# From Raw Timings to Actionable Insights

A Guide to Robust Benchmarking with `Perf_Benchmark`

`osbot-utils` v3.61.0 | A type-safe benchmarking framework

# The Familiar Chaos of Performance Testing

## The Old Way

```
LOG: Running test...
"Function X took: 0.0012345s"
    test_2_timing: 0.0021
>> BENCHMARK 'Y': 0.00098s
Timing for test_Z... 0.0015
```

## Key Pain Points

**Fragile:** String format changes break parsers.

**Ambiguous:** No clear success/failure/error state.

**Disconnected:** No built-in way to compare results over time.

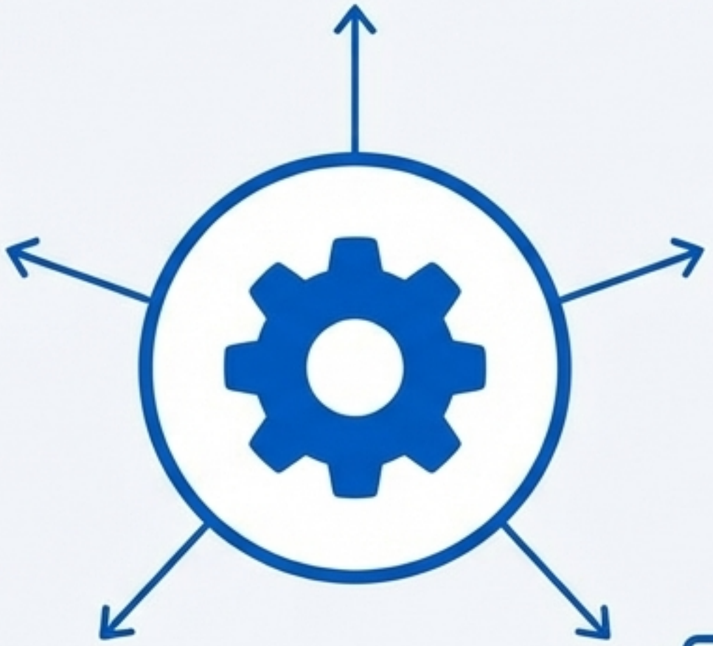
**Manual:** Reports are hand-crafted and inconsistent.

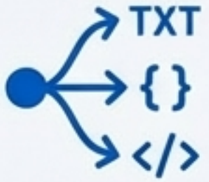
Raw timing measurements are a start, but they leave you swimming in a sea of disconnected data. Real-world benchmarking demands structure, context, and comparability.

# The Five Pillars of Principled Benchmarking

**Proven Foundation:** Built on the stable and reliable `Performance_Measure__Session` infrastructure.

**Schema-Driven:** All results are structured `Type_Safe` schemas, not formatted strings.






**Flexible Outputs:** A single data source generates multiple export formats: Text, HTML, JSON.

**Separation of Concerns:** Calculations are independent of presentation. Get the data first, then decide how to display it.

`Perf_Benchmark`

**Type-Safe State:** Status enums provide clear, unambiguous error states.

# Get Started in Three Steps

## 1 Run & Save

```
with Perf_Benchmark__Timing()
as timing:
    timing.benchmark('A_01__
dict_create', create_dict)

timing.reporter().save_json()
```

Execute your benchmarks within a context manager and save the full session data as a JSON file.

## 2 Compare Sessions

```
diff = Perf_Benchmark__Diff()
diff.load_folder('path/to/sess
ions')
comparison = diff.compare_two(
-2, -1)  # Compare last two
```

Load one or more saved sessions and run a comparison to analyse performance changes.

## 3 Export Results

```
exporter = Perf_Benchmark__Exp
ort__Text()

print(exporter.export_comparis
on(comparison))
```

Use a dedicated exporter to generate a clean, human-readable report from the comparison schema.

# The Core Shift: From Brittle Strings to Robust Schemas

## Before: Formatted Strings

`Returns: str`

```
# old_result = "Benchmark 'create_dict': 0.00123s"
```

❌ Prone to breaking on format changes.

❌ Requires manual parsing to extract data.

❌ Lacks metadata and error state.

## After: Type-Safe Schemas

`Returns: Schema__Perf__Benchmark__Result`

```
# new_result = Schema__Perf__Benchmark__Result(
#     id='A_01__dict_create',
#     duration=1.23e-06,
#     ...
# )
```

✅ Structurally guaranteed by `Type_Safe`.

✅ Data is directly accessible: `result.duration`.

✅ Rich with context and ready for comparison.

# Building Blocks for Reliable Insights

## Clear State with Status Enums

Every comparison returns a status, preventing silent failures. You always know if the result is valid before using it.

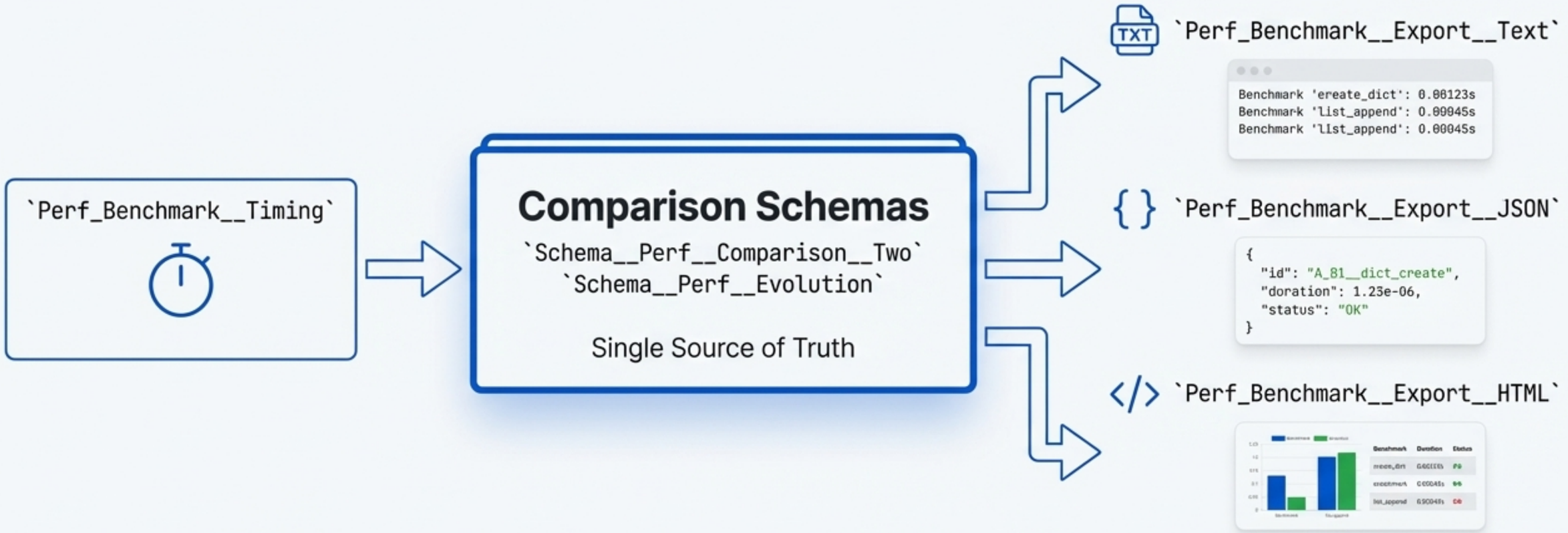
```
result = diff.compare_two()
if result.status == Enum__Comparison__Status.OK:
    # Process the data with confidence
    print(exporter.export_comparison(result))
else:
    # Handle the error gracefully
    print(f"Error: {result.error}")
```

## Instant Insight with Trend Detection

Performance changes between sessions are automatically classified into five distinct levels.

| Trend | Symbol | Change |
|---|---|---|
| STRONG_IMPROVEMENT | ▼▼▼ | > 10% faster |
| IMPROVEMENT | ▼ | 0-10% faster |
| UNCHANGED | — | 0% change |
| REGRESSION | ▲ | 0-10% slower |
| STRONG_REGRESSION | ▲▲▲ | > 10% slower |

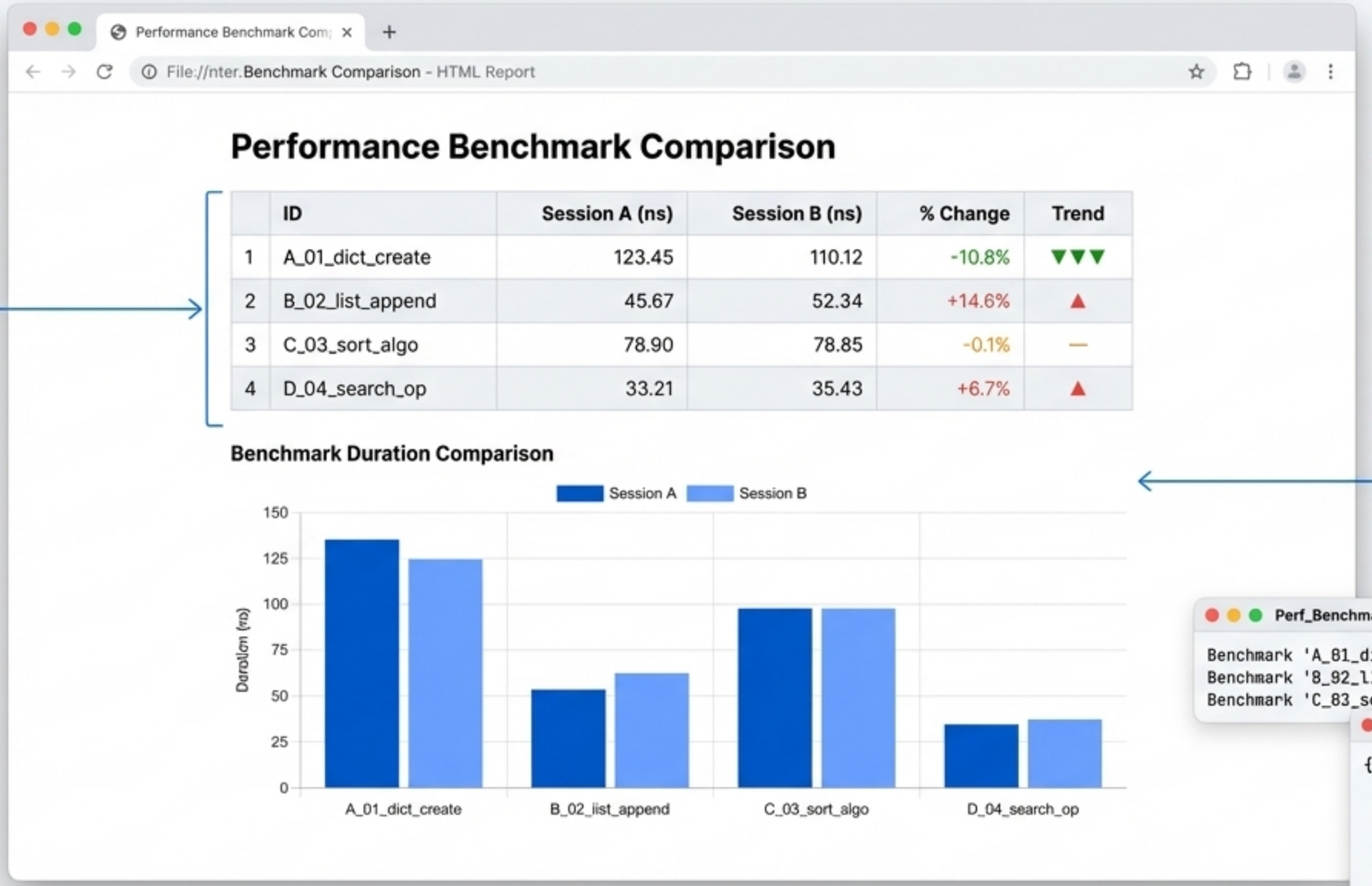
# The Architecture of Clarity: Calculate Once, Report Anywhere




The core schemas contain all the calculated data. Exporters are simply different "views" of that same data, ensuring consistency across all reports.

# From Data to Dashboard: The Power of Exporters

Clear, colour-coded trend analysis at a glance.




Rich data visualisation powered by Chart.js, built-in.

Perf_Benchmark__Export__Text

```
Benchmark 'A_01_dict_create': 118.12ns
Benchmark 'B_02_list_append': 52.34ns
Benchmark 'C_03_sort_algo': 78.85ns
```

Perf_Benchmark__Export__JSON

```
{
  "id": "A_01_dict_create",
  "duration_a": 123.45,
  "duration_b": 118.12,
  "change_percent": -10.8,
  "trend": "STRONG_IMPROVEMENT"
}
```

# Your Performance Playbook: Common Usage Patterns

## CI/CD Guardrails

Fail your builds on significant performance regressions.

```
timing.benchmark(..., assert_less_than=time_10_kns)
```

ⓘ Integrates perfectly with `Perf_Benchmark__Hypothesis` for robust testing.

## Historical Evolution Tracking

Analyse performance trends across an entire history of runs.

```
evolution = diff.compare_all()
```

## Two-Point Comparison

Compare any two specific sessions, like a feature branch vs. main.

```
comparison = diff.compare_two(session_a_index,
session_b_index)
```

## Summary Statistics

Get aggregate statistics (min, max, mean, stddev) across all sessions.

```
stats = diff.statistics()
```

# Best Practices for Building with Confidence

## DO

- **Use Consistent Benchmark IDs**: Essential for tracking the same test over time. (`A_01__dict_create`)
- **Save Every Session**: Build a rich history for meaningful `compare_all()` analysis. (`reporter().save_json()`)
- **Check Status Before Processing**: Always verify `result.status` is `OK` to avoid errors on invalid data.
- **Define Time Thresholds as Constants**: Keep your assertions clean and maintainable (`time_1_kns`, `time_10_kns`).

## DON'T

- **Mix String and Schema Approaches**: Commit to the schema-driven workflow for all its benefits.
- **Forget to Load Sessions**: Comparisons need data. Always `load_folder()` or `load_session()` first.
- **Ignore Trend Information**: The trend symbols (▲, ▼) are your first and best indicator of change.
- **Create New `Timing` Instances Per Benchmark**: Use a single instance for an entire suite to keep results grouped.

# Common Pitfalls, Solved

| Problem | Cause | Solution |
| --- | --- | --- |
| `` `compare_two()` returns `ERROR_NO_SESSIONS` `` | The `Perf_Benchmark__Diff` instance has no session data loaded. | Call `diff.load_session(filepath)` or `diff.load_folder(path)` before comparing. |
| Comparison result shows no common benchmarks found. | The benchmark IDs (`'A_01__dict_create'`) do not match between the sessions being compared. | Ensure you are using consistent, identical benchmark IDs across all runs. |
| Can't access individual raw measurements for a benchmark run. | You are looking in the `timing.results` dictionary, which only contains summaries. | Access the `timing.sessions` dictionary for the full, detailed measurement data. |

# Ready to Build with Confidence? Your Flight Check.

### Section 1: Setup

- Import `Perf_Benchmark__Timing` and `Perf_Benchmark__Diff`.
- Import your chosen exporters (e.g., `Perf_Benchmark__Export__Text`).

### Section 2: Running Benchmarks

- Use a consistent ID format (e.g., `{Section}__{Index}__{name}`).
- Use the `with` context manager for clean execution.
- Save sessions for future comparison using `reporter().save_json()`.

### Section 3: Comparing & Exporting

- Load sessions with `load_folder()`.
- Check the `result.status` enum before processing data.
- Pass the schema result to an exporter to generate your report.

### Key Types to Know

`Schema__Perf__Comparison__Two` `Schema__Perf__Evolution` `Enum__Comparison__Status` `Enum__Benchmark__Trend`

# Get `Perf_Benchmark`

## Install with pip

```
pip install osbot-utils
```

## Explore the Repository

https://github.com/owasp-sbot/OSBot-Utils

Located within `osbot_utils.helpers.performance.benchmark`