Package 'fluxCore'

September 22, 2026

Type Package

Title Probabilistic Simulation of Single-Entity Systems in Irregular Time

Version 2.1.0

Author Jarrod Dalton [aut, cre]

Maintainer Jarrod Dalton <daltonj@ccf.org>

Description A foundation for probabilistic simulation of single-entity systems in which events occur at irregular times and each event updates only a small, sparse subset of the entity's state. Models are assembled from a declared schema and a bundle of callback functions for event proposal, state transition, and stopping, with their contracts validated before simulation. Supports competing event processes, schema-declared decision points with user-supplied policies, typed parameter draws for representing uncertainty, and optional trajectory recording for auditing simulated decisions. Designed to be domain agnostic: this package contains no model of any particular system, only the scaffolding for building one.

License LGPL-3

Encoding UTF-8

Imports R6

Suggests future, future.apply, jsonlite, parallel, testthat (>= 3.0.0)

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

NeedsCompilation no

Repository CRAN

Date/Publication 2026-09-22 14:30:02 UTC

Contents

ActionEvent *Construct an ActionEvent*

Description

Timeline-native action proposed by a policy at a decision point. ActionEvents enter the same arbitration and refresh lifecycle as all other events — no side-channel state mutation.

Usage

```
ActionEvent(
  action_type,
  time_next,
  decision_point_id = NULL,
  params = NULL,
  metadata = NULL
)
```

Arguments

action_type	Character scalar; must match an allowed_actions entry if the decision point declared any.
time_next	Numeric scalar; realization time on the canonical timeline.
decision_point_id	Character scalar; which decision point produced this action. If NULL (the default), the engine fills this in automatically from the firing decision point's id. During a policy call, an explicitly supplied value must exactly match the firing decision point. Outside policy dispatch, the field may be used to construct a self-described action.
params	Optional named list of action-specific parameters.
metadata	Optional named list for policy provenance or audit fields.

Value

A list of class "ActionEvent".

as_time_spec *Coerce to time_spec*

Description

Coerces various inputs into a validated time_spec object. Accepts:

- A time_spec object (pass-through)
- A named list with unit (required) and optionally origin, zone

Usage

```
as_time_spec(x, ...)
```

Arguments

x	Object to coerce.
...	Additional arguments (unused).

Value

A validated time_spec object.

---

block_vars *Get variable names in a schema block*

Description

Convenience helper to look up the variables belonging to a named schema block (e.g., "bp", "cbc", "cbc_diff", "bmp", "cmp"). Membership is many-to-many: a variable may appear in multiple blocks.

Usage

```
block_vars(schema, block)
```

Arguments

schema	An entity state schema (named list).
block	Block name (character scalar).

Value

Character vector of variable names in the order they appear in schema.

check_derived *Register derived variables on an Entity*

Description

Validates and registers derived-variable functions for snapshot-time evaluation.

Usage

```
check_derived(entity, derived_vars, replace = FALSE)
```

Arguments

entity	A Entity object.
derived_vars	Named list of derived-variable functions, or NULL.
replace	If TRUE, overwrite existing derived vars with the same names.

Value

Returns entity invisibly.

combine_updates *Combine multiple update lists into one atomic update payload*

Description

Convenience helper for transition() implementations that need to apply multiple update sources (e.g., scalar tweaks plus one or more block updates). The function concatenates named lists and errors if any variable is updated more than once within the same event.

Usage

```
combine_updates(...)
```

Arguments

...	One or more named lists of updates (or NULL).

Value

A single named list of updates, or NULL if all inputs are NULL.

compose_bundles *Compose ModelBundles and transition components*

Description

These helpers make it easy to layer policy/intervention logic on top of baseline natural history dynamics without rewriting the baseline bundle.

Usage

```
compose_bundles(
  baseline,
  policy = NULL,
  merge = c("policy_wins", "baseline_wins", "error_on_conflict")
)
```

Arguments

baseline	Baseline model bundle list.
policy	Optional policy/intervention bundle list overriding selected components.
merge	Patch conflict strategy for transition updates: "policy_wins", "baseline_wins", or "error_on_conflict".

DecisionPlan *Construct a coordinated decision plan*

Description

Represents one complete set of selections returned by a grouped policy consultation. Each named entry is either one `ActionEvent()` or explicit `NULL`, meaning that the eligible leaf was considered but no new action was selected. Completeness against the eligible members is validated by the Engine, which has the activation context.

At runtime, `selections` must name every and only eligible member exactly once. An explicit `NULL` means "considered, but no new action selected"; it does not cancel an action already pending for that member. Core validates the complete plan and every pending-slot outcome before modifying any member slot. This all-or-none boundary covers plan acceptance and staging only. Accepted constituent actions subsequently arbitrate and realize as independent timeline events.

Usage

```
DecisionPlan(selections, metadata = NULL)
```

Arguments

selections	Non-empty named list with unique leaf decision-point ids. Every value must be an ActionEvent() or explicit NULL.
metadata	Optional named list of compact plan-level provenance. Core treats these values as opaque audit information, never as execution input. When trajectory logging is enabled, metadata is retained on raw grouped leaf records but is not flattened by trajectory_table().

Value

A list of class "DecisionPlan".

DecisionPoint *Construct a DecisionPoint specification*

Description

Declares a point in the simulation where a policy may be consulted. Decision points are declared in the Schema (schema$decision_points), not inferred at runtime.

Usage

```
DecisionPoint(
  id,
  trigger,
  allowed_actions = NULL,
  action_handlers = NULL,
  condition = NULL,
  audit = FALSE,
  observation_fn = NULL,
  label = NULL,
  on_pending_action = c("warn", "replace", "keep", "error")
)
```

Arguments

id	Character scalar; unique identifier (e.g., "post_dropoff").
trigger	Character vector of event types and/or a predicate function function(event) that returns TRUE when the decision point fires. Evaluated **pre-transition** on the raw event object; entity state is not yet updated at this point. An explicitly supplied NULL declares a group-only leaf that cannot fire directly and must be referenced by a GroupedDecisionPoint() in the full schema. Omitting trigger is an error.
allowed_actions	Optional character vector of named action types. If NULL, the policy is unconstrained.

`action_handlers`
: Optional named list of functions, keyed by action type. Each function has signature `function(entity, event)` (with optional `param_ctx`) and returns a named list of state updates, or `NULL` for no change. When present, the engine calls the matching handler directly when an ActionEvent of that type fires — bypassing `bundle$transition()`. Names must be a subset of `allowed_actions`.

`condition`
: Optional function `function(entity)` evaluated **post-transition** on the updated entity. If it returns `FALSE`, the policy is not consulted for this event cycle. Use `condition` to gate on entity state (e.g., `function(entity) entity$current$battery_pct < 25`). When `NULL` (default), the policy is always consulted when `trigger` fires.

`audit`
: Logical scalar (default `FALSE`). When `TRUE`, a `TrajectoryRecord` with `condition_met = FALSE` is emitted even for cycles where `condition` vetoed the policy call. Useful for auditing why a decision point did not fire.

`observation_fn`
: Optional function `function(entity)` that computes the observable state presented to the policy. Defaults to a full entity snapshot when `NULL`.

`label`
: Optional human-readable description.

`on_pending_action`
: What to do when the policy proposes an action at this decision point while a previously proposed action from the *same* decision point is still waiting to be realized. A decision point holds at most one pending action at a time.

  - `"warn"` (default) – replace the pending action and emit a warning.
  - `"replace"` – replace the pending action silently. Use this to state explicitly that superseding is the intended behavior, as for a decision point whose job is to reschedule something.
  - `"keep"` – keep the pending action and discard the new proposal.
  - `"error"` – stop with an error.

Value

A list of class `"DecisionPoint"`.

---

declare_variable *Target a variable for history-derived features*

Description

declare_variable() constructs a target that refers to a state variable's sparse history in entity$hist. It is used to define derived features based on recent values (e.g., counts, means, lags) when building model-ready snapshots.

Usage

```
declare_variable(name)
```

Arguments

name	Single variable name (character scalar).

Value

A target list used by derive() and lag_of().

---

derive *Define a derived feature*

Description

derive() returns a function computing a history-derived feature as of (j, t). Intended to populate Entity$derived_vars and be used by Entity$snapshot*().

Usage

```
derive(
  name,
  target,
  lookback_t = NULL,
  lookback_j = NULL,
  fn = c("count", "min", "max", "mean", "median", "sd", "iqr", "any", "all"),
  include_current = TRUE,
  force = FALSE,
  na_value = NA,
  clock = "time"
)
```

Arguments

name	Name of the derived feature.
target	A target created by event() or declare_variable().
lookback_t	Optional numeric lookback window on the time axis. Takes precedence over lookback_j.
lookback_j	Optional integer lookback window in event-index units.
fn	Summary function name. For event() targets, only count, any, and all are supported.
include_current	Logical; include the boundary event at t/j (default TRUE).
force	Logical; if TRUE, return a value when there is no history (default FALSE).
na_value	Value to return when force=TRUE and no history exists (default NA).
clock	Which column in entity$events defines the time axis for lookback_t (default "time").

Value

A function (entity, j, t) -> scalar or NULL.

`dp_fires` *Test whether a DecisionPoint fires for a given event*

Description

Used internally by the Engine to detect decision points during the simulation loop. A group-only DecisionPoint with an explicit `NULL` trigger never fires through this direct path.

Usage

```
dp_fires(dp, event)
```

Arguments

`dp`	A `DecisionPoint` object.
`event`	An event list with at least `event_type`.

Value

Logical scalar.

`Engine` *Engine*

Description

Orchestrates simulation by repeatedly proposing the next event(s), applying a transition patch, recording the event on a Entity, and stopping when bundle$stop() returns TRUE (or a max_time / max_events limit is reached).

Run result

`Engine$run()` returns a list containing the updated `entity`, its `events`, optional `observations`, and `stopped_by`. The termination value is one of `"stop"`, `"max_time"`, `"max_events"`, or `"no_proposals"`. When trajectory logging is configured, the result also contains `trajectory_records`.

Direct `Engine$run()` calls construct one `ParamContext` from `bundle$params`, or an empty parameter list, unless a Core cohort harness supplies a prevalidated context. `Engine$run_draw()` remains a raw parameter payload entry point. Its caller owns RNG setup, so a RuntimeContext stored on the Engine does not reseed the internal draw run.

Grouped decision dispatch

For an Engine assembled with `schema$decision_groups`, raw direct/group overlap is rejected before transition. After the event transition is applied once, Core freezes ordinary eligibility followed by grouped-member eligibility. Conditions are evaluated in ordinary schema order, then group declaration and member order, before any policy call. Core dispatches ordinary policies first and then each fired group in declaration order. A non-empty eligible group receives exactly one `policy$propose_plan()` call; an empty eligible group skips policy.

A grouped `DecisionPlan()` must name every and only eligible member and is validated completely, including every pending-slot outcome, before one group-level commit. This atomic boundary coordinates action selection and staging only; constituent actions later arbitrate and realize independently. Ordinary decisions and separate groups are independent local boundaries, not one event-wide transaction.

Grouped trajectory records

With trajectory logging enabled, a grouped activation emits ordinary-style leaf rows, not a synthetic parent row. Every eligible member gets a row, including an explicit `NULL` selection; an ineligible member gets a veto row only when its leaf declares `audit = TRUE`. All rows from one firing share `grouped_decision_point_id` and a deterministic run-local `group_activation_id`. A zero-eligible firing still advances activation identity and emits its opted-in veto rows, but has no policy call or plan metadata. Plan metadata remains opaque and is retained only in raw records.

Failure and partial progress

Ambiguous raw activation is rejected before transition. Otherwise the triggering transition and atomic Entity update occur before decision conditions and policies. A condition, policy, or plan error therefore stops the run without rolling back that triggering event. A failing action handler stops before its action event or state effect is committed. Ordinary work and each accepted group are independent local boundaries, so a later group error does not roll back an earlier policy call, diagnostic, or pending-slot commit.

Methods

Public methods:

- `Engine$new()`
- `Engine$run()`
- `Engine$run_draw()`
- `Engine$clone()`

`Engine$new()`:

Usage:

```
Engine$new(bundle = NULL, ...)
```

`Engine$run()`:

Usage:

```
Engine$run(
  entity,
  max_events = 1000,
  max_time = NULL,
  return_observations = TRUE,
  .internal_ctx = NULL
)
```

`Engine$run_draw()`:

Usage:

```
Engine$run_draw(
  entity,
  params = list(),
  draw_id = 1L,
  sim_id = 1L,
  max_events = 1000,
  max_time = NULL,
  return_observations = TRUE
)
```

`Engine$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Engine$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`Entity` *Entity*

Description

R6 simulation state container used by fluxCore::Engine. An Entity stores current state, event history, derived-variable registrations, and metadata such as id and optional meta$entity_type.

Methods

Public methods:

- `Entity$new()`
- `Entity$state()`
- `Entity$as_list()`
- `Entity$update()`
- `Entity$state_at_time()`
- `Entity$snapshot()`
- `Entity$snapshot_at()`

- Entity$snapshot_at_time()
- Entity$state_at()
- Entity$clone()

Entity$new():

Usage:

```
Entity$new(
  init = list(),
  schema,
  derived_vars = NULL,
  id = NULL,
  entity_type = NULL,
  time0 = 0,
  event_type0 = "init"
)
```

Entity$state():

Usage:

```
Entity$state(vars = NULL)
```

Entity$as_list():

Usage:

```
Entity$as_list(vars = NULL)
```

Entity$update():

Usage:

```
Entity$update(time, event_type, changes = NULL)
```

Entity$state_at_time():

Usage:

```
Entity$state_at_time(time, vars = NULL)
```

Entity$snapshot():

Usage:

```
Entity$snapshot(vars = NULL)
```

Entity$snapshot_at():

Usage:

```
Entity$snapshot_at(j, vars = NULL)
```

Entity$snapshot_at_time():

Usage:

```
Entity$snapshot_at_time(time, vars = NULL)
```

Entity$state_at():

Usage:

```
Entity$state_at(j, vars = NULL)
```

`Entity$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Entity$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`EnvironmentContext` *Construct an EnvironmentContext*

Description

Optional. Provides named external signals and world-step/reset hooks for ABM or RL scenarios. Designed to accommodate multi-entity ABM (an add-on sub-repo) without requiring Engine architectural changes.

Usage

```
EnvironmentContext(
  signals = NULL,
  step_fn = NULL,
  reset_fn = NULL,
  info = NULL
)
```

Arguments

`signals`	Optional named list of external signal values visible to the policy at each decision point.
`step_fn`	Optional function; advances the world by one step (ABM/RL use cases).
`reset_fn`	Optional function; resets the world state (RL use cases).
`info`	Optional named list of auxiliary metadata from the environment.

Value

A list of class `"EnvironmentContext"`.

event *Feature target: event type*

Description

Create a target describing events of a given type in entity$events.

Usage

```
event(event_type)
```

Arguments

event_type	Character scalar.

Value

A target list used by derive().

GroupedDecisionPoint *Construct a grouped decision point*

Description

Declares one shared trigger that opens a coordinated policy consultation across existing leaf DecisionPoint() objects. Group members are references to ids in schema$decision_points; a grouped declaration does not copy leaf action contracts or own pending-action slots.

When the group trigger fires, the Engine applies the triggering event's transition once and then evaluates each member's DecisionPoint() condition against the post-transition Entity. Members with no condition, or a condition returning TRUE, are eligible and are presented together in declared member order to policy$propose_plan(). An empty eligible set skips the policy call.

Usage

```
GroupedDecisionPoint(id, trigger, members, label = NULL)
```

Arguments

id	Non-empty character scalar. Group and leaf ids share one schema-wide namespace.
trigger	Non-empty character vector of event types or a predicate function function(event). Unlike a group-only leaf, a grouped decision point must own a trigger.
members	Character vector containing at least two distinct leaf decision-point ids. Declaration order is preserved and is the canonical member order.
label	Optional human-readable character scalar.

Value

A list of class "GroupedDecisionPoint".

---

lag_of *Define a lagged feature from variable history*

Description

Extract the k-th most recent value of a variable from sparse history as of (j, t).

Usage

```
lag_of(
  name,
  target,
  k = 1,
  lookback_t = NULL,
  lookback_j = NULL,
  include_current = FALSE,
  force = FALSE,
  na_value = NA,
  clock = "time"
)
```

Arguments

name	Name of the derived feature.
target	A target created by declare_variable().
k	Positive integer lag order (1 = most recent prior value by default).
lookback_t	Optional numeric lookback window on the time axis. Takes precedence over lookback_j.
lookback_j	Optional integer lookback window in event-index units.
include_current	Logical; include boundary event at t/j (default FALSE).
force	Logical; if TRUE return na_value when insufficient history exists (default FALSE).
na_value	Value to return when force=TRUE and insufficient history exists (default NA).
clock	Which column in entity$events defines the time axis for lookback_t (default "time").

Value

A function (entity, j, t) -> scalar or NULL.

load_model *Assemble and validate a simulation model (v2 API)*

Description

load_model() is the recommended entry point for the v2 architecture. It validates all supplied components against the schema and against each other, applies defaults, and returns a configured Engine in v2 mode.

Usage

```
load_model(
  schema,
  bundle,
  policy = NULL,
  environment = NULL,
  trajectory = NULL,
  runtime = NULL,
  param_source = NULL
)
```

Arguments

schema	A validated full schema list (from set_schema() or equivalent) containing at minimum $variables and a $time_spec of class "time_spec". For 2.1 compatibility, a variables-only named list is also accepted with a migration warning. A full schema may contain canonical leaf declarations in $decision_points and grouped references in $decision_groups. See set_schema().
bundle	A ModelBundle list with at minimum propose_events, transition, and stop callbacks, plus a $time_spec semantically equal to the full schema declaration.
policy	Optional for ordinary decisions. A function or list with a propose_action method called at ordinary decision points. A schema with non-empty decision_groups requires a list with an exact propose_plan callback as documented below; there is no implicit ordinary-policy fallback.
environment	Optional. An EnvironmentContext for ABM/RL scenarios.
trajectory	Optional. A TrajectoryLogger configuration list; enables TrajectoryRecord emission. Requires schema$decision_points to be non-empty.
runtime	Optional. A RuntimeContext specifying reproducibility and backend settings. Defaults are applied when NULL.
param_source	Optional. A ParamSource that resolves ParamContexts once per run.

Details

Engines returned by load_model() enforce the following at runtime:

- **Fail fast on `ctx`-style usage**: passing a `ctx=` argument to `Engine$run()` raises an immediate error. Use typed context objects (`SimContext`, `ParamContext`, `RuntimeContext`) instead.
- Bundle callbacks that need typed context declare supported `sim_ctx` and `param_ctx` formals. The removed v1.x-style `ctx` formal is rejected.

Value

An Engine object with `v2_mode = TRUE`.

Model time contract

A full schema and its ModelBundle must declare semantically equal `time_spec` objects. Equality covers unit, origin instant, origin class, and zone; object identity is not required. Engine`$time_spec` is the sole assembled runtime clock and is propagated through `SimContext`.

For 2.1 compatibility, a variables-only schema (a named list of variable definitions without the full `$variables` wrapper) is accepted with a migration warning and uses `bundle$time_spec`. A full schema, identified by the presence of a `$variables` field, must also contain `$time_spec`.

Decision declaration contract

Leaf `DecisionPoint()` objects remain in `schema$decision_points`. Shared trigger declarations live separately in `schema$decision_groups` and reference leaf ids rather than embedding leaf definitions. `load_model()` defensively repeats the global id, membership, no-nesting, and group-only trigger checks performed by `set_schema()` so manually assembled schemas do not bypass the declaration contract. When at least one group is declared, `policy` must be a list exposing `propose_plan()`; grouped dispatch never falls back implicitly to `propose_action()`.

For each fired group with at least one eligible member, Core makes exactly one call of the following shape:

```
policy$propose_plan(
  grouped_decision_point,
  eligible_decision_points,
  entity,
  sim_ctx = NULL,
  param_ctx = NULL
)
```

The first three named inputs are required. Core supplies `sim_ctx` and `param_ctx` only when the callback declares those formals. The eligible decision points are canonical schema objects in group member order, evaluated after the triggering transition; when none are eligible, Core skips the callback. The result must be a complete `DecisionPlan()` rather than `NULL`.

User experience tiers

Level	Entry point	What you supply
1	`load_model(schema, bundle)`	In-memory schema + bundle
2	`load_model(schema, bundle, ...)` + JSON schema export	JSON spec + language code

3	All optional components supplied	Full stack

Trajectory output contract

When `trajectory` is configured, the Engine returned by `load_model()` emits `trajectory_records` in run outputs.

- `Engine$run(...)` includes `trajectory_records` when trajectory logging is enabled.
- `run_cohort(...)` run entries include per-run `trajectory_records` when enabled.
- `trajectory_records` is a list of plain named lists (JSON-serializable).
- `trajectory$detail` controls state capture:
  - `none`: `state_before` and `state_after` are `NULL`.
  - `summary`: both are outputs of `summary_fn` (default `state_summary_default()`).
  - `full`: both are full snapshots of `entity$current`.

See Also

`SimContext()`, `ParamContext()`, `RuntimeContext()`, `EnvironmentContext()`, `DecisionPoint()`, `GroupedDecisionPoint()`, `DecisionPlan()`, `TrajectoryRecord()`

`ParamContext` *Construct a ParamContext*

Description

Encapsulates one parameter realization for a simulation run. Resolved once per draw by a `ParamSource` (or supplied directly) and injected into every bundle callback that accepts a `param_ctx` argument.

Usage

```
ParamContext(draw_id, params, provenance = NULL)
```

Arguments

`draw_id`	Positive integer-valued numeric scalar identifying this parameter draw. Whole-valued doubles such as `5.0` are accepted and stored as integers; fractional, non-finite, non-positive, and out-of-range values are rejected rather than truncated.
`params`	Named list of concrete parameter values.
`provenance`	Optional character scalar labelling the source of this draw (e.g., `"posterior_draw_42"`).

Value

A list of class `"ParamContext"`.

`RuntimeContext` *Construct a RuntimeContext*

Description

Captures reproducibility and backend settings. A context stored on an Engine configures direct `Engine$run()` calls. When supplied explicitly to `run_cohort()`, it configures the cohort and takes precedence over scalar seed/backend/worker controls; its `replicate_id` must be `NULL` because cohort replication is identified by `sim_id`.

Usage

```
RuntimeContext(
  seed = NULL,
  replicate_id = NULL,
  backend = "none",
  n_workers = NULL
)
```

Arguments

`seed`	Optional integer scalar; top-level seed. NULL = unseeded.
`replicate_id`	Optional integer scalar; stochastic replicate index for a direct Engine run. Cohorts use `sim_id` and reject a non-`NULL` value on an explicitly supplied RuntimeContext.
`backend`	Character scalar; one of `"none"` (default), `"cluster"`, `"mclapply"`, `"future"`.
`n_workers`	Optional integer; number of parallel workers.

Details

The public reproducibility contract is: `seed` + `draw_id` + `replicate_id` + `entity_id` = deterministic output. Internal stream allocation (`stream_id`) is set by the run harness and must not be set by users directly.

Value

A list of class `"RuntimeContext"`.

run_cohort *Run a cohort of entities (serial or parallel) with optional global parameter draws*

Description

These helpers support batch simulation with: global parameter draws reused across entities (parameter uncertainty) multiple stochastic sims per entity per draw (stochastic uncertainty) parallelization across entities

Usage

```
run_cohort(
  engine,
  entities,
  n_param_draws = 1,
  n_sims = 1,
  param_draws = NULL,
  runtime = NULL,
  max_events = 1000,
  max_time = NULL,
  return_observations = TRUE,
  backend = NULL,
  n_workers = NULL,
  seed = NULL
)
```

Arguments

engine	An Engine object (with a materialized bundle).
entities	List of Entity objects.
n_param_draws	Integer; number of global parameter draws (D). Default 1.
n_sims	Integer; number of stochastic sims per entity per draw (S). Default 1.
param_draws	Optional list of exactly n_param_draws ParamContext objects. Draw ids are stable identities: they must be positive and unique, need not be contiguous, and determine canonical cohort order. Bare parameter payload lists are not accepted. If NULL, the function calls engine$bundle$sample_params(D) when available; otherwise it constructs contexts numbered 1:D from bundle$params or an empty parameter list.
runtime	Optional RuntimeContext; carries seed, backend, and n_workers for v2-mode engines. Takes precedence over the individual seed, backend, and n_workers arguments when non-NULL. Its replicate_id must be NULL; cohort replicates are represented by sim_id. An explicit runtime with seed = NULL requests an unseeded cohort even when the Engine stores a seed.
max_events	Max events per run.

max_time	Optional max time per run.
return_observations	Logical; whether to return observations (if bundle provides observe()).
backend	Backend used to parallelize across entities. One of "none", "cluster", "mclapply", or "future". When NULL, inherits the Engine's stored runtime backend when available, then defaults to "none". Ignored when runtime is supplied.
n_workers	Integer; workers for parallel; default parallel::detectCores() - 1. When NULL, inherits the Engine's stored runtime setting when available. Ignored when runtime is supplied.
seed	Optional base seed for reproducibility. Public contract: fixed seed + draw_id + sim_id + entity_id = reproducible output. A non-NULL value overrides the Engine's stored runtime seed. Ignored when runtime is supplied.

Value

A list with: runs: list of per-run outputs (entity/events/observations; plus trajectory_records when enabled) with labels index: data.frame mapping run_id -> entity_id/param_draw_id/sim_id param_draws: the validated contexts in ascending draw-id order

The index run_id is a batch-local join key shared by the corresponding run, its SimContext, and its trajectory records. For replay across cohort calls, use the stable entity/draw/simulation coordinates rather than assuming that a sequential run_id will remain unchanged when the cohort shape changes.

---

schema_assert_levels *Assert that levels are declared in the schema*

Description

For binary/categorical/ordinal variables, stops if any provided levels are not declared in the schema.

Usage

```
schema_assert_levels(schema, var, levels)
```

Arguments

schema	A validated schema (or a raw schema that can be validated).
var	A single variable name.
levels	Character vector of levels to check.

Value

Invisibly returns TRUE.

schema_assert_types *Assert that variables have allowed schema types*

Description

Stops with an informative error if any variables have schema types outside the allowed set.

Usage

```
schema_assert_types(schema, vars, allowed_types)
```

Arguments

schema	A validated schema (or a raw schema that can be validated).
vars	Character vector of variable names.
allowed_types	Character vector of allowed schema types.

Value

Invisibly returns TRUE.

schema_assert_vars *Assert that variables exist in a schema*

Description

Stops with an informative error if any requested variable names are not present in the schema.

Usage

```
schema_assert_vars(schema, vars)
```

Arguments

schema	A validated schema (or a raw schema that can be validated).
vars	Character vector of variable names.

Value

Invisibly returns TRUE.

schema_blocks *List unique schema blocks*

Description

Extract the set of unique block names declared in a schema via the optional blocks metadata field on each variable entry. Variables may belong to multiple blocks (many-to-many).

Usage

```
schema_blocks(schema)
```

Arguments

schema	An entity state schema (named list).

Value

Character vector of unique block names.

schema_validate *Validate an entity schema*

Description

Validates the structure and required metadata of an entity schema. The schema is a named list where each entry describes one state variable.

Usage

```
schema_validate(schema)
```

Arguments

schema	A named list schema.

Value

The validated schema (invisibly), with normalized type fields.

schema_validator_integer *Create an integer schema validator*

Description

Returns a predicate function that checks a scalar integer value against inclusive numeric bounds and optional missingness.

Usage

```
schema_validator_integer(min = -Inf, max = Inf, allow_na = FALSE)
```

Arguments

min	A single numeric lower bound.
max	A single numeric upper bound.
allow_na	Logical scalar indicating whether missing values are permitted.

Value

A function of signature function(x) -> TRUE/FALSE.

schema_validator_levels *Create a levels-based schema validator*

Description

Returns a predicate function that checks a scalar value against a set of allowed character levels and optional missingness.

Usage

```
schema_validator_levels(levels, allow_na = FALSE)
```

Arguments

levels	A character vector of allowed level labels.
allow_na	Logical scalar indicating whether missing values are permitted.

Value

A function of signature function(x) -> TRUE/FALSE.

schema_validator_numeric *Create a numeric schema validator*

Description

Returns a predicate function that checks a scalar numeric value against inclusive numeric bounds and optional missingness.

Usage

```
schema_validator_numeric(min = -Inf, max = Inf, allow_na = FALSE)
```

Arguments

min	A single numeric lower bound.
max	A single numeric upper bound.
allow_na	Logical scalar indicating whether missing values are permitted.

Value

A function of signature function(x) -> TRUE/FALSE.

schema_var_info *Get schema metadata for variables*

Description

Returns schema metadata (type, levels, blocks) for the requested variables.

Usage

```
schema_var_info(schema, vars)
```

Arguments

schema	A validated schema (or a raw schema that can be validated).
vars	Character vector of variable names.

Value

A data.frame with columns var, type, levels, and blocks.

`set_schema` *Build or extend a fluxCore schema with hybrid shorthand*

Description

Constructs a validated fluxCore schema from a hybrid `vars` specification. Each element of `vars` may be either a type-name string (e.g. `"count"`) or a fully-specified list (e.g. `list(type = "positive_numeric", max = 20)`). Optionally merges onto an existing `schema`, with explicit `overwrite` and `remove` controls.

Usage

```
set_schema(
  vars = NULL,
  schema = NULL,
  remove = NULL,
  overwrite = FALSE,
  time_spec = NULL,
  decision_points = NULL,
  decision_groups = NULL
)
```

Arguments

`vars`	Named list (or character vector) of variable specs. Each element is either a single type-name string or a list containing `type` plus any recognized schema fields (`min`, `max`, `levels`, `default`, `coerce`, `validate`, `allow_na`, `required`, `blocks`).
`schema`	Optional existing schema to extend. If `NULL`, a new schema is created. May be either a plain variables list or a full schema list (with a `$variables` field) — both forms are accepted.
`remove`	Optional character vector of variable names to remove from `schema` before merging `vars`. Errors if any name is not present.
`overwrite`	Logical scalar. If `FALSE` (default), adding a variable already present in `schema` is an error. If `TRUE`, existing entries are replaced.
`time_spec`	Optional `time_spec()` object. When provided, the returned value is a full schema list with `$variables`, `$time_spec`, `$decision_points`, and `$decision_groups`, ready to pass directly to `load_model()`.
`decision_points`	Optional list of `DecisionPoint()` objects to attach to the schema. Requires `time_spec` to also be provided. When supplied, the returned value is a full schema list (see `time_spec` above).
`decision_groups`	Optional list of `GroupedDecisionPoint()` objects. Members reference ids in `decision_points`. Requires `time_spec`; group and leaf ids are validated in one shared namespace.

Details

When time_spec, decision_points, or decision_groups is supplied, set_schema() returns a **full schema list** (with $variables, $time_spec, $decision_points, and $decision_groups) suitable for direct use with load_model(). When none is supplied, it returns just the validated variables spec (backward-compatible with prior usage and with Entity$new(schema = ...)).

Value

A validated fluxCore variables spec (named list), or — when time_spec, decision_points, or decision_groups is supplied — a full schema list with $variables, $time_spec, $decision_points, and $decision_groups.

Examples

```
# Variables only (backward-compatible):
vars <- set_schema(vars = list(
  route_zone  = list(type = "categorical",
                     levels = c("urban", "suburban", "rural")),
  battery_pct = "percent",
  payload_kg  = list(type = "positive_numeric", max = 20),
  deliveries  = "count",
  prob_rain   = "probability"
))

# Full schema for load_model():
dp <- DecisionPoint(id = "dp1", trigger = "event_A",
                    allowed_actions = c("accept", "decline"))
schema <- set_schema(
  vars             = list(battery_pct = "percent"),
  time_spec        = time_spec(unit = "hours"),
  decision_points  = list(dp),
  decision_groups  = NULL
)
# Full schemas also contain schema$decision_groups (NULL here).
```

---

set_vars *Create a named list of updates from variable names and values*

Description

Many models generate multivariate predictions (e.g., SBP/DBP, CBC panels) but Entity$update() expects a named list of scalar changes. set_vars() converts vectors into the expected named-list format.

Usage

```
set_vars(vars, values)
```

Arguments

vars	Character vector of variable names.
values	Vector of values (same length as vars).

Value

Named list suitable for transition() returns.

---

set_vars_from_named *Create a named list of updates from a named vector*

Description

Convert a named vector into a named list suitable for returning from transition(). Optionally select and order a subset via vars.

Usage

```
set_vars_from_named(x, vars = NULL)
```

Arguments

x	Named vector (e.g., numeric) of updates.
vars	Optional character vector specifying which names to take and in what order. If provided, values are taken as x[vars] and names are set to vars.

Value

Named list suitable for transition() returns.

SimContext *Construct a SimContext*

Description

Captures invariant simulation-level metadata for one Engine run. Created once by load_model() / Engine$run() in v2 mode and passed to every bundle callback that accepts a sim_ctx argument.

Usage

```
SimContext(
  run_id,
  time_spec,
  model_id = NULL,
  scenario_id = NULL,
  horizon = NULL
)
```

Arguments

run_id	Character scalar; unique identifier for this simulation run.
time_spec	A time_spec object (from time_spec()). Resolved from the model schema.
model_id	Optional character scalar identifying the model.
scenario_id	Optional character scalar labelling an experimental condition.
horizon	Optional numeric scalar declaring the simulation horizon.

Value

A list of class "SimContext".

---

state_summary_default *Default state summary function for TrajectoryRecord*

Description

Captures a compact named list of all current variable values from an entity. Used as the default summary_fn when state_before or state_after is requested in a TrajectoryLogger with detail = "summary".

Usage

```
state_summary_default(entity, ...)
```

Arguments

entity	An Entity object.
...	Reserved for future arguments (ignored).

Details

Implementors may supply a custom summary function with the same signature: function(entity, ...) returning a named list.

Value

A named list of variable values.

time_from_model *Convert numeric model time to calendar time*

Description

Converts numeric model time to Date or POSIXct using a compiled time spec.

Usage

```
time_from_model(t, time_spec, class = c("origin", "Date", "POSIXct"))
```

Arguments

t	Numeric model time.
time_spec	A compiled time spec from time_spec(unit = ...).
class	Output class: 'origin' (match the origin class), 'Date', or 'POSIXct'.

Value

A Date or POSIXct vector.

time_spec *Compile and validate canonical time settings*

Description

Compiles and validates time settings from explicit unit, origin, and zone parameters. v2.0 only supports explicit time_spec() calls; ctx fallback was removed.

Usage

```
time_spec(unit = NULL, origin = NULL, zone = "UTC")
```

Arguments

unit	Required time unit. One of "seconds", "minutes", "hours", "days", "weeks", "months", "years".
origin	Optional Date or POSIXct/POSIXt origin used for calendar-time conversion. Defaults to Unix epoch.
zone	Time zone used for calendar-time conversion (default "UTC").

Value

An object of class time_spec with precomputed conversion constants.

time_to_model *Convert calendar time to numeric model time*

Description

Converts numeric, Date, or POSIXct/POSIXt time to numeric model time under a compiled time spec.

Usage

```
time_to_model(x, time_spec)
```

Arguments

x	A numeric vector, Date, or POSIXct/POSIXt vector of times.
time_spec	A compiled time spec from time_spec(unit = ...).

Value

Numeric model time in units of time_spec$unit.

TrajectoryRecord *Construct a TrajectoryRecord*

Description

Engine-owned audit record emitted at each decision point when a `TrajectoryLogger` is configured. This is the canonical surface for policy evaluation, audit, and RL reward computation.

Usage

```
TrajectoryRecord(
  run_id,
  entity_id,
  t,
  decision_point_id,
  observation,
  realized_event,
  candidate_actions = NULL,
  proposed_actions = NULL,
  selected_action = NULL,
  state_before = NULL,
  state_after = NULL,
  condition_met = NULL,
  reward = NULL,
```

```
  grouped_decision_point_id = NULL,
  group_activation_id = NULL,
  decision_plan_metadata = NULL
)
```

Arguments

run_id	Character scalar; run identifier (from SimContext).
entity_id	Character scalar; entity identifier.
t	Numeric scalar; time at which the decision point fired.
decision_point_id	Character scalar; which decision point produced this record.
observation	Named list; output of the decision point's observation_fn.
realized_event	The event record that triggered the decision point.
candidate_actions	Optional character vector of actions presented to the policy.
proposed_actions	Optional list of actions proposed by the policy.
selected_action	Optional ActionEvent; the action selected by the policy for this decision, recorded before pending-action resolution. It does not by itself establish that the action was retained or realized.
state_before	Optional named list; entity state before the event. NULL unless the TrajectoryLogger is configured to capture it.
state_after	Optional named list; entity state after the transition. NULL unless the TrajectoryLogger is configured to capture it.
condition_met	Logical scalar or NULL; whether the condition predicate was satisfied. TRUE when condition passed or was absent, FALSE for audit records emitted when condition vetoed the policy call.
reward	Optional numeric or named list; reward signal(s).
grouped_decision_point_id	Optional non-empty character scalar naming the grouped decision point that activated this leaf. Must be supplied with group_activation_id.
group_activation_id	Optional non-empty character scalar identifying one grouped firing within a run. Must be supplied with grouped_decision_point_id.
decision_plan_metadata	Optional named list copied from the accepted DecisionPlan() for grouped audit only. It requires both grouped identity fields, remains available only on raw records, and is never an execution input.

Details

state_before and state_after are NULL by default. Supply a summary_fn to the TrajectoryLogger to enable summary-level or full capture.

Consult the entity's event history to determine which selected actions actually realized. For example, a later policy selection can be discarded when its decision point uses `on_pending_action = "keep"` and an earlier action is still pending.

Grouped leaf records carry the static group id and one deterministic, run-local activation id shared by every emitted leaf row from that firing. Ordinary records leave both fields `NULL`. These fields correlate the grouped consultation; they do not provide proposal-to-realization lineage. `decision_plan_metadata` is copied unchanged for raw audit and is intentionally omitted from `trajectory_table()`.

Value

A list of class `"TrajectoryRecord"`.

`trajectory_table` *Compile trajectory records into a data frame*

Description

Takes the list of trajectory records from an engine run and returns a tidy data frame with one row per emitted leaf decision record. For grouped rows, `grouped_decision_point_id` and `group_activation_id` identify the shared consultation; they do not assert that a selected action later realized. Arbitrary `decision_plan_metadata` remains on raw records only.

Usage

```
trajectory_table(records, vars = NULL)
```

Arguments

`records`	List of trajectory records (from `engine$run(...)$trajectory_records`).
`vars`	Character vector of state variable names to extract from `state_before` and `state_after`. If `NULL` (default), all variables in `state_before` are included.

Value

A data.frame with columns: `run_id`, `entity_id`, `t`, `decision_point_id`, `grouped_decision_point_id`, `group_activation_id`, `trigger_event`, `selected_action`, `condition_met`, plus `<var>_before` and `<var>_after` for each requested variable. Grouped ids are `NA` for ordinary or legacy records. Opaque decision-plan metadata remains available only in raw trajectory records.

update_block *Block-oriented state updates for vectorized models*

Description

Many models generate multivariate outputs (e.g., SBP/DBP or lab panels). update_block() validates a named payload against an entity's schema and a named schema block, then returns a named list of per-variable state updates for use as a transition() return value.

Usage

```
update_block(
  entity,
  block,
  values,
  require_all = TRUE,
  unknown = c("error", "drop_warn_once", "drop_warn_always")
)
```

Arguments

entity	A Entity object.
block	Block name (character scalar) corresponding to schema$blocks.
values	Named vector or named list of values to update. Names must correspond to state variables in the entity's schema.
require_all	Logical; if TRUE (default) all variables belonging to block must be supplied in values.
unknown	Policy for variables supplied in values that are not present in the entity's schema. One of "error" (default), "drop_warn_once", or "drop_warn_always".

Value

A named list of state updates suitable for returning from transition(). Values are returned in schema block order.

Index