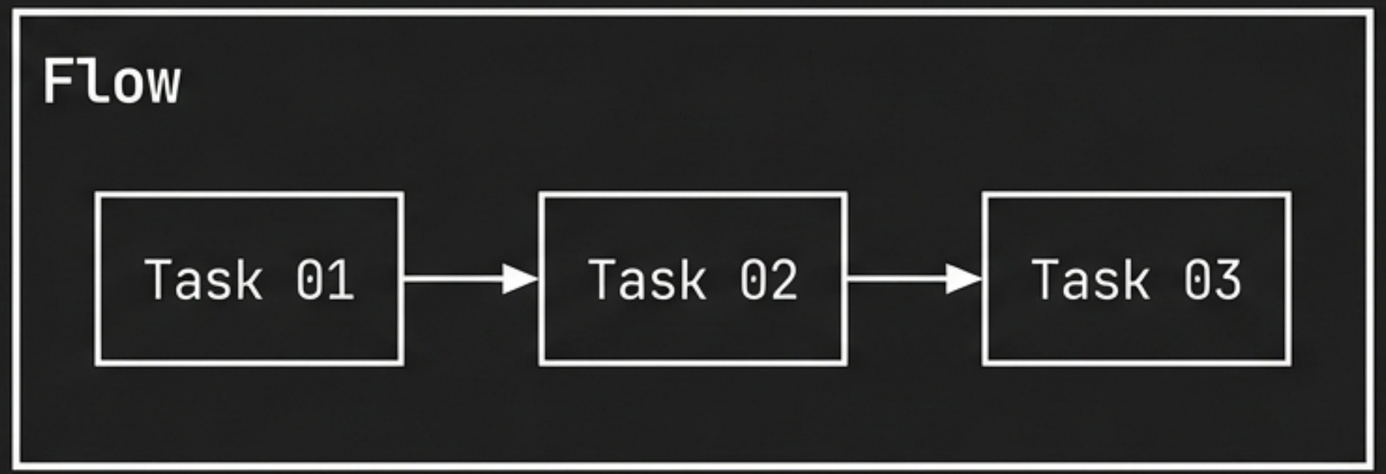
The Flow System

A Lightweight Workflow Orchestration Framework for Python.

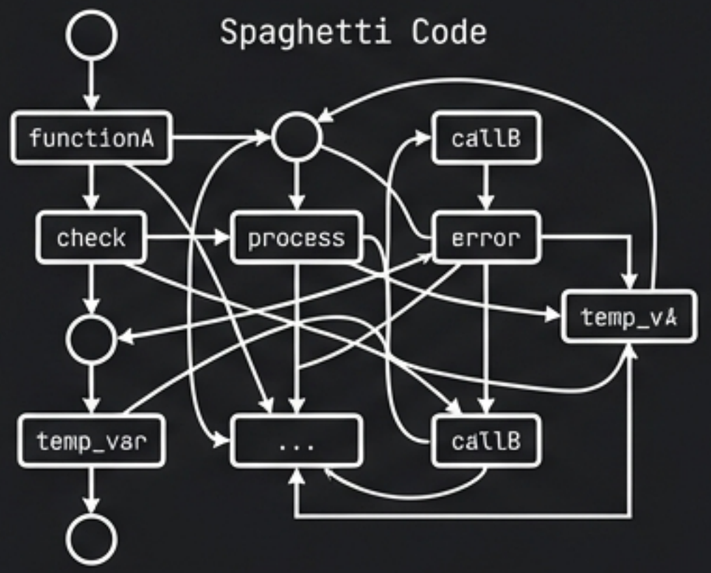
Structure

Observability

Error Handling

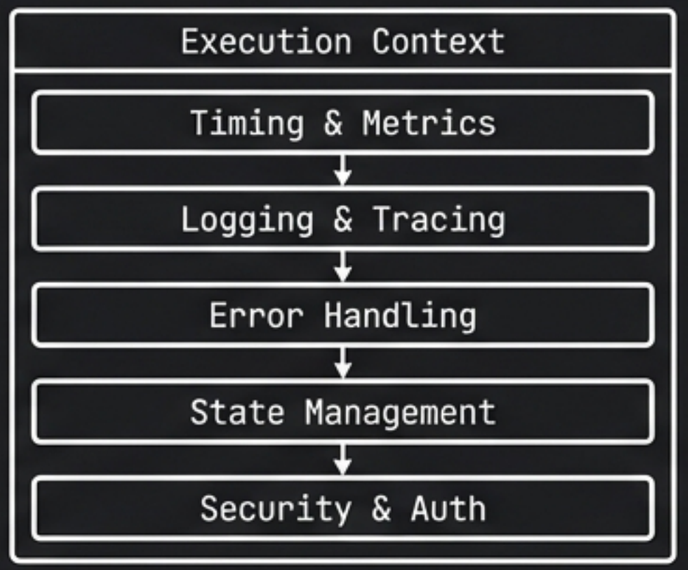


The Problem: Unmanaged State



- Verbose logging calls scattered everywhere
- Try/except blocks cluttering logic
- Undefined data sharing
- Print statement debugging

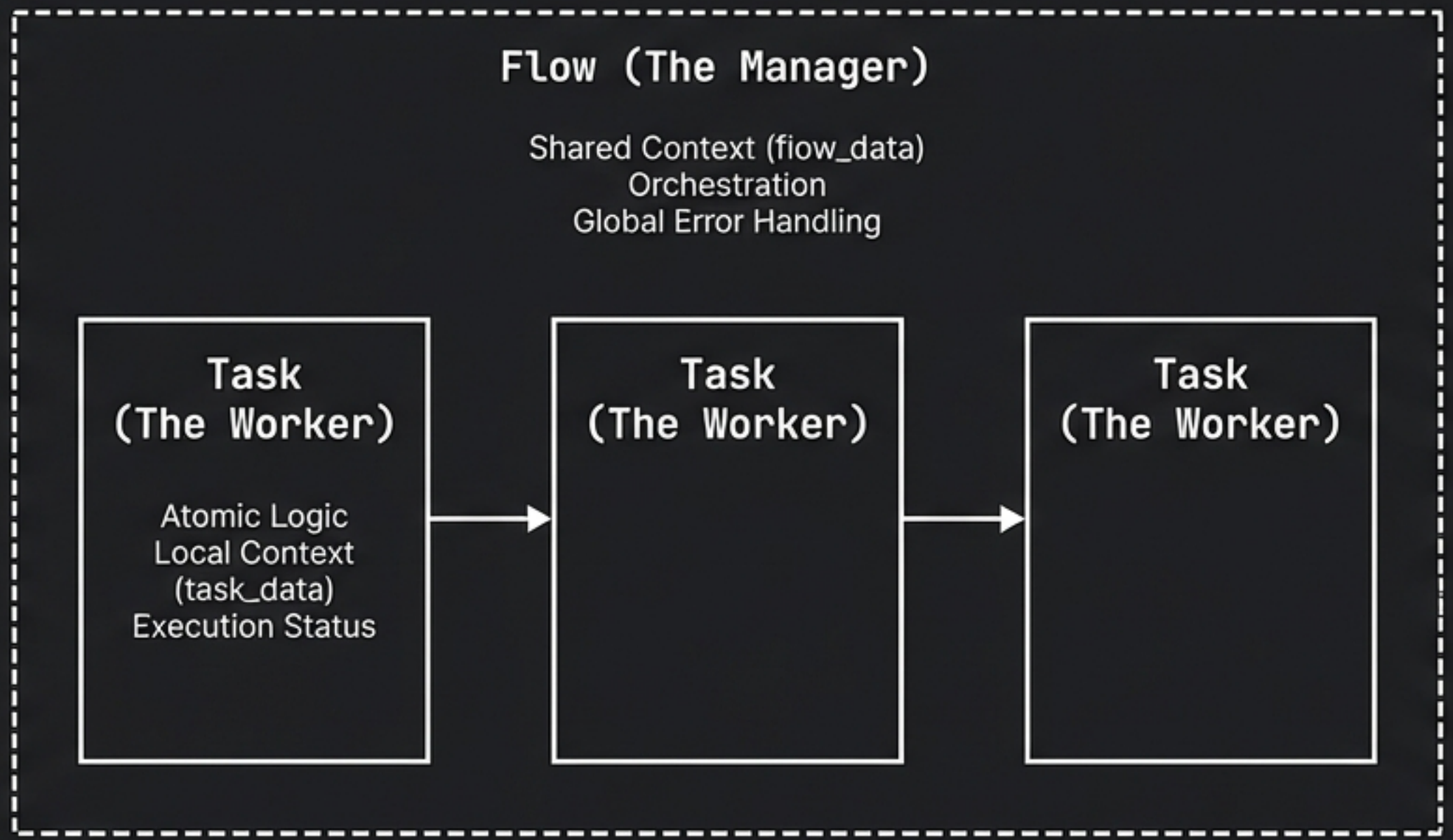
The Solution: Rich Execution Context



- **Zero Infrastructure**: No databases, message queues, or external services.
- **Type-Safe**: Runtime type checking and serialization.
- **Pythonic**: Decorators for simplicity, Classes for complexity.

Separation of Concerns: Keep business logic distinct from execution concerns like **timing**, **logging**, and **error handling**.

Architectural Anatomy: The Flow and The Task



The Flow is the container; the Task is the unit of work.

Feature List:

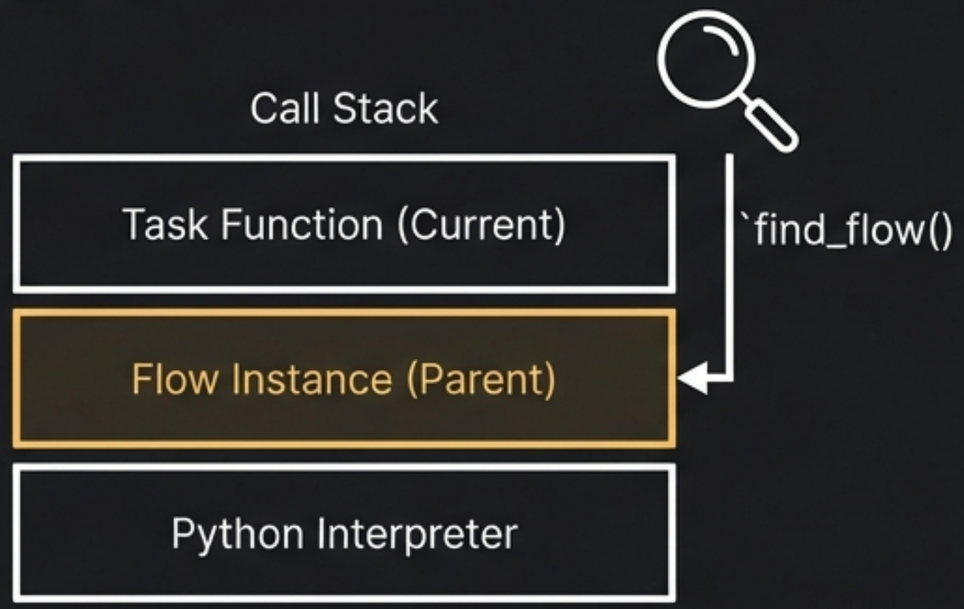
- **Flow Orchestration:** Coordinate multiple tasks with shared context.
- **Error Isolation:** Tasks can fail without crashing the entire Flow.
- **Artifact Tracking:** Store Store data products systematically.

Automatic Context Discovery

Tasks find their parents through Stack Inspection.

Concept

The framework eliminates the need to manually pass state arguments. A Task automatically identifies its execution context.



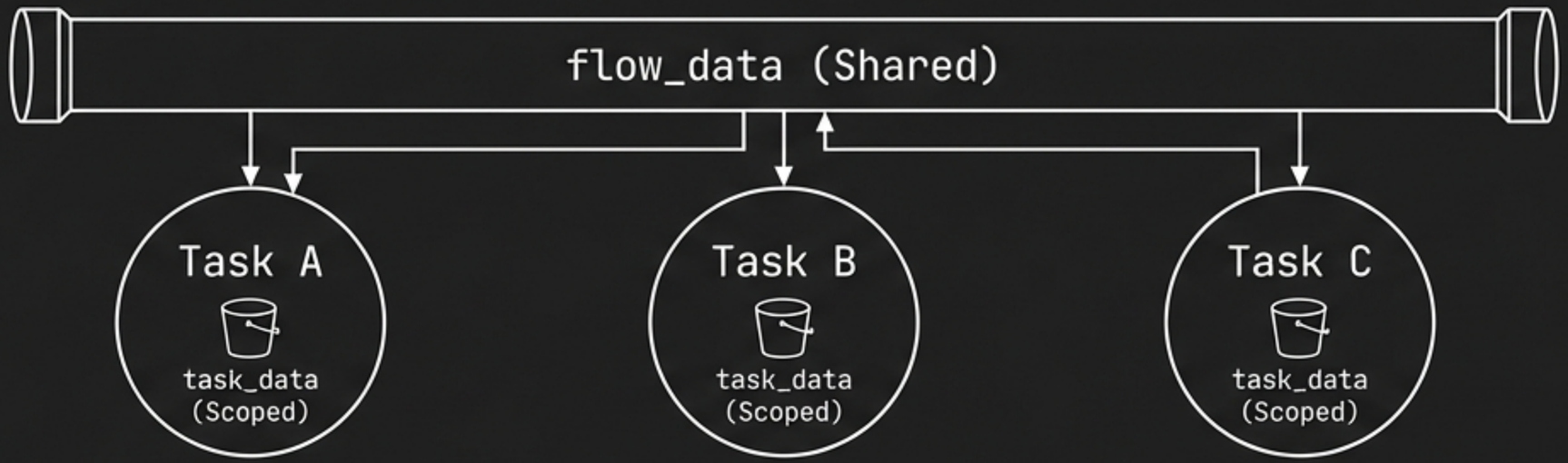
Code

```python
# The mechanism
def find_flow(self):
    # Walks stack frames to locate parent Flow
    for frame in inspect.stack():
        if isinstance(frame.locals.get('self'), Flow):
            return frame.locals['self']
    return None
```

No more passing 'parent' objects as arguments. The Task knows where it belongs.

Dependency Injection & Data Scope

What gets injected into your functions automatically.



	Parameter	Scope	Description
1	`this_flow`	Global	The current Flow instance.
2	`this_task`	Local	The current Task instance (tasks only).
3	`flow_data`	Global	Shared dictionary. Persists for lifecycle of flow.
4	`task_data`	Local	Private dictionary. Scoped to current task.

Implementation Styles

The Decorator Style

Best for: Simple cases, quick scripts.

```
@flow
def my_process(flow_data):
    step_one()

@task
def step_one(task_data):
    print("Doing work")
```

The Class-Based Style

Best for: Complex flows, inheritance.

```
class My_Process(Flow):
    def run(self):
        self.step_one()

    @task
    def step_one(self):
        self.log_info("Doing work")
```

Pro Tip: Use Context Managers for clean execution scopes.

The Execution Lifecycle

CRITICAL: You must call setup() before execute().

`setup(target, *args, **kwargs)` configures the flow. Calling `execute()` without it will raise a 'Setup has not been called' error.

Methods Reference

`setup()`	Configures target and arguments.
`execute()`	Runs the configured flow.
`execute_flow()`	Runs with optional runtime parameters.

Observability: Stop Guessing, Start Knowing

Automatic Logging: All print statements, `log_info`, and `log_error` calls are captured.

Statistics: Built-in timing for every `flow` and `task`.

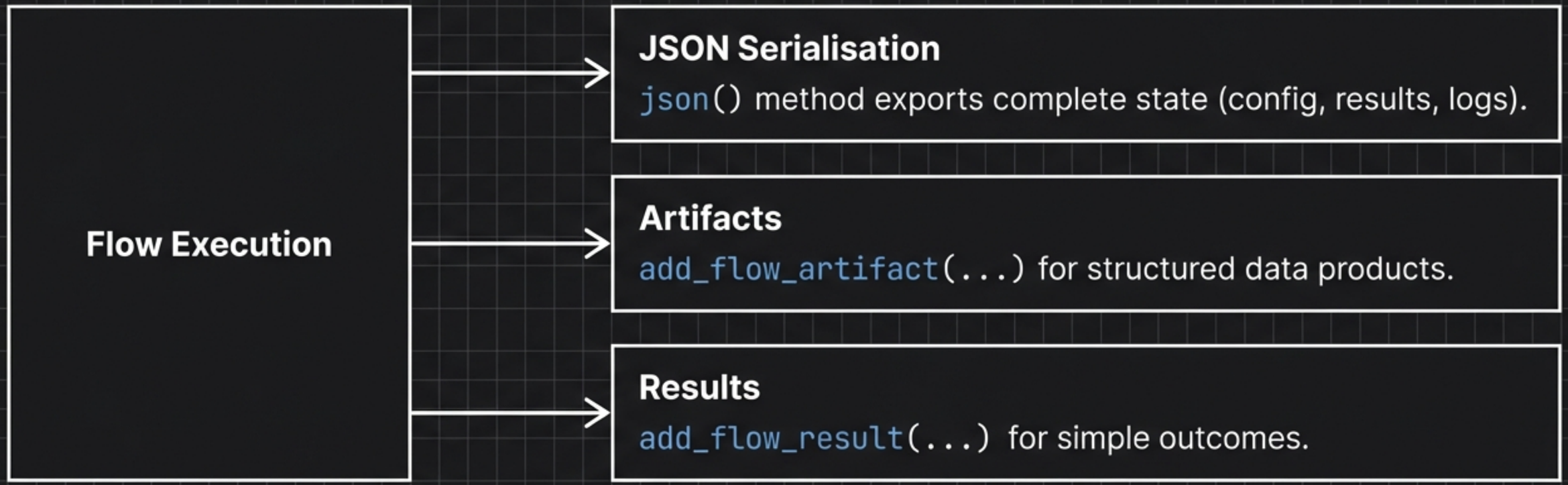
```
>>> flow.durations()
{ "flow_duration": 1.24s,
  "tasks": { "fetch_data": 0.8s, "process_data": 0.4s } }

>>> flow.captured_logs()
[ { "time": "10:00:01", "level": "INFO", "msg": "Starting process" },
  { "time": "10:00:02", "level": "INFO", "msg": "Data fetched successfully" } ]
```

`flow_id` correlation built-in.

Artifacts & Serialisation

Managing outputs and post-mortem data.



Use Case: Ideal for passing execution context to frontend systems or saving run logs for audit.

Resilience & Error Handling

Task Level*: `raise_on_error` (default: `True`). Set to `False` to continue flow on failure.

Flow Level': `flow_error` stores the exception if the flow fails.

The Error Recovery Pattern

“Configure `raise_on_error=False` for non-critical tasks to ensure flow completion.”

Async & Performance

Full support for async/await patterns.

Technical Details

- Use `execute__async()` for asynchronous task execution.
- Async flows execute in a **new event loop**.

Troubleshooting Callout

Problem: Async Flow Not Executing Properly?

Solution: Ensure you are `await`ing async tasks within the async flow function.

Developer's Cheat Sheet

DO

- Always call `setup()` before `execute()`.
- Use `flow_data` for cross-task sharing.

DO

- Register event listeners *before* execution.
- Use `durations()` for performance analysis.

DON'T

- Don't use `task_data` for shared communication.
- Don't forget to clean up event listeners.

DON'T

- Don't execute tasks outside a Flow context.

```
pip install osbot-utils
```

Documentation: `osbot_utils.helpers.flows`