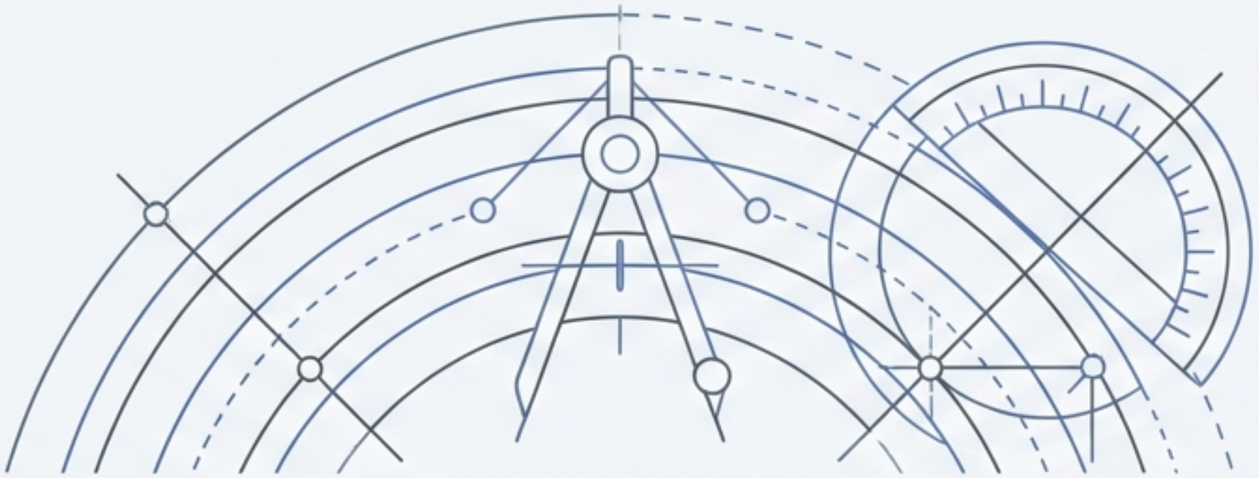
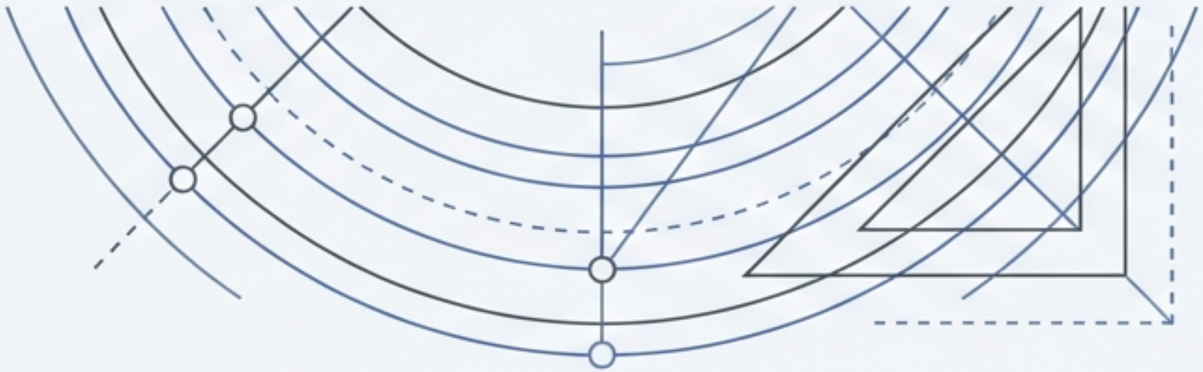


Engineering Performance with Confidence

A Reusable, Type-Safe Benchmarking Framework for `osbot_utils`



Project: Perf_Benchmark__Timing Infrastructure | Version: v3.66.0 | January 2026

The Need for Systematic Performance Measurement

As we perform critical optimisations on `osbot_utils`, particularly around `Type_Safe` object creation, we require a robust framework to move beyond ad-hoc timings. We need data, not guesses.

The Engineering Challenge

- Inconsistent measurement methods
- Disorganised and raw result data
- Manual before/after comparisons
- Unstructured performance experiments
- Lack of historical performance tracking
- Excessive boilerplate in test files
- Risk of runtime type errors

The Framework's Solution

- Leverage `Perf` class with Fibonacci sampling
- Automatic section/index extraction from IDs
- Built-in result comparison and diffing
- Structured hypothesis testing (baseline vs. optimised)
- Load and compare multiple sessions from disk
- Base `TestCase` handles all infrastructure
- Pure `Type_Safe` schemas for all components

Our Solution: A Reusable Benchmarking Toolkit

We are building a clean, `Type_Safe` framework to make performance measurement a seamless part of the development lifecycle.

Write Benchmarks Easily

Minimal boilerplate for rapid test creation.

Collect & Organise Results

Automatic section and index extraction for structured data.

Generate Rich Reports

Create text, JSON, markdown, and HTML outputs from any session.

Compare Between Runs

Natively diff results to see the impact of optimisations.

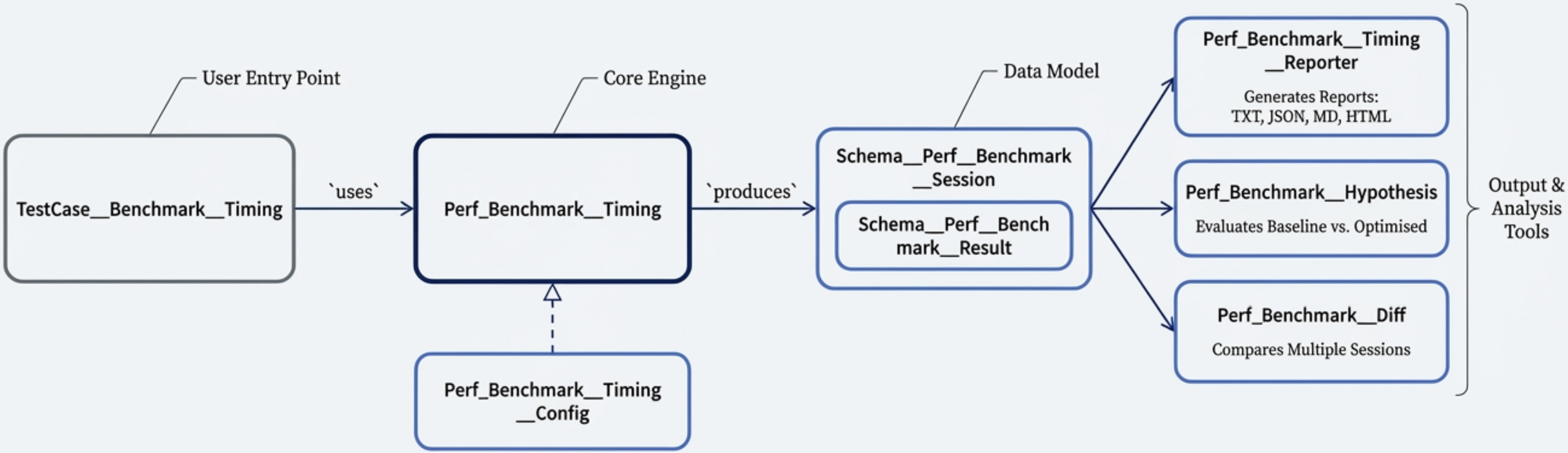
Test Performance Hypotheses

Formally validate improvements against a baseline.

Track Evolution Over Time

Analyse performance trends across multiple benchmark sessions.

The System Architecture at a Glance



Data Flow and Workflow Lifecycle

Workflow 1: Standard Benchmark Run

1. `TestCase__Benchmark__Timing` initialises `Perf_Benchmark__Timing`
2. User calls `benchmark()` method (or `start()`/`stop()`) within a test.
3. `Perf_Benchmark__Timing` executes the code block, using the underlying `Perf` class for timing.
4. A `Schema__Perf__Benchmark__Result` is created and stored.
5. On completion, a `Schema__Perf__Benchmark__Session` object is assembled.
6. The Session is passed to `Reporter` to generate output files (e.g., `results.json`, `report.txt`).

Workflow 2: Hypothesis Test

1. User wraps code in `Perf_Benchmark__Hypothesis`.
2. The `run()` method executes the baseline code block first, capturing a session.
3. It then executes the optimised code block, capturing a second session.
4. The two sessions are compared, and a `Schema__Perf__Hypothesis__Result` is generated.

The Foundation: A Pure and Type-Safe Data Model

Every piece of data in the framework is defined by a strict, pure `Type_Safe` schema. This eliminates ambiguity and ensures data integrity from measurement to serialisation. We use **no raw** `str`, `int`, or `dict`.

Custom Primitives

`Safe_Str__Benchmark_Id`
Full identifier, e.g., "Section A__001".

`Safe_Str__Benchmark__Section`
The "Section A" part.

`Safe_Str__Benchmark__Index`
The "001" part.

Core Schemas

`Schema__Perf__Benchmark__Result`
Holds data for a single timing result.

`Schema__Perf__Benchmark__Session`
The complete, serialisable session object. Contains a `Dict__Benchmark_Results` collection.

`Schema__Perf__Hypothesis__Result`
The structured outcome of a hypothesis test.

Enums & Collections

`Enum__Time_Unit`
Controls time display units (ns, µs, ms, s).

`Enum__Hypothesis__Status`
Defines outcomes (e.g., `IMPROVED` in `#28A745`, `REGRESSED` in `#DC3545`).

`Dict__Benchmark__Legend`
Maps sections to descriptions.

`List__Benchmark_Sessions`
A validated list for multi-session diffing.

From Raw Timings to Actionable Insights

The framework transforms raw performance data into clear, comparable reports in multiple formats.

Text Output (via `Print_Table`)

Section	Index	Time (ms)
Type_Creation	001	1.452
Type_Creation	002	1.449
Data_Access	001	0.023

Hypothesis Result

Hypothesis: 'Optimisation X improves Y'

Status: IMPROVED

Baseline: 1.2ms | Optimised: 0.8ms (Difference: -33%)

Comparison Output

Section	Index	Run_A (ms)	Run_B (ms)	Diff (ms)
Render	001	12.3	12.1	-0.2
Render	002	8.7	9.8	+1.1

Writing a Benchmark: Clean, Simple, and Effective

The `TestCase__Benchmark__Timing` base class handles all setup, teardown, and reporting logic, allowing you to focus purely on the code you need to measure.

```python
class Test_My_Feature(TestCase__Benchmark__Timing):
    def test_optimised_function(self):
        # self.benchmark is provided by the base class
        with self.benchmark("My Section__001"):
            my_optimised_function()

    def test_another_case(self):
        self.benchmark.start("My Section__002")
        some_other_code()
        self.benchmark.stop("My Section__002")
```

Inherit to get all functionality.

Context manager for simple, clean timing.

Manual control for complex scenarios.

Benchmark ID with automatic section/index parsing.

Advanced Workflows for Deeper Analysis

Go beyond simple timing to validate specific optimisations and track performance evolution across releases.

Validate Your Optimisations

A structured way to prove a performance change against a baseline, with enforceable assertions.

```
hypothesis = Perf_Benchmark__Hypothesis(self.benchmark)
hypothesis.baseline = lambda: old_code()
hypothesis.optimised = lambda: new_code()
result = hypothesis.run()
self.assert_improved(result)
```

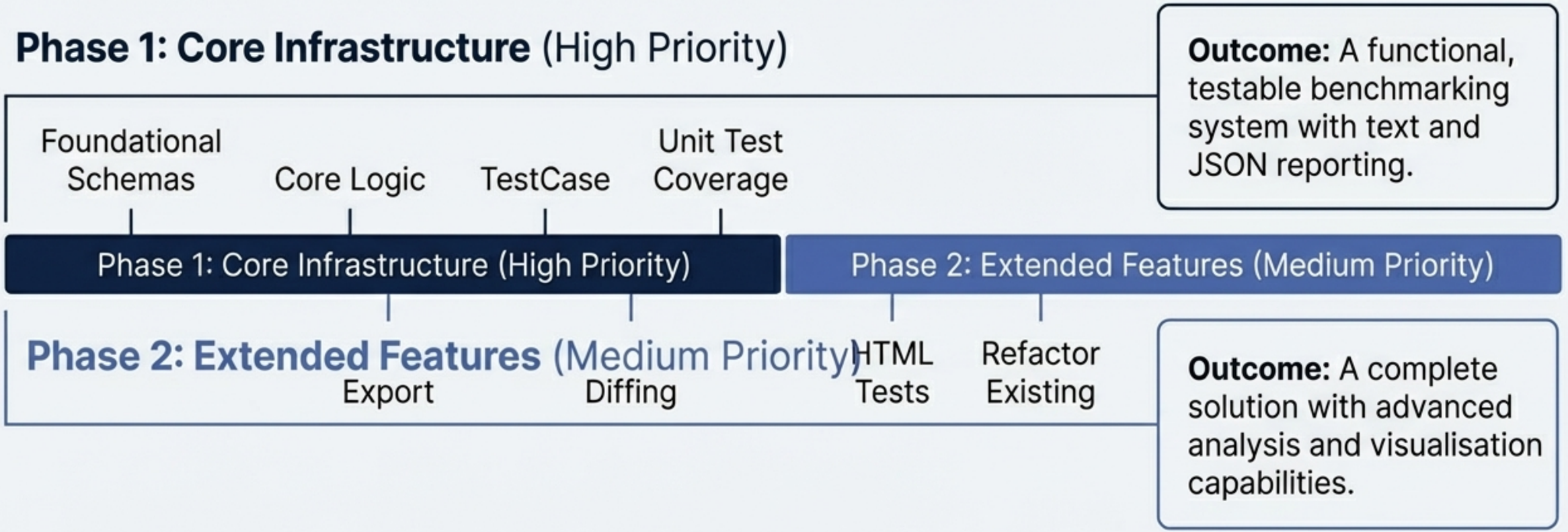
Track Performance Over Time

Load an entire history of benchmark runs from disk to identify trends and regressions.

```
diff = Perf_Benchmark__Diff()
# Loads all *.json session files from a folder
diff.load_sessions_from_folder(self.path_reports)
diff.print_results()
```

The Phased Implementation Roadmap

The implementation is broken down into two logical phases, prioritising the delivery of core functionality first, followed by extended features.



The Blueprint for Quality: Our Implementation Checklist

Success for this project is defined by adherence to these key architectural and quality principles.

Core Infrastructure

- All schemas are pure data, inheriting from `Type_Safe`.
- Strict, type-safe collections are used throughout (no raw dicts).
- `Perf_Benchmark__Timing` supports both context manager and `start()` / `stop()` patterns.
- `TestCase__Benchmark__Timing` provides a seamless test integration point.
- `auto_save_on_completion` provides fire-and-forget persistence.
- Code follows established Python formatting guides.

Extended Features & Data Integrity

- All reports (TXT, JSON, MD, HTML) can be losslessly reconstructed from the saved JSON files.
- `Perf_Benchmark__Hypothesis` enforces a baseline-first execution pattern.
- `Perf_Benchmark__Diff` capably loads and analyses entire folders of session files.
- HTML reports include interactive Chart.js visualisations.