DOCUMENTATION

A Python graph language for the SNN tool

ar063 · 30 July 2026 · Draft

Abstract

Translating a circuit idea into *tools/snn* is cumbersome. A paper may describe a small motif in a paragraph, while its implementation requires manual work across CLI flags, model construction, tensor dimensions, training choices, recordings, and artifact handling. This proposal introduces *snnlang*: a typed Python library for constructing a spiking computation graph and, when needed, a narrow training specification. One compile operation validates them and writes a portable bundle containing *graph.json* and optional *training.json*. *tools/snn* executes that bundle through a small Python API wrapped by its CLI.

The proposal is deliberately smaller than a language for whole experiments, but it is not restricted to the present `COBANet` architecture. Experiment runners continue to own hypotheses, condition and seed grids, custom interventions, derived analysis, figures, and publication. An audit of the current code identifies three workloads that should test the design: trained MNIST and SHD classifiers, including deeper layers; coupled untrained E/I circuits with population-level recordings; and online confidence feedback that modulates g_L to trade decision speed for additional PING cycles. The bundle is the canonical reproducibility boundary, while a small importable API preserves exploratory PyTorch flexibility. *tools/snn* lowers graphs into vectorized, compiled PyTorch rather than interpreting them. The implementation order is: draft the graph language, upgrade *tools/snn* to execute it, run one real experiment through both, and then prove that the combined system is flexible enough for the confidence experiment.

1. The existing separation of concerns

The repository intends to contain two different layers. *tools/snn*, documented in ar011, is the reusable simulation and training engine. Files under *experiments/* are scientific runners. The desired boundary is subprocesses and files, but the present code does not consistently maintain it.

```
experiment runner
      |
      | CLI invocation
      v
tools/snn/tool.py
      |
      | config, metrics, traces, weights
      v
experiment runner
      |
      | analysis, figures, numbers.json
      v
writings/expNNN.typ
```

This firewall is useful when it exists. It keeps reusable model science out of paper-specific code and prevents the simulator from accumulating every protocol ever used in the lab. The leaks are diagnostic: they identify missing reusable execution contracts more clearly than an abstract architecture exercise can.

1.1 What tools/snn owns

tools/snn owns operations that remain meaningful across experiments:

- neuron and synapse dynamics;
- graph execution and numerical integration;
- the standard training loop and surrogate-gradient implementation;
- checkpoint loading and saving;
- reusable input, recording, and perturbation primitives;
- generic rates, accuracy, and run metrics;
- deterministic seeding and run provenance;
- stable machine-readable outputs.

Its current use of *torch.compile* remains an internal execution detail. *snnlang* does not manage compilation caches or accelerator-specific artifacts.

1.2 What experiment runners own

A runner owns decisions whose meaning comes from one scientific question:

- the hypothesis and condition labels;
- parameter and seed grids;
- job ordering, caching, resumption, and dispatch;
- paper-specific data preparation;
- custom intervention streams built from earlier outputs;
- derived statistics and success criteria;
- figure design and *numbers.json*;
- staging and atomic publication.

Substantial runner code is not automatically a failure of abstraction. Code should move into a reusable tool only after a second use demonstrates a stable operation.

1.3 What the runners reveal

Existing runners make the boundary concrete. *exp022* defines a canonical family of trained MNIST cells. *exp041* computes spectra, gamma peaks, fits, and figures from tool outputs. *exp042* constructs experiment-specific inhibitory override streams from a reusable simulator primitive. *exp054* and *exp058* use untrained networks to study connectivity, synchrony, rhythmicity, and perturbation growth.

The newer SHD runners also expose failures in the current boundary. They import the CLI module in-process, replace the dataset directory, and construct networks directly for evaluation. *exp069* temporarily replaces `MetricsJsonl.write`, inspects its caller's local variables to obtain the live network, and saves a validation-selected checkpoint. This is ingenious in the same sense that opening a tin with a screwdriver is ingenious: it works, but it is evidence that the correct tool is missing.

The audit therefore identifies four engine seams needed before a general graph executor:

- an explicit data binding for dense samples and event streams, instead of fixed dataset paths;
- a standard checkpoint-selection policy or stable epoch event contract;
- evaluation of supplied event datasets without direct model construction;
- named recordings from every layer and population, rather than a privileged "primary" hidden layer.

These cases support one rule:

> The tool should understand one graph execution. The runner should understand why several executions constitute an experiment.

1.4 What the current model actually is

The present `COBANet` is a useful but narrow execution model:

- it builds a feedforward list of hidden E/I layers;
- E spikes alone feed the next hidden layer;
- each layer owns W_{EE}, W_{EI}, W_{IE}, and W_{II} recurrence;
- rate, mean-membrane, and cumulative-potential readouts are hard-coded modes;
- two hidden layers are already exercised by *exp071*;
- `train_leak` learns a static bounded leak per neuron, not an online feedback law;
- recording can capture all layers, although several consumers select only the deepest layer.

This means "support deeper networks" is not a distant grammar problem. The first graph schema should represent the existing sequential stack, but its ontology should be named populations and projections rather than `COBANet` constructor arguments. Coupling two independently named circuits is a genuinely new backend capability, even when neither circuit is trained. It is acceptable for the first compiler to emit graphs that the current backend reports as unsupported; the language and engine capability levels must be explicit rather than artificially identical.

1.5 Data is an execution binding, not graph structure

The standard trainer currently knows MNIST and SHD. MNIST is dense and Poisson encoded; SHD is an event stream binned lazily. The generic inference path still assumes 784 dense inputs, so SHD experiments work around it.

graph.json should declare typed input ports and output dimensions. *TrainSpec* should declare the target interface and encoder class. A runner should supply a small resolved data binding that maps actual train, validation, or evaluation sources onto those ports. Dataset splitting, sealing, and scientific cohort selection remain runner responsibilities. This avoids both extremes: hard-coding MNIST and SHD into the graph, or inventing a universal dataset language.

2. The proposed boundary

The proposal introduces one Python package with two related authoring objects:

```
snnlang.Network
snnlang.TrainSpec
```

Network describes forward computation. *TrainSpec* is an optional, narrow configuration for the standard trainer. It refers to parameters and outputs in a particular *Network*, but it is not part of that network and is not a second independent language.

One compile operation produces a bundle:

```
Python authoring
  Network
  optional TrainSpec
        |
        | snnlang.compile(...)
        v
ping.bundle/
  graph.json
  training.json
  manifest.json
        |
        | tools/snn API or CLI + data binding
        v
run artifacts
```

graph.json is the resolved executable graph. *training.json* records standard training policy and the digest of the exact graph against which it was validated. *manifest.json* binds the files into one convenient invocation unit.

A separate, resolved *data.json* is supplied when training or evaluating. It contains paths and representation metadata, not dataset contents. It belongs to an execution and may be replaced while the graph remains unchanged.

tools/snn depends on the versioned graph and training schemas, not on the *snnlang* Python package. Given an archived bundle, it must be possible to replay the run without installing the compiler that originally authored it.

2.1 Same repository, separate libraries

snnlang should begin in the same repository and Python environment as *tools/snn*, but as a sibling library:

```
tools/
    snn/
    snnlang/
schemas/
    snn-graph-v1.schema.json
    snn-training-v1.schema.json
```

Keeping both sides together makes early schema changes cheap. Keeping their package boundaries separate prevents the authoring library from becoming simulator internals. A separate repository or published project is justified only after another repository, simulator, or release cadence actually needs it.

3. The smallest useful authoring model

The first authoring surface is Python, not a custom textual grammar. Python already supplies composition, functions, loops, autocomplete, testing, and debugging.

```
import snnlang as snn
from snnlang import training

net = snn.Network("ping_classifier")

E = net.population(
    "E",
    size=1024,
    neuron=snn.COBA_LIF(...),
)
I = net.population(
    "I",
    size=256,
    neuron=snn.COBA_LIF(...),
)

ei = net.connect(
    E.spikes,
    I.excitatory,
    name="E_to_I",
    synapse=snn.AMPA(tau=...),
    weight=snn.Normal(0.5, 0.05),
    constraint=snn.NonNegative(),
)
ie = net.connect(
    I.spikes,
    E.inhibitory,
    name="I_to_E",
    synapse=snn.GABA(tau=9 * snn.ms),
```

```
    weight=snn.Normal(1.0, 0.1),
    constraint=snn.NonNegative(),
)

logits = snn.readouts.MeanVoltage(
    source=E.spikes,
    classes=10,
    tau=20 * snn.ms,
    name="classifier",
)

net.output("class_logits", logits)
net.expose(E.spikes, I.spikes)
```

The graph contains populations, projections, ordinary transformations, parameters, outputs, and observables. It does not contain experiment conditions, sweeps, losses, optimizers, or figures.

3.1 Components and layers are authoring conveniences

A reusable component may shorten construction:

```
cell = snn.components.ping(
    net,
    name="cell",
    n_e=1024,
    n_i=256,
    tau_gaba=9 * snn.ms,
)
```

Components and layers expand into the same small graph vocabulary. The serialized graph may retain group metadata for reports, but it need not introduce a special executable `layer` primitive.

The same rule applies to readouts. A helper such as:

```
logits = snn.readouts.SpikeRate(
    source=cell.E.spikes,
    classes=10,
    name="class_logits",
)
```

expands into ordinary population, projection, reduction, and output operations. Authoring functions execute while constructing the graph; the compiler serializes their result rather than the Python callable. The first graph schema therefore has no special `head` ontology. Named outputs provide the interface needed by inference and training.

3.2 Readouts are concise components with precise semantics

Readout switching is routine experimental work and must be a local edit. The initial library should include:

```
snn.readouts.MeanVoltage(...)
snn.readouts.FinalVoltage(...)
snn.readouts.SpikeCount(...)
snn.readouts.SpikeRate(...)
snn.readouts.CumulativePotential(...)
```

`MeanVoltage` expands into a trainable projection, a non-spiking stateful layer, and a temporal mean of its membrane voltage. The non-spiking layer remains explicit in the compiled graph and checkpoint even when hidden by the concise authoring helper.

`SpikeCount` means

$$z_c = \sum_t s_{c(t)}.$$

`SpikeRate` is distinct:

$$z_c = \frac{\sum_t s_{c(t)}}{T\Delta t}.$$

For padded or variable-duration samples it must use the valid-time mask independently for each sample. "Rate" without a declared duration and unit is rejected as ambiguous. Common window specifications should cover the full trial, a post-transient interval, or the final duration without requiring a Python callback.

Users may define their own authoring helpers from standard operations:

```
def my_readout(source, classes):
    x = snn.ops.sum(source, over="time")
    x = snn.ops.normalise(x)
    return snn.ops.linear(x, size=classes)
```

This remains portable because the function expands at compile time. Arbitrary PyTorch executed during the forward pass is a different extension level and is not embedded as a callable in JSON.

3.3 Parameters are not intrinsically selected for this run

The graph distinguishes a *Parameter* from a *Constant* and records structural constraints such as non-negativity, sparsity masks, or tying. It does not permanently declare that every parameter must be updated.

Selection belongs to a particular training specification:

```
train = snn.TrainSpec(
    objectives=[
        training.CrossEntropy(
            prediction=net.outputs["class_logits"],
            target="digit",
        ),
    ],
    parameter_groups=[
```

```
        training.ParameterGroup(
            [ei.weight, ie.weight],
            lr=1e-4,
        ),
        training.ParameterGroup(
            logits.parameters,
            lr=1e-3,
        ),
    ],
    regularizers=[
        training.UpperRatePenalty(
            signal=E.spikes,
            threshold=1.0,
            strength=1e-4,
        ),
    ],
    optimizer=training.AdamW(),
    epochs=50,
)
```

The same graph can support readout-only training, recurrent fine-tuning, or inference without changing its structural identity.

3.4 Forward and backward concerns

Anything executed during inference belongs in the graph: populations, reductions, projections, classifier parameters, and named outputs. Anything used only to calculate or apply gradients belongs in *TrainSpec*: objectives, targets, parameter groups, regularizers, surrogate choice, clipping, and backward-only stop boundaries.

Checkpoints remain separate evolving state. They contain realised parameter values and, when resuming training, optimizer state. A checkpoint may initialise, resume, or partially map onto a graph, but its path is not part of the immutable graph.

3.5 Online feedback belongs to the combined execution system

Confidence-controlled leak is not a training recipe. At each timestep it computes evidence from the current readout, derives confidence, and modulates a population parameter before a later state update. The combined *snnlang* plus *tools/snn* system must support that causal loop, but version one need not give confidence or g_L modulation dedicated language constructs.

One eventual declarative spelling could be:

```
evidence = snn.readouts.cumulative(cell.E.spikes)
confidence = snn.signals.max_probability(evidence)

snn.controls.bounded_leak(
    source=confidence,
    target=cell.E,
    low_confidence_tau=30 * snn.ms,
    high_confidence_tau=10 * snn.ms,
    delay=1 * snn.step,
)
```

The spelling is provisional and is not a version-one requirement. Three implementation routes are legitimate:

- ordinary graph operations, if the initial vocabulary can express the loop cleanly;
- a generic stateful controller or modulatory-node extension implemented by *tools/snn*;
- custom experiment code using a small, stable runtime hook around a compiled graph.

Whichever route is chosen must state the source signal, transform, target, bounds, initial state, and whether the effect is immediate or delayed. A next-timestep default avoids an implicit algebraic loop. If controller parameters later become trainable, *TrainSpec* can select them like any other parameter.

The third route does not mean reaching into model locals or monkey-patching the training loop. It means an intentional engine extension point. The graph and checkpoint remain portable; the run manifest records the controller implementation and configuration. If the same controller pattern recurs, it can then be promoted into snnlang.

4. What remains outside snnlang

Neither *Network* nor *TrainSpec* should initially describe:

- experimental conditions or hypothesis labels;
- parameter and seed sweeps;
- matched-control searches;
- multi-stage curricula or alternating optimizers;
- arbitrary Python callbacks;
- experiment-specific adaptive interventions driven by intermediate results;
- paper-specific metrics;
- figures and publication;
- RunPod or Modal orchestration;
- automatic prose or claim generation.

A custom experiment may use *graph.json* with custom Python training code and omit *training.json*. A feature enters *TrainSpec* only when the standard *tools/snn* trainer supports it and repeated experiments share it.

A paper-specific rule that decides which condition to run next remains in the runner. A within-trial control loop must execute inside the simulator runtime, whether authored directly in the graph or attached through a stable extension point.

5. Compilation and static analysis

Compilation is pure and deterministic:

```
bundle = snn.compile(
    net,
    training=train,
    target="tools/snn",
)
bundle.write("ping.bundle")
```

It performs no simulation, accelerator allocation, realised random initialization, or mutation of *tools/snn* globals. Python object references become stable graph identifiers at serialization.

5.1 Hard validation

The early compiler can provide substantial value before execution:

- schema and version validation;
- unique names and resolved references;
- tensor and projection shapes;
- physical units;
- port compatibility;
- graph reachability from inputs to outputs;
- disconnected populations and unused projections;
- Dale and sign-constraint compatibility;
- parameter inventory and optimizer-group membership;
- objective and target compatibility;
- differentiable reachability from each objective;
- surrogate-gradient coverage on objective paths;
- feedback-cycle delays and a deterministic within-step update order;
- controller output units, target compatibility, and declared bounds;
- checkpoint names and shapes;
- backend capability checks;
- deterministic canonical serialization.

Example diagnostics should identify the object and failed relation:

```
error E203: projection "E_to_I" has shape [800, 200]
  expected [1024, 256] from E.spikes -> I.excitatory

error E501: selected parameter "I_to_E.weight"
  has no differentiable path to objective "classification"
```

5.2 Reports and estimates

The analyser can also report:

- population and projection tables;
- strongly connected components and recurrent motifs;
- feedforward, recurrent, and modulatory paths between named populations;

- parameter counts and selected versus frozen parameters;
- expected sparse edge counts and fan-in;
- approximate parameter, optimizer, state, recording, and BPTT memory;
- semantic differences between two bundles;
- whether two runs share a graph but differ in training policy;
- whether a bundle contains unresolved environment-dependent defaults.

Numerical risk checks should be warnings rather than proofs:

```
warning W311: tau_gaba=0.2 ms is only two timesteps
  at dt=0.1 ms; the decay may be poorly resolved
```

5.3 Visual graph reports

snnlang should optionally render a compiled bundle as a polished circuit diagram. This is a compiler report, not a simulator feature:

```
bundle.visualise(
    "network.svg",
    view="circuit",
)
bundle.visualise(
    "network.png",
    view="circuit",
    scale=2,
)
```

The implementation should generate styled DOT from *graph.json*, use Graphviz for layout, and treat SVG as the canonical visual output. PNG and PDF are derived publication formats. Graphviz is responsible for ranks, routing, and crossing reduction; *snnlang* remains responsible for visual meaning.

The visual grammar should be stable:

- rounded cards for populations, with size and neuron model;
- restrained, consistent colours for excitatory, inhibitory, input, output, and modulatory nodes;
- solid excitatory projections, inhibitory bar endings, and dashed modulatory paths;
- softly bounded clusters for layers, PING cells, and user-defined components;
- feedback routed separately from the primary feedforward direction;
- optional edge width or annotation for density, fan-in, delay, or weight summary;
- subtle marks for trainable parameters and stop-gradient boundaries.

One overloaded picture is not the goal. The renderer should offer at least:

- `circuit`, a paper-ready population and projection view;
- `training`, showing outputs, objectives, trainable groups, and gradient boundaries;
- `expanded`, showing lower-level operations and stateful nodes for debugging.

Components are collapsed by default and expandable on request. Parallel projections may be grouped, default parameters suppressed, and a selected path highlighted. If a graph is too

dense for a legible static image, the renderer should warn and require filtering or collapsing rather than emit decorative spaghetti. Layout, colours, fonts, node ordering, and identifiers must be deterministic so an unchanged bundle produces an unchanged diagram.

The canonical SVG should retain stable object identifiers and classes. This permits tooltips or interactive inspection later without changing the graph schema. Visualisation failure must never make an otherwise valid bundle unexecutable.

5.4 What static analysis does not promise

The early analyser does not predict whether gamma emerges, its frequency, a Hopf threshold, firing rates, accuracy, or successful optimization. Those are dynamical claims. Later restricted analysers may attach approximate semantics to supported motifs, but every result must state its assumptions and unsupported components.

6. Runtime and CLI boundary

The bundle is the canonical reproducibility boundary, not the only execution route. *tools/snn* should expose one small request-based Python API:

```
model = tools_snn.build(graph)
result = tools_snn.train(
    graph,
    training,
    data,
    request,
)
result = tools_snn.simulate(
    graph,
    data,
    request,
)
```

The CLI is a thin wrapper over the same functions. Ordinary experiment runners should receive a typed subprocess helper when they want process isolation:

```
from experiments.helpers.snn import run_training

run_training(
    bundle=bundle_path,
    seed=seed,
    out_dir=cell_dir,
)
```

The helper invokes:

```
uv run python tools/snn/tool.py train \
  --bundle ping.bundle \
  --data data.json \
  --seed 42 \
  --out-dir cell/
```

Inference needs only the graph and optional checkpoint:

```
uv run python tools/snn/tool.py sim \
  --graph ping.bundle/graph.json \
  --weights weights.pth \
  --out-dir inference/
```

Scientific graph and standard training settings live in validated files rather than long argument lists. The data binding carries resolved sources. CLI arguments carry paths, seeds, output locations, and lifecycle controls.

The process boundary remains valuable because a fresh process contains failures, accelerator memory, compiler state, and legacy globals. The importable API remains available for notebooks, debugging, composition into custom training code, and experiments needing deliberate runtime extensions. Runners must not obtain that flexibility by importing the CLI module, inspecting stack frames, or monkey-patching engine internals.

6.1 Three extension levels

The system supports increasing flexibility with decreasing portability:

1. **Authoring components.** Python functions compose standard snnlang operations and expand before serialization. This is the preferred route for readouts and circuit templates.
2. **Registered backend operations.** The graph records a versioned operation identifier and configuration; *tools/snn* supplies its PyTorch implementation. The manifest records the required extension.
3. **Direct Python execution.** Exploratory code uses the importable API with a custom `torch.nn.Module`, controller, or training loop. The run records its source identity, configuration, environment, and compiled graph.

Every standard experiment should replay from an archived bundle. Experimental Python extensions are allowed when recorded explicitly; they should be promoted into a portable operation only after their interface stabilizes.

6.2 PyTorch performance is non-negotiable

snnlang is an authoring language and intermediate representation. It is not a runtime interpreter. *tools/snn* lowers a validated graph into coarse, vectorized PyTorch modules before simulation:

```
graph.json
    -> tools_snn.build(...)
    -> torch.nn.Module
    -> torch.compile
    -> CUDA, MPS, or CPU execution
```

Populations are batched tensors; projections become dense, sparse, or structured tensor operations; recurrent state remains in tensors; and training uses PyTorch autograd with surrogate gradients. Compatible operations may be fused, and future Triton or custom CUDA kernels remain backend implementation choices.

The runtime must not execute a Python loop over graph nodes inside every timestep. Arbitrary Python callbacks may cause graph breaks and are an explicitly slower exploratory path. Stateful controllers intended for production execution should be PyTorch modules with fixed tensor interfaces.

Backend conformance includes performance. Each reference graph is benchmarked against its specialized legacy implementation after warm-up and compilation. The initial target is no more than approximately 5–10% steady-state overhead for an equivalent graph, with peak memory and compilation time reported separately. A pleasant API does not compensate for a materially slower simulator.

Existing flag-driven experiments remain supported. The manifest path is additive:

```
legacy flags -> compatibility adapter -\
                                       +-> ExecutionSpec
graph bundle -> bundle loader --------/
```

During migration, *ExecutionSpec* may select either the legacy `COBANet` builder or the graph-native executor. The target is one graph-native PyTorch execution core with the legacy CLI retained as a compatibility frontend. This permits gradual adoption without rewriting the historical experiment corpus.

7. Staged implementation

Implementation is divided into cumulative, goal-sized milestones. Each milestone states separate *snnlang* and *tools/snn* deliverables, the legacy behaviour that must remain unchanged, and executable acceptance tests. "Implement up to milestone N" therefore means: inspect the repository's declared milestone status, complete every unmet requirement through N, run each intervening gate, and stop. It does not authorize beginning milestone $N + 1$.

Milestone 0 records the bundle compiler, visualization, narrow legacy adapter, MNIST training smoke test, and lifecycle equivalence work already demonstrated by *exp074–exp076*. Milestones 1–3 create the execution boundary, prove one graph natively, and then unlock arbitrary coupled circuits. Later milestones add a real gamma-coupling experiment, graph-native training, SHD and deeper networks, and online control.

Appendix B is the normative milestone ledger. The sequence is additive and reversible: no milestone deletes the working legacy path, changes an old runner, or changes a default in order to prove the next one.

8. Migration policy

Completed experiments are scientific records, not merely old application code. They should remain executable through their existing runners.

- New experiments use *snnlang* once the needed feature set exists.
- A small MNIST, SHD, and untrained simulation suite becomes the conformance set.
- An old experiment migrates when it is materially extended.
- A migrated version is treated as a new implementation and compared explicitly.

- Custom training or intervention code remains acceptable when it is genuinely experiment-specific.

Mass retrofitting would risk changing defaults, seed handling, initialization, or simulator configuration while producing reassuringly similar figures. Architectural purity is not worth damaged provenance.

9. Success criterion

The first new scientific capability milestone is milestone 3:

> *snnlang* compiles two independently driven PING circuits with reciprocal, delayed inhibitory coupling; the graph-native *tools/snn* executor simulates them without changing the legacy executor; and named population recordings make their phase relationship measurable by an experiment runner.

The confidence-to-g_L experiment is the next system-level acceptance test. It succeeds whether the feedback is expressed through ordinary graph operations, a generic controller extension, or a stable runtime hook. The important constraint is that it composes with a compiled graph, remains reproducible, and does not require surgery on simulator internals.

Appendix A. Bundle and artifact lifecycle

A.1 Construction and compilation

An experiment-specific definition may begin inside its runner:

```
def make_bundle(tau_gaba_ms: float):
    net = make_ping_classifier(
        tau_gaba_ms=tau_gaba_ms,
    )
    train = make_standard_training(net)
    return snnlang.compile(
        net,
        training=train,
        target="tools/snn",
    )
```

The first use remains local. A repeated component may later move into a reusable circuit module. Reuse must be observed rather than predicted.

Compilation writes:

```
ping-tg9.bundle/
    manifest.json
    graph.json
    training.json
    reports/
        circuit.svg
        circuit.png
```

The executable files contain data only. They contain no Python callables, live instances, pickled authoring objects, accelerator tensors, or *torch.compile* products. The optional *reports/* directory contains derived human-readable views and may be regenerated exactly from the bundle.

A.2 Bundle contract

graph.json contains:

- populations, operations, ports, and projections;
- dynamics and structural parameters;
- initializer specifications;
- constants and optimizable parameters;
- constraints, masks, and ties;
- named outputs and exposed observables;
- grouping metadata for source-level components.

training.json contains:

- the exact graph digest;
- the expected data interface and encoder supported by the standard trainer;
- objectives, targets, and regularizers;
- optimizer parameter groups;
- surrogate-gradient and clipping settings;
- epochs and an explicit standard checkpoint-selection policy.

A resolved *data.json* contains:

- the input representation, such as dense samples or timestamped events;
- train, validation, or evaluation source paths;
- the mapping from source fields to graph inputs and training targets;
- shape, class-vocabulary, split, and content digests needed for validation and provenance.

It does not decide how a scientific split was created. That remains runner code.

manifest.json binds the files:

```
{
  "schema": "snnlang.bundle/v1",
  "graph": {
    "path": "graph.json",
    "sha256": "..."
  },
  "training": {
    "path": "training.json",
    "sha256": "...",
    "graph_sha256": "..."
  }
}
```

A simulation-only bundle may omit *training.json*. *tools/snn* rejects missing files, unsupported schema versions, digest mismatches, or backend capabilities absent from the graph.

A.3 Run state

config.json, written by *tools/snn*, records the resolved execution:

- execution mode, seed, device, timestep, and duration;
- graph and training schema versions and digests;
- data-binding, input, and checkpoint paths and digests;
- requested outputs;
- code and environment provenance.

A checkpoint contains evolving parameter values and optional optimizer state. Initializing, resuming, fine-tuning, and inference are invocation choices rather than changes to *graph.json*.

Graph-runtime state is a separate artifact again. It contains membrane voltages, refractory counters, per-projection conductances, recurrent delay histories, delayed-input context, completed steps, and a structural compatibility signature—but no weights. The initial portable representation is an authenticated `manifest.json` plus `tensors.npz`. This separation permits a mature zero-weight graph to branch into a state-compatible non-zero-weight graph without confusing dynamic continuation with parameter checkpointing. *snnlang* continues to author topology only; *tools/snn* owns state validation and I/O, while the experiment runner owns checkpoint selection and branching.

A.4 Scratch layout

Each experiment uses regenerable scratch separately from its committed publication record:

```
temp/experiments/expNNN/
    bundles/
        ping-tg4p5.bundle/
        ping-tg9.bundle/
        ping-tg18.bundle/
    reports/
        ping-tg4p5.svg
        ping-tg9.svg
        ping-tg18.svg
    cells/
        ping-tg4p5-seed42/
        ping-tg4p5-seed43/
        ping-tg9-seed42/

artifacts/data/expNNN/
```

A runner compiles once per unique graph and training policy, not once per seed. It may reuse that bundle with different validated data bindings.

A.5 Execution

The runner uses a typed subprocess helper:

```
run_training(
    bundle=bundles[condition],
    data=data_binding,
    seed=seed,
    out_dir=cell_dir(condition, seed),
)
```

At the shell boundary:

```
tools/snn train \
  --bundle ping-tg9.bundle \
  --data data.json \
  --seed 42 \
  --out-dir cells/ping-tg9-seed42
```

The tool copies the exact files it executed into the run directory:

```
cells/ping-tg9-seed42/
    graph.json
    training.json
    data.json
    config.json
    metrics.json
    weights.pth
    output.log
    run.jsonl
    run.sh
```

Copying prevents later edits to a source bundle from changing the apparent meaning of existing weights and metrics.

A.6 Publication

The committed experiment record deduplicates descriptions while retaining the mapping from every reported cell:

```
artifacts/data/expNNN/
    numbers.json
    bundle_manifest.json
    graphs/
        ping-tg4p5.json
        ping-tg9.json
        ping-tg18.json
    diagrams/
        ping-tg4p5.svg
        ping-tg9.svg
        ping-tg18.svg
    training/
        train-tg4p5.json
        train-tg9.json
        train-tg18.json
    figure.svg
    raster.png
```

The manifest maps each cell to graph and training digests:

```
{
  "cells": [
    {
      "condition": "tg9",
      "seed": 42,
      "graph": "ping-tg9",
      "training": "train-tg9"
    }
  ]
}
```

Compiler caches and accelerator-specific products are disposable implementation state and are not published scientific artifacts.

A.7 Atomic runner lifecycle

```
def main():
    run_id = next_run_id(SLUG)

    with published_run(
        SLUG,
        run_id,
        scale=SCALE,
    ) as (scratch, staging):

        bundles = {}

        for condition in CONDITIONS:
            bundle = make_bundle(condition)
```

```
            path = (
                scratch
                / "bundles"
                / f"{condition}.bundle"
            )
            bundle.write(path)
            bundles[condition] = path

        for condition in CONDITIONS:
            for seed in SEEDS:
                run_training(
                    bundle=bundles[condition],
                    data=data_binding(condition),
                    seed=seed,
                    out_dir=cell_dir(
                        condition,
                        seed,
                    ),
                )

        rows = analyse_cells(
            CONDITIONS,
            SEEDS,
        )
        render_figures(rows, staging)
        publish_descriptions(
            bundles,
            staging,
        )
        write_bundle_manifest(
            bundles,
            CONDITIONS,
            SEEDS,
            staging,
        )
        write_numbers(
            staging,
            run_id=run_id,
            duration_s=duration,
            payload=summarise(rows),
        )
```

The final artifact directory is replaced only after compilation, execution, analysis, plotting, and summary generation all succeed. A failed run cannot publish new graph descriptions beside figures from an older run.

Appendix B. Implementation milestone ledger

B.1 How milestone goals are interpreted

A goal phrased as `Implement ar063 up to milestone N` is cumulative. The implementation agent must:

1. read this ledger and inspect the current code rather than assuming its status;
2. complete unmet deliverables from milestone 0 through N;
3. run every acceptance gate through N;
4. update the status ledger and supporting documentation with evidence;
5. preserve all compatibility invariants; and
6. stop before work unique to milestone $N + 1$.

A milestone is complete only when its tests and experiment evidence are committed. Merely creating types, schemas, or command-line flags does not count. If a gate reveals a numerical or compatibility regression, the milestone remains incomplete.

The status labels are `demonstrated`, `partial`, and `not started`. They describe evidence in the repository at the date of this proposal and must be revised as the implementation advances.

B.2 Compatibility invariants

Every stage keeps the existing CLI and historical runners operational. New functionality is added beside the legacy path and becomes the default only after correctness, checkpoint, artifact, and performance gates pass.

The transition begins with two frontends and, temporarily, two construction paths:

```
old runner
    -> legacy CLI flags
    -> compatibility adapter --\
                                +-> ExecutionSpec
new runner                     /
    -> snnlang bundle --------/

ExecutionSpec
    -> legacy COBANet executor
    or
    -> graph-native executor
```

The target is:

```
legacy CLI flags
    -> compatibility adapter --\
                                +-> ExecutionSpec
snnlang bundle                 /
    -> bundle loader ---------/

ExecutionSpec
    -> graph-native PyTorch executor
```

The compatibility adapter survives after convergence. The legacy executor remains selected for legacy CLI invocations until a later milestone explicitly changes that routing. In particular:

- old commands, defaults, configuration files, checkpoints, parameter names, recordings, and artifact schemas remain valid;
- no existing experiment runner is edited merely to adopt *snnlang*;
- `--executor graph` or an equivalent explicit bundle field selects new execution during the compatibility period;
- graph checkpoints use their own manifest until an explicit, tested parameter map permits cross-loading; and
- low-level numerical kernels may be shared, but the legacy `COBANet` construction and update path are not refactored as a prerequisite for graph execution.

B.3 Milestone 0: established bundle baseline

Status at 2026-08-05: demonstrated for the narrow MNIST COBANet slice; broader legacy characterization remains partial.

snnlang deliverables: Python authoring objects; graph and training IRs; PING and readout helpers; deterministic bundles and manifests; basic validation; circuit, expanded, and training diagrams.

tools/snn deliverables: load a supported bundle without importing *snnlang*; translate the exact one-layer PING plus mean-voltage graph into the legacy executor; translate its narrow MNIST recipe into the legacy trainer.

Evidence: exp074 simulates supplied spikes and records rasters; *exp075* trains a small MNIST subset; *exp076* compares seeded initialization, forward values, loss, gradients, one optimizer step, checkpoint interchange, and replay.

Gate: those experiments and their focused unit tests pass through the unchanged CLI. This baseline must remain green at every later milestone.

B.4 Milestone 1: freeze the compatibility seam

Status at 2026-08-05: demonstrated. The typed request seam, legacy-default selector, versioned element-level capability vocabulary, and data-only bundle boundary are implemented in commits `5be1cdb` and `cf11906`. Focused tests lower legacy MNIST, SHD, checkpoint, recording, and bundle invocations into the same request type while retaining legacy routing. *exp074–exp076* reran successfully through their historical interfaces; the validation evidence and anomalies are recorded in *exp077*.

Purpose: create a safe place for a second executor without changing numerical behaviour. This milestone adds architecture and tests, not new circuit science.

snnlang deliverables: define a versioned backend capability vocabulary and make compilation report required capabilities such as neuron kinds, synapse kinds, delays, feedback, operations, training, and recording modes. Capability failure must identify the graph element and missing feature rather than merely saying that a bundle is unsupported.

tools/snn deliverables: introduce typed `ExecutionSpec` and `ExecutionResult` objects and a small internal request API for build, simulate, train, and infer. Add an explicit executor

selector with `legacy` as the default. The existing CLI becomes a thin caller of this API, while bundle loading remains data-only and does not import *snnlang*. A `graph` executor entry may initially fail with a precise "not implemented" diagnostic.

Compatibility gate: run the milestone-0 suite plus representative existing MNIST, SHD, untrained, checkpoint-load, and recording CLI smoke tests. Commands, resolved defaults, parameter names and shapes, seeded outputs, artifacts, and exit behaviour must remain unchanged. No historical runner imports the new API.

Exit criterion: both legacy flags and bundles produce an `ExecutionSpec`; all existing invocations still select the untouched legacy executor; capability reports agree with what the legacy bundle adapter actually accepts.

B.5 Milestone 2: graph-native single-PING forward execution

Status at 2026-08-05: demonstrated. Commit `cf11906` makes the existing PING state, refractory counts, membrane constants, update order, silent recurrence, readout reset, delays, initialisers, constraints, outputs, and observables explicit. *exp077* records zero parameter error, zero E/I spike mismatches, zero named-output error, and zero checkpoint-replay error on an active matched CPU fixture. The graph path is faster than the legacy path on the matched steady-state workload, so the overhead gate passes without an exception. CPU Inductor compilation time, warm runtime, replay error, and traced peak memory are reported separately. A larger CPU compile attempt was killed after five minutes; accelerator compilation remains unmeasured.

Purpose: prove the new lowering and scheduling machinery on a topology whose legacy result is known.

snnlang deliverables: emit all explicit state, update-order, delay, initializer, constraint, output, and observable information required to execute the existing one-layer PING graph without backend guesses.

tools/snn deliverables: add a graph planner and vectorized PyTorch executor for COBA-LIF E/I populations, AMPA/GABA projections, dense weights, direct spike inputs, non-spiking leaky-integrator populations, mean-voltage reduction, and named recordings. Lower the complete graph before the timestep loop; do not interpret graph nodes dynamically at every step. Keep `torch.compile` behind the same internal boundary used by the legacy engine.

Compatibility gate: legacy remains the default. Run the same bundle once through its legacy translation and once through the graph executor. Compare parameter identities and shapes, seeded initialization, short CPU state trajectories, spikes, named outputs, recordings, and checkpoint round trips. Benchmark compiled steady-state runtime, compilation time, and peak memory separately.

Exit criterion: the graph executor matches the legacy single-PING forward path within declared tolerances and is no more than approximately 5–10% slower at steady state for the reference workload, unless a measured exception is recorded and accepted before proceeding.

B.6 Milestone 3: arbitrary coupled forward graphs

Status at 2026-08-05: demonstrated. Commit `cf11906` adds deterministic topological scheduling, arbitrary named population sizes, independent inputs, dense feedforward/

recurrent/feedback projections, multiple incoming conductance streams, integral delay buffers, and all-population recordings. *exp077* (fixture commit `3b7a935`, evidence commit `fdfb19f`) archives uncoupled, unidirectional, reciprocal zero-additional-delay, and reciprocal explicitly delayed two-PING graphs. Each variant differs only in graph data and retains its authenticated graph, manifest, canonical diagram, named input/E/I recordings, delay evidence, and execution provenance. The fixture computes only compact recording diagnostics; the scientific coupling sweep is performed separately by Milestone 4 in *exp078*.

Purpose: deliver the first capability that the legacy `COBANet` architecture cannot express.

snnlang deliverables: compile multiple independently named components, feedforward/ recurrent/feedback projections, explicit positive delays, arbitrary E/I sizes, independent inputs, and observables from every population. Tighten validation for temporal causality, projection dimensions, polarity, and delayed feedback.

tools/snn deliverables: execute arbitrary named populations and projections, multiple incoming conductance streams, delay buffers, deterministic recurrent and feedback scheduling, and population-level recordings. Initial support may remain dense; sparse and structured lowering are later optimizations.

Acceptance fixture: two independently driven PING circuits with reciprocal GABA projections from each inhibitory population to the other excitatory population. Include uncoupled, unidirectional, reciprocal, and delayed variants. Exact tests establish which timestep receives each pulse; the experiment runner, not the engine, computes phase locking and synchrony.

Compatibility gate: milestone-0 and milestone-2 parity suites remain green and legacy CLI routing remains unchanged.

Exit criterion: all coupling variants require only graph changes, not simulator edits, and archive their graph, manifest, diagram, recordings, and execution provenance.

B.7 Milestone 4: first native gamma-coupling experiment

Status: executor capability complete; scientific experiment planned. Exact split-run tests cover refractory state, live conductances, recurrent delays, and delayed inputs. Runtime-state compatibility excludes parameter values but rejects changes to timestep, population size, projection identity, delay, synapse, and state layout. This is sufficient to burn in a structurally complete graph with reciprocal E-to-I weights at zero and continue the same causal trajectory with nonzero weights. General time-dependent weights are not required.

exp078 executes the registered graph-native Arnold-tongue test in two 80 E / 20 I circuits. It calibrates uncoupled gamma frequency, freezes locking tolerances and a bounded coupling pilot, then crosses measured natural-frequency detuning with reciprocal coupling. The 80/20 result recovers the widening tongue but narrowly fails the strict phase-lead clause. A reduced 800 E / 200 I confirmation near that phase-sign boundary is the appropriate finite-size follow-up; the full sweep is not silently redefined at tenfold scale.

B.8 Milestone 5: graph-native readouts and input bindings

snnlang deliverables: finish precise shape and unit inference for mean voltage, final voltage, spike count, duration-normalized spike rate, and cumulative potential. Define resolved dense

and event-stream input bindings separately from the graph, including masks and durations; paths live in execution data, not in the reusable graph.

tools/snn deliverables: execute all five readouts, supplied dense spike arrays, event streams, masks, named outputs, and recordings through stable request and CLI contracts. Dataset generation remains optional: callers may provide an input artifact or request a reusable tool-side encoder.

Gate: hand-calculated micro-fixtures verify every readout and masked-duration case; existing input and artifact behaviour remains unchanged.

B.9 Milestone 6: graph-native MNIST training

snnlang deliverables: validate cross-entropy objectives, complete parameter-group partitions, frozen parameters, AdamW settings, gradient clipping, checkpoint selection, and optimizer-state replay. Reject objectives without a differentiable path to a trainable parameter.

tools/snn deliverables: train the milestone-2 graph natively with surrogate gradients and the standard MNIST input binding. Support save, resume, selected and final checkpoints, inference, and optimizer-state replay.

Gate: compare the legacy and graph paths on initialization, forward values, gradients, one update, short learning curves, checkpoint replay, artifacts, compiled runtime, and memory. Keep bundle execution opt-in.

B.10 Milestone 7: SHD and deeper trained graphs

Extend data binding and graph-native training to event-stream SHD and multiple hidden PING layers. Add named recordings from every layer and an explicit standard checkpoint-selection contract. Use the existing one-layer SHD and two-layer SHD cells as conformance cases rather than retrofitting their runners.

Gate: dense MNIST, one-layer SHD, and two-layer SHD train and evaluate without runner access to live model internals, while the original commands and checkpoints remain supported.

B.11 Milestone 8: regularization and training boundaries

Implement declared surrogate choices, rate regularizers, multiple optimizer groups, constraints, stop-gradient boundaries, and differentiable-reachability diagnostics. Add only features exercised by current or imminent experiments.

Gate: each training construct has a hand-checkable gradient or update fixture; unsupported constructs fail during compilation or planning, never halfway through an expensive run.

B.12 Milestone 9: online confidence-controlled leak

Make an MNIST classifier's online confidence modulate g_L, allowing low- confidence trials more PING cycles. Attempt the least specialized route in order:

1. compose existing graph operations;
2. add a generic stateful controller node;
3. add a registered PyTorch backend operation;
4. use the importable API as an explicitly experimental extension.

Do not create a dedicated `ConfidenceToLeak` primitive solely for one experiment. The chosen route declares causal timing, bounds, initial state, evidence source, target parameter, checkpoint behaviour, and differentiability.

Gate: fixed-confidence fixtures reproduce expected g_L or τ_m trajectories; a one-step feedback delay has an exact causal test; the experiment compares a matched fixed-leak control and reports accuracy, calibration, decision time, PING cycles, and spike-count or energetic cost.

B.13 Historical experiment policy

Completed experiments remain unchanged by default. They are scientific records, not migration chores.

- Run selected old cells as regression cases through their original interface.
- Create separate snnlang conformance definitions rather than editing historical runners.
- Do not mass-retrofit the experiment corpus.
- Migrate an old experiment only when it is materially extended.
- Treat a migrated implementation as new and compare it explicitly with the archived result.
- Preserve the legacy CLI even after the graph executor becomes the default.

This policy avoids changing defaults, random initialization, dataset splits, checkpoint selection, or artifact meaning merely to achieve architectural uniformity. There is no scientific prize for deleting working compatibility code early.

B.14 Milestone 10: optional convergence and retirement

Changing the default executor or retiring `COBANet` is a separate, optional milestone, not an automatic consequence of graph-native success. It requires every selected conformance case to pass through the compatibility adapter, historical checkpoints to load or have a documented conversion, artifact contracts to remain stable, compiled performance not to be materially worse, and an explicit decision to accept the migration. The legacy CLI compatibility adapter survives retirement of the legacy execution architecture.