Type_Safe Performance Schemas
Implementation Briefing

Patterns and primitives for OSBot-Utils v3.69.1

```
REFERENCE: v3.69.1__perf_schemas
TARGET: Developers & Engineers
DOC TYPE: Technical Masterclass / Standard Protocol
STATUS: Implementation Guide
```

THE SHIFT FROM HINTS TO ENFORCEMENT

THE CONTEXT

THE PROBLEM

Python standard type hints are advisory only. They are ignored at runtime, leading to silent failures and data corruption.

```
def calculate(val: int):
    # Ignored at runtime!
```

THE SOLUTION

THE TYPE_SAFE FRAMEWORK

A strict runtime environment that validates data integrity at the moment of assignment.

- **RUNTIME ENFORCEMENT:** Validates every operation during execution.

- **AUTO-INITIALIZATION:** Attributes are automatically instantiated.

- **DOMAIN PRIMITIVES:** Access to 100+ specialized types.

Critical Comparison

Feature	Standard Python Hints	Type_Safe Framework
Runtime Enforcement	Ignored ❌	Every Operation ✅
Auto-initialization	Manual ❌	Automatic ✅
Validation	None ❌	On Assignment ✅
Domain Primitives	None ❌	100+ Specialized Types ✅

Protocol Shift: Moving from a flexible, risky environment to a strict, predictable one.

Protocol 1: Ban Raw Primitives

NEVER use raw 'str', 'int', or 'float' in Type_Safe classes.

Raw ‘str’

Susceptible to SQL injection, XSS, buffer overflows, and command injection.

Raw ‘int’

Prone to overflow bugs and logic errors (e.g., negative values where positive are expected).

Raw ‘float’

Causes financial calculation errors and precision loss due to floating point arithmetic.

Anatomy of a Schema

Schema classes must be Pure Data Containers.

POLLUTED SCHEMA

```
class UserSchema:
    name: str
    email: str
    age: int
    ---
    def validate(self):
        # Business logic here
        ...
    ---
    def save(self):
        # DB operations
        ...
```

Error Red: NO Methods. NO Business Logic.

CLEAN SCHEMA

```
class UserSchema:
    name: Safe_Str
    email: Safe_Email
    age: Positive_Int
```

Success Green: Pure Data Only.

Numeric Primitives & Timing Standards

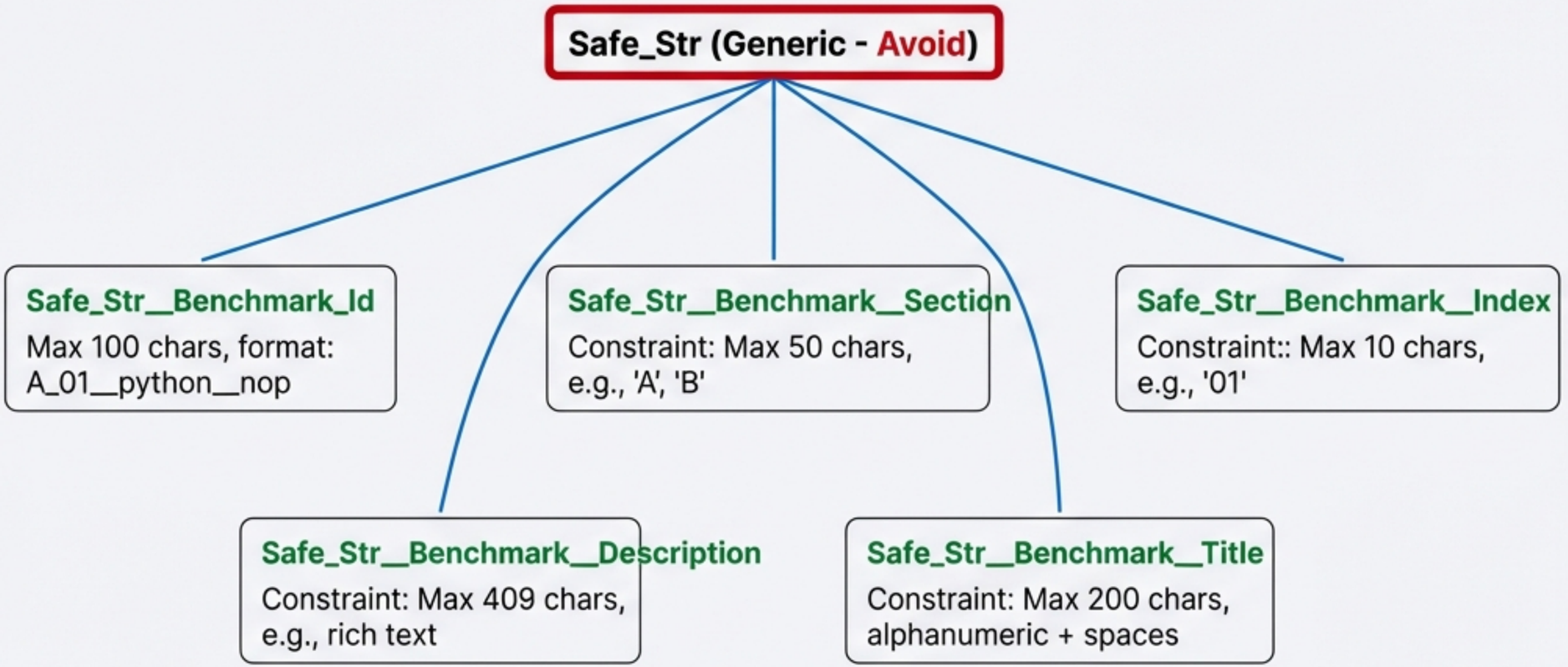
`Safe_Int`	`Safe_UInt`	`Safe_Float`
Signed integer with validation.	Unsigned integer (≥0).	Float with precision control.
Default: `0`	Default: `0`	Default: `0.0`

Performance Rule:
For performance timing values, ALWAYS use ‘`Safe_UInt`’.

Nanoseconds cannot be negative. Using signed integers for time measurements introduces potential logic bugs.

Domain-Specific String Primitives

Move from generic strings to specific constraints.



SIDEBAR NOTE:

These primitives are imported from the 'osbot_utils' library, not standard Python.

Automation: Identifiers & Time

Auto-Generating Primitives

`Timestamp_Now`

Automatically generates current timestamp (ms) upon instantiation.

`Random_Guid`

Automatically generates a random UUID4 string.

Safety Standards

`Safe_Id`	General safe identifier (512 chars).
`Safe_Str__File__Path`	Max 1024 chars, file system path validation.
`Safe_Str__File__Name`	Strict filename validation.

Precision Logic: Percentages & Enums

Percentage Primitives

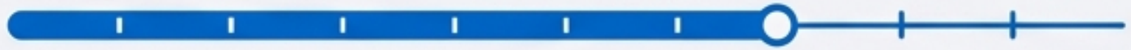
`Safe_Float__Percentage_Change`
Tracking variance.

Range: -∞ to +∞

`Safe_Float__Percentage_Exact`
Range 0.0-100.0 (2 decimal precision).

`0.0` Value `100.0`

`Safe_UInt__Percentage`
Integer 0-100.



`0` Value `100`

Configuration Modes

~~`mode = "fast"`~~

Unsafe: String literal

`mode = Enum__Measure_Mode.FAST`

Safe: Defined Enum

Mandate: Use defined Enums for all configuration states.

The Collections Strategy

Stop using 'List[str]' or 'Dict[str, int]'.

`{CollectionType} __ {Domain} __ {Description}`

Base collection type (e.g., Dict, List, Set)

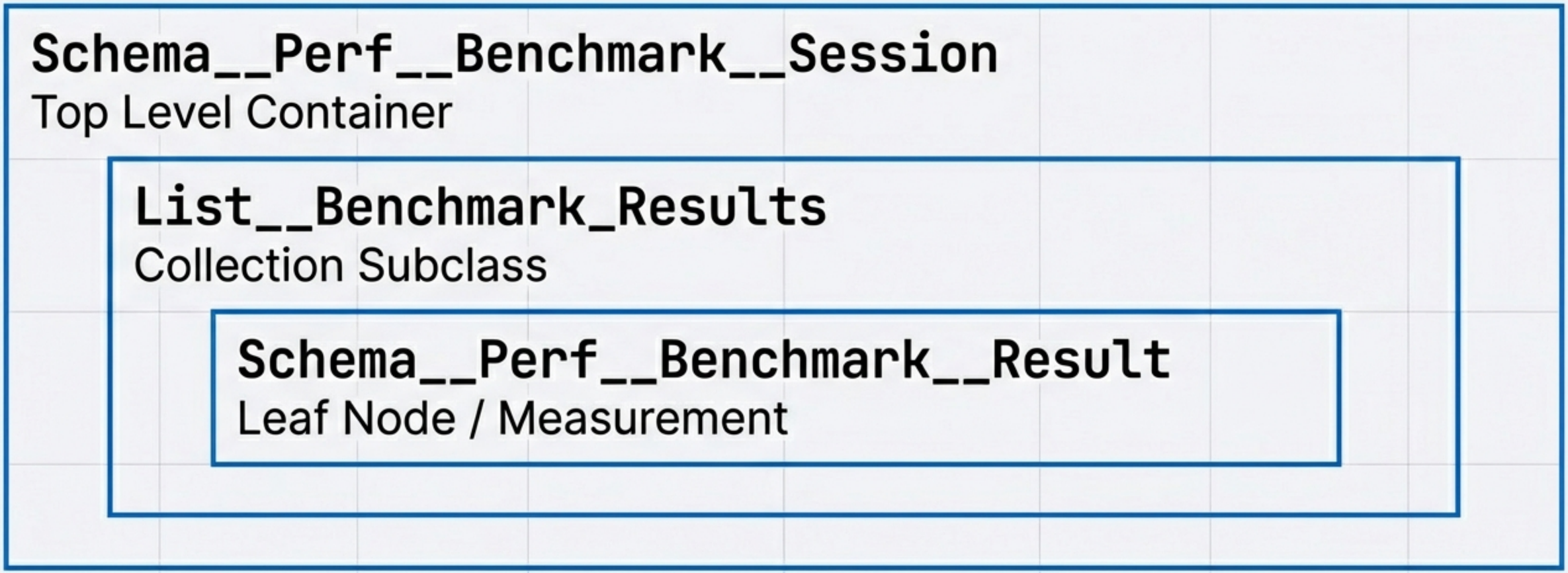
Business domain context

Specific use case or details

Examples

Collection	Named Subclasses
Dicts	`Dict__Benchmark_Results, Dict__Benchmark__Legend`
Lists	`List__Scores, List__Titles, List__Benchmark_Comparisons`
Sets	`Set__Permission__Ids`

Schema Composition Architecture



Complex data structures are built by nesting simple, type-safe schemas.

Migration Case Study

Transforming “Schema__Perf_Report__Benchmark”

BEFORE - Raw Types

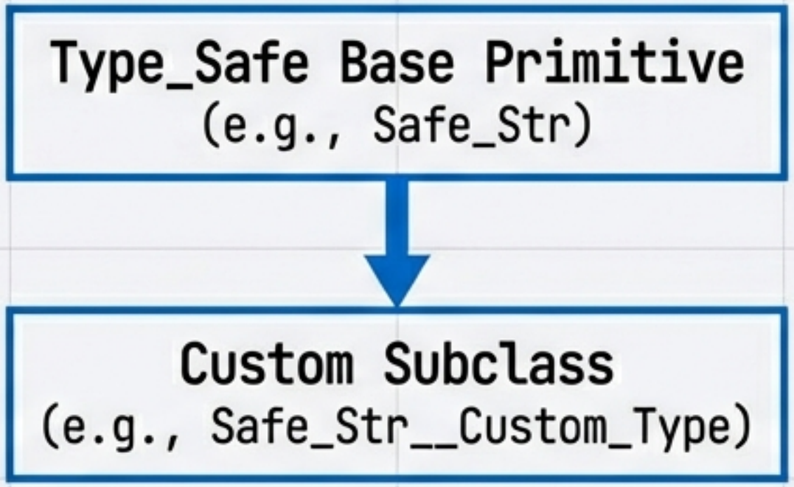
```
class Schema__Perf_Report__Benchmark:
    name: str
    name: str
    timestamp: int
    score: float
```

AFTER - Type_Safe

```
class Schema__Perf_Report__Benchmark
    (Type_Safe):
    name: Safe_Str__Benchmark_Id
    timestamp: Timestamp_Now
    score: Safe_UInt
```

Extending the Framework

When the exact primitive you need does not exist, create a custom subclass to inherit safety features.



Strings: Subclass
(e.g., Safe_Str')

Integers: Subclass
(e.g., Safe_UInt')

Floats: Subclass
(e.g., Safe_Float')

Custom types inherit auto-validation and initialization automatically.

The Implementation Checklist

- [ ] **Inherit** from `Type_Safe` (All schemas must extend base).
- [ ] **Ban raw primitives** (No `str`, `int`, `float`).
- [ ] Use `Safe_UInt` for timing (Positive nanoseconds).
- [ ] Use **Enums** for modes (No string literals).
- [ ] Use `Timestamp_Now` (Auto-generated).
- [ ] Create collection subclasses (e.g., `List__Scores`).
- [ ] Follow naming prefixes (`Dict__*`, `List__*`, `Set__*`).
- [ ] No methods (Pure data containers only).
- [ ] Use domain primitives (e.g., `Safe_Str__Benchmark_Id`).
- [ ] Inline comments (Explain constraints).

The Ecosystem & Reference

Reuse existing schemas before building new ones.

- `Schema__Perf__Benchmark__Result`
- `Schema__Perf__Benchmark__Session`
- `Schema__Perf__Benchmark__Comparison`
- `Schema__Perf__Benchmark__Evolution`
- `Schema__Perf__Benchmark__Evolution`
- `Schema__Perf__Statistics`
- `Schema__Perf__Hypothesis__Result`
- `Schema__Perf_Benchmark__Timing__Config`

Build on these foundations.
Ensure reliability through strict typing.