Plugins

Eleanor exposes four extension points that share a single plugin model: a generic in-process registry, lazy entry-point discovery, and a common collision/override policy. The registry primitive lives at `eleanor.plugin.PluginRegistry`; each extension point instantiates it with its own spec shape and built-in name set.

Extension points

Kind	Entry-point group	Spec type
Executor	`eleanor.executors`	`ConfigurablePluginSpec`
Kernel	`eleanor.kernels`	`ConfigurablePluginSpec`
Navigator	`eleanor.navigators`	`SimplePluginSpec`
Output sink	`eleanor.outputs`	`ConfigurablePluginSpec`

Every entry point must resolve to a `SimplePluginSpec` or `ConfigurablePluginSpec` instance (from `eleanor.plugin`). Bare callables are not accepted.

Names are short strings (e.g. `random`, `eq36`, `dask`). Built-in names are reserved and cannot be overridden through the registry; see Collision policy below.

Configuration shape

Kernel, navigator, and output sink selection all use the same flat `kind`-keyed shape. The `kind` key selects the plugin; all remaining keys are forwarded to the plugin's `parse_settings` as a flat dict:

```yaml
kernel:
  kind: eq36
  model: b-dot
  charge_balance: H+

navigator:
  kind: random

output:
  kind: postgres
  database:
    host: localhost
    port: 5432
    database: eleanor
```

The short-string form `navigator: random` is accepted as sugar for `{kind: random}`. ### Navigator contract Navigator plugins must expose:

- `num_systems(order: Order, scale: int) -> int`: total number of VS points the navigator will produce for the given scale.
- `navigate(order: Order, kernel: AbstractKernel, scale: int, batch_size: int, *args, order_id=None, **kwargs) -> Iterator[list[vs.Point]]`: a batch iterator that yields lists of up to `batch_size` points.

Eleanor passes `order_id` and `max_attempts` as keyword arguments to `navigate`; plugins should accept `**kwargs` to tolerate future additions.

Across a complete `navigate(...)` iteration, the total number of yielded points must match `num_systems(order, scale)`. Eleanor raises an error if the counts diverge.

Distributing a third-party plugin

Third-party plugins register themselves through Python entry points declared in the distribution's `pyproject.toml`:

```
[project.entry-points."eleanor.navigators"]
my_nav = "my_project.navigators:my_nav_spec"

[project.entry-points."eleanor.kernels"]
my_kernel = "my_project.kernels:my_kernel_spec"

[project.entry-points."eleanor.outputs"]
my_sink = "my_project.outputs:my_sink_spec"
```

Discovery runs lazily on the first call to `available_*()` or `get_*()` for the relevant extension point. Any entry-point load or validation failure is a hard error — a broken plugin fails fast rather than silently disappearing from the available set.

End-to-end example: third-party output sink package

Minimal package layout:

```
my_sink_plugin/
  pyproject.toml
  src/my_sink_plugin/output.py
```

`pyproject.toml`:

```
[project]
name = "my-sink-plugin"
version = "0.1.0"
dependencies = ["eleanor>=0.0.0"]
```

```
[project.entry-points."eleanor.outputs"]
my_sink = "my_sink_plugin.output:my_sink_spec"
```

src/my_sink_plugin/output.py:

```
from eleanor.output.interface import ComputeResult, AbstractOutputSink,
WriteOutcome
from eleanor.output.settings import OutputSinkSettings
from eleanor.plugin import ConfigurablePluginSpec


class MySink(AbstractOutputSink):
    def begin_run(self, order):
        return order.id or 0

   def write_batch(self, order_id, results: list[ComputeResult], progress=None)
-> list[WriteOutcome]:
        return [WriteOutcome(exit_code=r.point.exit_code, committed=True) for r
in results]

    def finalize_run(self) -> None:
        return None

    def finalize(self) -> None:
        return None


def _parse_settings(raw: dict) -> OutputSinkSettings:
    return OutputSinkSettings.from_dict(raw)


def _build_sink(settings: object) -> MySink:
    return MySink()


my_sink_spec = ConfigurablePluginSpec(
    parse_settings=_parse_settings,
    build=_build_sink,
    plugin_api_version=1,
)
```

Install the package, then reference it in Eleanor config:

```
output:
  kind: my_sink
```

You can verify discovery with:

```
eleanor doctor
```

Inline plugins

Every extension point supports two non-packaged registration paths:

1. **Programmatic name registration.** Call `register_<kind>(name, spec)` from the host script, then reference the name in the order file or CLI:

   ```
   from eleanor.navigator.registry import register_navigator
   from eleanor.plugin import SimplePluginSpec
   from my_script import MyNavigator

   register_navigator("my_nav", SimplePluginSpec(build=MyNavigator))
   ```

2. **Direct instance override.** `Eleanor.run` accepts already-constructed objects that bypass the registry entirely:

   ```
   from eleanor import Eleanor

   Eleanor(config=config).run(
       order,
       simulation_size,
       navigator=my_navigator_instance,
       kernel=my_kernel_instance,
       output_sink=my_output_sink_instance,
   )
   ```

   The override, when supplied, short-circuits the registry lookup for that extension point only. The `executor` override must be supplied at construction time (`Eleanor(executor=my_executor)`), not per-run.

Writing a kernel plugin

Kernel plugins register a `ConfigurablePluginSpec` that bundles two entry points: parsing order-file settings and constructing the kernel at run time.

```
from eleanor.kernel.registry import register_kernel
from eleanor.plugin import ConfigurablePluginSpec
```

```
from my_project.kernel import MyKernel, MySettings

my_kernel_spec = ConfigurablePluginSpec(
    parse_settings=MySettings.from_dict,
    build=lambda settings: MyKernel(settings),
    plugin_api_version=1,
)
register_kernel("my_kernel", my_kernel_spec)
```

Commercial plugins

Entry points are pure metadata, so proprietary plugin wheels shipped through a private index (and optionally Cython-compiled) participate in discovery identically to open-source plugins. The registered factory can live inside a compiled `.so` and perform license verification on first use; Eleanor will surface any failure through the registry's standard warning/error path.

Collision policy

No shadowing is allowed:

- Registering under a built-in name (via `register_<kind>()` or an entry point) is always a hard error.
- Two entry points claiming the same name — whether built-in or not — is a hard error listing both offending entry-point values.
- Programmatic `register_<kind>()` calls that collide with an already-registered name are a hard error.

To replace a built-in at run time without touching the registry, use the direct-instance-override path on `Eleanor.run(kernel=..., ...)` or pass an executor at construction time via `Eleanor(executor=...)`.

The `ELEANOR_<KIND>_OVERRIDES` environment variable only affects API-version checks (downgrading version mismatches to warnings); it has no effect on name collision handling.

Writing a new extension point

If you add a new pluggable concept to Eleanor, instantiate another `PluginRegistry` with the appropriate spec shape and entry-point group. The generic primitive handles discovery, validation, collision resolution, and override semantics, so the new registry module stays small:

```
from eleanor.plugin import PluginRegistry

registry = PluginRegistry(
    kind='thing',
    entry_point_group='eleanor.things',
    override_env_var='ELEANOR_THING_OVERRIDES',
```

```
    builtin_names=frozenset({'default'}),
)

register_thing = registry.register
available_things = registry.available
get_factory = registry.get
```

`builtin_names` reserves names whose specs will arrive via entry-point discovery. In production registries `builtin_names` lists the names declared in `pyproject.toml`.

Then declare the built-in in `pyproject.toml` so discovery can load it:

```
[project.entry-points."eleanor.things"]
default = "eleanor.thing.factories:default_spec"
```