Perf_Benchmark v4: A Case Study in Schema-Driven Design

How radical type safety and separation of concerns lead to robust, maintainable systems.

Key Metrics

300+
Passing Tests

140ms
Total Test Execution

22
Core Implementation Files

Zero
Raw `str` Usage

The Pitfalls of Ad-Hoc Performance Benchmarking

Brittle, Untyped Data

- Methods returning formatted **strings** are difficult to parse and use programmatically.
- **`Magic strings`** for status codes (e.g., `"OK"`, `"ERROR"`) lead to typos and runtime errors.
- Generic types like `str` and `dict` hide the true domain model and constraints.

Mixed Logic and Presentation

- Calculation **logic** (e.g., diffing performance results) is tangled with output formatting (`print` statements).
- This makes the core logic difficult to test in isolation.
- Changing the output format requires changing the calculation code.

Ambiguous and Error-Prone

- Without strict schemas, the **structure** of results is **implicit** and can drift over time.
- Error handling becomes a mix of returned strings and exceptions, with no consistent pattern.

Principle 1: Achieve Radical Type Safety

We treated type safety not as a feature, but as the foundation of the entire architecture.

Prone to Runtime Errors

```
# What if 'status' is misspelled? Or the dict key changes?
result = {"status": "ERROR_INSUFFICIENT_SESSIONS", "message": "…"}

if result['status'] == "ERROR_INSUFICIENT_SESSIONS": ...
```

Compile-Time Guarantees

```
# IDE provides autocomplete; typos are compile-time errors.
result: Schema__Perf__Comparison

if result.status == StatusEnum.ERROR_INSUFFICIENT_SESSIONS: ...
```

Benefits

Compile-Time Error Detection

Catches bugs before code is ever run.

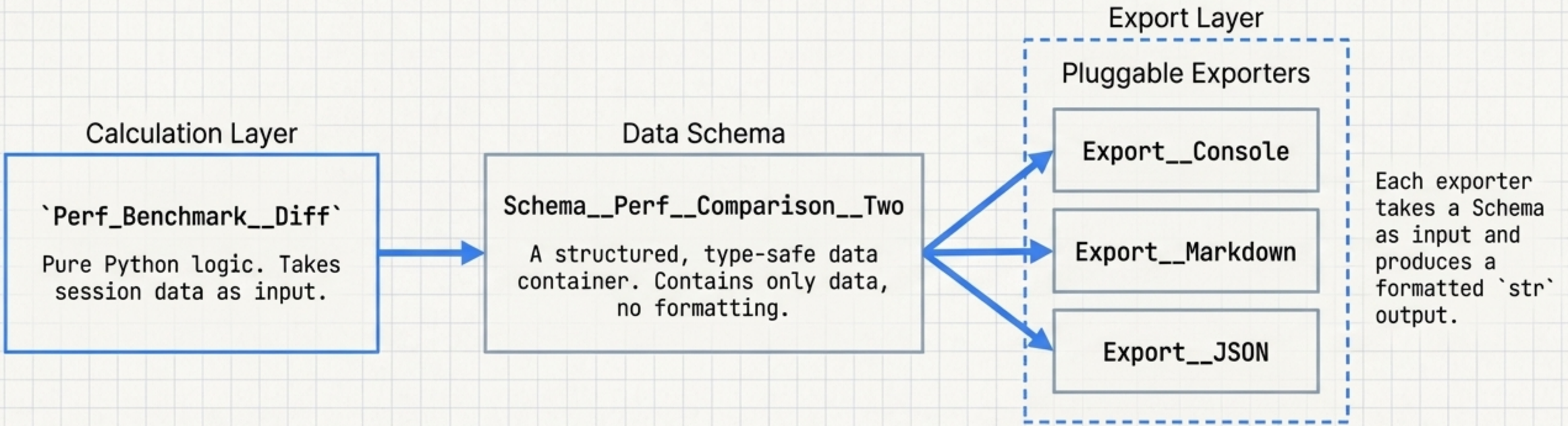
Self-Documenting Code

Types make function signatures and data structures explicit and clear.

Automatic Validation

Schemas enforce constraints on data automatically.

Principle 2: Aggressively Separate Calculation from Presentation



Benefits

- **Independent Testability**: Core logic can be tested without worrying about UI or string formatting.
- **Data Reusability**: Schemas can be saved, serialised to JSON, or sent to a UI without modification.
- **Architectural Flexibility**: New output formats (e.g., HTML, CSV) can be added simply by creating a new exporter, with zero changes to the calculation logic.

Principle 3: Define Domain-Specific Primitives from Day One

Why not just use `Safe_Str` everywhere?

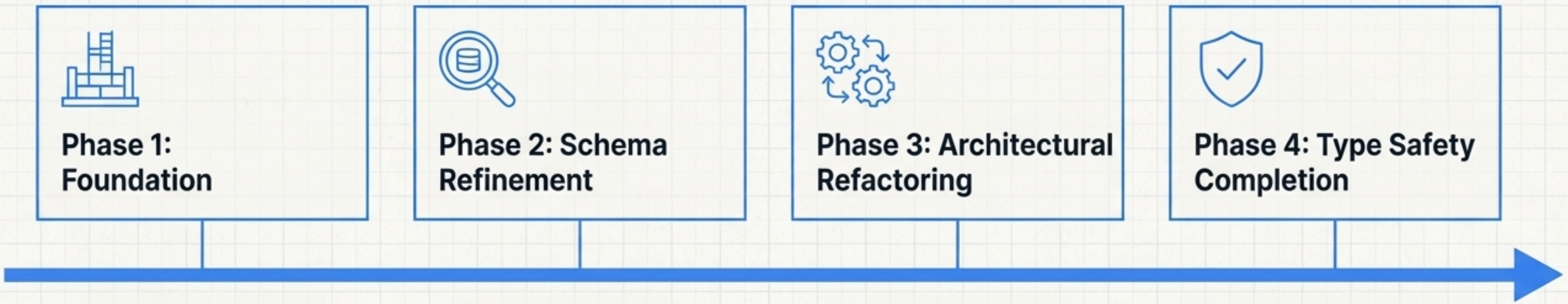
Because different concepts have different constraints. An ID is not a Title, and a Title is not a Description.

Semantic String Types

Type	Max Length	Character Set	Purpose
`Safe_Str__Benchmark_Id`	100	[a-zA-Z0-9_]	Machine-readable identifiers
`Safe_Str__Benchmark__Title`	200	[a-zA-Z0-9_ ()-:,.v]	Short UI headers
`Safe_Str__Benchmark__Description`	4096	Extended ASCII + backticks	Technical documentation
`Safe_Str__Benchmark__Section`	50	[a-zA-Z0-9_]	Dict keys (identifier-safe)

Takeaway: Starting with primitives forces clear thinking about domain boundaries and validation rules early in the process, preventing significant refactoring later.

The Process: A Phased Approach to Architectural Purity



Phase 1: Foundation

- Established `Type_Safe` base classes.
- Created initial schema structures.
- Built basic timing infrastructure.

Phase 2: Schema Refinement

- Moved from generic `Safe_Str` to domain-specific string types.
- Introduced typed collections (`Type_Safe__List`, `Type_Safe__Dict`).
- Added validation constraints to schemas.

Phase 3: Architectural Refactoring

- Separated calculation from presentation in `Perf_Benchmark__Diff`.
- Introduced status enums for robust error handling.
- Created the pluggable export system.

Phase 4: Type Safety Completion

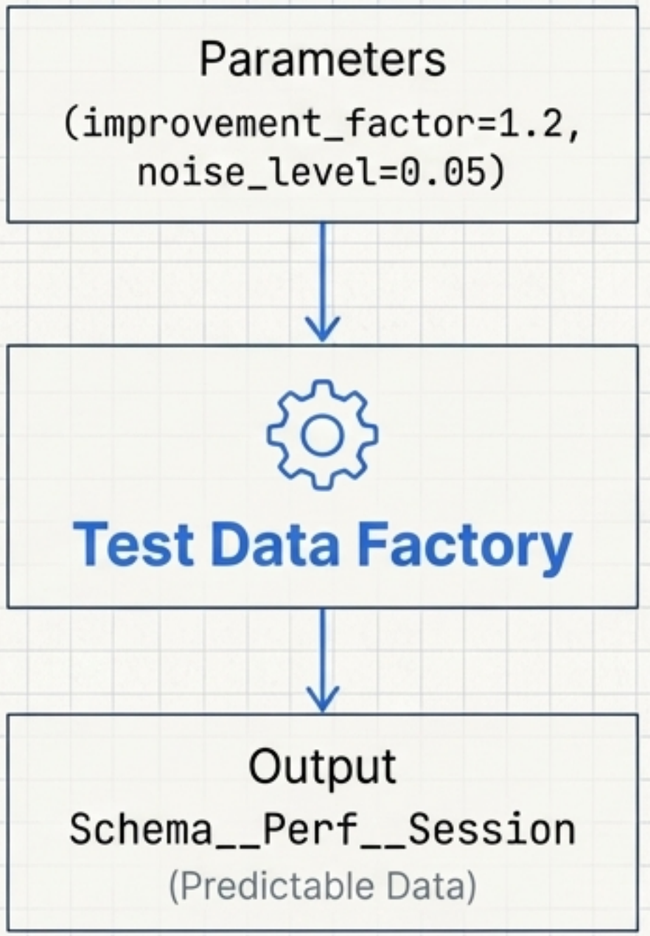
- Removed all remaining raw `str` usage.
- Enforced strictness with the `@type_safe` decorator on all methods.
- Finalised specialised primitives for each domain.

The Method: Building with Confidence Through TDD

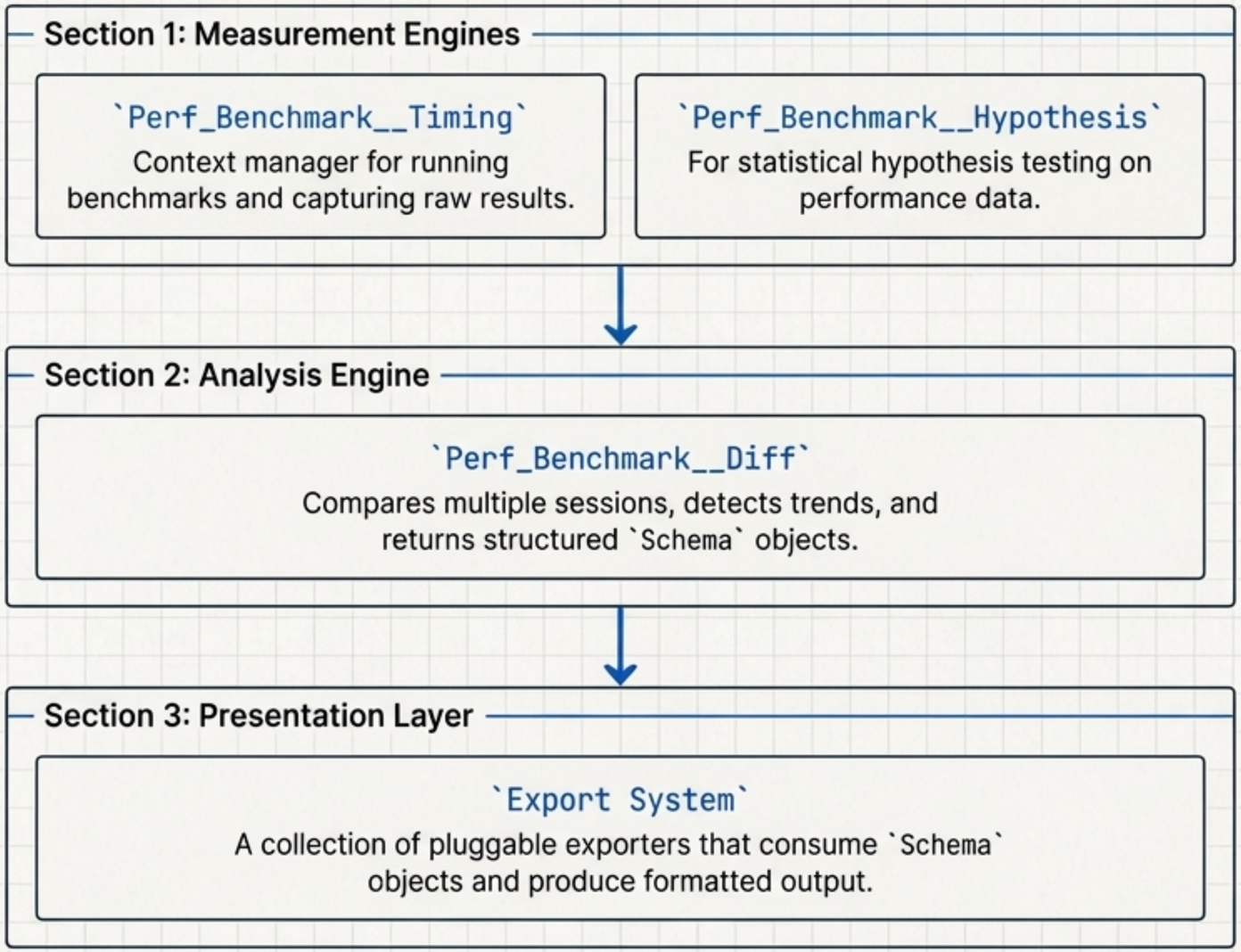
Comprehensive tests were written for every component, ensuring correctness at each step.

The Test Data Factory Pattern

- **Problem**: Initial tests used identical data, which only proved the code ran but didn't exercise all logical paths (e.g., performance improvements vs. regressions).
- **Solution**: A parameterised test data factory was built to generate deterministic, varied test sessions on demand.
- **Benefit**: This enabled exhaustive testing of all comparison logic, trend detection, and edge cases with predictable, repeatable results.



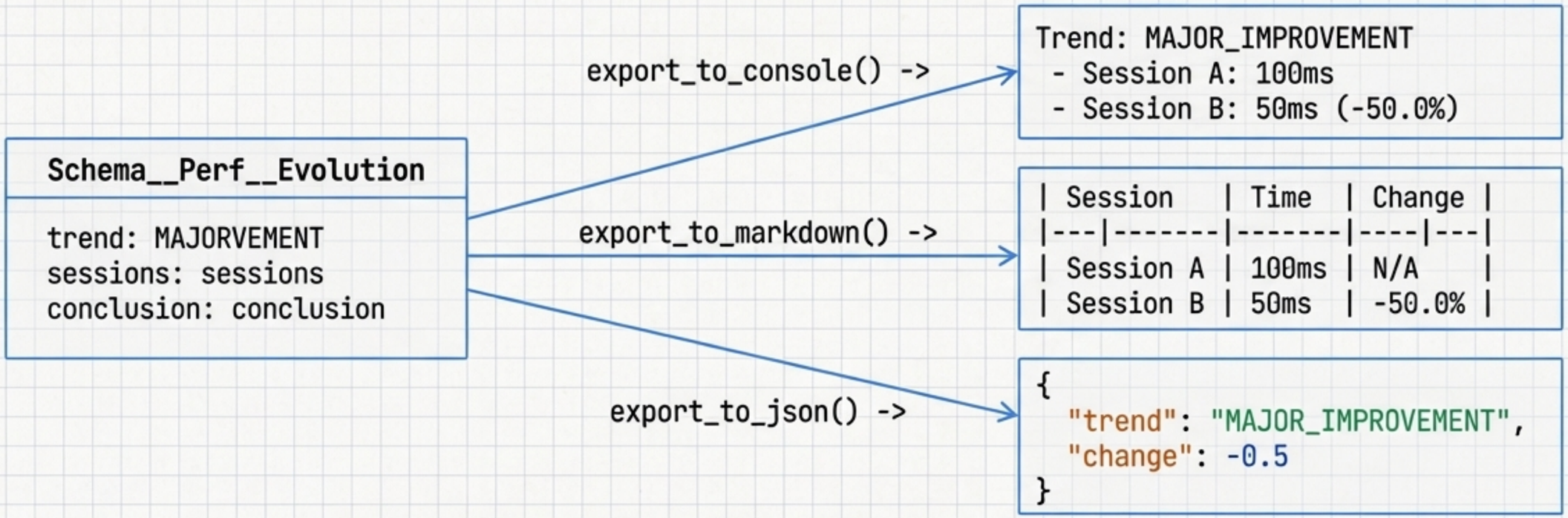
The Result: An Overview of the `Perf_Benchmark v4` Architecture



This layered architecture ensures that measurement, analysis, and presentation are fully decoupled.

The Result: The Pluggable Export System

Key Concept*: "Print is Just Another Export." The realisation that console output is semantically identical to writing to a file, and should be handled by the same architecture.



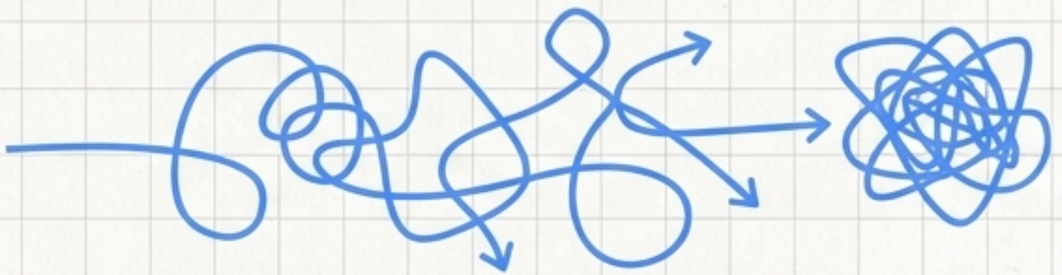
Implementation Detail: Each exporter simply implements a common interface:

```
export_comparison(...) -> str
export_evolution(...) -> str
export_statistics(...) -> str
```

Key Lesson: Start with Primitives, Not Generic Types

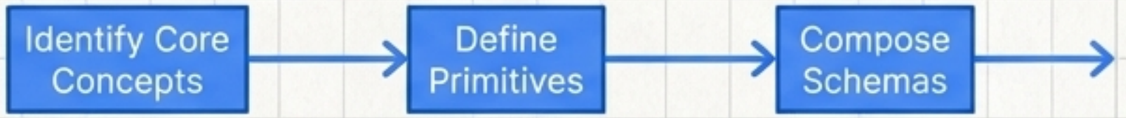
Defining the smallest units of your domain first is the most critical architectural decision.

The Anti-Pattern



- Start with `str`, `int`, `dict`.
- Realise constraints are needed later.
- Begin a large, painful refactoring process to introduce domain-specific types.

The Better Approach

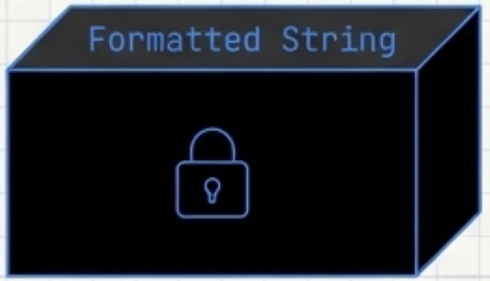


- Identify core domain concepts first (e.g., `Benchmark_Id`, `Test_Title`).
- Define primitive types for them with built-in validation.
- Compose these primitives into larger schemas.

Result: The intent of the code is clear from the start, and refactoring is drastically reduced.

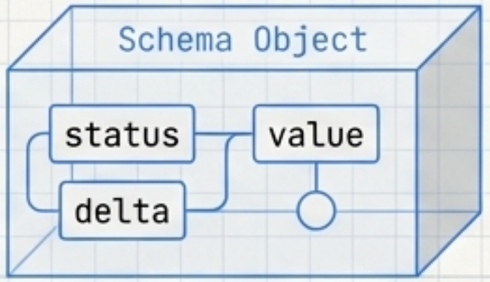
Key Lesson: A Clean Boundary Between Data and Display is Non-Negotiable

The Problem with Returning Formatted Strings



- **No Programmatic Inspection**: You can't easily check `` `if result.is_improvement` ``. You have to parse strings.
- **Untestable Logic**: Core business logic is mixed with presentation details.
- **Brittle**: A small change to the output text can break downstream consumers.
- **Single-Purpose**: The data cannot be easily saved, sent over an API, or rendered in a different format.

The Solution: Return Schemas, Export Separately



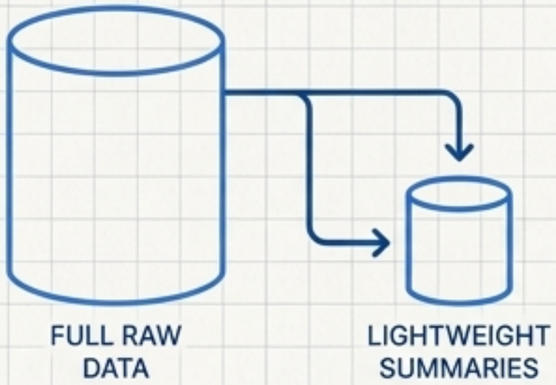
- This provides a stable, machine-readable contract.
- Enables rich, programmatic access to all comparison data.
- Allows any number of presentation formats to be built on top of the same data structure.

Key Lessons: High-Impact Patterns and Refinements

Enums Over Magic Strings

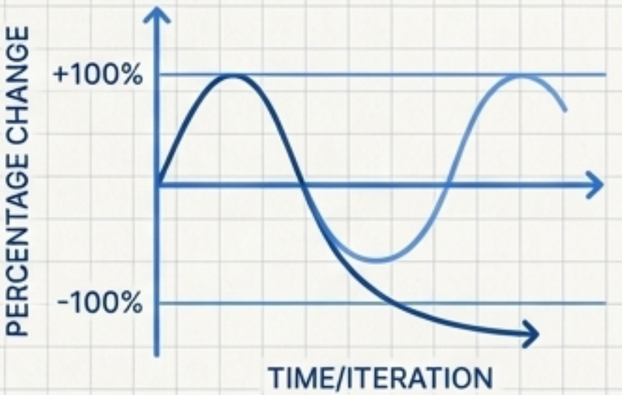
- **Problem:** `if status == "ERROR_NO_SESSIONS"` is prone to typos.
- **Solution:** Use an **Enum**. `if status == Status.ERROR_NO_SESSIONS`.
- **Benefit:** IDE autocomplete and compile-time correctness. The full set of possible states is explicitly enumerated.

The Dual Storage Pattern



- **Problem:** Storing only summary results loses valuable detailed measurement data.
- **Solution:** Store both lightweight summaries and the full raw session data.
- **Benefit:** Allows for both quick access (`timing.results['A'].final_score`) and deep analysis (`timing.sessions['A'].result.stddev_time`).

Symmetric Percentage Bounds



- **Problem:** Percentage change is asymmetric. Improvement is capped at +100%, but regression is unbounded.
- **Solution:** Design schemas with practical, symmetric high bounds for easier handling in UIs and reporting.

Future Considerations: Building on a Solid Foundation

The schema-driven design makes the system highly extensible.

Potential Enhancements

- **Richer Analytics:** Percentile tracking (P50, P95, P99) and automatic outlier detection.

- **Trend Prediction:** Statistical projection of performance trajectory based on historical data.

- **CI/CD Integration:** Native reporters for GitHub Actions, Jenkins, etc., to flag performance regressions in pull requests.

- **Interactive Visualisations:** An HTML exporter that generates interactive drill-down charts.

Architectural Evolution

- **Schema Versioning:** A strategy for introducing non-breaking changes to schemas.
- **Migration Utilities:** Tools to upgrade existing saved session data to new schema versions.

Proven Principles for Robust Systems

`` `300+ Passing Tests in 140ms` `Zero Raw str Usage` ``

The `Perf_Benchmark v4` system demonstrates that a rigorous, type-safe architecture is not a barrier to productivity, but an enabler of quality and maintainability. Its success rests on five key principles:

1. **Domain-specific primitives** over generic types.
2. **Schemas** for **data**, exporters for presentation.
3. **Enums** for **status**, strings for messages.
4. **Test data factories** for predictable, comprehensive testing.
5. **Consistent formatting** for long-term code health.