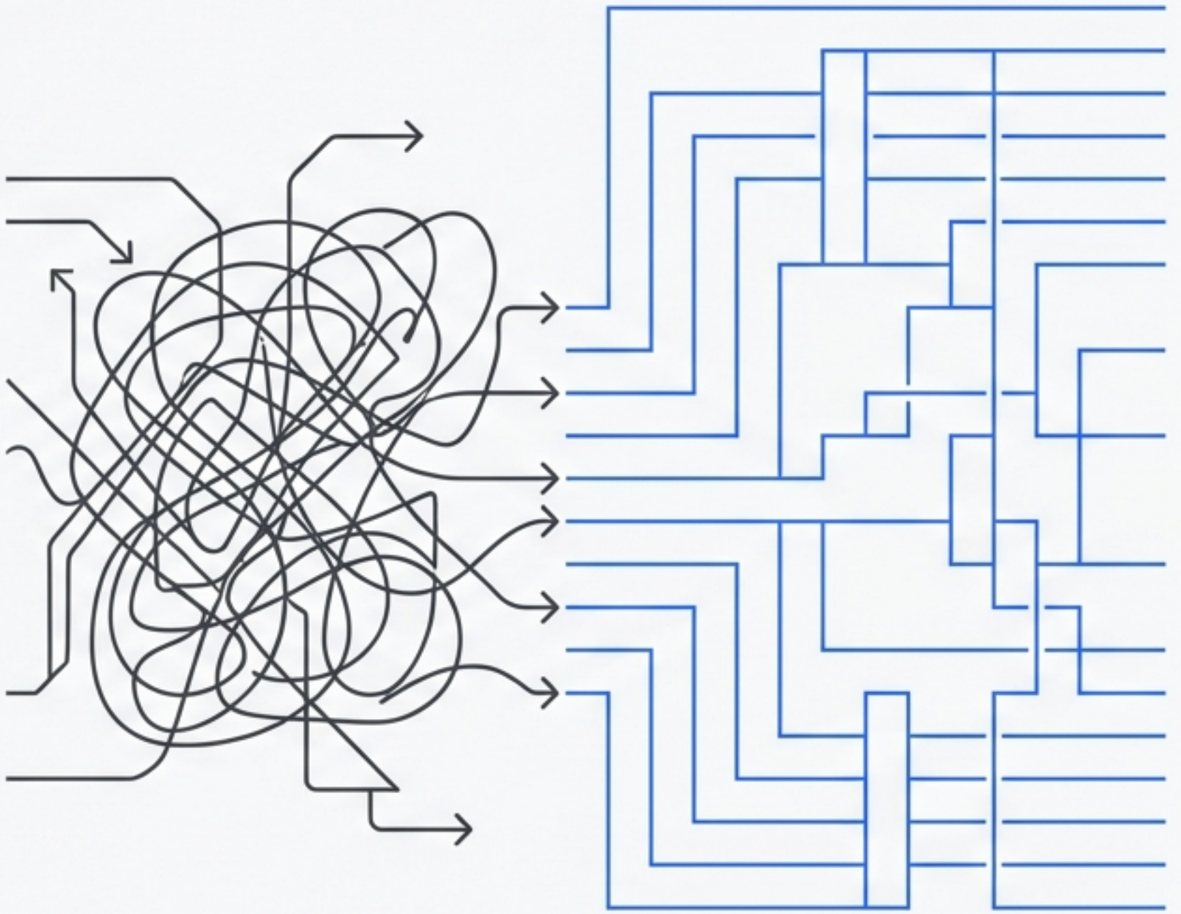
LETS Pipeline Refactoring: The Shift to Flow/Task Architecture

Moving to a Modular, Declarative, and Type-Safe System.



PACKAGE:	mgraph_ai_service_html_graph.service.lets_pipeline
VERSION:	v1.4.44
DATE:	January 2025
DEPENDENCY:	osbot_utils.helpers.flows

Core Objectives of the v1.4.44 Refactor

Reusable Actions

Logic is decoupled into atomic units using @task() decorated functions. No more monolithic methods.

Declarative Steps

Steps are no longer logic containers; they ‘wire’ actions together using simple class attributes.

Type-Safe Injection

Context is managed via a custom task_dependencies() method, ensuring strict typing for config and document objects.

Standardised Execution

All steps inherit a common execute() method from the base class, removing boilerplate error handling and logging.

The Foundation: Updates to osbot_utils

Enabling typed injection in the core Flow/Task system (**backward compatible**).

File: osbot_utils/helpers/flows/Flow.py

```
162         def execute_task(self, task, *args, **kwargs):
163             return self.task_executor.execute_task(task, *args, **kwargs)
164
165 →       def task_dependencies(self): return {}
166
167         def wait_for_tasks(self, tasks):
```

New overridable method for injecting objects

File: osbot_utils/helpers/flows/Task.py

```
 70         def run(self, *args, **kwargs):
 71             if self.execute_task_is_valid():
 72 →               args, kwargs = self.resolve_args_and_kwargs(*args, **kwargs)
 73               return self.execute_task(*args, **kwargs)
```

Modified to consume flow-defined dependencies

Note: Existing flows continue to work unchanged. Only flows overriding task_dependencies() gain new behaviour.

System Architecture Overview

Html_LETS__Flow (The Orchestrator)

Extends core Flow. Manages lifecycle. Holds 'document' state.

Html_LETS__Base (The Logic)

Refactored Base Class. Contains shared LETS operations.

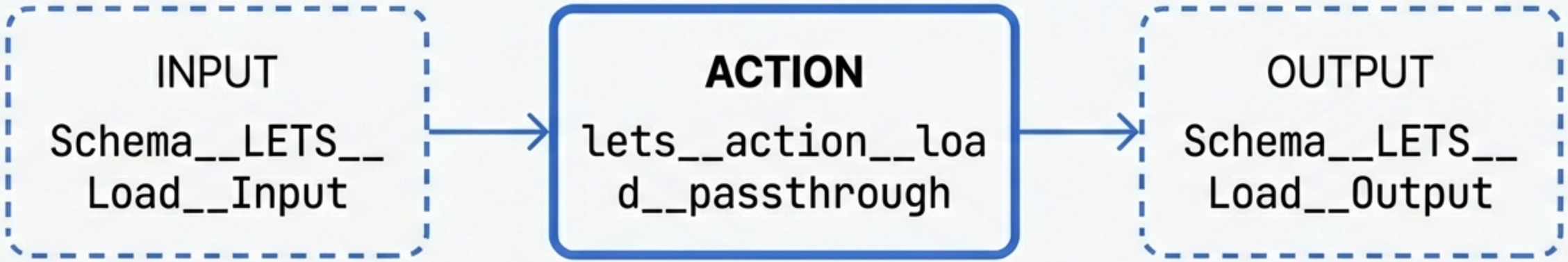
Schema__LETS__Config (The Configuration)

Expanded Schema. Centralises configuration.

The new structure separates orchestration, business logic, and configuration into distinct, testable layers.

The Atomic Unit: The 'Action'

An Action is a single-purpose function decorated with `@task`.
It represents one phase (Load, Extract, Transform, or Save).

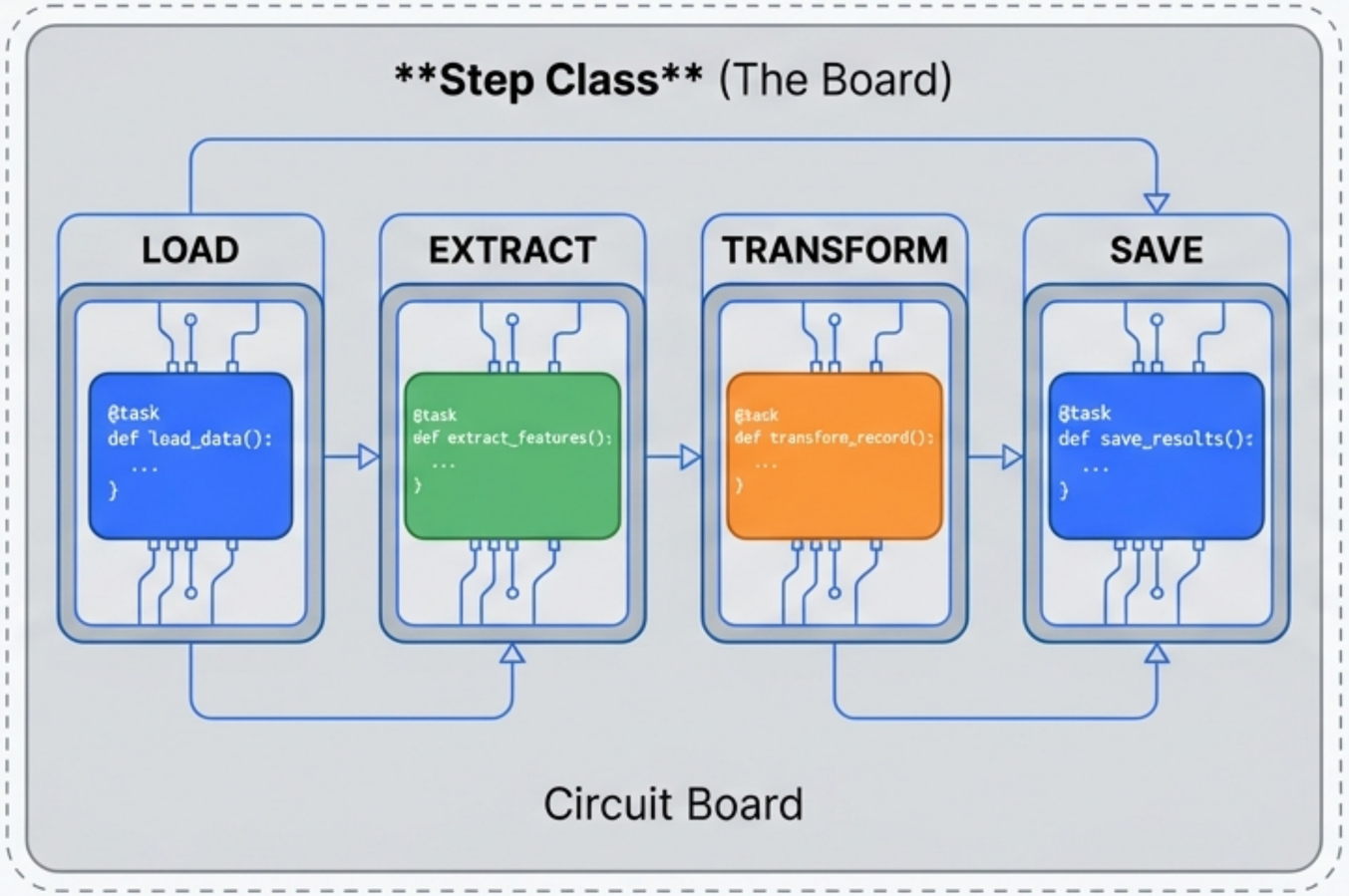


Example Actions:

- `lets__action__extract__passthrough.py`
- `lets__action__transform__compute_stats.py`
- `lets__action__save__to__cache_service.py`

The Structural Unit: The “Step”

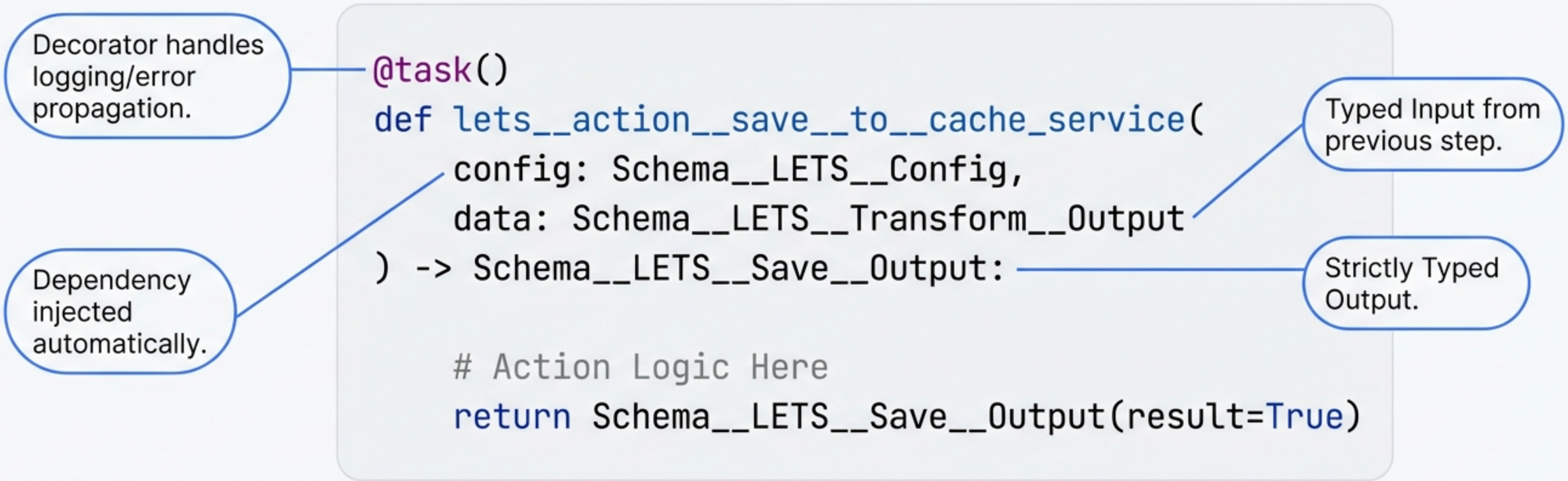
From Imperative Logic to Declarative Wiring



Key Characteristics

- **Declarative:** Steps define *what* happens, not *how*.
- **Wiring:** Uses class attributes to link actions.
- **No Logic:** Directs the Flow without calculation code.

Anatomy of an Action (Code View)



Anatomy of a Step (Code View)

Replacing 50+ lines of imperative logic with 4 lines of configuration.

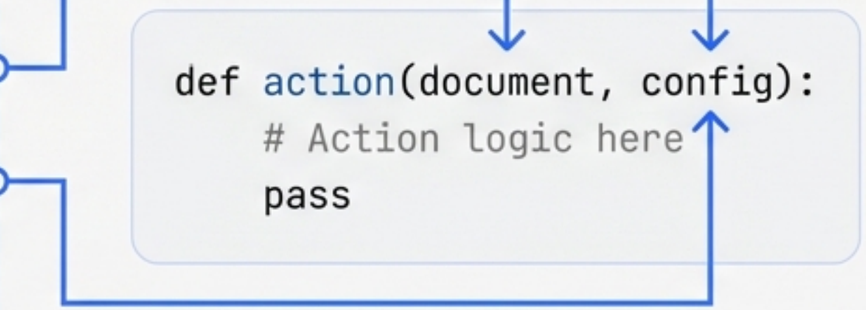
```
class Html_LETS__Html__From__Raw(Html_LETS__Base):

    action_load      = lets__action__load__passthrough
    action_extract   = lets__action__extract__passthrough
    action_transform = lets__action__transform__passthrough
    action_save      = lets__action__save__to__cache_service
```

The execution flow is handled entirely by the base class.

Type-Safe Dependency Injection

Dependencies are injected automatically based on the function signature via `task_dependencies`.

Parameter Name	Type	Source
document	Html_Cache__Document	Html_LETS__Flow.document
config	Schema__LETS__Config	Html_LETS__Flow.config
this_flow	Html_LETS__Flow	Built-in (Task)
task_data	dict	Built-in (Task)



Testing Strategy: Split and Conquer

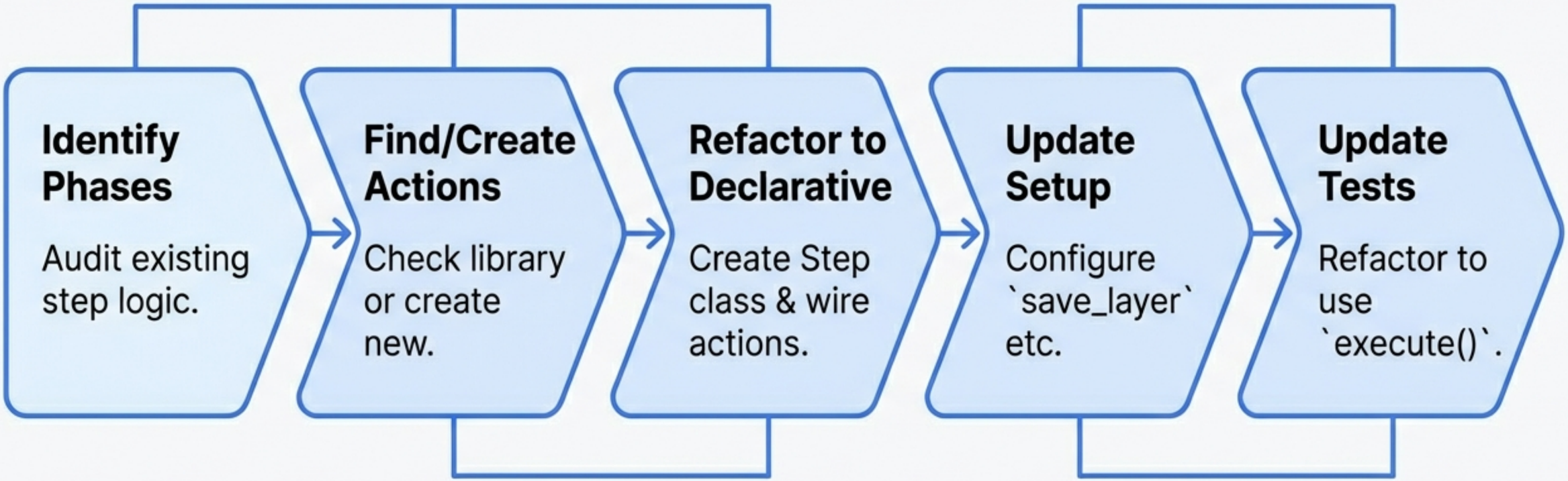
Unit Tests (Actions)

- Focus: Input -> Output
- Method: Test each `@task` function independently.
- Benefit: Verify transformations without spinning up the pipeline.

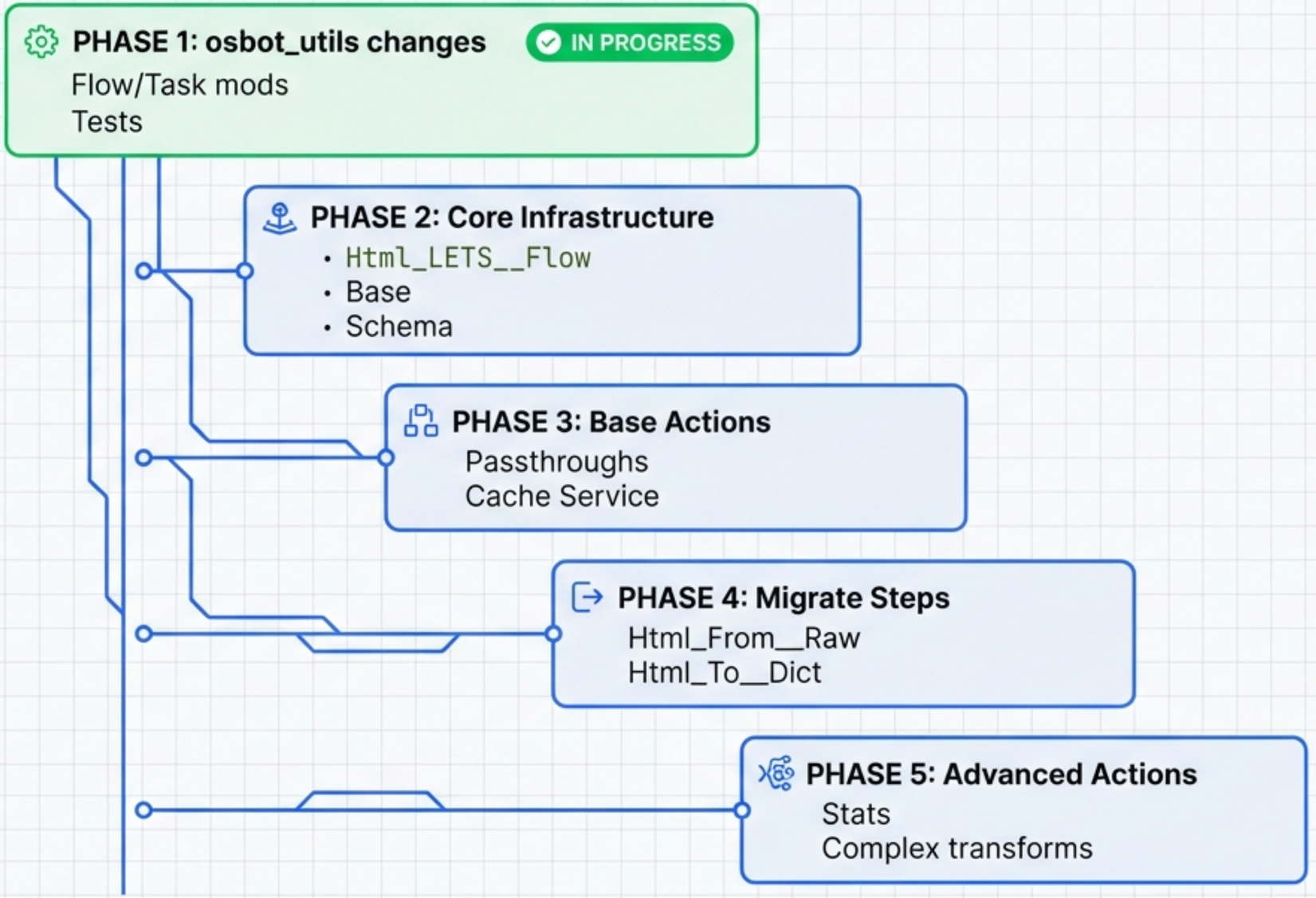
Integration Tests (Steps)

- Focus: Wiring & Execution
- Method: Use the `step.execute(input)` pattern.
- Benefit: Ensure data passes correctly between phases.

Migration Guide for Existing Steps



Implementation Roadmap

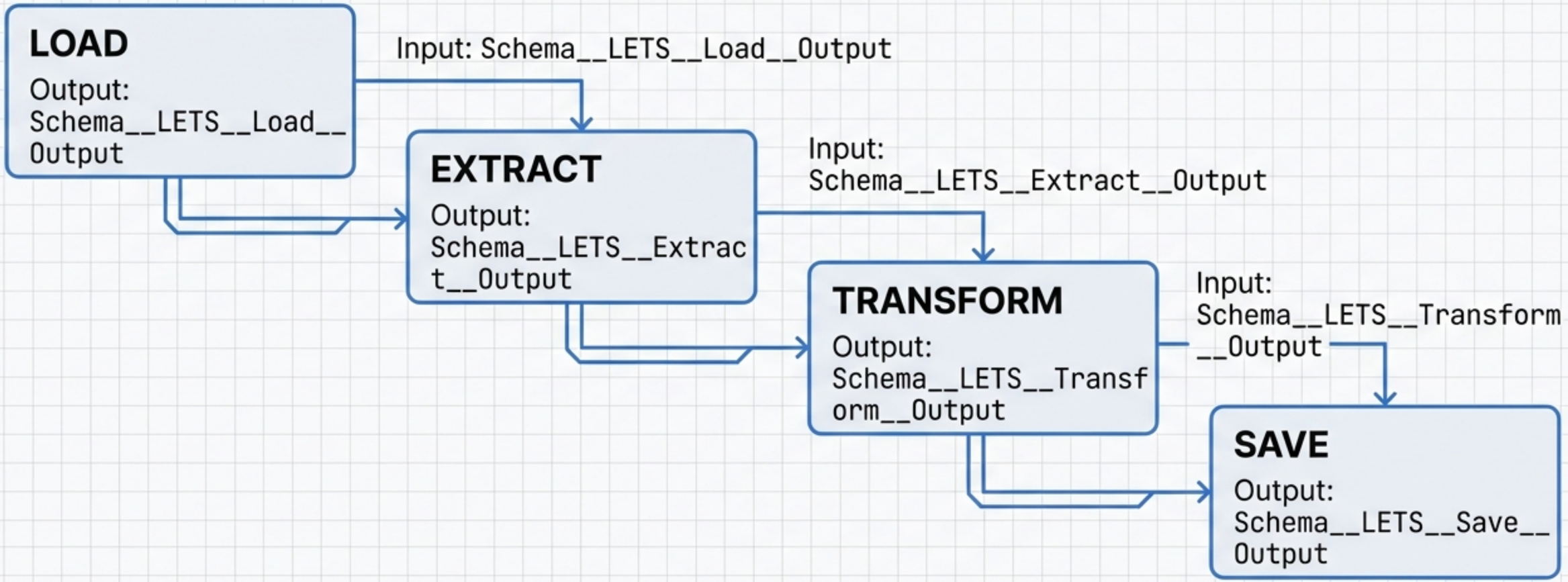


Before vs. After Analysis

Aspect	Before (Imperative)	After (Declarative)
Code Size	50+ lines/step	4 lines (wiring)
Reusability	Copy/Paste	Import & Wire
Observability	Manual Logging	Automatic via Flow
Type Safety	Varies	Enforced via Schema
Error Handling	Manual try/catch	`@task(raise_on_error)`

Reference: Action Phase Schemas

Enforcing Compatibility Across the Pipeline



Immediate Next Steps

Execute Phase 1

- [ ] Verify `osbot_utils` dependency update
- [ ] Run regression tests on `osbot_utils.helpers.flows`
- [ ] Begin scaffolding `Html_LETS__Flow`

Modular, Declarative, Type-Safe.