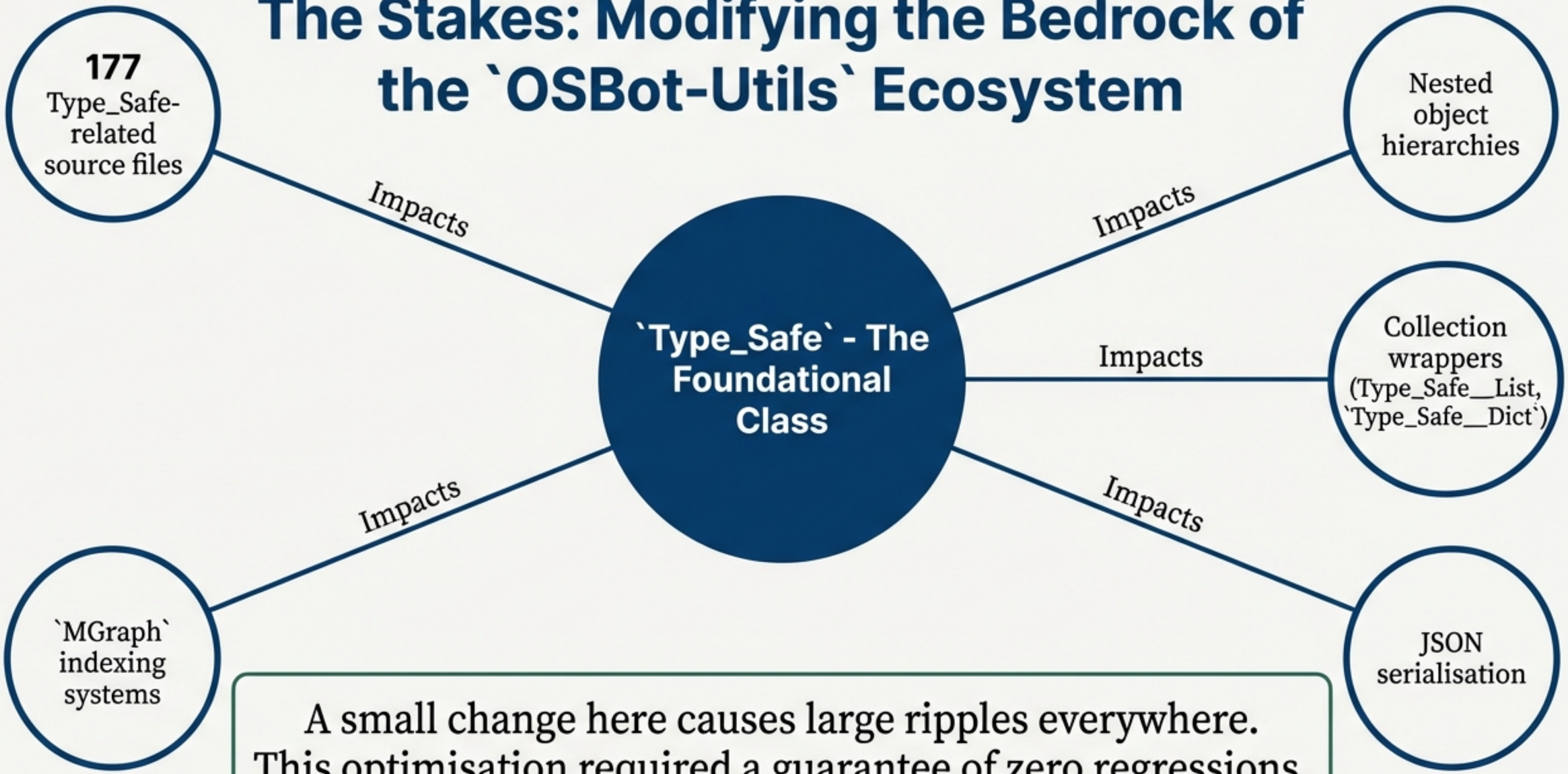
Taming the Backbone: How Comprehensive Testing Unlocked a 20x Performance Gain in `Type_Safe`

A Technical Debrief on `Type_Safe__On_Demand` Test Coverage

Version: 1.0.0 | Total Tests: 41 | Status: ✅ All Passing

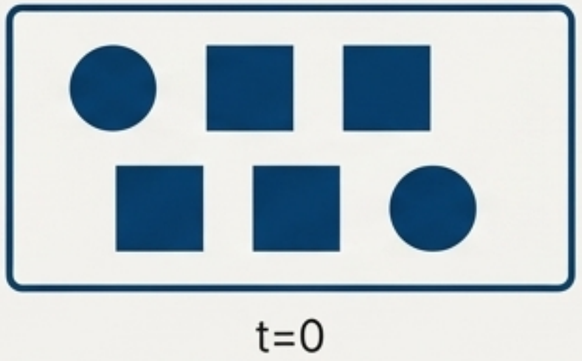
The Stakes: Modifying the Bedrock of the `OSBot-Utils` Ecosystem



A small change here causes large ripples everywhere.
This optimisation required a guarantee of zero regressions.

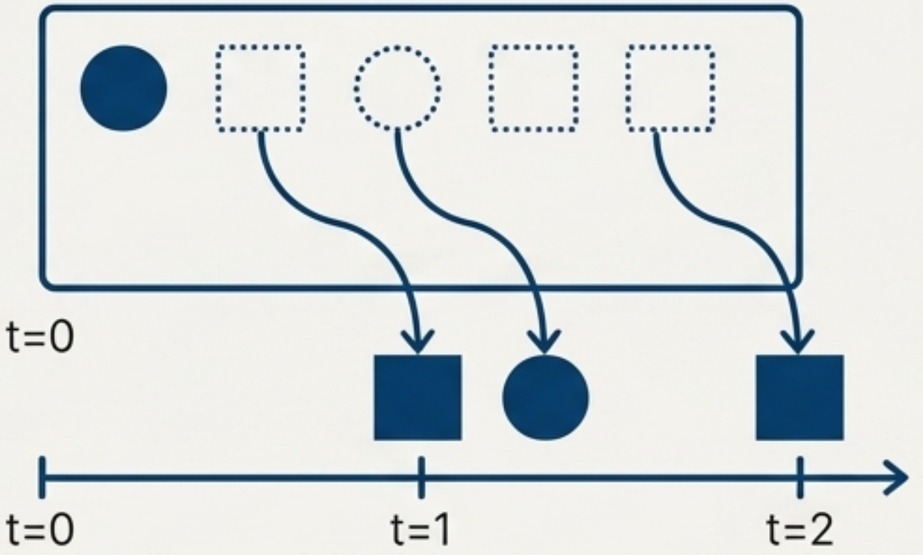
Changing the Nature of Time: The Challenge of On-Demand Initialisation

`Type_Safe` (Eager)



All objects exist immediately after construction.

`Type_Safe__On_Demand` (Lazy)



Objects exist only when first accessed.

This fundamental shift in *when* objects exist introduces significant new risks.

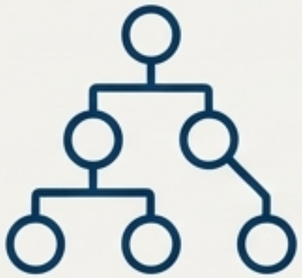
The Four Core Testing Challenges

1. Temporal Dependencies

Code assuming immediate object existence could now fail or trigger unexpected creation.

2. State Visibility

The internal `_on_demand__types` state must be correctly maintained throughout the entire object lifecycle.



3. Inheritance Complexity

Mixed hierarchies of eager (`Type_Safe`) and lazy (`Type_Safe__On_Demand`) classes must interoperate predictably.

4. Serialisation Timing

`json()` and `obj()` calls now interact with a partially-realised object state, creating non-obvious failure modes.

The Solution: A 41-Test Gauntlet for Guaranteed Stability

29

Functional Tests

Validating correctness across all features and edge cases.

12

Performance Tests

Measuring speed, object creation, and memory behaviour against concrete targets.

A systematic approach to neutralise every identified risk.

Functional Tests (1/3): Core Mechanics and Collection Integration

Core Mechanics

Test	Purpose	Assertion
`test__getattribute__nested_on_demand`	Validates that on-demand creation cascades correctly through deep hierarchies, the primary MGraph__Index use case.	Each level defers creation independently until accessed.
`test__init__`	Ensures basic construction, inheritance, and internal state are initialised correctly.	Primitives are created; on-demand types are registered.
`test_json`	Verifies that JSON serialisation works correctly with a mix of accessed and pending attributes.	Accessed attributes appear in output; pending do not.

Collection Integration

Test	Purpose
`test__dict_attribute`	Ensures `Type_Safe__Dict` wrappers are handled correctly.
`test__list_attribute`	Ensures `Type_Safe__List` wrappers are handled correctly.

Why Collections Are NOT Deferred

Collections are cheap to create (empty containers) and frequently accessed immediately. Deferring them adds overhead for no meaningful benefit, so they are created eagerly.

Functional Tests (2/3): Primitives and Mixed Inheritance

Primitive Handling

Test	`test__primitive_attributes` `test__safe_primitive_attributes`
Purpose	Validates that native primitives (`str`, `int`) and `Type_Safe__Primitive` types (`Safe_Str`) are handled correctly.
Key Takeaway	Primitives are never deferred, as they are cheap and provide immediate validation.

Mixed Inheritance

Test	`test__type_safe_child_of_on_demand` `test__on_demand_child_of_type_safe`
Purpose	Validates interoperability when Type_Safe and Type_Safe__On_Demand are mixed in the same hierarchy.

Key Insight: Direction Matters

An `On_Demand` parent *defers* its `Type_Safe` child. A `Type_Safe` parent *immediately creates* its `On_Demand` child. The parent's initialisation strategy always wins.

Functional Tests (3/3): Probing Edge Cases and Preventing Regressions

Edge Case Spotlight: Assignment to Pending Attributes

`test__attribute_assignment_after_init`

This test documents and validates a subtle behaviour: assigning a value to an attribute that has not yet been created on-demand may be overwritten when that attribute is first accessed. This shows we've considered state interaction issues.

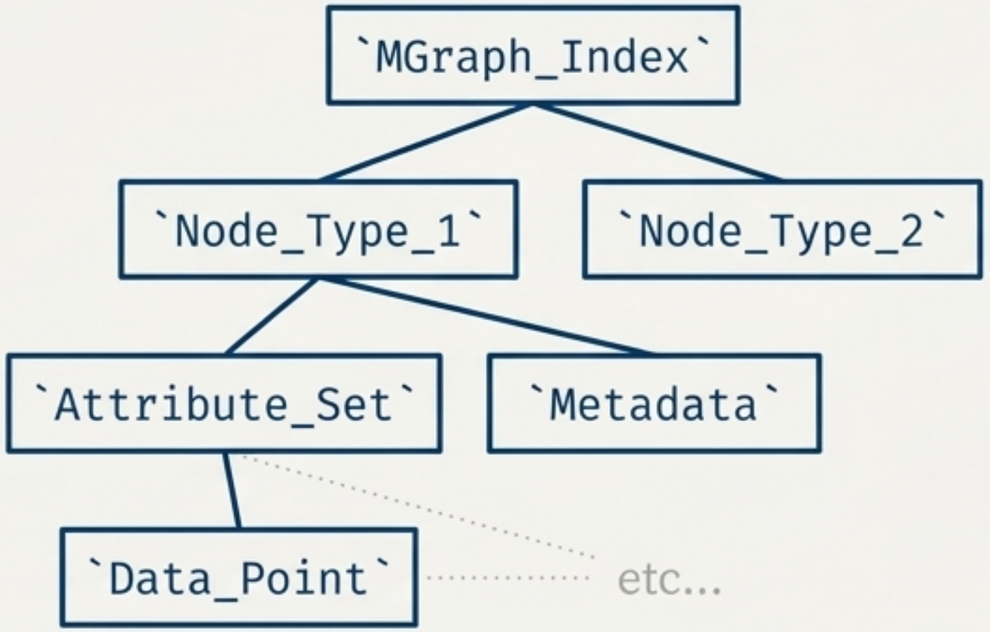
Regression Guard: State Independence

`test__multiple_instances_independent`

A critical guard to ensure that two instances of an `On_Demand` class do not share internal state, preventing class-level pollution bugs from being reintroduced.

Tests also cover empty classes, private attributes, and circular reference potential to ensure robustness.

Performance Tests: Simulating the Real-World `MGraph__Index` Scenario



The performance tests don't use abstract models. They create a realistic simulation of the `MGraph__Index` object hierarchy which can have 10+ levels of nesting.

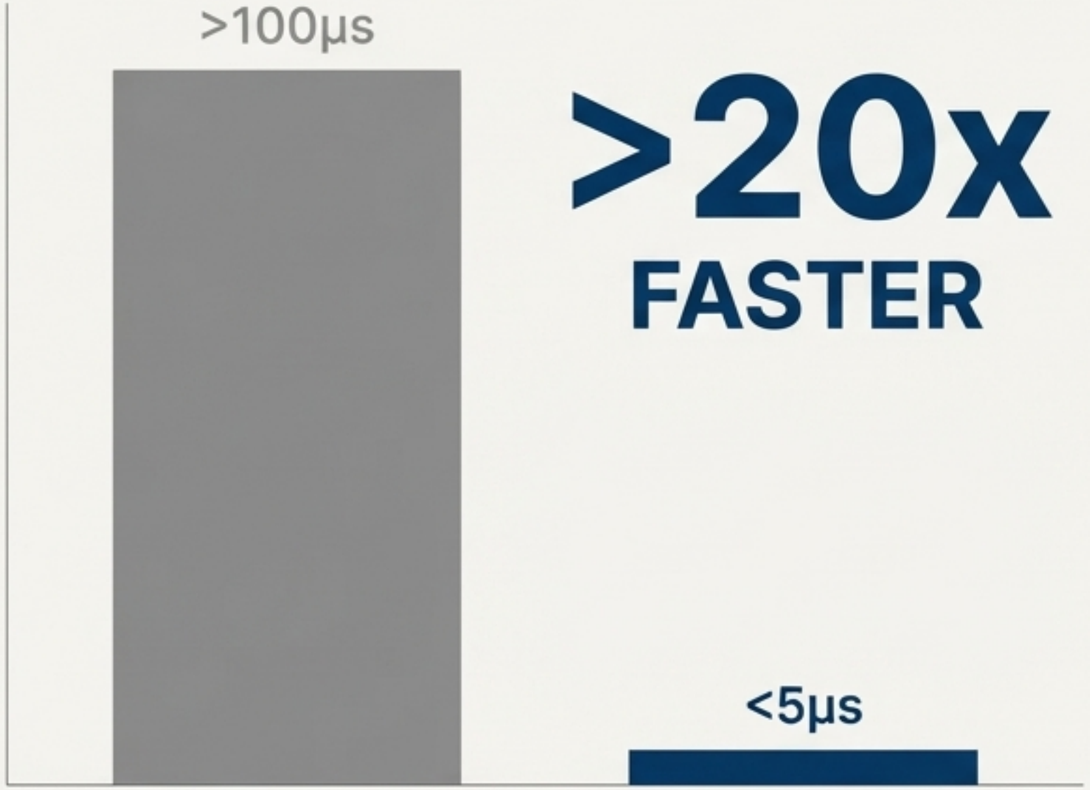
This ensures our benchmarks directly reflect the performance gains seen in the production system that drove this optimisation.

The Results: A Staggering Reduction in Construction Overhead

Objects Created During Initialisation

>98%
REDUCTION

Time to Construct `MGraph__Index`



>20x
FASTER

Beyond Construction: Analysing Attribute Access Performance

Answering the question: “What happens when you actually use the objects?”

- **Partial Access:** For the common case of accessing only a few attributes, On-Demand remains ≥5x faster overall.
- **Full Access:** In the worst-case scenario of accessing every attribute, performance is at least not slower than the original eager implementation.

A Foundation of Rigour: Test Methodology and Patterns

Context Manager Pattern

Functional tests use `Type_Safe` context managers to isolate and inspect object state cleanly.

```
with Type_Safe(obj) as _:
    # assertions about internal state
    assert _.get('_on_demand__pending_
count') == 10
```

Performance Measurement Pattern

All benchmarks use `Performance_Measure__Session` for consistent and reliable timing.

```
with Performance_Measure__Session() as
performance:
    My_On_Demand_Class()

# performance.duration_in_microseconds
```

Object Counting Pattern

We monkey-patch `Type_Safe.__init__` to precisely count object instantiations without altering source code.

```
# In test setup
original_init = Type_Safe.__init__
self.counter = 0
def new_init(self, *args, **kwargs):
    self.counter += 1
    original_init(self, *args, **kwargs)

...
```

The Full Picture: A Comprehensive Coverage Summary

✅ What IS Tested

- Basic construction and inheritance
- On-demand attribute creation
- Nested hierarchy behaviour
- Collection type handling (Dict, List)
- Primitive type handling (native and Safe)
- Mixed `Type_Safe` / `On_Demand` inheritance
- JSON serialisation/deserialisation
- Performance vs target (200µs)
- Object count reduction (98%)
- `Html_MGraph` 6-index scenario

☐ Future Work

- Thread safety tests
- Pickle serialisation tests
- Deep copy behaviour tests
- Integration with `Type_Safe` validators
- Memory usage benchmarks

Verdict: Performance Unlocked, Stability Guaranteed

The Challenge Revisited

Safely optimise the foundational `Type_Safe` class without breaking `177` dependent files and multiple core systems.

>20x

Faster Construction

>98%

Fewer Objects Created

41

Robust Tests Passed

The `Type_Safe__On_Demand` optimisation is not just fast; it is proven safe and fully compatible across the entire ecosystem, thanks to a comprehensive and meticulously designed test suite.

Appendix: Resources & Deeper Dive

Test Files

- `test_Type_Safe__On_Demand.py` (Functional Tests)
- `test_perf_Type_Safe__On_Demand.py` (Performance Tests)

Related Documentation

- Link to original technical debrief document