

PAL Architecture and Design

This document explains PAL's organization, the reasons behind its interfaces, and its implementation trade-offs. It focuses on execution architecture, state semantics, and subsystem design rather than teaching the API from the beginning. Use the [PAL Guide](#) for runnable modeling workflows and public API behavior, the [API Reference](#) for exact signatures, and the [Cheatsheet](#) for quick lookup.

Contents

1. [Foundations: responsibilities and execution](#) — How PAL's Python, compiled, and native parts work together to run a model.
2. [State, ownership, and execution semantics](#) — Where model data lives, how changes take effect, and what safe and fast modes mean.
3. [Subsystem design choices](#) — Why PAL uses different kinds of grids, agent collections, random sampling, and visualization tools.

1. Foundations: responsibilities and execution

1.1 The design problem PAL is solving

PAL targets models that repeatedly update many small pieces of state. A timestep might inspect thousands of neighborhoods, move individual agents, update a population count at every site, solve a field equation, or combine several of these operations. Expressing every inner operation as a Python call is flexible but can incur substantial interpreter overhead. Writing the whole model in C could reduce that overhead, but would make iteration and experimentation less convenient for Python users.

PAL's architecture splits the problem by frequency and responsibility. Python remains the control language for constructing the model, selecting parameters, loading inputs, managing experiments, and deciding when to draw or checkpoint. Repeated computation is expressed in functions that PAL can prepare for Numba compilation. The native core supplies low-level state and kernels behind that interface.

This division lets a model combine state representations without rewriting its timestep in C, and lets low-level implementations evolve without changing the model-facing API. Maintaining that interface requires coordination among wrappers, compiled overloads, and native state. Not all Python code is slow, and not every model function needs compilation. The split focuses optimization where repeated work makes it worthwhile. A function called once during setup usually needs less attention than a loop executed at every site over millions of timesteps. Start with clear model code, identify the repeated work, and benchmark before moving more logic into compiled kernels.

1.2 The layers and their responsibilities

PAL has five implementation layers:

1. **Public API and Protocols.** The public API defines what users can do with PAL: which objects and methods exist and what arguments they accept. **Protocols** describe those interfaces for type checking and autocomplete; they do not implement them.
2. **Python constructors and wrappers.** **Constructors** create PAL objects. **Wrappers** handle calls made from ordinary Python, translating user-friendly arguments into the operations PAL performs internally.
3. **Numba implementations and overloads.** Numba **compiles** Python functions into machine code for faster repeated execution. PAL's **overloads** provide versions of supported operations that work inside those compiled functions.

4. `pal.njit` **source transformation**. Before Numba compiles a function, PAL can **transform** certain Python constructs—such as neighborhood and box loops—into forms Numba can compile.
5. **C layer (native core)**. **Native** means code compiled from a lower-level language rather than executed by the Python interpreter. PAL uses C for some of its data storage and computational routines; not every operation calls C.

Tests check agreement between implementations; examples and documentation describe the public API's intended behavior.

The central idea is that users work with the same PAL objects and methods whether their code runs in Python or is compiled with Numba. PAL handles the different implementations behind the scenes. Not every operation passes through every layer.

1.3 Python, Numba, and the native core

PAL uses Python to construct models, configure experiments, inspect results, and perform work that does not need millions of repetitions. Repeated model calculations can instead run in functions compiled by Numba, avoiding much of the overhead of executing Python loops. Both paths operate on PAL objects, with the C core providing storage and selected lower-level operations.

Python wrappers handle ordinary calls; Numba overloads implement supported calls in compiled functions. Numba cannot compile arbitrary Python code: it needs supported operations and argument types. The implementations can differ, but supported operations should have the same observable behavior in both paths, including return values, state changes, boundary handling, and errors where those are part of the API. Not every operation reaches the C core; some work is handled directly by Python or Numba.

1.4 Model-code transformation with `pal.njit`

PAL's `njit` interface does more than call Numba's decorator. It inspects available Python source and rewrites supported neighborhood and Box loops into forms Numba can compile efficiently. This keeps spatial operations readable without requiring model code to manually convert coordinates at every iteration. If source inspection fails, PAL passes the function to Numba unchanged, so PAL-specific rewriting is not guaranteed for dynamically defined functions.

Neighborhood patterns used by the transformer must be compile-time tuples of integer-offset tuples, defined literally or through a resolvable module or closure constant. With `unroll=True`, PAL expands a fixed neighborhood loop so its individual neighbor operations are visible to the compiler. Unrolling can reduce loop overhead, but duplicates the loop body and can increase compilation time and code size. It is most appropriate for small stencils. Because of this expansion, Hood-level `break` and `continue` are unsupported with `unroll=True`; use an ordinary Hood loop when those operations are needed. In Python execution, `unroll=True` offers no compilation-time optimization.

Source transformation introduces constraints and failure modes beyond those of ordinary Python. Errors may arise from the rewriting stage, Numba typing, PAL wrappers, or native operations; tracebacks can also refer to transformed code rather than the original source. Argument annotations improve the type information available for diagnostics, but cannot guarantee exact source-line attribution or matching behavior in every case.

When a failure occurs under `@pal.njit`, first reproduce it in safe mode without calling `pal.FastMode()`. If that does not identify the problem, temporarily remove `@pal.njit` from the functions involved, starting near the failure, and inspect execution with a Python debugger. For compiled-only failures, reduce the case and check source availability, neighborhood definitions, annotations, and supported API usage before rewriting the scientific algorithm. Verify the fix in both Python and compiled execution.

2. State, ownership, and execution semantics

2.1 Native-backed state and object lifetime

A PAL object's public methods operate on state managed through PAL's native-backed implementation, not necessarily an ordinary Python container. The Python object provides access to that state, but internal values are not necessarily freely editable Python lists or NumPy arrays.

This distinction affects object lifetime and mutation. Creating, moving, or disposing an agent must keep position, occupancy, and handle validity consistent. Writing directly into an internal count or pointer-like representation could bypass those invariants, so PAL exposes explicit state-changing methods. Likewise, a returned array or slice must be interpreted according to its contract: a detached copy is a snapshot, not a live view into native memory, and editing it changes the copy until the result is explicitly assigned back.

Native-backed objects can support pickling, but pickling an object is not equivalent to serializing the entire simulation process. A model's complete checkpoint may also need parameters, timestep, other coupled objects, external state, and random-number state. See the [PAL Guide's saving section](#) for the checkpoint contract and its limitations.

2.2 Relative-offset neighborhoods

A neighborhood such as Moore or Von Neumann is naturally a pattern around a location. PAL represents it as relative integer offsets, then applies it to a specific grid position with `grid.Hood(...)`. This separates the reusable pattern from the grid location where it is used.

When `grid.Hood(...)` is called, PAL converts the hood's relative offsets into grid coordinates. It wraps coordinates along periodic axes and excludes neighbors that fall outside nonperiodic boundaries.

Relative offsets avoid storing absolute-neighbor lists for every cell and allow the compiled path to transform high-level neighborhood loops. Neighborhood destinations are not necessarily unique: small or wrapped domains can map multiple offsets to the same site. Whether to count those repeated contributions or deduplicate them depends on the model, so choose and test the intended behavior.

2.3 Separate update and commit operations

PAL distinguishes pending changes from committed state. In `PopGrid` and `PDEgrid`, `Add` queues deltas and `Update` commits them, making the timestep's commit boundary explicit.

This distinction has mathematical consequences. If every new site value must be computed from the same old field, immediate mutation can make results depend on iteration order. Queued deltas let the model calculate changes before committing them. This does not guarantee synchronous updates in every case: reads observe the current committed state, while direct indexing can mutate it immediately. PAL permits order-dependent algorithms but makes the choice explicit.

Agent updates have different semantics. `All()` snapshots which handles will be visited, but it does not snapshot the world. If one agent reproduces, later agents may see the newborn's occupancy even though the newborn is not part of the original handle snapshot. This is a deliberate and useful pattern for sequential agent models, but it is not equivalent to a synchronous cellular update.

PAL does not impose one scheduler because these update schemes represent different models. For example, a two-site population transfer can queue `Add(A, -1)` and `Add(B, 1)`, then call `Update()` once: both changes are calculated against the old committed counts. By contrast, moving an agent immediately changes which site subsequent agents encounter. The

representations can model similar populations without implementing the same update schedule. Choose the intended timing and test it with a small case whose result can be calculated by hand.

2.4 Process-wide safe and fast modes

Safe mode performs selected validation and provides diagnostics that are useful while developing a model. Fast mode removes selected checks from performance-sensitive paths. Keeping one mode active for a process makes the execution contract predictable for objects created within that process; PAL locks the choice at first construction.

A process-wide mode also avoids pretending that an individual object can be switched independently when calls across objects may share compiled functions or native machinery. The consequence is operational: call `pal.FastMode()` before creating any PAL object or window, and run safe and fast test suites in separate processes.

Fast mode should be treated as a measured optimization, not a correctness mode. It can reduce validation overhead; it cannot repair invalid model assumptions, unsafe numerical parameters, or a wrong update order. Safe mode can catch selected contract violations, but it cannot establish that the biological or physical equations are appropriate.

3. Subsystem design choices

The first two chapters cover PAL's implementation layers and the state and timing rules shared across execution paths. This chapter explains what each subsystem's representation enables and what remains the modeler's responsibility. For signatures, examples, performance options, safety-mode selection, and checkpointing, consult the API Reference and PAL Guide. Usage instructions complement this design rationale but cannot replace tests of implementation behavior or scientific validity.

3.1 Grid: typed site state and explicit indexing

A `Grid` represents one typed value per lattice site. The element type is chosen at construction, which makes the intended state representation explicit and allows the implementation to use compact native storage rather than a Python object at every site. Use a type that matches the model's meaning: integer labels for categories, Boolean values for masks, and floating-point values for continuous attributes.

Indexing supports both linear site indices and coordinate forms, but these are different views of the same lattice. Use coordinate indexing when the equation is naturally spatial; use a linear index when iterating sites or passing a site identifier between APIs. In more than one dimension, explicit coordinate indexing is often easier to review because the axes remain visible. The conversions `ToI` and `ItoX / ItoY / ItoZ` are explicit so the mapping can be tested rather than guessed.

The fast compiled path has a specific optimization: scalar `Grid` indexing can lower to a direct memory load or store, avoiding a PAL function call for each site access. This is possible because the native-backed grid has a fixed element type and its compiled representation carries the data address. The trade-off is that fast scalar access bypasses some checks performed by the safe path; the model must use valid indices and representable values. This optimization applies to scalar indexing; it does not make arbitrary NumPy slices or Python operations zero-overhead in compiled code.

Grid slices are detached NumPy copies, not live native-backed views. Editing the result changes only the copy; to update the grid, assign through the grid itself. For example, `region = grid[2:5, 3:8]; region[:] = 7` does not change `grid`, whereas `grid[2:5, 3:8] = 7` does. This makes ownership clear at the Python boundary. In safe mode, assignment validates relevant type, range, and nonfinite conditions; do not rely on fast mode to diagnose invalid values.

3.2 AgentGrid: handles, occupancy, and mutation

An agent handle is distinct from its position. The handle identifies the living agent; position determines where it is located. This separation lets a model retain identity while an agent moves, and lets queries report handles without copying whole Python objects. The modeler remains responsible for the meaning and lifecycle of any additional per-agent state. Handles may be reused after disposal, so a saved handle value is not a permanent identity across an agent's death and a later allocation. A snapshot of handles preserves the values selected for iteration, not historical identities; models that need persistent lineage identifiers should maintain them separately.

Stackability is an explicit constructor choice because allowing multiple agents at a site changes the model's state space and affects occupancy logic. A nonstackable grid rejects a second occupant in safe mode. Do not enable stackability merely to suppress an error: decide whether multiple occupancy is biologically or physically meaningful.

The distinction between `All()` and a live iteration matters for mutation. `All()` returns a snapshot of handles and is therefore the appropriate pattern when the loop can create or dispose agents. A snapshot freezes membership, not agent state or grid occupancy. Each later query observes current occupancy. If newborns must not act until the next timestep, build the snapshot before processing the timestep.

The `counts` property is a read-only occupancy query view, not a writable alternate path to agent state. This is a deliberate consistency boundary: occupancy is derived from agent placement and lifecycle, not a second independently editable source of truth. Changing a count directly would bypass agent handles, placement rules, and lifecycle bookkeeping. Query occupancy through this view; create, move, and dispose agents through the AgentGrid API.

3.3 PopGrid: counts instead of individual objects

A `PopGrid` stores integer population counts per site. It is intended for populations whose members are interchangeable for the questions being asked. This is a modeling assumption, not just a memory optimization: populations with equal site counts are indistinguishable even when their individual histories differ. This can greatly reduce storage and update overhead relative to individual agents, at the cost of individual histories and identities.

Its `Add` operation queues a delta; `Update` applies pending changes; `Reset` clears current state and pending deltas. The delta buffer lets the model compute changes before committing them. For a movement event from site A to site B, queue a negative delta at A and a positive delta at B, then update once. If the model reads current counts during this phase, it sees the pre-commit counts, which helps make the update independent of visitation order.

The `capacity` argument selects the smallest integer storage type that can represent the requested per-site population. It is a storage-sizing hint, not a population limit. In safe mode, PAL checks direct writes and committed updates against the selected storage type's range, not the requested capacity. Queued deltas have their own storage bounds, and the total population must fit a signed 64-bit integer. The modeler must enforce any model-specific population limit.

`GetPop()` reports the total across all sites. `All()` instead returns indices of nonzero sites, not their counts; read each indexed site to obtain its local count. The set of occupied sites and the population values are distinct.

3.4 PDEgrid: numerical operations with explicit assumptions

A `PDEgrid` stores a lattice scalar field. Unlike an `AgentGrid`, its state is a spatially distributed quantity rather than a collection of individually tracked entities. This representation supports field-level transport and source terms, but the modeler must choose a discretization and use consistent units for stored values and coefficients. Its transport methods represent different mathematical choices: scalar-coefficient diffusion, spatially varying or interface-defined coefficients, masked interfaces, alternating-direction implicit diffusion, radial diffusion on circular or spherical domains, and constant or spatially varying advection. Choose the method that matches the equation and geometry, not merely the fastest one.

`SetTimeSpaceStep` configures the temporal step `dt` and spatial steps `dx`, `dy`, and `dz` used by transport methods. These values are part of the discretized equation. A timestep that is too large can produce unstable or inaccurate results even when every API call is valid. Boundary conditions likewise belong to the mathematical problem: face-specific values and periodic wrapping can produce different solutions.

The distinction between queued and immediate mutations is especially important for fields. `Add()` queues a field delta for `Update()` to apply, whereas direct indexed assignment changes a site immediately. Do not assume every write is queued. Neither queued updates nor an implicit or explicit numerical method guarantees stability, positivity, or conservation on its own; those properties depend on the scheme, coefficients, timestep, and boundaries. Validate against a known field, test boundary behavior, and check convergence as timestep and grid spacing change.

3.5 IList: reusable collections for compiled workflows

`IList` stores reusable integer collections in Python and compiled code. For example, a model can collect empty neighboring sites and choose one for reproduction without allocating a new collection at every step.

`All()` returns a detached NumPy copy; `Iter()` exposes the current native-backed values without making that detached copy. This avoids allocations in tight compiled loops, but it is not a snapshot: changing the list while consuming an `Iter()` result can alter the values being traversed or invalidate assumptions about its storage. Use `All()` when iteration must be insulated from later mutations, and `Iter()` only while the underlying list remains unchanged. Clear reused lists before repopulating them, and check that a list is nonempty before calling `Random()`.

3.6 Randomness: thread-local PAL streams

`Seed`, `Random`, and `RandInt` use the same PAL random stream within an execution thread, including across Python and compiled calls. Each draw advances that thread's stream, so an additional draw changes its subsequent outcomes. PAL gives each thread its own PCG32 stream derived from a master seed, avoiding contention over a single shared RNG state. In multithreaded runs, reproducing results also requires consistent assignment of streams and work to threads.

Draw order matters: changing when one component samples can change later outcomes, even if the later component is unchanged. External RNGs have independent state and require separate seeding or checkpoint handling.

PAL object checkpoints do not save RNG state. To reproduce a continuation, call `pal.Seed(checkpointSeed)` immediately after saving the model, before continuing. In the resumed run, load the checkpoint and call `pal.Seed(checkpointSeed)` before any random draws. Both runs then start a new random sequence from the same model state. Reproduction still requires identical parameters, execution order, and subsequent draws; this approach is most reliable in single-threaded runs and does not guarantee identical multithreaded outcomes if thread assignments change.

3.7 Multinomial sampling: configuration versus sample state

A `Multinomial` sampler separates configuration from the state accumulated while sampling. `Setup(n)` configures the number of trials for repeated `Sample(p)` calls; `Binomial(n, p)` performs a draw with an explicit trial count. `NewMultinomial(other)` copies the sampler's configuration, but not its current multinomial remainder, so the new sampler starts with independent sampling state. Both samplers must use the same process-wide safety mode. Cloning therefore preserves the configuration without continuing a partially consumed sample.

3.8 Visualization: simulation state versus presentation

Visualization is separate from simulation state transitions, allowing headless execution and drawing at selected steps. `PixWindow` provides a writable pixel buffer for lattice-style images; `OpenGLWindow` draws 2D and 3D geometry.

After writing pixels or drawing geometry, call `update()` to request display. If the window cannot keep up, display frames are skipped instead of blocking the simulation. Saving images and recording GIF frames differ from display updates: output is written asynchronously so the simulation can continue, but requested saved frames are not skipped. The window thread remains open until pending drawing and saving work finishes.

Constructing frames still costs computation, even when presentation is nonblocking. Keep drawing separate from model updates when measuring performance. Headless rendering exercises the off-screen output path but cannot validate interactive window behavior.