that returns all relevant versions and configurations, which is stored alongside results. This practice aligns with emerging standards in reproducible research: *reproducibility packages* often include everything needed to rebuild the run [18]. In fact, a recent framework called *SciRep* automates packaging experiments with their code, data, dependencies, and execution commands, producing a *capsule that holds everything needed to re-run the experiment anywhere* [18]. We should strive for a similar all-inclusive packaging. By capturing the exact versions and configuration in use, we make it possible to re-launch an experiment even on a different machine or years later, confident that it's the "same" experiment.

- **Versioning and API Stability:** The plugin API itself must be **versioned** and kept stable (following semantic versioning). Each plugin should declare which version of the reproducibility API it supports. This prevents confusion if the standard evolves – older plugins can still advertise v1.0 while newer ones use v2.0, etc. The system (JENS) can then check compatibility easily. Additionally, the *neuromorphic frameworks* should continue to version their own software in the usual way, and those versions appear in the metadata. It's wise to encourage authors to use *fixed releases* of simulators for their experiments (rather than a moving development branch), and to ideally link to a DOI or repository for the exact code used. Journals like JOSS (Journal of Open Source Software) already mandate that a *software release* (with version number and archive) accompany the paper, which encourages proper versioning and documentation. JENS can similarly require that any plugin submitted is based on a specific, citeable version of the underlying tool. This practice ensures that if NEST or Brian2 updates in the future, one can still obtain the old version that was used to produce a paper's results, avoiding reproducibility failure due to software drift.

- **Ease of Installation and Distribution:** To maximize adoption, the plugin system should be easy to obtain and use. Container images help here (one `docker pull` gets everything). For Python-based frameworks, publishing the plugin as a PyPI package could also be useful (so `pip install jens-nest-plugin` installs the interface for NEST, for example). The packaging should handle all dependencies (possibly by installing the simulator or by bundling the interface to call an existing install). In some cases, a **thin wrapper** might just invoke a command provided by the framework – e.g., if a simulator already has a CLI, the plugin could call that internally. The key is that *from the user's perspective*, there is a uniform way to install and invoke any supported backend.

- **Example and Template**: It might be helpful to provide a reference implementation or template plugin (perhaps for a simple reference simulator) to guide framework developers on how to implement the standard. This reference could demonstrate how to parse a standard input (like a JSON experiment spec), how to run the simulation, and how to collect outputs and metadata. By following a template, different teams can create plugins with less guesswork, leading to a more consistent behavior across platforms.

Sandboxing and Reproducibility Constraints

When executing arbitrary experiments (potentially provided by users in a journal system), **sandboxing** is critical for security and consistency. We want to *isolate each execution* so that it doesn't depend on or affect other runs, and to prevent any harmful operations:

- **Isolated Execution Environments:** Running the experiments in containers or virtual machines inherently provides a sandbox. The code runs inside a confined environment with only the resources we allocate. This protects the host system from malicious or accidental damage (e.g. a runaway process). It also ensures that experiments don't unintentionally access internet or local files that aren't part of the package (which could compromise reproducibility). The JENS platform can enforce that the container has no network access (unless explicitly needed for the experiment and allowed), and only specific mount points (like input data and an output directory). Using container sandboxing, one can even allow privileged operations inside the container (some neuromorphic hardware APIs might need low-level access) without jeopardizing the host – *for example, Singularity and other HPC container tools let users run as non-root inside the container but still access needed devices in a controlled way* [19] .

- **Deterministic Execution and Randomness:** Reproducibility demands that running the same experiment twice produces the same result (assuming the same platform). Therefore, the plugin standard should encourage or enforce setting **random seeds** for any stochastic aspects of simulations. The plugin API might include a parameter for a random seed or a directive that all random number generators in the framework be initialized predictably. Many simulators like NEST and Brian allow seeding their RNGs; the plugin can call those APIs. For hardware platforms (which might have more variability), it's trickier, but at least the software driving them can be deterministic. Additionally, if a framework has parallel execution, it should offer a mode where results don't depend on thread scheduling (e.g., using a fixed scheduling algorithm or single-threaded mode if necessary) – these details might be documented in the reproducibility guidelines for each plugin. The **SciRep** study noted that for AI experiments, explicitly controlling seeds in nondeterministic analyses is important for reproducibility [20] . We carry that principle into neuromorphic experiments.

- **Resource and Time Limits:** In a journal execution environment, it's wise to impose limits so that one experiment doesn't hog all resources indefinitely. The plugin could include *sensible defaults or hooks* for the orchestrator to set a maximum runtime or memory usage. For example, JENS might run each plugin inside a container orchestrated by a system that kills the job if it exceeds X minutes or uses more than Y GB of RAM. These ensure that even if an experiment has an infinite loop or bug, it won't stall the entire system. From a design perspective, the plugin interface might not directly handle this, but documentation to users should state any limits and the need to design experiments accordingly (e.g., run smaller test versions for verification).

- **Reproducibility Checksums/Verification:** For true reproducibility, one could go a step further and implement automated verification. For instance, the plugin might generate a hash or fingerprint of key result files, and the JENS system can compare it to a published value. If they mismatch, it signals that something in the environment differed. As an example, if a paper claims certain output, the container could be run to re-generate that output and the results compared. This might be optional or a future extension, but having the infrastructure in place (consistent output formats, logging of results) sets the stage for such verification. The API standard could specify that results should be written in a certain directory or named in a certain way to facilitate automated collection and comparison.

- **Security Considerations:** Because plugins will execute arbitrary user-provided code (models could include custom components, especially in systems like Brian2 that allow on-the-fly code generation), security is important. Container sandboxing covers a lot, but we should also recommend best practices like running as a *non-root user* inside containers (except where hardware access requires root, in which case use specific privileges rather than full root). If the plugin system uses a communication protocol (e.g., a service listening for commands), it must be secured (for instance, listening only on localhost or within the container, to prevent external intrusion). These are standard precautions to make sure the **reproducibility framework doesn't become an attack vector** on the host running it.

In summary, sandboxing ensures that each experiment runs in a clean, controlled bubble – improving reproducibility (no hidden state or dependency on outside services) and providing safety for shared execution infrastructure.

Recommendations for a Neuromorphic Reproducibility API/Plugin Standard

Bringing the above points together, here are **actionable recommendations** to build a general-purpose plugin/API standard for neuromorphic research tools:

1. **Define a Unified Experiment Interface:** Develop a minimal *language-agnostic API specification* for running neuromorphic experiments. For example, specify that a plugin must implement functions like `load_model(model_spec)`, `run(parameters)`, and `export_results(output_path)`. Keep this interface high-level (covering common needs of SNN simulations) so it can wrap diverse backends. Document the interface clearly for implementers.

2. **Provide Reference Implementations:** Create a reference plugin (or skeleton) demonstrating how to adhere to the API. This could be a dummy simulator that just echoes inputs, or a simple integration with an existing framework (e.g., wrapping a call to Brian2 or NEST in a trivial way). This helps framework developers get started quickly.

3. **Implement Adapters for Major Frameworks:** Work with maintainers of Brian2, NEST, SpiNNaker (sPyNNaker), Loihi (Lava or NxSDK), etc., to implement the plugin for each. In many cases, this will involve writing a Python module or CLI tool that uses the framework's native API under the hood. Leverage any existing cross-platform tools: e.g., a NEST plugin could potentially use PyNN under the hood – when given a PyNN model, it directs it to NEST, whereas a Brian2 plugin could directly execute Brian scripts. Aim for *minimal code in the plugin* – reuse the framework's normal usage patterns.

4. **Adopt Common Model Formats (if feasible):** Encourage (but don't require) the use of standard model description formats. The plugin API might support multiple ways to specify the experiment: perhaps the user can provide either (a) a self-contained Python script (for script-driven experiments), or (b) a model description in a standard format (PyNN, NeuroML, etc.) along with a separate config for simulation parameters. Supporting a standard format means any model in that format could run on all backends that support it, enhancing reproducibility. For instance, if all plugins accept NeuroML, an author could submit a NeuroML model once and it would run via NEST, Brian, or Loihi plugins, enabling direct comparison.

5. **Standardize Command-Line Entrypoint:** Define a uniform CLI for the plugins. For example: `neuromorphic-run --platform <name> --experiment <file> --output <dir> [--seed N]`. This single command, when pointed at different `--platform`, would internally call the respective plugin code. Implement a dispatcher that recognizes the platform and invokes the correct container or module. This way, users and automated systems have one command to remember. Within each container, an entrypoint script can parse such arguments and invoke the plugin logic. (If using Docker, you might even label each image with the platform name and use one master script to run them.)

6. **Containerize Each Plugin Environment:** For every plugin, publish a Docker image (and/or Singularity recipe) that includes the plugin and the neuromorphic framework with the correct version. This image should also include all necessary dependencies (Python packages, MPI if needed, etc.). Test that running an example experiment with the image yields the expected result. These images can be versioned (e.g., `jens-nest-plugin:v1.0`). When authors submit an experiment, they can either use an official image or provide a custom image if their work needs a custom build – but in most cases, using the standard images for each framework ensures consistency. The **reproducible containers preserve the exact environment** used by the authors [21], so others can literally "click and run" the experiment in the same setup.

7. **Embed Metadata and Ensure Logging:** Configure the plugin to produce a metadata report (versions, timestamps, etc.) each time it runs. Also, capture all console logs and outputs from the simulator. This can be as simple as directing stdout/stderr to a log file and writing a JSON summary of the run (including random seed used, duration, etc.). These artifacts should be saved to the output directory for the experiment. Having this information is invaluable when troubleshooting a failed reproduction attempt – one can quickly see if a version mismatch or other issue occurred. It also helps for long-term archival of the experiment's context.

8. **Enforce Reproducibility Checks in Plugins:** As part of the plugin implementation, include assertions or settings that promote reproducibility. For example, a plugin might check that a seed is provided (or set a default) for random number generation. It could also configure the simulator to use deterministic mode if available (e.g., single-threaded or fixed scheduling). If the underlying framework has known non-deterministic behavior, document it and, if possible, have the plugin warn the user or mitigate it (for instance, NEST can randomize neuron IDs in certain runs – a plugin could sort the outputs to ensure consistency in ordering before comparison).

9. **Sandbox Execution and Limit Access:** Run each plugin in isolation. If using containers, this is largely handled, but ensure that, for example, the working directory is cleared or reset for each run (to avoid leftover state). The plugin should not rely on any external network resources unless

absolutely necessary (and if so, those should be specified and cached if possible). For instance, if an experiment needs a particular dataset, it should be included with the submission or downloaded as part of a setup step in a controlled manner, not fetched from an unstable URL at runtime. Sandboxing also means *no reading random system state*: the experiment shouldn't depend on the current time, hardware serial numbers, etc. – these are typical sources of unreproducible variation.

10. **Testing Across Platforms:** Finally, validate the system by running the *same simple test case* on all plugins to ensure they produce equivalent outputs. For example, a basic SNN model (maybe a two-neuron network) could be specified, and each plugin run it for 1 second of simulation time. The spike outputs or summary metrics should be compared. This kind of cross-platform test, akin to a unit test for the standard, can catch discrepancies early. It's similar to how PyNN was used to verify that standard neuron models behave the same on different simulators [22]. Regularly running these tests (possibly in CI for the JENS platform) will ensure that updates to plugins or frameworks don't silently break reproducibility.

By following these steps, we create a **modular, robust plugin architecture** for executable experiments in neuromorphic computing. The approach draws on known best practices (from plugin design to containerization) to balance flexibility with control. Researchers gain a convenient way to package and share their *entire experiment pipeline* in a standardized format, and journals or platforms like JENS gain the ability to *automatically execute and verify* those experiments across different neuromorphic substrates. Ultimately, this increases confidence that neuromorphic research findings are **truly reproducible and comparable**, accelerating progress through easier validation and integration of results.

Sources:

- Davison et al., *Front. Neuroinform.* 2009 – PyNN common interface for neuronal simulators [23] [7]
- Gleeson et al., *PLoS Comput. Biol.* 2010 – NeuroML for model sharing across simulators [11] [12]
- Pedersen et al., *Nature Comm.* 2024 – Neuromorphic Intermediate Representation (NIR) for cross-platform spiking models [1] [13]
- Jupyter Project Documentation – Design of Jupyter kernel plugin system [2] [3]
- OpenMM Developer Guide – Plugin architecture for adding new simulation components [4] [5]
- PyTorch/XLA Documentation – Plugin support for custom hardware in PyTorch [6]
- Kurtzer et al., *PLOS ONE* 2017 – Singularity containers for portable, reproducible scientific computing [17]
- Costa et al., *SciRep Framework (arXiv preprint)* 2023 – Packaging experiments with code, data, and environment for exact reproducibility [18]
- Pedersen et al., *Open Neuromorphic Blog* 2025 – Vision of executable papers with preserved containerized environments [21]

---

[1] [7] [8] [9] [10] [13] [15] Neuromorphic intermediate representation: A unified instruction set for interoperable brain-inspired computing | Nature Communications
https://www.nature.com/articles/s41467-024-52259-9?error=cookies_not_supported&code=8b5142d9-90f2-41cb-b511-d611eef46e7a

[2] [3] Architecture — Jupyter Documentation 4.1.1 alpha documentation
https://docs.jupyter.org/en/stable/projects/architecture/content-architecture.html

[4] [5] 3. Writing Plugins — OpenMM Dev Guide 8.3 documentation
https://docs.openmm.org/latest/developerguide/03_writing_plugins.html

[6] Custom Hardware Plugins — PyTorch/XLA master documentation
https://docs.pytorch.org/xla/master/contribute/plugins.html

[11] [12] NeuroML: A Language for Describing Data Driven Models of Neurons and Networks with a High Degree of Biological Detail | PLOS Computational Biology
https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1000815

[14] [22] [23] Introduction | PyNN 0.12.3 documentation
https://neuralensemble.org/docs/PyNN/introduction.html

[16] [17] [19] Singularity: Scientific containers for mobility of compute | PLOS One
https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0177459

[18] [20] A Framework for Supporting the Reproducibility of Computational Experiments in Multiple Scientific Domains
https://arxiv.org/html/2503.07080v1

[21] Strategic Vision for Open Neuromorphic - Open Neuromorphic
https://open-neuromorphic.org/blog/strategic-vision-open-neuromorphic/